

# Isabelle/HOL — Higher-Order Logic

June 21, 2010

## Contents

<b>1</b>	<b>Code-Generator: Loading the code generator modules</b>	<b>23</b>
<b>2</b>	<b>HOL: The basis of Higher-Order Logic</b>	<b>23</b>
2.1	Primitive logic . . . . .	24
2.1.1	Core syntax . . . . .	24
2.1.2	Additional concrete syntax . . . . .	25
2.1.3	Axioms and basic definitions . . . . .	26
2.2	Fundamental rules . . . . .	27
2.2.1	Equality . . . . .	27
2.2.2	Congruence rules for application . . . . .	28
2.2.3	Equality of booleans – iff . . . . .	28
2.2.4	True . . . . .	29
2.2.5	Universal quantifier . . . . .	29
2.2.6	False . . . . .	30
2.2.7	Negation . . . . .	30
2.2.8	Implication . . . . .	30
2.2.9	Existential quantifier . . . . .	31
2.2.10	Conjunction . . . . .	31
2.2.11	Disjunction . . . . .	32
2.2.12	Classical logic . . . . .	32
2.2.13	Unique existence . . . . .	33
2.2.14	THE: definite description operator . . . . .	34
2.2.15	Classical intro rules for disjunction and existential quantifiers . . . . .	35
2.2.16	Intuitionistic Reasoning . . . . .	36
2.2.17	Atomizing meta-level connectives . . . . .	37
2.2.18	Atomizing elimination rules . . . . .	38
2.3	Package setup . . . . .	39
2.3.1	Sledgehammer setup . . . . .	39
2.3.2	Classical Reasoner setup . . . . .	39
2.3.3	Simplifier . . . . .	42

2.3.4	Generic cases and induction . . . . .	51
2.3.5	Coherent logic . . . . .	54
2.3.6	Reorienting equalities . . . . .	55
2.4	Other simple lemmas and lemma duplicates . . . . .	55
2.5	Basic ML bindings . . . . .	56
2.6	Code generator setup . . . . .	57
2.6.1	SML code generator setup . . . . .	57
2.6.2	Generic code generator preprocessor setup . . . . .	58
2.6.3	Equality . . . . .	58
2.6.4	Generic code generator foundation . . . . .	59
2.6.5	Generic code generator target languages . . . . .	61
2.6.6	Evaluation and normalization by evaluation . . . . .	62
2.7	Counterexample Search Units . . . . .	63
2.7.1	Quickcheck . . . . .	63
2.7.2	Nitpick setup . . . . .	63
2.8	Preprocessing for the predicate compiler . . . . .	64
2.9	Legacy tactics and ML bindings . . . . .	64
<b>3</b>	<b>Orderings: Abstract orderings</b>	<b>66</b>
3.1	Syntactic orders . . . . .	66
3.2	Quasi orders . . . . .	67
3.3	Partial orders . . . . .	68
3.4	Linear (total) orders . . . . .	69
3.5	Reasoning tools setup . . . . .	72
3.6	Bounded quantifiers . . . . .	78
3.7	Transitivity reasoning . . . . .	79
3.8	Monotonicity, least value operator and min/max . . . . .	85
3.9	Top and bottom elements . . . . .	87
3.10	Dense orders . . . . .	87
3.11	Wellorders . . . . .	88
3.12	Order on bool . . . . .	89
3.13	Order on functions . . . . .	90
3.14	Name duplicates . . . . .	91
<b>4</b>	<b>Groups: Groups, also combined with orderings</b>	<b>92</b>
4.1	Fact collections . . . . .	92
4.2	Abstract structures . . . . .	93
4.3	Generic operations . . . . .	94
4.4	Semigroups and Monoids . . . . .	95
4.5	Groups . . . . .	97
4.6	(Partially) Ordered Groups . . . . .	101
4.7	Support for reasoning about signs . . . . .	104
4.8	Tools setup . . . . .	116

<b>5</b>	<b>Lattices: Abstract lattices</b>	<b>118</b>
5.1	Abstract semilattice . . . . .	118
5.2	Idempotent semigroup . . . . .	119
5.3	Concrete lattices . . . . .	119
5.3.1	Intro and elim rules . . . . .	120
5.3.2	Equational laws . . . . .	121
5.3.3	Strict order . . . . .	124
5.4	Distributive lattices . . . . .	125
5.5	Bounded lattices and boolean algebras . . . . .	125
5.6	Uniqueness of inf and sup . . . . .	129
5.7	$\min/\max$ on linear orders as special case of $op \sqcap/op \sqcup$ . . . . .	129
5.8	Bool as lattice . . . . .	130
5.9	Fun as lattice . . . . .	131
<b>6</b>	<b>Set: Set theory for higher-order logic</b>	<b>132</b>
6.1	Sets as predicates . . . . .	132
6.2	Subsets and bounded quantifiers . . . . .	134
6.3	Basic operations . . . . .	141
6.3.1	Subsets . . . . .	141
6.3.2	Equality . . . . .	142
6.3.3	The universal set – UNIV . . . . .	142
6.3.4	The empty set . . . . .	143
6.3.5	The Powerset operator – Pow . . . . .	144
6.3.6	Set complement . . . . .	144
6.3.7	Binary union – Un . . . . .	144
6.3.8	Binary intersection – Int . . . . .	145
6.3.9	Set difference . . . . .	146
6.3.10	Augmenting a set – <i>insert</i> . . . . .	146
6.3.11	Singletons, using insert . . . . .	147
6.3.12	Image of a set under a function . . . . .	148
6.3.13	Some rules with <i>if</i> . . . . .	149
6.4	Further operations and lemmas . . . . .	150
6.4.1	The “proper subset” relation . . . . .	150
6.4.2	Derived rules involving subsets. . . . .	151
6.4.3	Equalities involving union, intersection, inclusion, etc. . . . .	151
6.4.4	Monotonicity of various operations . . . . .	161
6.4.5	Inverse image of a function . . . . .	162
6.4.6	Getting the Contents of a Singleton Set . . . . .	164
6.4.7	Least value operator . . . . .	164
6.5	Misc . . . . .	164
<b>7</b>	<b>Typedef: HOL type definitions</b>	<b>166</b>

<b>8 Complete-Lattice: Complete lattices, with special focus on sets</b>	<b>168</b>
8.1 Syntactic infimum and supremum operations . . . . .	169
8.2 Abstract complete lattices . . . . .	169
8.3 <i>bool</i> and $- \Rightarrow -$ as complete lattice . . . . .	171
8.4 Union . . . . .	173
8.5 Unions of families . . . . .	174
8.6 Inter . . . . .	177
8.7 Intersections of families . . . . .	179
8.8 Distributive laws . . . . .	181
8.9 Complement . . . . .	182
8.10 Miniscoping and maxiscoping . . . . .	182
<b>9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions</b>	<b>185</b>
9.1 Least and greatest fixed points . . . . .	185
9.2 Proof of Knaster-Tarski Theorem using <i>lfp</i> . . . . .	186
9.3 General induction rules for least fixed points . . . . .	186
9.4 Proof of Knaster-Tarski Theorem using <i>gfp</i> . . . . .	187
9.5 Coinduction rules for greatest fixed points . . . . .	188
9.6 Even Stronger Coinduction Rule, by Martin Coen . . . . .	189
9.7 Inductive predicates and sets . . . . .	190
9.8 Inductive datatypes and primitive recursion . . . . .	190
<b>10 Fun: Notions about functions</b>	<b>191</b>
10.1 The Identity Function <i>id</i> . . . . .	191
10.2 The Composition Operator $f \circ g$ . . . . .	192
10.3 The Forward Composition Operator <i>fcomp</i> . . . . .	192
10.4 Injectivity and Surjectivity . . . . .	193
10.5 Function Updating . . . . .	198
10.6 <i>override-on</i> . . . . .	200
10.7 <i>swap</i> . . . . .	200
10.8 Inversion of injective functions . . . . .	201
10.9 Proof tool setup . . . . .	202
10.10 Code generator setup . . . . .	203
<b>11 Product-Type: Cartesian products</b>	<b>204</b>
11.1 <i>bool</i> is a datatype . . . . .	204
11.2 The <i>unit</i> type . . . . .	204
11.3 The product type . . . . .	206
11.3.1 Type definition . . . . .	206
11.3.2 Tuple syntax . . . . .	207
11.3.3 Code generator setup . . . . .	209
11.3.4 Fundamental operations and properties . . . . .	211

11.3.5	Derived operations . . . . .	219
11.4	Inductively defined sets . . . . .	227
11.5	Legacy theorem bindings and duplicates . . . . .	227
<b>12</b>	<b>Sum-Type: The Disjoint Sum of Two Types</b>	<b>227</b>
12.1	Construction of the sum type and its basic abstract operations	228
12.2	Projections . . . . .	229
12.3	The Disjoint Sum of Sets . . . . .	231
<b>13</b>	<b>Rings: Rings</b>	<b>231</b>
<b>14</b>	<b>Fields: Fields</b>	<b>259</b>
<b>15</b>	<b>Nat: Natural numbers</b>	<b>277</b>
15.1	Type <i>ind</i> . . . . .	278
15.2	Type <i>nat</i> . . . . .	278
15.3	Arithmetic operators . . . . .	280
15.3.1	Addition . . . . .	282
15.3.2	Difference . . . . .	282
15.3.3	Multiplication . . . . .	283
15.4	Orders on <i>nat</i> . . . . .	284
15.4.1	Operation definition . . . . .	284
15.4.2	Introduction properties . . . . .	286
15.4.3	Elimination properties . . . . .	286
15.4.4	Inductive (?) properties . . . . .	287
15.4.5	<i>min</i> and <i>max</i> . . . . .	290
15.4.6	Monotonicity of Addition . . . . .	291
15.4.7	Additional theorems about $op \leq$ . . . . .	293
15.4.8	More results about difference . . . . .	297
15.4.9	Monotonicity of Multiplication . . . . .	299
15.5	Natural operation of natural numbers on functions . . . . .	301
15.6	Embedding of the Naturals into any <i>semiring-1: of-nat</i> . . . . .	302
15.7	The Set of Natural Numbers . . . . .	304
15.8	Further Arithmetic Facts Concerning the Natural Numbers . . . . .	305
15.9	The divides relation on <i>nat</i> . . . . .	309
15.10	size of a datatype value . . . . .	311
15.11	code module namespace . . . . .	311
<b>16</b>	<b>Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums</b>	<b>311</b>
16.1	The datatype universe . . . . .	311
16.2	Freeness: Distinctness of Constructors . . . . .	314
16.3	Set Constructions . . . . .	317

<b>17 Record: Extensible records with structural subtyping</b>	<b>322</b>
17.1 Introduction . . . . .	322
17.2 Operators and lemmas for types isomorphic to tuples . . . . .	323
17.3 Logical infrastructure for records . . . . .	324
17.4 Concrete record syntax . . . . .	330
17.5 Record package . . . . .	331
<b>18 Power: Exponentiation</b>	<b>331</b>
18.1 Powers for Arbitrary Monoids . . . . .	331
18.2 Exponentiation for the Natural Numbers . . . . .	339
18.3 Code generator tweak . . . . .	340
<b>19 Option: Datatype option</b>	<b>341</b>
19.0.1 Operations . . . . .	341
19.0.2 Code generator setup . . . . .	342
<b>20 Finite-Set: Finite sets</b>	<b>343</b>
20.1 Predicate for finite sets . . . . .	343
20.2 Class <i>finite</i> . . . . .	353
20.3 A basic fold functional for finite sets . . . . .	354
20.3.1 From <i>fold-graph</i> to <i>fold</i> . . . . .	355
20.3.2 Expressing set operations via <i>fold</i> . . . . .	358
20.4 The derived combinator <i>fold-image</i> . . . . .	360
20.5 A fold functional for non-empty sets . . . . .	363
20.5.1 Determinacy for <i>fold1Set</i> . . . . .	368
20.5.2 Lemmas about <i>fold1</i> . . . . .	368
20.6 Locales as mini-packages for fold operations . . . . .	368
20.6.1 The natural case . . . . .	368
20.6.2 The natural case with idempotency . . . . .	370
20.6.3 The image case with fixed function . . . . .	371
20.6.4 The image case with flexible function . . . . .	373
20.6.5 The image case with fixed function and idempotency . . . . .	375
20.6.6 The image case with flexible function and idempotency . . . . .	376
20.6.7 The neutral-less case . . . . .	376
20.6.8 The neutral-less case with idempotency . . . . .	378
20.7 Finite cardinality . . . . .	379
20.7.1 Cardinality of image . . . . .	384
20.7.2 Cardinality of sums . . . . .	385
20.7.3 Cardinality of the Powerset . . . . .	386
20.7.4 Relating injectivity and surjectivity . . . . .	386

<b>21 Relation: Relations</b>	<b>387</b>
21.1 Definitions	387
21.2 The identity relation	389
21.3 Diagonal: identity over a set	389
21.4 Composition of two relations	390
21.5 Reflexivity	391
21.6 Antisymmetry	391
21.7 Symmetry	392
21.8 Transitivity	392
21.9 Irreflexivity	393
21.10 Totality	393
21.11 Converse	393
21.12 Domain	394
21.13 Range	395
21.14 Field	396
21.15 Image of a set under a relation	397
21.16 Single valued relations	398
21.17 Graphs given by <i>Collect</i>	399
21.18 Inverse image	399
21.19 Finiteness	399
21.20 Miscellaneous	400
<b>22 Predicate: Predicates as relations and enumerations</b>	<b>400</b>
22.1 Predicates as (complete) lattices	400
22.1.1 Equality	401
22.1.2 Order relation	401
22.1.3 Top and bottom elements	402
22.1.4 Binary union	402
22.1.5 Binary intersection	403
22.1.6 Unions of families	403
22.1.7 Intersections of families	404
22.2 Predicates as relations	405
22.2.1 Composition	405
22.2.2 Converse	405
22.2.3 Domain	406
22.2.4 Range	406
22.2.5 Inverse image	407
22.2.6 Powerset	407
22.2.7 Properties of relations	407
22.3 Predicates as enumerations	407
22.3.1 The type of predicate enumerations (a monad)	407
22.3.2 Emptiness check and definite choice	410
22.3.3 Derived operations	412
22.3.4 Implementation	414

<b>23 Transitive-Closure: Reflexive and Transitive closure of a relation</b>	<b>419</b>
23.1 Reflexive closure . . . . .	420
23.2 Reflexive-transitive closure . . . . .	420
23.3 Transitive closure . . . . .	425
23.4 The power operation on relations . . . . .	433
23.5 Setup of transitivity reasoner . . . . .	436
<b>24 Wellfounded: Well-founded Recursion</b>	<b>437</b>
24.1 Basic Definitions . . . . .	437
24.2 Basic Results . . . . .	439
24.3 Well-Foundedness Results for Unions . . . . .	442
24.4 Acyclic relations . . . . .	445
24.5 <i>nat</i> is well-founded . . . . .	446
24.6 Accessible Part . . . . .	447
24.7 Tools for building wellfounded relations . . . . .	450
24.8 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize. . . . .	454
24.9 size of a datatype value . . . . .	455
<b>25 FunDef: Function Definitions and Termination Proofs</b>	<b>455</b>
25.1 Definitions with default value. . . . .	456
25.2 Measure Functions . . . . .	458
25.3 Congruence Rules . . . . .	458
25.4 Simp rules for termination proofs . . . . .	458
25.5 Decomposition . . . . .	459
25.6 Reduction Pairs . . . . .	459
25.7 Concrete orders for SCNP termination proofs . . . . .	460
25.8 Tool setup . . . . .	461
<b>26 Extraction: Program extraction for HOL</b>	<b>462</b>
26.1 Setup . . . . .	462
26.2 Type of extracted program . . . . .	462
26.3 Realizability . . . . .	463
26.4 Computational content of basic inference rules . . . . .	465
<b>27 Plain: Plain HOL</b>	<b>470</b>
<b>28 Big-Operators: Big operators and finite (non-empty) sets</b>	<b>470</b>
28.1 Generic monoid operation over a set . . . . .	470
28.2 Generalized summation over a set . . . . .	471
28.2.1 Properties in more restricted classes of structures . . .	477
28.2.2 Cardinality as special case of <i>setsum</i> . . . . .	486
28.2.3 Cardinality of products . . . . .	487



28.3	Generalized product over a set . . . . .	487
28.3.1	Properties in more restricted classes of structures . . . . .	491
28.4	Versions of <i>inf</i> and <i>sup</i> on non-empty sets . . . . .	494
28.5	Versions of <i>min</i> and <i>max</i> on non-empty sets . . . . .	500
<b>29</b>	<b>Equiv-Relations: Equivalence Relations in Higher-Order Set Theory</b>	<b>507</b>
29.1	Equivalence relations . . . . .	507
29.2	Equivalence classes . . . . .	508
29.3	Quotients . . . . .	509
29.4	Defining unary operations upon equivalence classes . . . . .	510
29.5	Defining binary operations upon equivalence classes . . . . .	511
29.6	Quotients and finiteness . . . . .	513
<b>30</b>	<b>Int: The Integers as Equivalence Classes over Pairs of Natural Numbers</b>	<b>513</b>
30.1	The equivalence relation underlying the integers . . . . .	514
30.2	Construction of the Integers . . . . .	515
30.3	Arithmetic Operations . . . . .	515
30.4	The $\leq$ Ordering . . . . .	517
30.5	Embedding of the Integers into any <i>ring-1: of-int</i> . . . . .	519
30.6	Magnitude of an Integer, as a Natural Number: <i>nat</i> . . . . .	521
30.7	Lemmas about the Function <i>of-nat</i> and Orderings . . . . .	523
30.8	Cases and induction . . . . .	524
30.9	Binary representation . . . . .	525
30.9.1	The constructors <i>Bit0</i> , <i>Bit1</i> , <i>Pls</i> and <i>Min</i> . . . . .	525
30.9.2	Successor and predecessor functions . . . . .	527
30.9.3	Binary arithmetic . . . . .	528
30.9.4	Binary comparisons . . . . .	529
30.10	Converting Numerals to Rings: <i>number-of</i> . . . . .	532
30.10.1	Equality of Binary Numbers . . . . .	534
30.10.2	Comparisons, for Ordered Rings . . . . .	535
30.10.3	The Less-Than Relation . . . . .	536
30.10.4	Simplification of arithmetic operations on integer constants. . . . .	537
30.10.5	Simplification of arithmetic when nested to the right. . . . .	538
30.11	The Set of Integers . . . . .	538
30.12	<i>setsum</i> and <i>setprod</i> . . . . .	541
30.13	Inequality Reasoning for the Arithmetic Simproc . . . . .	541
30.14	Special Arithmetic Rules for Abstract 0 and 1 . . . . .	542
30.15	Setting up simplification procedures . . . . .	543
30.16	Lemmas About Small Numerals . . . . .	543
30.17	More Inequality Reasoning . . . . .	544
30.18	The functions <i>nat</i> and <i>int</i> . . . . .	544

30.19	Induction principles for <code>int</code>	546
30.20	Intermediate value theorems	549
30.21	Products and 1, by T. M. Rasmussen	549
30.22	Further theorems on numerals	551
30.22.1	Special Simplification for Constants	551
30.22.2	Optional Simplification Rules Involving Constants	553
30.23	The divides relation	554
30.24	Configuration of the code generator	557
30.25	Legacy theorems	561
<b>31</b>	<b>Nat-Numeral: Binary numerals for the natural numbers</b>	<b>562</b>
31.1	Numerals for natural numbers	562
31.2	Special case: squares and cubes	563
31.3	Predicate for negative binary numbers	567
31.4	Function <i>nat</i> : Coercion from Type <i>int</i> to <i>nat</i>	568
31.5	Function <i>int</i> : Coercion from Type <i>nat</i> to <i>int</i>	569
31.5.1	Successor	569
31.5.2	Addition	569
31.5.3	Subtraction	570
31.5.4	Multiplication	570
31.6	Comparisons	570
31.6.1	Equals (=)	570
31.6.2	Less-than (<)	571
31.6.3	Less-than-or-equal (<=)	571
31.7	Powers with Numeric Exponents	571
31.7.1	Nat	571
31.7.2	Arith	572
31.8	Comparisons involving (0::nat)	572
31.9	Comparisons involving <i>Suc</i>	572
31.10	Max and Min Combined with <i>Suc</i>	574
31.11	Literal arithmetic involving powers	574
31.12	Literal arithmetic and <i>of-nat</i>	577
31.12.1	For simplifying <i>Suc m - K</i> and <i>K - Suc m</i>	577
31.12.2	For <i>nat-case</i> and <i>nat-rec</i>	578
31.12.3	Various Other Lemmas	578
<b>32</b>	<b>Nat-Transfer: Generic transfer machinery; specific transfer from nats to ints and back.</b>	<b>579</b>
32.1	Generic transfer machinery	579
32.2	Set up transfer from nat to int	580
32.3	Set up transfer from int to nat	584

<b>33 Divides: The division operators <code>div</code> and <code>mod</code></b>	<b>587</b>
33.1 Syntactic division operations . . . . .	587
33.2 Abstract division in commutative semirings. . . . .	587
33.3 Division on <i>nat</i> . . . . .	596
33.3.1 Quotient . . . . .	601
33.3.2 Remainder . . . . .	601
33.3.3 Quotient and Remainder . . . . .	602
33.3.4 Further Facts about Quotient and Remainder . . . . .	603
33.3.5 An “induction” law for modulus arithmetic. . . . .	608
33.4 Division on <i>int</i> . . . . .	611
33.4.1 Uniqueness and Monotonicity of Quotients and Re- mainders . . . . .	613
33.4.2 Correctness of <i>posDivAlg</i> , the Algorithm for Non-Negative Dividends . . . . .	614
33.4.3 Correctness of <i>negDivAlg</i> , the Algorithm for Negative Dividends . . . . .	614
33.4.4 Existence Shown by Proving the Division Algorithm to be Correct . . . . .	615
33.4.5 General Properties of <code>div</code> and <code>mod</code> . . . . .	617
33.4.6 Laws for <code>div</code> and <code>mod</code> with Unary Minus . . . . .	618
33.4.7 Division of a Number by Itself . . . . .	619
33.4.8 Computation of Division and Remainder . . . . .	620
33.4.9 Monotonicity in the First Argument (Dividend) . . . . .	622
33.4.10 Monotonicity in the Second Argument (Divisor) . . . . .	623
33.4.11 More Algebraic Laws for <code>div</code> and <code>mod</code> . . . . .	624
33.4.12 Proving $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$ . . . . .	626
33.4.13 Splitting Rules for <code>div</code> and <code>mod</code> . . . . .	627
33.4.14 Speeding up the Division Algorithm with Shifting . . . . .	629
33.4.15 Computing <code>mod</code> by Shifting (proofs resemble those for <code>div</code> ) . . . . .	630
33.4.16 Quotients of Signs . . . . .	631
33.4.17 The Divides Relation . . . . .	632
33.4.18 Nitpick . . . . .	636
33.4.19 Code generation . . . . .	636
<b>34 Code-Numeral: Type of target language numerals</b>	<b>638</b>
34.1 Datatype of target language numerals . . . . .	638
34.2 Indices as datatype of ints . . . . .	641
34.3 Basic arithmetic . . . . .	641
34.3.1 Lazy Evaluation of an indexed function . . . . .	644
34.4 Code generator setup . . . . .	644
<b>35 Numeral-Simprocs: Combination and Cancellation Simprocs for Numeral Expressions</b>	<b>646</b>

<b>36 Semiring-Normalization: Semiring normalization</b>	<b>648</b>
<b>37 Groebner-Basis: Groebner bases</b>	<b>653</b>
37.1 Groebner Bases . . . . .	653
<b>38 SetInterval: Set intervals</b>	<b>654</b>
38.1 Various equivalences . . . . .	656
38.2 Logical Equivalences for Set Inclusion and Equality . . . . .	656
38.3 Two-sided intervals . . . . .	657
38.3.1 Emptiness, singletons, subset . . . . .	658
38.3.2 Intersection . . . . .	659
38.4 Intervals of natural numbers . . . . .	660
38.4.1 The Constant <i>lessThan</i> . . . . .	660
38.4.2 The Constant <i>greaterThan</i> . . . . .	660
38.4.3 The Constant <i>atLeast</i> . . . . .	660
38.4.4 The Constant <i>atMost</i> . . . . .	661
38.4.5 The Constant <i>atLeastLessThan</i> . . . . .	661
38.4.6 Intervals of nats with <i>Suc</i> . . . . .	662
38.4.7 Image . . . . .	662
38.4.8 Finiteness . . . . .	664
38.4.9 Proving Inclusions and Equalities between Unions . . . . .	665
38.4.10 Cardinality . . . . .	667
38.5 Intervals of integers . . . . .	668
38.5.1 Finiteness . . . . .	668
38.5.2 Cardinality . . . . .	668
38.6 Lemmas useful with the summation operator <i>setsum</i> . . . . .	670
38.6.1 Disjoint Unions . . . . .	670
38.6.2 Disjoint Intersections . . . . .	671
38.6.3 Some Differences . . . . .	672
38.6.4 Some Subset Conditions . . . . .	672
38.7 Summation indexed over intervals . . . . .	672
38.8 Shifting bounds . . . . .	676
38.9 The formula for geometric sums . . . . .	676
38.10 The formula for arithmetic sums . . . . .	677
38.11 Products indexed over intervals . . . . .	679
38.12 Transfer setup . . . . .	679
<b>39 Presburger: Decision Procedure for Presburger Arithmetic</b>	<b>680</b>
39.1 The $-\infty$ and $+\infty$ Properties . . . . .	680
39.2 The A and B sets . . . . .	681
39.3 Cooper's Theorem $-\infty$ and $+\infty$ Version . . . . .	685
39.3.1 First some trivial facts about periodic sets or predicates	685
39.3.2 The $-\infty$ Version . . . . .	685
39.3.3 The $+\infty$ Version . . . . .	687

<b>40 Hilbert-Choice: Hilbert’s Epsilon-Operator and the Axiom of Choice</b>	<b>690</b>
40.1 Hilbert’s epsilon	690
40.2 Hilbert’s Epsilon-operator	691
40.3 Axiom of Choice, Proved Using the Description Operator	692
40.4 Function Inverse	692
40.5 Other Consequences of Hilbert’s Epsilon	695
40.6 Least value operator	696
40.7 Greatest value operator	697
40.8 The Meson proof procedure	699
40.8.1 Negation Normal Form	699
40.8.2 Pulling out the existential quantifiers	699
40.8.3 Generating clauses for the Meson Proof Procedure	700
40.9 Lemmas for Meson, the Model Elimination Procedure	700
40.9.1 Lemmas for Forward Proof	700
40.10 Meson package	701
40.11 Specification package – Hilbertized version	701
<b>41 Sledgehammer: Sledgehammer: Isabelle–ATP Linkup</b>	<b>701</b>
41.1 Setup of external ATPs	703
41.2 The MESON prover	703
41.3 The Metis prover	703
<b>42 Recdef: TFL: recursive function definitions</b>	<b>704</b>
42.1 Well-Founded Recursion	704
<b>43 List: The datatype of finite lists</b>	<b>707</b>
43.1 Basic list processing functions	708
43.1.1 List comprehension	712
43.1.2 $[]$ and $op \#$	716
43.1.3 <i>length</i>	716
43.1.4 $@$ – append	719
43.1.5 <i>map</i>	722
43.1.6 <i>rev</i>	724
43.1.7 <i>set</i>	725
43.1.8 <i>filter</i>	728
43.1.9 List partitioning	731
43.1.10 <i>concat</i>	732
43.1.11 <i>nth</i>	732
43.1.12 <i>list-update</i>	734
43.1.13 <i>last</i> and <i>butlast</i>	736
43.1.14 <i>take</i> and <i>drop</i>	738
43.1.15 <i>takeWhile</i> and <i>dropWhile</i>	743
43.1.16 <i>zip</i>	746

43.1.17	<i>list-all2</i>	750
43.1.18	<i>foldl</i> and <i>foldr</i>	753
43.1.19	List summation: <i>listsum</i> and $\sum$	757
43.1.20	<i>upt</i>	760
43.1.21	<i>upto</i> : interval-list on <i>int</i>	762
43.1.22	<i>distinct</i> and <i>remdups</i>	763
43.1.23	<i>insert</i>	766
43.1.24	<i>remove1</i>	767
43.1.25	<i>removeAll</i>	768
43.1.26	<i>replicate</i>	769
43.1.27	<i>rotate1</i> and <i>rotate</i>	771
43.1.28	<i>sublist</i> — a generalization of <i>nth</i> to sets	773
43.1.29	<i>splice</i>	775
43.1.30	Transpose	775
43.1.31	(In)finiteness	778
43.2	Sorting	779
43.2.1	<i>transpose</i> on sorted lists	785
43.2.2	<i>sorted-list-of-set</i>	788
43.2.3	<i>lists</i> : the list-forming operator over sets	789
43.2.4	Inductive definition for membership	790
43.2.5	Lists as Cartesian products	790
43.3	Relations on Lists	791
43.3.1	Length Lexicographic Ordering	791
43.3.2	Lexicographic Ordering	793
43.4	Lexicographic combination of measure functions	794
43.4.1	Lifting a Relation on List Elements to the Lists	795
43.5	Size function	796
43.6	Transfer	797
43.7	Code generator	797
43.7.1	Setup	797
43.7.2	Generation of efficient code	799
<b>44</b>	<b>Random: A HOL random engine</b>	<b>806</b>
44.1	Auxiliary functions	806
44.2	Random seeds	806
44.3	Base selectors	807
44.4	<i>ML</i> interface	808
<b>45</b>	<b>String: Character and string types</b>	<b>810</b>
45.1	Characters	810
45.2	Strings	811
45.3	Strings as dedicated datatype	813
45.4	Code generator	813

<b>46 Typerep: Reflecting Pure types into HOL</b>	<b>814</b>
<b>47 Code-Evaluation: Term evaluation using the generic code generator</b>	<b>816</b>
47.1 Term representation . . . . .	816
47.1.1 Terms and class <i>term-of</i> . . . . .	816
47.1.2 <i>term-of</i> instances . . . . .	817
47.1.3 Code generator setup . . . . .	819
47.1.4 Syntax . . . . .	820
47.2 Numeric types . . . . .	821
47.3 Obfuscate . . . . .	822
47.4 Tracing of generated and evaluated code . . . . .	822
47.5 Evaluation setup . . . . .	822
<b>48 Quickcheck: A simple counterexample generator</b>	<b>823</b>
48.1 The <i>random</i> class . . . . .	823
48.2 Fundamental and numeric types . . . . .	823
48.3 Complex generators . . . . .	825
48.4 Code setup . . . . .	826
48.5 The Random-Predicate Monad . . . . .	826
<b>49 Lazy-Sequence: Lazy sequences</b>	<b>828</b>
49.1 Code setup . . . . .	830
49.2 With Hit Bound Value . . . . .	831
<b>50 DSequence: Depth-Limited Sequences with failure element</b>	<b>832</b>
<b>51 New-DSequence: Depth-Limited Sequences with failure element</b>	<b>835</b>
51.1 Positive Depth-Limited Sequence . . . . .	836
51.2 Negative Depth-Limited Sequence . . . . .	836
51.3 Negation . . . . .	837
<b>52 Predicate-Compile: A compiler for predicates defined by introduction rules</b>	<b>840</b>
<b>53 Map: Maps</b>	<b>840</b>
53.1 <i>empty</i> . . . . .	842
53.2 <i>map-upd</i> . . . . .	842
53.3 <i>map-of</i> . . . . .	843
53.4 <i>Option.map</i> related . . . . .	845
53.5 <i>map-comp</i> related . . . . .	846
53.6 <i>++</i> . . . . .	846
53.7 <i>restrict-map</i> . . . . .	847
53.8 <i>map-upds</i> . . . . .	848

53.9	<i>dom</i>	850
53.10	<i>ran</i>	852
53.11	<i>map-le</i>	852
53.12	Various	854
<b>54</b>	<b>Quotient: Definition of Quotient Types</b>	<b>855</b>
54.1	Respects predicate	856
54.2	Function map and function relation	856
54.3	Quotient Predicate	857
54.4	lemmas for regularisation of ball and bex	860
54.5	Bounded abstraction	863
54.6	<i>Bex1-rel</i> quantifier	864
54.7	Various respects and preserve lemmas	866
54.8	ML setup	869
54.9	Methods / Interface	870
<b>55</b>	<b>Refute: Refute</b>	<b>871</b>
<b>56</b>	<b>SAT: Reconstructing external resolution proofs for propositional logic</b>	<b>873</b>
<b>57</b>	<b>Nitpick: Nitpick: Yet Another Counterexample Generator for Isabelle/HOL</b>	<b>874</b>
<b>58</b>	<b>SMT: Bindings to Satisfiability Modulo Theories (SMT) solvers</b>	<b>879</b>
58.1	Triggers for quantifier instantiation	879
58.2	Higher-order encoding	880
58.3	First-order logic	880
58.4	Integer division and modulo for Z3	880
58.5	Setup	880
58.6	Configuration	881
58.7	General configuration options	881
58.8	Certificates	881
58.9	Tracing	882
58.10	Z3-specific options	882
58.11	Schematic rules for Z3 proof reconstruction	882
<b>59</b>	<b>Main: Main HOL</b>	<b>884</b>
<b>60</b>	<b>Lubs: Definitions of Upper Bounds and Least Upper Bounds</b>	<b>884</b>
60.1	Rules for the Relations $*\leq$ and $\leq*$	885
60.2	Rules about the Operators <i>leastP</i> , <i>ub</i> and <i>lub</i>	885



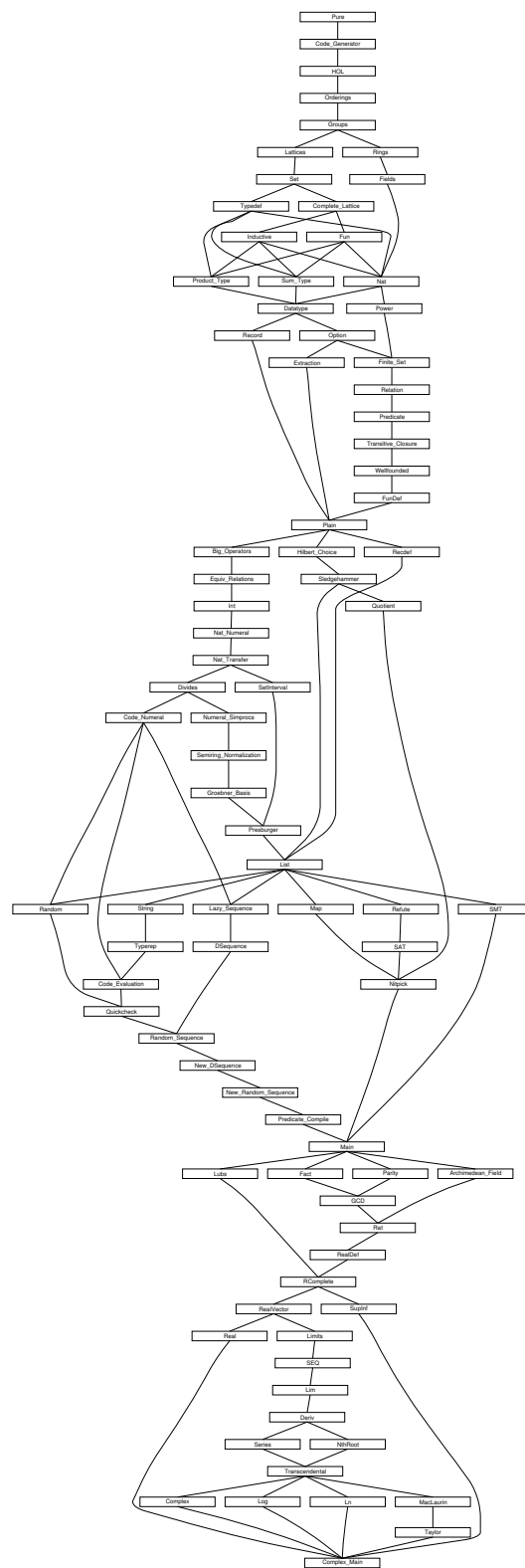
<b>61 Fact: Factorial Function</b>	<b>886</b>
61.1 Set up Transfer . . . . .	887
61.2 Factorial . . . . .	888
61.3 <i>fact</i> and <i>of-nat</i> . . . . .	891
<b>62 Parity: Even and Odd for int and nat</b>	<b>892</b>
62.1 Even and odd are mutually exclusive . . . . .	893
62.2 Behavior under integer arithmetic operations . . . . .	893
62.3 Equivalent definitions . . . . .	894
62.4 even and odd for nats . . . . .	894
62.5 Equivalent definitions . . . . .	895
62.6 Parity and powers . . . . .	895
62.7 More Even/Odd Results . . . . .	898
62.8 An Equivalence for $0 \leq a^n$ . . . . .	899
62.9 Miscellaneous . . . . .	900
<b>63 GCD: Greatest common divisor and least common multiple</b>	<b>900</b>
63.1 GCD and LCM definitions . . . . .	900
63.2 Transfer setup . . . . .	901
63.3 GCD properties . . . . .	902
63.4 Coprimality . . . . .	911
63.5 Bezout's theorem . . . . .	920
63.6 LCM properties . . . . .	925
63.6.1 The complete divisibility lattice . . . . .	929
<b>64 Archimedean-Field: Archimedean Fields, Floor and Ceiling Functions</b>	<b>935</b>
64.1 Class of Archimedean fields . . . . .	935
64.2 Existence and uniqueness of floor function . . . . .	937
64.3 Floor function . . . . .	938
64.4 Ceiling function . . . . .	940
64.5 Negation . . . . .	943
<b>65 Rat: Rational numbers</b>	<b>943</b>
65.1 Rational numbers as quotient . . . . .	943
65.1.1 Construction of the type of rational numbers . . . . .	943
65.1.2 Representation and basic operations . . . . .	944
65.1.3 Function <i>normalize</i> . . . . .	949
65.1.4 The field of rational numbers . . . . .	952
65.1.5 Various . . . . .	952
65.1.6 The ordered field of rational numbers . . . . .	953
65.1.7 Rationals are an Archimedean field . . . . .	958
65.2 Linear arithmetic setup . . . . .	958
65.3 Embedding from Rationals to other Fields . . . . .	959

65.4	The Set of Rational Numbers . . . . .	961
65.5	Implementation of rational numbers as pairs of integers . . .	964
65.6	Float syntax . . . . .	968
<b>66</b>	<b>RealDef: Development of the Reals using Cauchy Sequences</b>	<b>968</b>
66.1	Preliminary lemmas . . . . .	968
66.2	Sequences that converge to zero . . . . .	969
66.3	Cauchy sequences . . . . .	970
66.4	Equivalence relation on Cauchy sequences . . . . .	974
66.5	The field of real numbers . . . . .	974
66.6	Positive reals . . . . .	980
66.7	Completeness . . . . .	984
66.8	Hiding implementation details . . . . .	989
66.9	Legacy theorem names . . . . .	989
66.10	More Lemmas . . . . .	989
66.11	Embedding numbers into the Reals . . . . .	990
66.12	Embedding the Naturals into the Reals . . . . .	993
66.13	Rationals . . . . .	997
66.14	Numerals and Arithmetic . . . . .	998
66.15	Simprules combining $x+y$ and 0: ARE THEY NEEDED? . .	999
66.16	Lemmas about powers . . . . .	1000
66.17	Density of the Reals . . . . .	1001
66.18	Absolute Value Function for the Reals . . . . .	1001
66.19	Implementation of rational real numbers . . . . .	1002
<b>67</b>	<b>RComplete: Completeness of the Reals; Floor and Ceiling Functions</b>	<b>1005</b>
67.1	Completeness of Positive Reals . . . . .	1005
67.2	The Archimedean Property of the Reals . . . . .	1006
67.3	Density of the Rational Reals in the Reals . . . . .	1007
67.4	Floor and Ceiling Functions from the Reals to the Integers .	1008
67.5	Versions for the natural numbers . . . . .	1012
67.6	Exponentiation with floor . . . . .	1020
<b>68</b>	<b>RealVector: Vector Spaces and Algebras over the Reals</b>	<b>1020</b>
68.1	Locale for additive functions . . . . .	1020
68.2	Vector spaces . . . . .	1021
68.3	Real vector spaces . . . . .	1023
68.4	Embedding of the Reals into any <i>real-algebra-1: of-real</i> . . .	1024
68.5	The Set of Real Numbers . . . . .	1026
68.6	Topological spaces . . . . .	1028
68.7	Metric spaces . . . . .	1030
68.8	Real normed vector spaces . . . . .	1031
68.9	Class instances for real numbers . . . . .	1036

68.10	Extra type constraints . . . . .	1037
68.11	Sign function . . . . .	1037
68.12	Bounded Linear and Bilinear Operators . . . . .	1038
<b>69</b>	<b>SupInf: Sup and Inf Operators on Sets of Reals.</b>	<b>1042</b>
69.1	Supremum of a set of reals . . . . .	1042
69.2	Infimum of a set of reals . . . . .	1047
69.3	Relate max and min to Sup and Inf. . . . .	1051
<b>70</b>	<b>Limits: Filters and Limits</b>	<b>1052</b>
70.1	Nets . . . . .	1052
70.2	Eventually . . . . .	1053
70.3	Finer-than relation . . . . .	1054
70.4	Map function for nets . . . . .	1057
70.5	Sequentially . . . . .	1057
70.6	Standard Nets . . . . .	1058
70.7	Boundedness . . . . .	1059
70.8	Convergence to Zero . . . . .	1060
70.9	Limits . . . . .	1063
70.10	Continuity of Inverse . . . . .	1066
<b>71</b>	<b>SEQ: Sequences and Convergence</b>	<b>1069</b>
71.1	Bounded Sequences . . . . .	1070
71.2	Limits of Sequences . . . . .	1071
71.3	Convergence . . . . .	1078
71.4	Bounded Monotonic Sequences . . . . .	1085
71.4.1	Upper Bounds and Lubs of Bounded Sequences . . . .	1086
71.4.2	A Bounded and Monotonic Sequence Converges . . . .	1086
71.4.3	Increasing and Decreasing Series . . . . .	1087
71.4.4	A Few More Equivalence Theorems for Boundedness . .	1089
71.5	Cauchy Sequences . . . . .	1089
71.5.1	Cauchy Sequences are Bounded . . . . .	1090
71.5.2	Cauchy Sequences are Convergent . . . . .	1091
71.6	Power Sequences . . . . .	1094
<b>72</b>	<b>Lim: Limits and Continuity</b>	<b>1096</b>
72.1	Limits of Functions . . . . .	1096
72.1.1	Derived theorems about <i>LIM</i> . . . . .	1104
72.2	Continuity . . . . .	1104
72.3	Uniform Continuity . . . . .	1107
72.4	Relation of LIM and LIMSEQ . . . . .	1108

<b>73 Deriv: Differentiation</b>	<b>1110</b>
73.1 Derivatives . . . . .	1110
73.2 Differentiability predicate . . . . .	1117
73.3 Nested Intervals and Bisection . . . . .	1119
73.4 Intermediate Value Theorem . . . . .	1123
73.5 Boundedness of continuous functions . . . . .	1124
73.6 Local extrema . . . . .	1126
73.7 Rolle's Theorem . . . . .	1129
73.8 Mean Value Theorem . . . . .	1131
73.9 Continuous injective functions . . . . .	1136
73.10 Generalized Mean Value Theorem . . . . .	1140
73.11 Theorems about Limits . . . . .	1142
<b>74 Series: Finite Summation and Infinite Series</b>	<b>1143</b>
74.1 Infinite Sums, by the Properties of Limits . . . . .	1145
74.2 The Ratio Test . . . . .	1154
74.3 Cauchy Product Formula . . . . .	1155
<b>75 NthRoot: Nth Roots of Real Numbers</b>	<b>1157</b>
75.1 Existence of Nth Root . . . . .	1158
75.2 Nth Root . . . . .	1158
75.3 Square Root . . . . .	1166
75.4 Square Root of Sum of Squares . . . . .	1169
<b>76 Transcendental: Power Series, Transcendental Functions etc</b>	<b>1171</b>
76.1 Properties of Power Series . . . . .	1171
76.2 Alternating series test / Leibniz formula . . . . .	1175
76.3 Term-by-Term Differentiability of Power Series . . . . .	1178
76.4 Some properties of factorials . . . . .	1184
76.5 Derivability of power series . . . . .	1184
76.6 Exponential Function . . . . .	1189
76.6.1 Properties of the Exponential Function . . . . .	1191
76.6.2 Properties of the Exponential Function on Reals . . . . .	1192
76.7 Natural Logarithm . . . . .	1195
76.8 Sine and Cosine . . . . .	1198
76.9 Properties of Sine and Cosine . . . . .	1200
76.10 The Constant Pi . . . . .	1205
76.11 Tangent . . . . .	1214
76.12 Inverse Trigonometric Functions . . . . .	1219
76.13 More Theorems about Sin and Cos . . . . .	1224
76.14 Machins formula . . . . .	1226
76.15 Introducing the arcus tangens power series . . . . .	1229
76.16 Existence of Polar Coordinates . . . . .	1238

<b>77 Complex: Complex Numbers: Rectangular and Polar Representations</b>	<b>1239</b>
77.1 Addition and Subtraction . . . . .	1240
77.2 Multiplication and Division . . . . .	1241
77.3 Numerals and Arithmetic . . . . .	1242
77.4 Scalar Multiplication . . . . .	1243
77.5 Properties of Embedding from Reals . . . . .	1244
77.6 Vector Norm . . . . .	1244
77.7 Completeness of the Complexes . . . . .	1246
77.8 The Complex Number $i$ . . . . .	1247
77.9 Complex Conjugation . . . . .	1248
77.10 The Functions $sgn$ and $arg$ . . . . .	1250
77.11 Finally! Polar Form for Complex Numbers . . . . .	1251
<b>78 Log: Logarithms: Standard Version</b>	<b>1254</b>
<b>79 Ln: Properties of <math>\ln</math></b>	<b>1260</b>
<b>80 MacLaurin: MacLaurin Series</b>	<b>1268</b>
80.1 Maclaurin's Theorem with Lagrange Form of Remainder . . .	1268
80.2 More Convenient "Bidirectional" Version. . . . .	1273
80.3 Version for Exponential Function . . . . .	1275
80.4 Version for Sine Function . . . . .	1275
80.5 Maclaurin Expansion for Cosine Function . . . . .	1277
<b>81 Taylor: Taylor series</b>	<b>1280</b>
<b>82 Complex-Main: Comprehensive Complex Theory</b>	<b>1283</b>



## 1 Code-Generator: Loading the code generator modules

```

theory Code-Generator
imports Pure
uses
  ~~ /src/Tools/auto-solve.ML
  ~~ /src/Tools/auto-counterexample.ML
  ~~ /src/Tools/quickcheck.ML
  ~~ /src/Tools/value.ML
  ~~ /src/Tools/Code/code-preproc.ML
  ~~ /src/Tools/Code/code-thingol.ML
  ~~ /src/Tools/Code/code-printer.ML
  ~~ /src/Tools/Code/code-target.ML
  ~~ /src/Tools/Code/code-ml.ML
  ~~ /src/Tools/Code/code-eval.ML
  ~~ /src/Tools/Code/code-haskell.ML
  ~~ /src/Tools/Code/code-scala.ML
  ~~ /src/Tools/nbe.ML
begin

setup ⟨⟨
  Code-Preproc.setup
  #> Code-ML.setup
  #> Code-Eval.setup
  #> Code-Haskell.setup
  #> Code-Scala.setup
  #> Nbe.setup
  #> Quickcheck.setup
  ⟩⟩

end

```

## 2 HOL: The basis of Higher-Order Logic

```

theory HOL
imports Pure ~~ /src/Tools/Code-Generator
uses
  (Tools/hologic.ML)
  ~~ /src/Tools/IsaPlanner/zipper.ML
  ~~ /src/Tools/IsaPlanner/isand.ML
  ~~ /src/Tools/IsaPlanner/rw-tools.ML
  ~~ /src/Tools/IsaPlanner/rw-inst.ML
  ~~ /src/Tools/intuitionistic.ML
  ~~ /src/Tools/project-rule.ML
  ~~ /src/Tools/cong-tac.ML
  ~~ /src/Provers/hypsubst.ML

```

```

~~/src/Provers/splitter.ML
~~/src/Provers/classical.ML
~~/src/Provers/blast.ML
~~/src/Provers/clasimp.ML
~~/src/Tools/coherent.ML
~~/src/Tools/eqsubst.ML
~~/src/Provers/quantifier1.ML
(Tools/simpdata.ML)
~~/src/Tools/random-word.ML
~~/src/Tools/atomize-elim.ML
~~/src/Tools/induct.ML
(~~/src/Tools/induct-tacs.ML)
(Tools/recfun-codegen.ML)
begin

setup << Intuitionistic.method-setup @{binding iprover} >>

```

## 2.1 Primitive logic

### 2.1.1 Core syntax

```

classes type
default-sort type
setup << Object-Logic.add-base-sort @{sort type} >>

arities
  fun :: (type, type) type
  itself :: (type) type

global

typeddecl bool

judgment
  Trueprop      :: bool => prop                ((-) 5)

consts
  Not           :: bool => bool                (~ - [40] 40)
  True          :: bool
  False         :: bool

  The           :: ('a => bool) => 'a
  All           :: ('a => bool) => bool          (binder ALL 10)
  Ex            :: ('a => bool) => bool          (binder EX 10)
  Ex1           :: ('a => bool) => bool          (binder EX! 10)
  Let           :: ['a, 'a => 'b] => 'b

  op =          :: ['a, 'a] => bool             (infixl = 50)
  op &          :: [bool, bool] => bool         (infixr & 35)
  op |          :: [bool, bool] => bool         (infixr | 30)

```



*op*  $-->$       $:: [bool, bool] => bool$      (**infixr**  $-->$  25)

## local

### consts

*If*      $:: [bool, 'a, 'a] => 'a$      ((*if* (-)/ *then* (-)/ *else* (-)) [0, 0, 10] 10)

## 2.1.2 Additional concrete syntax

### notation (output)

*op* = (**infix** = 50)

### abbreviation

*not-equal*  $:: ['a, 'a] => bool$  (**infixl**  $\sim =$  50) **where**  
*x*  $\sim =$  *y* ==  $\sim (x = y)$

### notation (output)

*not-equal* (**infix**  $\sim =$  50)

### notation (xsymbols)

*Not* ( $\neg$  - [40] 40) **and**  
*op* & (**infixr**  $\wedge$  35) **and**  
*op* | (**infixr**  $\vee$  30) **and**  
*op*  $-->$  (**infixr**  $\longrightarrow$  25) **and**  
*not-equal* (**infix**  $\neq$  50)

### notation (HTML output)

*Not* ( $\neg$  - [40] 40) **and**  
*op* & (**infixr**  $\wedge$  35) **and**  
*op* | (**infixr**  $\vee$  30) **and**  
*not-equal* (**infix**  $\neq$  50)

### abbreviation (iff)

*iff*  $:: [bool, bool] => bool$  (**infixr**  $<->$  25) **where**  
*A*  $<->$  *B* == *A* = *B*

### notation (xsymbols)

*iff* (**infixr**  $\longleftrightarrow$  25)

### nonterminals

*letbinds* *letbind*  
*case-syn* *cases-syn*

### syntax

*-The*      $:: [pttrn, bool] => 'a$      ((3*THE* -./ -) [0, 10] 10)  
*-bind*      $:: [pttrn, 'a] => letbind$      ((2- =/ -) 10)  
             $:: letbind => letbinds$      (-)  
*-binds*     $:: [letbind, letbinds] => letbinds$      (-;/ -)

$-Let \quad :: [letbinds, 'a] \Rightarrow 'a \quad ((let (-) / in (-)) [0, 10] 10)$   
 $-case-syntax :: ['a, cases-syn] \Rightarrow 'b \quad ((case - of / -) 10)$   
 $-case1 \quad :: ['a, 'b] \Rightarrow case-syn \quad ((2- \Rightarrow / -) 10)$   
 $\quad \quad :: case-syn \Rightarrow cases-syn \quad (-)$   
 $-case2 \quad :: [case-syn, cases-syn] \Rightarrow cases-syn \quad (- / | -)$

**translations**

$THE \ x. P \quad == \ CONST \ The \ (\%x. P)$   
 $-Let \ (-binds \ b \ bs) \ e \ == \ -Let \ b \ (-Let \ bs \ e)$   
 $let \ x = a \ in \ e \quad == \ CONST \ Let \ a \ (\%x. e)$

**print-translation**  $\ll$ 

$[(\@ \{const-syntax \ The\}, fn \ [Abs \ abs] \Rightarrow$   
 $\quad let \ val \ (x, t) = atomic-abs-tr' \ abs$   
 $\quad in \ Syntax.const \ \@ \{syntax-const -The\} \ \$ \ x \ \$ \ t \ end)]$   
 $\gg$  — To avoid eta-contraction of body

**syntax** (*xsymbols*)

$-case1 \quad :: ['a, 'b] \Rightarrow case-syn \quad ((2- \Rightarrow / -) 10)$

**notation** (*xsymbols*)

$All \ (binder \ \forall \ 10) \ and$   
 $Ex \ (binder \ \exists \ 10) \ and$   
 $Ex1 \ (binder \ \exists! \ 10)$

**notation** (*HTML output*)

$All \ (binder \ \forall \ 10) \ and$   
 $Ex \ (binder \ \exists \ 10) \ and$   
 $Ex1 \ (binder \ \exists! \ 10)$

**notation** (*HOL*)

$All \ (binder \ ! \ 10) \ and$   
 $Ex \ (binder \ ? \ 10) \ and$   
 $Ex1 \ (binder \ ?! \ 10)$

**2.1.3 Axioms and basic definitions****axioms**

$refl: \quad t = (t :: 'a)$   
 $subst: \quad s = t \Longrightarrow P \ s \Longrightarrow P \ t$   
 $ext: \quad (!x :: 'a. (f \ x :: 'b) = g \ x) \Longrightarrow (\%x. f \ x) = (\%x. g \ x)$

— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL

$the-eq-trivial: \ (THE \ x. x = a) = (a :: 'a)$

$impI: \quad (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$

*mp*:  $[| P \multimap Q; P |] \implies Q$

### defs

*True-def*:  $True \implies (\%x::bool. x) = (\%x. x)$   
*All-def*:  $All(P) \implies (P = (\%x. True))$   
*Ex-def*:  $Ex(P) \implies !Q. (!x. P x \multimap Q) \multimap Q$   
*False-def*:  $False \implies (!P. P)$   
*not-def*:  $\sim P \implies P \multimap False$   
*and-def*:  $P \ \& \ Q \implies !R. (P \multimap Q \multimap R) \multimap R$   
*or-def*:  $P \ | \ Q \implies !R. (P \multimap R) \multimap (Q \multimap R) \multimap R$   
*Ex1-def*:  $Ex1(P) \implies ? x. P(x) \ \& \ (! y. P(y) \multimap y=x)$

### axioms

*iff*:  $(P \multimap Q) \multimap (Q \multimap P) \multimap (P=Q)$   
*True-or-False*:  $(P=True) \ | \ (P=False)$

### defs

*Let-def* [code]:  $Let \ s \ f \implies f(s)$   
*if-def*:  $If \ P \ x \ y \implies THE \ z::'a. (P=True \multimap z=x) \ \& \ (P=False \multimap z=y)$

### finalconsts

*op* =  
*op*  $\multimap$   
*The*

### axiomatization

*undefined* :: 'a

### class default =

*fixes* *default* :: 'a

## 2.2 Fundamental rules

### 2.2.1 Equality

**lemma** *sym*:  $s = t \implies t = s$

**by** (*erule subst*) (*rule refl*)

**lemma** *ssubst*:  $t = s \implies P \ s \implies P \ t$

**by** (*drule sym*) (*erule subst*)

**lemma** *trans*:  $[| r=s; s=t |] \implies r=t$

**by** (*erule subst*)

**lemma** *meta-eq-to-obj-eq*:

**assumes** *meq*:  $A = B$

**shows**  $A = B$

**by** (*unfold meq*) (*rule refl*)

Useful with *erule* for proving equalities from known equalities.

```
lemma box-equals: [| a=b; a=c; b=d |] ==> c=d
  apply (rule trans)
  apply (rule trans)
  apply (rule sym)
  apply assumption+
done
```

For calculational reasoning:

```
lemma forw-subst: a = b ==> P b ==> P a
  by (rule ssubst)
```

```
lemma back-subst: P a ==> a = b ==> P b
  by (rule subst)
```

### 2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon’s HOL.

```
lemma fun-cong: (f::'a=>'b) = g ==> f(x)=g(x)
  apply (erule subst)
  apply (rule refl)
done
```

Similar to *AP-TERM* in Gordon’s HOL and FOL’s *subst-context*.

```
lemma arg-cong: x=y ==> f(x)=f(y)
  apply (erule subst)
  apply (rule refl)
done
```

```
lemma arg-cong2: [| a = b; c = d |] ==> f a c = f b d
  apply (erule ssubst)+
  apply (rule refl)
done
```

```
lemma cong: [| f = g; (x::'a) = y |] ==> f x = g y
  apply (erule subst)+
  apply (rule refl)
done
```

```
ML << val cong-tac = Cong-Tac.cong-tac @ {thm cong} >>
```

### 2.2.3 Equality of booleans – iff

```
lemma iffI: assumes P ==> Q and Q ==> P shows P=Q
  by (iprover intro: iff [THEN mp, THEN mp] impI assms)
```

```
lemma iffD2: [| P=Q; Q |] ==> P
  by (erule ssubst)
```

**lemma** *rev-iffD2*:  $[\mid Q; P=Q \mid] ==> P$   
**by** (*erule iffD2*)

**lemma** *iffD1*:  $Q = P \implies Q \implies P$   
**by** (*drule sym*) (*rule iffD2*)

**lemma** *rev-iffD1*:  $Q \implies Q = P \implies P$   
**by** (*drule sym*) (*rule rev-iffD2*)

**lemma** *iffE*:  
**assumes** *major*:  $P=Q$   
**and** *minor*:  $[\mid P \dashv\vdash Q; Q \dashv\vdash P \mid] ==> R$   
**shows**  $R$   
**by** (*iprover intro: minor impI major [THEN iffD2] major [THEN iffD1]*)

#### 2.2.4 True

**lemma** *TrueI*:  $\text{True}$   
**unfolding** *True-def* **by** (*rule refl*)

**lemma** *eqTrueI*:  $P ==> P = \text{True}$   
**by** (*iprover intro: iffI TrueI*)

**lemma** *eqTrueE*:  $P = \text{True} ==> P$   
**by** (*erule iffD2*) (*rule TrueI*)

#### 2.2.5 Universal quantifier

**lemma** *allI*: **assumes**  $!!x::'a. P(x)$  **shows**  $\text{ALL } x. P(x)$   
**unfolding** *All-def* **by** (*iprover intro: ext eqTrueI assms*)

**lemma** *spec*:  $\text{ALL } x::'a. P(x) ==> P(x)$   
**apply** (*unfold All-def*)  
**apply** (*rule eqTrueE*)  
**apply** (*erule fun-cong*)  
**done**

**lemma** *allE*:  
**assumes** *major*:  $\text{ALL } x. P(x)$   
**and** *minor*:  $P(x) ==> R$   
**shows**  $R$   
**by** (*iprover intro: minor major [THEN spec]*)

**lemma** *all-dupE*:  
**assumes** *major*:  $\text{ALL } x. P(x)$   
**and** *minor*:  $[\mid P(x); \text{ALL } x. P(x) \mid] ==> R$   
**shows**  $R$   
**by** (*iprover intro: minor major major [THEN spec]*)

### 2.2.6 False

Depends upon *spec*; it is impossible to do propositional logic before quantifiers!

```
lemma FalseE: False ==> P
  apply (unfold False-def)
  apply (erule spec)
  done
```

```
lemma False-neg-True: False = True ==> P
  by (erule eqTrueE [THEN FalseE])
```

### 2.2.7 Negation

```
lemma notI:
  assumes P ==> False
  shows ~P
  apply (unfold not-def)
  apply (iprover intro: impI assms)
  done
```

```
lemma False-not-True: False ~ = True
  apply (rule notI)
  apply (erule False-neg-True)
  done
```

```
lemma True-not-False: True ~ = False
  apply (rule notI)
  apply (erule sym)
  apply (erule False-neg-True)
  done
```

```
lemma notE: [| ~P; P |] ==> R
  apply (unfold not-def)
  apply (erule mp [THEN FalseE])
  apply assumption
  done
```

```
lemma notI2: (P ==> ~ Pa) ==> (P ==> Pa) ==> ~ P
  by (erule notE [THEN notI]) (erule meta-mp)
```

### 2.2.8 Implication

```
lemma impE:
  assumes P-->Q P Q ==> R
  shows R
  by (iprover intro: assms mp)
```

**lemma** *rev-mp*:  $[ [ P; P \multimap Q ] \implies Q ]$   
**by** (*iprover intro: mp*)

**lemma** *contrapos-nn*:  
**assumes** *major*:  $\sim Q$   
**and** *minor*:  $P \implies Q$   
**shows**  $\sim P$   
**by** (*iprover intro: notI minor major [THEN notE]*)

**lemma** *contrapos-pn*:  
**assumes** *major*:  $Q$   
**and** *minor*:  $P \implies \sim Q$   
**shows**  $\sim P$   
**by** (*iprover intro: notI minor major notE*)

**lemma** *not-sym*:  $t \sim s \implies s \sim t$   
**by** (*erule contrapos-nn*) (*erule sym*)

**lemma** *eq-neq-eq-imp-neq*:  $[ [ x = a ; a \sim b; b = y ] \implies x \sim y ]$   
**by** (*erule subst, erule ssubst, assumption*)

**lemma** *rev-contrapos*:  
**assumes** *pq*:  $P \implies Q$   
**and** *nq*:  $\sim Q$   
**shows**  $\sim P$   
**apply** (*rule nq [THEN contrapos-nn]*)  
**apply** (*erule pq*)  
**done**

## 2.2.9 Existential quantifier

**lemma** *exI*:  $P\ x \implies \exists x::'a. P\ x$   
**apply** (*unfold Ex-def*)  
**apply** (*iprover intro: allI allE impI mp*)  
**done**

**lemma** *exE*:  
**assumes** *major*:  $\exists x::'a. P(x)$   
**and** *minor*:  $!!x. P(x) \implies Q$   
**shows**  $Q$   
**apply** (*rule major [unfolded Ex-def, THEN spec, THEN mp]*)  
**apply** (*iprover intro: impI [THEN allI] minor*)  
**done**

## 2.2.10 Conjunction

**lemma** *conjI*:  $[ [ P; Q ] \implies P \& Q ]$   
**apply** (*unfold and-def*)

```

apply (iprover intro: impI [THEN allI] mp)
done

```

```

lemma conjunct1: [| P & Q |] ==> P
apply (unfold and-def)
apply (iprover intro: impI dest: spec mp)
done

```

```

lemma conjunct2: [| P & Q |] ==> Q
apply (unfold and-def)
apply (iprover intro: impI dest: spec mp)
done

```

```

lemma conjE:
  assumes major: P&Q
    and minor: [| P; Q |] ==> R
  shows R
apply (rule minor)
apply (rule major [THEN conjunct1])
apply (rule major [THEN conjunct2])
done

```

```

lemma context-conjI:
  assumes P P ==> Q shows P & Q
by (iprover intro: conjI assms)

```

### 2.2.11 Disjunction

```

lemma disjI1: P ==> P|Q
apply (unfold or-def)
apply (iprover intro: allI impI mp)
done

```

```

lemma disjI2: Q ==> P|Q
apply (unfold or-def)
apply (iprover intro: allI impI mp)
done

```

```

lemma disjE:
  assumes major: P|Q
    and minorP: P ==> R
    and minorQ: Q ==> R
  shows R
by (iprover intro: minorP minorQ impI
      major [unfolded or-def, THEN spec, THEN mp, THEN mp])

```

### 2.2.12 Classical logic

```

lemma classical:
  assumes prem: ~P ==> P

```



```

shows P
apply (rule True-or-False [THEN disjE, THEN eqTrueE])
apply assumption
apply (rule notI [THEN prem, THEN eqTrueI])
apply (erule subst)
apply assumption
done

```

```

lemmas ccontr = FalseE [THEN classical, standard]

```

```

lemma rev-notE:
  assumes premp: P
    and premnot:  $\sim R \implies \sim P$ 
  shows R
apply (rule ccontr)
apply (erule notE [OF premnot premp])
done

```

```

lemma notnotD:  $\sim\sim P \implies P$ 
apply (rule classical)
apply (erule notE)
apply assumption
done

```

```

lemma contrapos-pp:
  assumes p1: Q
    and p2:  $\sim P \implies \sim Q$ 
  shows P
by (iprover intro: classical p1 p2 notE)

```

### 2.2.13 Unique existence

```

lemma ex1I:
  assumes P a !!x. P(x)  $\implies x=a$ 
  shows EX! x. P(x)
by (unfold Ex1-def, iprover intro: assms exI conjI allI impI)

```

Sometimes easier to use: the premises have no shared variables. Safe!

```

lemma ex-ex1I:
  assumes ex-prem: EX x. P(x)
    and eq: !!x y. [| P(x); P(y) |]  $\implies x=y$ 
  shows EX! x. P(x)
by (iprover intro: ex-prem [THEN exE] ex1I eq)

```

```

lemma ex1E:
  assumes major: EX! x. P(x)
    and minor: !!x. [| P(x); ALL y. P(y)  $\longrightarrow y=x$  |]  $\implies R$ 

```

```

shows R
apply (rule major [unfolded Ex1-def, THEN exE])
apply (erule conjE)
apply (iprover intro: minor)
done

```

```

lemma ex1-implies-ex: EX! x. P x ==> EX x. P x
apply (erule ex1E)
apply (rule exI)
apply assumption
done

```

#### 2.2.14 THE: definite description operator

```

lemma the-equality:
  assumes prema: P a
    and premx: !!x. P x ==> x=a
  shows (THE x. P x) = a
apply (rule trans [OF - the-eq-trivial])
apply (rule-tac f = The in arg-cong)
apply (rule ext)
apply (rule iffI)
  apply (erule premx)
apply (erule ssubst, rule prema)
done

```

```

lemma theI:
  assumes P a and !!x. P x ==> x=a
  shows P (THE x. P x)
by (iprover intro: assms the-equality [THEN ssubst])

```

```

lemma theI': EX! x. P x ==> P (THE x. P x)
apply (erule ex1E)
apply (erule theI)
apply (erule allE)
apply (erule mp)
apply assumption
done

```

```

lemma theI2:
  assumes P a !!x. P x ==> x=a !!x. P x ==> Q x
  shows Q (THE x. P x)
by (iprover intro: assms theI)

```

```

lemma theI12: assumes EX! x. P x ^ x. P x ==> Q x shows Q (THE x. P x)
by (iprover intro: assms(2) theI2 [where P=P and Q=Q] ex1E [OF assms(1)]
    elim: allE impE)

```

```

lemma the1-equality [elim?]: [| EX!x. P x; P a |] ==> (THE x. P x) = a
apply (rule the-equality)
apply assumption
apply (erule ex1E)
apply (erule all-dupE)
apply (drule mp)
apply assumption
apply (erule ssubst)
apply (erule allE)
apply (erule mp)
apply assumption
done

```

```

lemma the-sym-eq-trivial: (THE y. x=y) = x
apply (rule the-equality)
apply (rule refl)
apply (erule sym)
done

```

### 2.2.15 Classical intro rules for disjunction and existential quantifiers

```

lemma disjCI:
  assumes  $\sim Q \implies P$  shows  $P \mid Q$ 
apply (rule classical)
apply (iprover intro: assms disjI1 disjI2 notI elim: notE)
done

```

```

lemma excluded-middle:  $\sim P \mid P$ 
by (iprover intro: disjCI)

```

case distinction as a natural deduction rule. Note that  $\neg P$  is the second case, not the first

```

lemma case-split [case-names True False]:
  assumes prem1:  $P \implies Q$ 
  and prem2:  $\sim P \implies Q$ 
  shows  $Q$ 
apply (rule excluded-middle [THEN disjE])
apply (erule prem2)
apply (erule prem1)
done

```

```

lemma impCE:
  assumes major:  $P \dashv\dashv Q$ 
  and minor:  $\sim P \implies R \quad Q \implies R$ 
  shows  $R$ 
apply (rule excluded-middle [of P, THEN disjE])
apply (iprover intro: minor major [THEN mp]) +

```

done

lemma *impCE'*:  
 assumes *major*:  $P \multimap Q$   
 and *minor*:  $Q \implies R \sim P \implies R$   
 shows *R*  
 apply (rule *excluded-middle* [of *P*, THEN *disjE*])  
 apply (iprover *intro*: *minor major* [THEN *mp*])+  
 done

lemma *iffCE*:  
 assumes *major*:  $P = Q$   
 and *minor*:  $[P; Q] \implies R \quad [\sim P; \sim Q] \implies R$   
 shows *R*  
 apply (rule *major* [THEN *iffE*])  
 apply (iprover *intro*: *minor elim*: *impCE notE*)  
 done

lemma *exCI*:  
 assumes *ALL* *x*.  $\sim P(x) \implies P(a)$   
 shows *EX* *x*.  $P(x)$   
 apply (rule *ccontr*)  
 apply (iprover *intro*: *assms exI allI notI notE* [of  $\exists x. P x$ ])  
 done

## 2.2.16 Intuitionistic Reasoning

lemma *impE'*:  
 assumes 1:  $P \multimap Q$   
 and 2:  $Q \implies R$   
 and 3:  $P \multimap Q \implies P$   
 shows *R*  
 proof –  
 from 3 and 1 have *P* .  
 with 1 have *Q* by (rule *impE*)  
 with 2 show *R* .  
 qed

lemma *allE'*:  
 assumes 1: *ALL* *x*. *P x*  
 and 2:  $P x \implies \text{ALL } x. P x \implies Q$   
 shows *Q*  
 proof –  
 from 1 have *P x* by (rule *spec*)  
 from *this* and 1 show *Q* by (rule 2)  
 qed

```

lemma notE':
  assumes 1:  $\sim P$ 
    and 2:  $\sim P \implies P$ 
  shows R
proof -
  from 2 and 1 have P .
  with 1 show R by (rule notE)
qed

lemma TrueE:  $True \implies P \implies P$  .
lemma notFalseE:  $\sim False \implies P \implies P$  .

lemmas [Pure.elim!] = disjE iffE FalseE conjE exE TrueE notFalseE
  and [Pure.intro!] = iffI conjI impI TrueI notI allI refl
  and [Pure.elim 2] = allE notE' impE'
  and [Pure.intro] = exI disjI2 disjI1

lemmas [trans] = trans
  and [sym] = sym not-sym
  and [Pure.elim?] = iffD1 iffD2 impE

use Tools/hologic.ML

2.2.17 Atomizing meta-level connectives

axiomatization where
  eq-reflection:  $x = y \implies x \equiv y$ 

lemma atomize-all [atomize]:  $(!!x. P x) == Trueprop (ALL x. P x)$ 
proof
  assume !!x. P x
  then show ALL x. P x ..
next
  assume ALL x. P x
  then show !!x. P x by (rule allE)
qed

lemma atomize-imp [atomize]:  $(A \implies B) == Trueprop (A \longrightarrow B)$ 
proof
  assume r:  $A \implies B$ 
  show  $A \longrightarrow B$  by (rule impI) (rule r)
next
  assume  $A \longrightarrow B$  and A
  then show B by (rule mp)
qed

lemma atomize-not:  $(A \implies False) == Trueprop (\sim A)$ 
proof
  assume r:  $A \implies False$ 

```

```

  show  $\sim A$  by (rule notI) (rule r)
next
  assume  $\sim A$  and  $A$ 
  then show False by (rule notE)
qed

```

```

lemma atomize-eq [atomize]:  $(x == y) == \text{Trueprop } (x = y)$ 
proof
  assume  $x == y$ 
  show  $x = y$  by (unfold  $\langle x == y \rangle$ ) (rule refl)
next
  assume  $x = y$ 
  then show  $x == y$  by (rule eq-reflection)
qed

```

```

lemma atomize-conj [atomize]:  $(A \&\&\& B) == \text{Trueprop } (A \& B)$ 
proof
  assume conj:  $A \&\&\& B$ 
  show  $A \& B$ 
  proof (rule conjI)
    from conj show  $A$  by (rule conjunctionD1)
    from conj show  $B$  by (rule conjunctionD2)
  qed
next
  assume conj:  $A \& B$ 
  show  $A \&\&\& B$ 
  proof -
    from conj show  $A$  ..
    from conj show  $B$  ..
  qed
qed

```

```

lemmas [symmetric, rulify] = atomize-all atomize-imp
and [symmetric, defn] = atomize-all atomize-imp atomize-eq

```

### 2.2.18 Atomizing elimination rules

setup *AtomizeElim.setup*

```

lemma atomize-exL[atomize-elim]:  $(!!x. P\ x ==> Q) == ((EX\ x. P\ x) ==> Q)$ 
by rule iprover+

```

```

lemma atomize-conjL[atomize-elim]:  $(A ==> B ==> C) == (A \& B ==> C)$ 
by rule iprover+

```

```

lemma atomize-disjL[atomize-elim]:  $((A ==> C) ==> (B ==> C) ==> C)$ 
 $== ((A \mid B ==> C) ==> C)$ 
by rule iprover+

```

**lemma** *atomize-elimL*[*atomize-elim*]: ( $\neg B. (A \implies B) \implies B$ ) == *Trueprop* *A* ..

## 2.3 Package setup

### 2.3.1 Sledgehammer setup

Theorems blacklisted to Sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

```
ML <<
structure No-ATPs = Named-Thms
(
  val name = no-atp
  val description = theorems that should be filtered out by Sledgehammer
)
>>

setup << No-ATPs.setup >>
```

### 2.3.2 Classical Reasoner setup

**lemma** *imp-elim*:  $P \dashv\dashv Q \implies (\sim R \implies P) \implies (Q \implies R) \implies R$   
**by** (*rule classical*) *iprover*

**lemma** *swap*:  $\sim P \implies (\sim R \implies P) \implies R$   
**by** (*rule classical*) *iprover*

**lemma** *thin-refl*:  
 $\bigwedge X. \llbracket x=x; \text{PROP } W \rrbracket \implies \text{PROP } W$  .

```
ML <<
structure Hypsubst = HypsubstFun(
  struct
    structure Simplifier = Simplifier
    val dest-eq = HOLogic.dest-eq
    val dest-Trueprop = HOLogic.dest-Trueprop
    val dest-imp = HOLogic.dest-imp
    val eq-reflection = @{thm eq-reflection}
    val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
    val imp-intr = @{thm impI}
    val rev-mp = @{thm rev-mp}
    val subst = @{thm subst}
    val sym = @{thm sym}
    val thin-refl = @{thm thin-refl};
    val prop-subst = @{lemma PROP P t ==> PROP prop (x = t ==> PROP P
x)
                                by (unfold prop-def) (drule eq-reflection, unfold)}
  end);
```

```
open Hypsubst;
```

```
structure Classical = ClassicalFun(
struct
  val imp-elim = @{thm imp-elim}
  val not-elim = @{thm notE}
  val swap = @{thm swap}
  val classical = @{thm classical}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
end);
```

```
structure Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;
```

```
ML-Antiquote.value claset
(Scan.succeed Classical.claset-of (ML-Context.the-local-context ()));
>>
```

```
setup Classical.setup
```

```
setup <<
let
  fun non-bool-eq (@{const-name op =}, Type (-, [T, -])) = T <> @{typ bool}
    | non-bool-eq - = false;
  val hyp-subst-tac' =
    SUBGOAL (fn (goal, i) =>
      if Term.exists-Const non-bool-eq goal
      then Hypsubst.hyp-subst-tac i
      else no-tac);
in
  Hypsubst.hypsubst-setup
  (*prevent substitution on bool*)
  #> Context-Rules.addSWrapper (fn tac => hyp-subst-tac' ORELSE' tac)
end
>>
```

```
declare iffI [intro!]
and notI [intro!]
and impI [intro!]
and disjCI [intro!]
and conjI [intro!]
and TrueI [intro!]
and refl [intro!]
```

```
declare iffCE [elim!]
and FalseE [elim!]
and impCE [elim!]
and disjE [elim!]
```



```

and conjE [elim!]

declare ex-ex1I [intro!]
  and allI [intro!]
  and the-equality [intro]
  and exI [intro]

declare exE [elim!]
  allE [elim]

ML ⟨⟨ val HOL-cs = @{claset} ⟩⟩

lemma contrapos-np:  $\sim Q \implies (\sim P \implies Q) \implies P$ 
  apply (erule swap)
  apply (erule (1) meta-mp)
  done

declare ex-ex1I [rule del, intro! 2]
  and ex1I [intro]

lemmas [intro?] = ext
  and [elim?] = ex1-implies-ex

lemma alt-ex1E [elim!]:
  assumes major:  $\exists!x. P x$ 
  and prem:  $\bigwedge x. \llbracket P x; \forall y y'. P y \wedge P y' \longrightarrow y = y' \rrbracket \implies R$ 
  shows R
  apply (rule ex1E [OF major])
  apply (rule prem)
  apply (tactic ⟨⟨ ares-tac @{thms allI} 1 ⟩⟩)+
  apply (tactic ⟨⟨ etac (Classical.dup-elim @{thm allE}) 1 ⟩⟩)
  apply iprover
  done

ML ⟨⟨
  structure Blast = Blast
  (
    val thy = @{theory}
    type claset = Classical.claset
    val equality-name = @{const-name op =}
    val not-name = @{const-name Not}
    val notE = @{thm notE}
    val ccontr = @{thm ccontr}
    val contr-tac = Classical.contr-tac
    val dup-intr = Classical.dup-intr
    val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
    val rep-cs = Classical.rep-cs
    val cla-modifiers = Classical.cla-modifiers
  )
  ⟩⟩

```

```

    val cla-meth' = Classical.cla-meth'
  );
  val blast-tac = Blast.blast-tac;
  >>

```

```

setup Blast.setup

```

### 2.3.3 Simplifier

**lemma** *eta-contract-eq*:  $(\%s. f\ s) = f \ ..$

**lemma** *simp-thms*:

```

shows not-not:  $(\sim \sim P) = P$ 
and Not-eq-iff:  $((\sim P) = (\sim Q)) = (P = Q)$ 
and
   $(P \sim = Q) = (P = (\sim Q))$ 
   $(P \mid \sim P) = \text{True} \quad (\sim P \mid P) = \text{True}$ 
   $(x = x) = \text{True}$ 
and not-True-eq-False [code]:  $(\neg \text{True}) = \text{False}$ 
and not-False-eq-True [code]:  $(\neg \text{False}) = \text{True}$ 
and
   $(\sim P) \sim = P \quad P \sim = (\sim P)$ 
   $(\text{True} = P) = P$ 
and eq-True:  $(P = \text{True}) = P$ 
and  $(\text{False} = P) = (\sim P)$ 
and eq-False:  $(P = \text{False}) = (\neg P)$ 
and
   $(\text{True} \dashrightarrow P) = P \quad (\text{False} \dashrightarrow P) = \text{True}$ 
   $(P \dashrightarrow \text{True}) = \text{True} \quad (P \dashrightarrow P) = \text{True}$ 
   $(P \dashrightarrow \text{False}) = (\sim P) \quad (P \dashrightarrow \sim P) = (\sim P)$ 
   $(P \ \& \ \text{True}) = P \quad (\text{True} \ \& \ P) = P$ 
   $(P \ \& \ \text{False}) = \text{False} \quad (\text{False} \ \& \ P) = \text{False}$ 
   $(P \ \& \ P) = P \quad (P \ \& \ (P \ \& \ Q)) = (P \ \& \ Q)$ 
   $(P \ \& \ \sim P) = \text{False} \quad (\sim P \ \& \ P) = \text{False}$ 
   $(P \mid \text{True}) = \text{True} \quad (\text{True} \mid P) = \text{True}$ 
   $(P \mid \text{False}) = P \quad (\text{False} \mid P) = P$ 
   $(P \mid P) = P \quad (P \mid (P \mid Q)) = (P \mid Q)$  and
   $(\text{ALL } x. P) = P \quad (\text{EX } x. P) = P \quad \text{EX } x. x=t \quad \text{EX } x. t=x$ 
and
  !!P.  $(\text{EX } x. x=t \ \& \ P(x)) = P(t)$ 
  !!P.  $(\text{EX } x. t=x \ \& \ P(x)) = P(t)$ 
  !!P.  $(\text{ALL } x. x=t \dashrightarrow P(x)) = P(t)$ 
  !!P.  $(\text{ALL } x. t=x \dashrightarrow P(x)) = P(t)$ 
by (blast, blast, blast, blast, blast, iprover+)

```

**lemma** *disj-absorb*:  $(A \mid A) = A$

**by** *blast*

**lemma** *disj-left-absorb*:  $(A \mid (A \mid B)) = (A \mid B)$

**by** *blast*

**lemma** *conj-absorb*:  $(A \ \& \ A) = A$   
**by** *blast*

**lemma** *conj-left-absorb*:  $(A \ \& \ (A \ \& \ B)) = (A \ \& \ B)$   
**by** *blast*

**lemma** *eq-ac*:  
**shows** *eq-commute*:  $(a=b) = (b=a)$   
**and** *eq-left-commute*:  $(P=(Q=R)) = (Q=(P=R))$   
**and** *eq-assoc*:  $((P=Q)=R) = (P=(Q=R))$  **by** (*iprover*, *blast*+)   
**lemma** *neg-commute*:  $(a\sim=b) = (b\sim=a)$  **by** *iprover*

**lemma** *conj-comms*:  
**shows** *conj-commute*:  $(P \ \& \ Q) = (Q \ \& \ P)$   
**and** *conj-left-commute*:  $(P \ \& \ (Q \ \& \ R)) = (Q \ \& \ (P \ \& \ R))$  **by** *iprover*+  
**lemma** *conj-assoc*:  $((P \ \& \ Q) \ \& \ R) = (P \ \& \ (Q \ \& \ R))$  **by** *iprover*

**lemmas** *conj-ac* = *conj-commute conj-left-commute conj-assoc*

**lemma** *disj-comms*:  
**shows** *disj-commute*:  $(P|Q) = (Q|P)$   
**and** *disj-left-commute*:  $(P|(Q|R)) = (Q|(P|R))$  **by** *iprover*+  
**lemma** *disj-assoc*:  $((P|Q)|R) = (P|(Q|R))$  **by** *iprover*

**lemmas** *disj-ac* = *disj-commute disj-left-commute disj-assoc*

**lemma** *conj-disj-distribL*:  $(P \ \& \ (Q|R)) = (P \ \& \ Q \ | \ P \ \& \ R)$  **by** *iprover*  
**lemma** *conj-disj-distribR*:  $((P|Q) \ \& \ R) = (P \ \& \ R \ | \ Q \ \& \ R)$  **by** *iprover*

**lemma** *disj-conj-distribL*:  $(P|(Q \ \& \ R)) = ((P|Q) \ \& \ (P|R))$  **by** *iprover*  
**lemma** *disj-conj-distribR*:  $((P \ \& \ Q)|R) = ((P|R) \ \& \ (Q|R))$  **by** *iprover*

**lemma** *imp-conjR*:  $(P \ \dashrightarrow \ (Q \ \& \ R)) = ((P \ \dashrightarrow \ Q) \ \& \ (P \ \dashrightarrow \ R))$  **by** *iprover*  
**lemma** *imp-conjL*:  $((P \ \& \ Q) \ \dashrightarrow \ R) = (P \ \dashrightarrow \ (Q \ \dashrightarrow \ R))$  **by** *iprover*  
**lemma** *imp-disjL*:  $((P|Q) \ \dashrightarrow \ R) = ((P \ \dashrightarrow \ R) \ \& \ (Q \ \dashrightarrow \ R))$  **by** *iprover*

These two are specialized, but *imp-disj-not1* is useful in *Auth/Yahalom*.

**lemma** *imp-disj-not1*:  $(P \ \dashrightarrow \ Q \ | \ R) = (\sim Q \ \dashrightarrow \ P \ \dashrightarrow \ R)$  **by** *blast*  
**lemma** *imp-disj-not2*:  $(P \ \dashrightarrow \ Q \ | \ R) = (\sim R \ \dashrightarrow \ P \ \dashrightarrow \ Q)$  **by** *blast*

**lemma** *imp-disj1*:  $((P \ \dashrightarrow \ Q)|R) = (P \ \dashrightarrow \ Q|R)$  **by** *blast*  
**lemma** *imp-disj2*:  $(Q|(P \ \dashrightarrow \ R)) = (P \ \dashrightarrow \ Q|R)$  **by** *blast*

**lemma** *imp-cong*:  $(P = P') \implies (P' \implies (Q = Q')) \implies ((P \ \dashrightarrow \ Q) = (P' \ \dashrightarrow \ Q'))$   
**by** *iprover*

**lemma** *de-Morgan-disj*:  $(\sim(P \mid Q)) = (\sim P \ \& \ \sim Q)$  **by** *iprover*  
**lemma** *de-Morgan-conj*:  $(\sim(P \ \& \ Q)) = (\sim P \mid \sim Q)$  **by** *blast*  
**lemma** *not-imp*:  $(\sim(P \dashrightarrow Q)) = (P \ \& \ \sim Q)$  **by** *blast*  
**lemma** *not-iff*:  $(P \sim Q) = (P = (\sim Q))$  **by** *blast*  
**lemma** *disj-not1*:  $(\sim P \mid Q) = (P \dashrightarrow Q)$  **by** *blast*  
**lemma** *disj-not2*:  $(P \mid \sim Q) = (Q \dashrightarrow P)$  — changes orientation :-(  
**by** *blast*  
**lemma** *imp-conv-disj*:  $(P \dashrightarrow Q) = ((\sim P) \mid Q)$  **by** *blast*  
  
**lemma** *iff-conv-conj-imp*:  $(P = Q) = ((P \dashrightarrow Q) \ \& \ (Q \dashrightarrow P))$  **by** *iprover*  
  
**lemma** *cases-simp*:  $((P \dashrightarrow Q) \ \& \ (\sim P \dashrightarrow Q)) = Q$   
— Avoids duplication of subgoals after *split-if*, when the true and false  
— cases boil down to the same thing.  
**by** *blast*  
  
**lemma** *not-all*:  $(\sim (! x. P(x))) = (? x. \sim P(x))$  **by** *blast*  
**lemma** *imp-all*:  $((! x. P \ x) \dashrightarrow Q) = (? x. P \ x \dashrightarrow Q)$  **by** *blast*  
**lemma** *not-ex*:  $(\sim (? x. P(x))) = (! x. \sim P(x))$  **by** *iprover*  
**lemma** *imp-ex*:  $((? x. P \ x) \dashrightarrow Q) = (! x. P \ x \dashrightarrow Q)$  **by** *iprover*  
**lemma** *all-not-ex*:  $(ALL \ x. P \ x) = (\sim (EX \ x. \sim P \ x))$  **by** *blast*  
  
**declare** *All-def* [*no-atp*]  
  
**lemma** *ex-disj-distrib*:  $(? x. P(x) \mid Q(x)) = ((? x. P(x)) \mid (? x. Q(x)))$  **by** *iprover*  
**lemma** *all-conj-distrib*:  $(! x. P(x) \ \& \ Q(x)) = ((! x. P(x)) \ \& \ (! x. Q(x)))$  **by** *iprover*  
  
The  $\&$  congruence rule: not included by default! May slow rewrite proofs  
down by as much as 50%  
  
**lemma** *conj-cong*:  
 $(P = P') ==> (P' ==> (Q = Q')) ==> ((P \ \& \ Q) = (P' \ \& \ Q'))$   
**by** *iprover*  
  
**lemma** *rev-conj-cong*:  
 $(Q = Q') ==> (Q' ==> (P = P')) ==> ((P \ \& \ Q) = (P' \ \& \ Q'))$   
**by** *iprover*  
  
The  $\mid$  congruence rule: not included by default!  
  
**lemma** *disj-cong*:  
 $(P = P') ==> (\sim P' ==> (Q = Q')) ==> ((P \mid Q) = (P' \mid Q'))$   
**by** *blast*  
  
if-then-else rules  
  
**lemma** *if-True* [*code*]:  $(if \ True \ then \ x \ else \ y) = x$   
**by** (*unfold if-def*) *blast*  
  
**lemma** *if-False* [*code*]:  $(if \ False \ then \ x \ else \ y) = y$

```

by (unfold if-def) blast

lemma if-P:  $P \implies (if\ P\ then\ x\ else\ y) = x$ 
by (unfold if-def) blast

lemma if-not-P:  $\sim P \implies (if\ P\ then\ x\ else\ y) = y$ 
by (unfold if-def) blast

lemma split-if:  $P (if\ Q\ then\ x\ else\ y) = ((Q \implies P(x)) \ \&\ (\sim Q \implies P(y)))$ 
apply (rule case-split [of Q])
apply (simplesubst if-P)
prefer 3 apply (simplesubst if-not-P, blast+)
done

lemma split-if-asm:  $P (if\ Q\ then\ x\ else\ y) = (\sim((Q \ \&\ \sim P\ x) \mid (\sim Q \ \&\ \sim P\ y)))$ 
by (simplesubst split-if, blast)

lemmas if-splits [no-atp] = split-if split-if-asm

lemma if-cancel:  $(if\ c\ then\ x\ else\ x) = x$ 
by (simplesubst split-if, blast)

lemma if-eq-cancel:  $(if\ x = y\ then\ y\ else\ x) = x$ 
by (simplesubst split-if, blast)

lemma if-bool-eq-conj:  $(if\ P\ then\ Q\ else\ R) = ((P \implies Q) \ \&\ (\sim P \implies R))$ 
  — This form is useful for expanding ifs on the RIGHT of the  $\implies$  symbol.
by (rule split-if)

lemma if-bool-eq-disj:  $(if\ P\ then\ Q\ else\ R) = ((P \ \&\ Q) \mid (\sim P \ \&\ R))$ 
  — And this form is useful for expanding ifs on the LEFT.
apply (simplesubst split-if, blast)
done

lemma Eq-TrueI:  $P \implies P == True$  by (unfold atomize-eq) iprover
lemma Eq-FalseI:  $\sim P \implies P == False$  by (unfold atomize-eq) iprover

let rules for simproc

lemma Let-folded:  $f\ x \equiv g\ x \implies Let\ x\ f \equiv Let\ x\ g$ 
by (unfold Let-def)

lemma Let-unfold:  $f\ x \equiv g \implies Let\ x\ f \equiv g$ 
by (unfold Let-def)

The following copy of the implication operator is useful for fine-tuning congruence rules. It instructs the simplifier to simplify its premise.

definition simp-implies ::  $[prop, prop] \implies prop$  (infixr =simp=> 1) where
  [code del]:  $simp-implies \equiv op \implies$ 

```

```

lemma simp-impliesI:
  assumes PQ: (PROP P  $\implies$  PROP Q)
  shows PROP P =simp=> PROP Q
  apply (unfold simp-implies-def)
  apply (rule PQ)
  apply assumption
  done

lemma simp-impliesE:
  assumes PQ: PROP P =simp=> PROP Q
  and P: PROP P
  and QR: PROP Q  $\implies$  PROP R
  shows PROP R
  apply (rule QR)
  apply (rule PQ [unfolded simp-implies-def])
  apply (rule P)
  done

lemma simp-implies-cong:
  assumes PP': PROP P == PROP P'
  and P'QQ': PROP P' ==> (PROP Q == PROP Q')
  shows (PROP P =simp=> PROP Q) == (PROP P' =simp=> PROP Q')
proof (unfold simp-implies-def, rule equal-intr-rule)
  assume PQ: PROP P  $\implies$  PROP Q
  and P': PROP P'
  from PP' [symmetric] and P' have PROP P
    by (rule equal-elim-rule1)
  then have PROP Q by (rule PQ)
  with P'QQ' [OF P'] show PROP Q' by (rule equal-elim-rule1)
next
  assume P'Q': PROP P'  $\implies$  PROP Q'
  and P: PROP P
  from PP' and P have P': PROP P' by (rule equal-elim-rule1)
  then have PROP Q' by (rule P'Q')
  with P'QQ' [OF P', symmetric] show PROP Q
    by (rule equal-elim-rule1)
qed

lemma uncurry:
  assumes P  $\longrightarrow$  Q  $\longrightarrow$  R
  shows P  $\wedge$  Q  $\longrightarrow$  R
  using assms by blast

lemma iff-allI:
  assumes  $\bigwedge x. P\ x = Q\ x$ 
  shows  $(\forall x. P\ x) = (\forall x. Q\ x)$ 
  using assms by blast

```

```

lemma iff-exI:
  assumes  $\bigwedge x. P\ x = Q\ x$ 
  shows  $(\exists x. P\ x) = (\exists x. Q\ x)$ 
  using assms by blast

```

```

lemma all-comm:
   $(\forall x\ y. P\ x\ y) = (\forall y\ x. P\ x\ y)$ 
  by blast

```

```

lemma ex-comm:
   $(\exists x\ y. P\ x\ y) = (\exists y\ x. P\ x\ y)$ 
  by blast

```

```

use Tools/simpdata.ML
ML  $\langle\langle$  open Simpdata  $\rangle\rangle$ 

```

```

setup  $\langle\langle$ 
  Simplifier.method-setup Splitter.split-modifiers
   $\#>$  Simplifier.map-simpset (K Simpdata.simpset-simprocs)
   $\#>$  Splitter.setup
   $\#>$  clasimp-setup
   $\#>$  EqSubst.setup
 $\rangle\rangle$ 

```

Simproc for proving  $(y = x) == \text{False}$  from premise  $\sim(x = y)$ :

```

simproc-setup neq  $(x = y) = \langle\langle$  fn  $- =>$ 
  let
    val neq-to-EQ-False =  $\@ \{ \text{thm not-sym} \}$  RS  $\@ \{ \text{thm Eq-FalseI} \}$ ;
    fun is-neq eq lhs rhs thm =
      (case Thm.prop-of thm of
        -  $\$$  (Not  $\$$  (eq'  $\$$  l'  $\$$  r'))  $=>$ 
          Not = HOLogic.Not andalso eq' = eq andalso
            r' aconv lhs andalso l' aconv rhs
        | -  $=>$  false);
    fun proc ss ct =
      (case Thm.term-of ct of
        eq  $\$$  lhs  $\$$  rhs  $=>$ 
          (case find-first (is-neq eq lhs rhs) (Simplifier.premis-of-ss ss) of
            SOME thm  $=>$  SOME (thm RS neq-to-EQ-False)
            | NONE  $=>$  NONE)
        | -  $=>$  NONE);
  in proc end;
 $\rangle\rangle$ 

```

```

simproc-setup let-simp (Let x f) =  $\langle\langle$ 
  let
    val (f-Let-unfold, x-Let-unfold) =
      let val [ $-$  (f  $\$$  x)  $\$$   $-$ ] = prems-of  $\@ \{ \text{thm Let-unfold} \}$ 
      in (cterm-of  $\@ \{ \text{theory} \}$  f, cterm-of  $\@ \{ \text{theory} \}$  x) end

```

```

val (f-Let-folded, x-Let-folded) =
  let val [(- $ (f $ x) $ -)] = prems-of @{thm Let-folded}
  in (cterm-of @{theory} f, cterm-of @{theory} x) end;
val g-Let-folded =
  let val [(- $ - $ (g $ -))] = prems-of @{thm Let-folded}
  in cterm-of @{theory} g end;
fun count-loose (Bound i) k = if i >= k then 1 else 0
  | count-loose (s $ t) k = count-loose s k + count-loose t k
  | count-loose (Abs (-, -, t)) k = count-loose t (k + 1)
  | count-loose - - = 0;
fun is-trivial-let (Const (@{const-name Let}, -) $ x $ t) =
  case t
  of Abs (-, -, t') => count-loose t' 0 <= 1
  | - => true;
in fn - => fn ss => fn ct => if is-trivial-let (Thm.term-of ct)
  then SOME @{thm Let-def} (*no or one occurrence of bound variable*)
  else let (*Norbert Schirmer's case*)
    val ctxt = Simplifier.the-context ss;
    val thy = ProofContext.theory-of ctxt;
    val t = Thm.term-of ct;
    val ([t'], ctxt') = Variable.import-terms false [t] ctxt;
  in Option.map (hd o Variable.export ctxt' ctxt o single)
    (case t' of Const (@{const-name Let},-) $ x $ f => (* x and f are already in
normal form *))
    if is-Free x orelse is-Bound x orelse is-Const x
    then SOME @{thm Let-def}
    else
      let
        val n = case f of (Abs (x, -, -)) => x | - => x;
        val cx = cterm-of thy x;
        val {T = xT, ...} = rep-cterm cx;
        val cf = cterm-of thy f;
        val fx-g = Simplifier.rewrite ss (Thm.capply cf cx);
        val (- $ - $ g) = prop-of fx-g;
        val g' = abstract-over (x,g);
      in (if (g aconv g')
        then
          let
            val rl =
              cterm-instantiate [(f-Let-unfold, cf), (x-Let-unfold, cx)] @{thm
Let-unfold};
          in SOME (rl OF [fx-g]) end
        else if Term.betapply (f, x) aconv g then NONE (*avoid identity
conversion*)
        else let
          val abs-g' = Abs (n,xT,g');
          val g'x = abs-g'$x;
          val g-g'x = Thm.symmetric (Thm.beta-conversion false (cterm-of
thy g'x));

```



```

      val rl = cterm-instantiate
        [(f-Let-folded, cterm-of thy f), (x-Let-folded, cx),
         (g-Let-folded, cterm-of thy abs-g')]
        @{thm Let-folded};
    in SOME (rl OF [Thm.transitive fx-g g-g'x])
    end)
  end
| - => NONE)
end
end >>

```

**lemma** *True-implies-equals*:  $(\text{True} \implies \text{PROP } P) \equiv \text{PROP } P$

**proof**

**assume**  $\text{True} \implies \text{PROP } P$

**from** *this* [OF TrueI] **show**  $\text{PROP } P$  .

**next**

**assume**  $\text{PROP } P$

**then show**  $\text{PROP } P$  .

**qed**

**lemma** *ex-simps*:

!!P Q.  $(\text{EX } x. P \ x \ \& \ Q) = ((\text{EX } x. P \ x) \ \& \ Q)$

!!P Q.  $(\text{EX } x. P \ \& \ Q \ x) = (P \ \& \ (\text{EX } x. Q \ x))$

!!P Q.  $(\text{EX } x. P \ x \ | \ Q) = ((\text{EX } x. P \ x) \ | \ Q)$

!!P Q.  $(\text{EX } x. P \ | \ Q \ x) = (P \ | \ (\text{EX } x. Q \ x))$

!!P Q.  $(\text{EX } x. P \ x \ \longrightarrow \ Q) = ((\text{ALL } x. P \ x) \ \longrightarrow \ Q)$

!!P Q.  $(\text{EX } x. P \ \longrightarrow \ Q \ x) = (P \ \longrightarrow \ (\text{EX } x. Q \ x))$

— Miniscoping: pushing in existential quantifiers.

**by** (*iprover* | *blast*)+

**lemma** *all-simps*:

!!P Q.  $(\text{ALL } x. P \ x \ \& \ Q) = ((\text{ALL } x. P \ x) \ \& \ Q)$

!!P Q.  $(\text{ALL } x. P \ \& \ Q \ x) = (P \ \& \ (\text{ALL } x. Q \ x))$

!!P Q.  $(\text{ALL } x. P \ x \ | \ Q) = ((\text{ALL } x. P \ x) \ | \ Q)$

!!P Q.  $(\text{ALL } x. P \ | \ Q \ x) = (P \ | \ (\text{ALL } x. Q \ x))$

!!P Q.  $(\text{ALL } x. P \ x \ \longrightarrow \ Q) = ((\text{EX } x. P \ x) \ \longrightarrow \ Q)$

!!P Q.  $(\text{ALL } x. P \ \longrightarrow \ Q \ x) = (P \ \longrightarrow \ (\text{ALL } x. Q \ x))$

— Miniscoping: pushing in universal quantifiers.

**by** (*iprover* | *blast*)+

**lemmas** [*simp*] =

*triv-forall-equality*

*True-implies-equals*

*if-True*

*if-False*

*if-cancel*

*if-eq-cancel*

*imp-disjL*

*conj-assoc*  
*disj-assoc*  
*de-Morgan-conj*  
*de-Morgan-disj*  
*imp-disj1*  
*imp-disj2*  
*not-imp*  
*disj-not1*  
*not-all*  
*not-ex*  
*cases-simp*  
*the-eq-trivial*  
*the-sym-eq-trivial*  
*ex-simps*  
*all-simps*  
*simp-thms*

**lemmas** [*cong*] = *imp-cong simp-implies-cong*

**lemmas** [*split*] = *split-if*

**ML**  $\ll \text{val } HOL\text{-ss} = @\{\text{simpset}\} \gg$

Simplifies x assuming c and y assuming  $\neg c$

**lemma** *if-cong*:

**assumes**  $b = c$

**and**  $c \implies x = u$

**and**  $\neg c \implies y = v$

**shows**  $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$

**unfolding** *if-def* **using** *assms* **by** *simp*

Prevents simplification of x and y: faster and allows the execution of functional programs.

**lemma** *if-weak-cong* [*cong*]:

**assumes**  $b = c$

**shows**  $(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } x \text{ else } y)$

**using** *assms* **by** (*rule arg-cong*)

Prevents simplification of t: much faster

**lemma** *let-weak-cong*:

**assumes**  $a = b$

**shows**  $(\text{let } x = a \text{ in } t \ x) = (\text{let } x = b \text{ in } t \ x)$

**using** *assms* **by** (*rule arg-cong*)

To tidy up the result of a simproc. Only the RHS will be simplified.

**lemma** *eq-cong2*:

**assumes**  $u = u'$

**shows**  $(t \equiv u) \equiv (t \equiv u')$

**using** *assms* **by** *simp*

**lemma** *if-distrib*:

$f \text{ (if } c \text{ then } x \text{ else } y) = (\text{if } c \text{ then } f \ x \text{ else } f \ y)$   
**by** *simp*

This lemma restricts the effect of the rewrite rule  $u=v$  to the left-hand side of an equality. Used in  $\{Integ, Real\}/simproc.ML$

**lemma** *restrict-to-left*:

**assumes**  $x = y$   
**shows**  $(x = z) = (y = z)$   
**using** *assms* **by** *simp*

### 2.3.4 Generic cases and induction

Rule projections:

**ML**  $\langle\langle$   
 $structure \ Project\text{-}Rule = \ Project\text{-}Rule$   
 $($   
 $\quad val \ conjunct1 = @\{thm \ conjunct1\}$   
 $\quad val \ conjunct2 = @\{thm \ conjunct2\}$   
 $\quad val \ mp = @\{thm \ mp\}$   
 $)$   
 $\rangle\rangle$

**definition** *induct-forall* **where**

$induct\text{-}forall \ P == \forall x. \ P \ x$

**definition** *induct-implies* **where**

$induct\text{-}implies \ A \ B == A \longrightarrow B$

**definition** *induct-equal* **where**

$induct\text{-}equal \ x \ y == x = y$

**definition** *induct-conj* **where**

$induct\text{-}conj \ A \ B == A \wedge B$

**definition** *induct-true* **where**

$induct\text{-}true == True$

**definition** *induct-false* **where**

$induct\text{-}false == False$

**lemma** *induct-forall-eq*:  $(!!x. \ P \ x) == Trueprop \ (induct\text{-}forall \ (\lambda x. \ P \ x))$

**by** *(unfold atomize-all induct-forall-def)*

**lemma** *induct-implies-eq*:  $(A ==> B) == Trueprop \ (induct\text{-}implies \ A \ B)$

**by** *(unfold atomize-imp induct-implies-def)*

```

lemma induct-equal-eq:  $(x == y) == \text{Trueprop } (\text{induct-equal } x \ y)$ 
  by (unfold atomize-eq induct-equal-def)

lemma induct-conj-eq:  $(A \ \&\&\& \ B) == \text{Trueprop } (\text{induct-conj } A \ B)$ 
  by (unfold atomize-conj induct-conj-def)

lemmas induct-atomize' = induct-forall-eq induct-implies-eq induct-conj-eq
lemmas induct-atomize = induct-atomize' induct-equal-eq
lemmas induct-rulify' [symmetric, standard] = induct-atomize'
lemmas induct-rulify [symmetric, standard] = induct-atomize
lemmas induct-rulify-fallback =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-true-def induct-false-def

lemma induct-forall-conj:  $\text{induct-forall } (\lambda x. \text{induct-conj } (A \ x) \ (B \ x)) =$ 
   $\text{induct-conj } (\text{induct-forall } A) \ (\text{induct-forall } B)$ 
  by (unfold induct-forall-def induct-conj-def) iprover

lemma induct-implies-conj:  $\text{induct-implies } C \ (\text{induct-conj } A \ B) =$ 
   $\text{induct-conj } (\text{induct-implies } C \ A) \ (\text{induct-implies } C \ B)$ 
  by (unfold induct-implies-def induct-conj-def) iprover

lemma induct-conj-curry:  $(\text{induct-conj } A \ B ==> \text{PROP } C) == (A ==> B ==>$ 
   $\text{PROP } C)$ 
proof
  assume r:  $\text{induct-conj } A \ B ==> \text{PROP } C$  and  $A \ B$ 
  show  $\text{PROP } C$  by (rule r) (simp add: induct-conj-def  $\langle A \rangle \langle B \rangle$ )
next
  assume r:  $A ==> B ==> \text{PROP } C$  and  $\text{induct-conj } A \ B$ 
  show  $\text{PROP } C$  by (rule r) (simp-all add:  $\langle \text{induct-conj } A \ B \rangle$  [unfolded induct-conj-def])
qed

lemmas induct-conj = induct-forall-conj induct-implies-conj induct-conj-curry

lemma induct-trueI: induct-true
  by (simp add: induct-true-def)

```

Method setup.

```

ML <<
  structure Induct = Induct
  (
    val cases-default = @{thm case-split}
    val atomize = @{thms induct-atomize}
    val rulify = @{thms induct-rulify}
    val rulify-fallback = @{thms induct-rulify-fallback}
    val equal-def = @{thm induct-equal-def}
    fun dest-def (Const (@{const-name induct-equal}, -) $ t $ u) = SOME (t, u)
    | dest-def - = NONE
  )

```

```

    val trivial-tac = match-tac @{thms induct-trueI}
  )
  >>

setup <<
  Induct.setup #>
  Context.theory-map (Induct.map-simpset (fn ss => ss
    setmksimps (fn ss => Simpdata.mksimps Simpdata.mksimps-pairs ss #>
      map (Simplifier.rewrite-rule (map Thm.symmetric
        @{thms induct-rulify-fallback}))))
  addsimprocs
  [Simplifier.simproc @{theory} swap-induct-false
    [induct-false ==> PROP P ==> PROP Q]
    (fn - => fn - =>
      (fn - $ (P as - $ @{const induct-false}) $ (- $ Q $ -) =>
        if P <> Q then SOME Drule.swap-prems-eq else NONE
        | - => NONE)),
  Simplifier.simproc @{theory} induct-equal-conj-curry
    [induct-conj P Q ==> PROP R]
    (fn - => fn - =>
      (fn - $ (- $ P) $ - =>
        let
          fun is-conj (@{const induct-conj} $ P $ Q) =
            is-conj P andalso is-conj Q
            | is-conj (Const (@{const-name induct-equal}, -) $ - $ -) = true
            | is-conj @{const induct-true} = true
            | is-conj @{const induct-false} = true
            | is-conj - = false
          in if is-conj P then SOME @{thm induct-conj-curry} else NONE end
          | - => NONE)))]))
  >>

```

Pre-simplification of induction and cases rules

```

lemma [induct-simp]: (!!x. induct-equal x t ==> PROP P x) == PROP P t
unfolding induct-equal-def

```

**proof**

```

  assume R: !!x. x = t ==> PROP P x
  show PROP P t by (rule R [OF refl])

```

**next**

```

  fix x assume PROP P t x = t
  then show PROP P x by simp

```

**qed**

```

lemma [induct-simp]: (!!x. induct-equal t x ==> PROP P x) == PROP P t
unfolding induct-equal-def

```

**proof**

```

  assume R: !!x. t = x ==> PROP P x
  show PROP P t by (rule R [OF refl])

```

**next**

```

fix  $x$  assume  $PROP\ P\ t\ t = x$ 
then show  $PROP\ P\ x$  by simp
qed

```

```

lemma [induct-simp]: ( $induct\ false ==> P$ ) == Trueprop induct-true
unfolding induct-false-def induct-true-def
by (iprover intro: equal-intr-rule)

```

```

lemma [induct-simp]: ( $induct\ true ==> PROP\ P$ ) ==  $PROP\ P$ 
unfolding induct-true-def
proof
  assume  $R: True \implies PROP\ P$ 
  from TrueI show  $PROP\ P$  by (rule R)
next
  assume  $PROP\ P$ 
  then show  $PROP\ P$  .
qed

```

```

lemma [induct-simp]: ( $PROP\ P ==> induct\ true$ ) == Trueprop induct-true
unfolding induct-true-def
by (iprover intro: equal-intr-rule)

```

```

lemma [induct-simp]: ( $!!x. induct\ true$ ) == Trueprop induct-true
unfolding induct-true-def
by (iprover intro: equal-intr-rule)

```

```

lemma [induct-simp]:  $induct\ implies\ induct\ true\ P == P$ 
by (simp add: induct-implies-def induct-true-def)

```

```

lemma [induct-simp]: ( $x = x$ ) = True
by (rule simp-thms)

```

```

hide-const induct-forall induct-implies induct-equal induct-conj induct-true induct-false

```

```

use  $\sim\sim/src/Tools/induct-tacs.ML$ 
setup InductTacs.setup

```

### 2.3.5 Coherent logic

```

ML <<
  structure Coherent = Coherent
  (
    val atomize-elimL = @{thm atomize-elimL}
    val atomize-exL = @{thm atomize-exL}
    val atomize-conjL = @{thm atomize-conjL}
    val atomize-disjL = @{thm atomize-disjL}
    val operator-names =
      [ @{const-name op |}, @{const-name op &}, @{const-name Ex} ]
  );

```

»

**setup** *Coherent.setup*

### 2.3.6 Reorienting equalities

```

ML <<
signature REORIENT-PROC =
sig
  val add : (term -> bool) -> theory -> theory
  val proc : morphism -> simpset -> cterm -> thm option
end;

structure Reorient-Proc : REORIENT-PROC =
struct
  structure Data = Theory-Data
  (
    type T = ((term -> bool) * stamp) list;
    val empty = [];
    val extend = I;
    fun merge data : T = Library.merge (eq-snd op =) data;
  );
  fun add m = Data.map (cons (m, stamp ())) ;
  fun matches thy t = exists (fn (m, -) => m t) (Data.get thy);

  val meta-reorient = @{thm eq-commute [THEN eq-reflection]};
  fun proc phi ss ct =
    let
      val ctxt = Simplifier.the-context ss;
      val thy = ProofContext.theory-of ctxt;
    in
      case Thm.term-of ct of
        (- $ t $ u) => if matches thy u then NONE else SOME meta-reorient
      | - => NONE
    end;
end;
>>

```

## 2.4 Other simple lemmas and lemma duplicates

**lemma** *ex1-eq* [*iff*]:  $EX! x. x = t \rightarrow EX! x. t = x$   
**by** *blast+*

**lemma** *choice-eq*:  $(ALL x. EX! y. P x y) = (EX! f. ALL x. P x (f x))$   
**apply** (*rule iffI*)  
**apply** (*rule-tac a = %x. THE y. P x y in ex1I*)  
**apply** (*fast dest!: theI'*)  
**apply** (*fast intro: ext the1-equality [symmetric]*)  
**apply** (*erule ex1E*)  
**apply** (*rule allI*)

```

apply (rule ex1I)
apply (erule spec)
apply (erule-tac  $x = \%z.$  if  $z = x$  then  $y$  else  $f z$  in allE)
apply (erule impE)
apply (rule allI)
apply (case-tac  $xa = x$ )
apply (drule-tac  $[\beta]$   $x = x$  in fun-cong, simp-all)
done

```

**lemmas** eq-sym-conv = eq-commute

**lemma** nnf-simps:

```

 $(\neg(P \wedge Q)) = (\neg P \vee \neg Q) \quad (\neg(P \vee Q)) = (\neg P \wedge \neg Q) \quad (P \longrightarrow Q) = (\neg P \vee Q)$ 
 $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q)) \quad (\neg(P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$ 
 $(\neg \neg(P)) = P$ 
by blast+

```

## 2.5 Basic ML bindings

```

ML ⟨⟨
  val FalseE = @{thm FalseE}
  val Let-def = @{thm Let-def}
  val TrueI = @{thm TrueI}
  val allE = @{thm allE}
  val allI = @{thm allI}
  val all-dupE = @{thm all-dupE}
  val arg-cong = @{thm arg-cong}
  val box-equals = @{thm box-equals}
  val ccontr = @{thm ccontr}
  val classical = @{thm classical}
  val conjE = @{thm conjE}
  val conjI = @{thm conjI}
  val conjunct1 = @{thm conjunct1}
  val conjunct2 = @{thm conjunct2}
  val disjCI = @{thm disjCI}
  val disjE = @{thm disjE}
  val disjI1 = @{thm disjI1}
  val disjI2 = @{thm disjI2}
  val eq-reflection = @{thm eq-reflection}
  val ex1E = @{thm ex1E}
  val ex1I = @{thm ex1I}
  val ex1-implies-ex = @{thm ex1-implies-ex}
  val exE = @{thm exE}
  val exI = @{thm exI}
  val excluded-middle = @{thm excluded-middle}
  val ext = @{thm ext}
  val fun-cong = @{thm fun-cong}
  val iffD1 = @{thm iffD1}

```



```

val iffD2 = @{thm iffD2}
val iffI = @{thm iffI}
val impE = @{thm impE}
val impI = @{thm impI}
val meta-eq-to-obj-eq = @{thm meta-eq-to-obj-eq}
val mp = @{thm mp}
val notE = @{thm notE}
val notI = @{thm notI}
val not-all = @{thm not-all}
val not-ex = @{thm not-ex}
val not-iff = @{thm not-iff}
val not-not = @{thm not-not}
val not-sym = @{thm not-sym}
val refl = @{thm refl}
val rev-mp = @{thm rev-mp}
val spec = @{thm spec}
val ssubst = @{thm ssubst}
val subst = @{thm subst}
val sym = @{thm sym}
val trans = @{thm trans}
>>

```

## 2.6 Code generator setup

### 2.6.1 SML code generator setup

```
use Tools/recfun-codegen.ML
```

```

setup <<
  Codegen.setup
  #> RecfunCodegen.setup
  #> Codegen.map-unfold (K HOL-basic-ss)
>>

types-code
  bool (bool)
attach (term-of) <<
  fun term-of-bool b = if b then HOLogic.true-const else HOLogic.false-const;
>>
attach (test) <<
  fun gen-bool i =
    let val b = one-of [false, true]
    in (b, fn () => term-of-bool b) end;
>>
  prop (bool)
attach (term-of) <<
  fun term-of-prop b =
    HOLogic.mk-Trueprop (if b then HOLogic.true-const else HOLogic.false-const);
>>

```

**consts-code**

```

Trueprop ((-))
True      (true)
False     (false)
Not       (Bool.not)
op |      ((- orelse/ -))
op &      ((- andalso/ -))
If        ((if -/ then -/ else -))

```

**setup**  $\ll$ 

```
let
```

```

fun eq-codegen thy defs dep thyname b t gr =
  (case strip-comb t of
    (Const (@{const-name op =}, Type (-, [Type (fun, -), -]), -) => NONE
   | (Const (@{const-name op =}, -, [t, u]) =>
      let
        val (pt, gr') = Codegen.invoke-codegen thy defs dep thyname false t gr;
        val (pu, gr'') = Codegen.invoke-codegen thy defs dep thyname false u gr';
        val (-, gr''') = Codegen.invoke-tycodegen thy defs dep thyname false
        HOLogic.boolT gr'';
      in
        SOME (Codegen.parens
              (Pretty.block [pt, Codegen.str =, Pretty.brk 1, pu]), gr''')
        end
      | (t as Const (@{const-name op =}, -, ts) => SOME (Codegen.invoke-codegen
        thy defs dep thyname b (Codegen.eta-expand t ts 2) gr)
      | - => NONE);
  in
    Codegen.add-codegen eq-codegen eq-codegen
  end
  >>

```

**2.6.2 Generic code generator preprocessor setup****setup**  $\ll$ 

```

Code-Preproc.map-pre (K HOL-basic-ss)
#> Code-Preproc.map-post (K HOL-basic-ss)
>>

```

**2.6.3 Equality****class** *eq* =

```

fixes eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
assumes eq-equals: eq x y  $\longleftrightarrow$  x = y

```

**begin**

```

lemma eq [code-unfold, code-inline del]: eq = (op =)
  by (rule ext eq-equals)+

```

```

lemma eq-refl:  $eq\ x\ x \longleftrightarrow True$ 
  unfolding eq by rule+

lemma equals-eq:  $(op =) \equiv eq$ 
  by (rule eq-reflection) (rule ext, rule ext, rule sym, rule eq-equals)

declare equals-eq [symmetric, code-post]

end

declare equals-eq [code]

setup ⟨⟨
  Code-Preproc.map-pre (fn simpset =>
    simpset addsimprocs [Simplifier.simproc-i @{theory} eq [@{term op =}]]
    (fn thy => fn - => fn t as Const (-, T) => case strip-type T
      of ((T as Type -) :: -, -) => SOME @{thm equals-eq}
      | - => NONE))
  ⟩⟩

```

## 2.6.4 Generic code generator foundation

Datatypes

**code-datatype** *True False*

**code-datatype** *TYPE('a::{})*

**code-datatype** *prop Trueprop*

Code equations

```

lemma [code]:
  shows  $(True \implies PROP\ Q) \equiv PROP\ Q$ 
  and  $(PROP\ Q \implies True) \equiv Trueprop\ True$ 
  and  $(P \implies R) \equiv Trueprop\ (P \longrightarrow R)$  by (auto intro!: equal-intr-rule)

```

```

lemma [code]:
  shows  $False \wedge P \longleftrightarrow False$ 
  and  $True \wedge P \longleftrightarrow P$ 
  and  $P \wedge False \longleftrightarrow False$ 
  and  $P \wedge True \longleftrightarrow P$  by simp-all

```

```

lemma [code]:
  shows  $False \vee P \longleftrightarrow P$ 
  and  $True \vee P \longleftrightarrow True$ 
  and  $P \vee False \longleftrightarrow P$ 
  and  $P \vee True \longleftrightarrow True$  by simp-all

```

```

lemma [code]:

```

```

shows (False  $\longrightarrow$  P)  $\longleftrightarrow$  True
and (True  $\longrightarrow$  P)  $\longleftrightarrow$  P
and (P  $\longrightarrow$  False)  $\longleftrightarrow$   $\neg$  P
and (P  $\longrightarrow$  True)  $\longleftrightarrow$  True by simp-all

instantiation itself :: (type) eq
begin

definition eq-itself :: 'a itself  $\Rightarrow$  'a itself  $\Rightarrow$  bool where
  eq-itself x y  $\longleftrightarrow$  x = y

instance proof
qed (fact eq-itself-def)

end

lemma eq-itself-code [code]:
  eq-class.eq TYPE('a) TYPE('a)  $\longleftrightarrow$  True
  by (simp add: eq)

Equality

declare simp-thms(6) [code nbe]

setup  $\langle\langle$ 
  Sign.add-const-constraint (@{const-name eq}, SOME @{typ 'a::type  $\Rightarrow$  'a  $\Rightarrow$  bool})
 $\rangle\rangle$ 

lemma equals-alias-cert: OFCLASS('a, eq-class)  $\equiv$  ((op = :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\equiv$ 
eq) (is ?ofclass  $\equiv$  ?eq)
proof
  assume PROP ?ofclass
  show PROP ?eq
  by (tactic  $\langle\langle$  ALLGOALS (rtac (Thm.unconstrainT @{thm equals-eq})  $\rangle\rangle$ )
    (fact  $\langle$ PROP ?ofclass $\rangle$ )
  next
  assume PROP ?eq
  show PROP ?ofclass proof
  qed (simp add:  $\langle$ PROP ?eq $\rangle$ )
qed

setup  $\langle\langle$ 
  Sign.add-const-constraint (@{const-name eq}, SOME @{typ 'a::eq  $\Rightarrow$  'a  $\Rightarrow$  bool})
 $\rangle\rangle$ 

setup  $\langle\langle$ 
  Nbe.add-const-alias @{thm equals-alias-cert}
 $\rangle\rangle$ 

```

**hide-const** (**open**) *eq*

Cases

**lemma** *Let-case-cert*:

**assumes**  $CASE \equiv (\lambda x. \text{Let } x \text{ } f)$   
**shows**  $CASE \ x \equiv f \ x$   
**using** *assms* **by** *simp-all*

**lemma** *If-case-cert*:

**assumes**  $CASE \equiv (\lambda b. \text{If } b \text{ } f \text{ } g)$   
**shows**  $(CASE \ True \equiv f) \ \&\&\& \ (CASE \ False \equiv g)$   
**using** *assms* **by** *simp-all*

**setup**  $\langle\langle$

*Code.add-case*  $\@ \{thm \text{ Let-case-cert} \}$   
 $\#>$  *Code.add-case*  $\@ \{thm \text{ If-case-cert} \}$   
 $\#>$  *Code.add-undefined*  $\@ \{const-name \text{ undefined} \}$   
 $\rangle\rangle$

**code-abort** *undefined*

### 2.6.5 Generic code generator target languages

type *bool*

**code-type** *bool*

(*SML bool*)  
(*OCaml bool*)  
(*Haskell Bool*)  
(*Scala Boolean*)

**code-const** *True and False and Not and op & and op | and If*

(*SML true and false and not*  
**and** **infixl** 1 *andalso* **and** **infixl** 0 *orelse*  
**and**  $!(if \ (-) / \ then \ (-) / \ else \ (-))$ )  
(*OCaml true and false and not*  
**and** **infixl** 4  $\&\&$  **and** **infixl** 2  $||$   
**and**  $!(if \ (-) / \ then \ (-) / \ else \ (-))$ )  
(*Haskell True and False and not*  
**and** **infixl** 3  $\&\&$  **and** **infixl** 2  $||$   
**and**  $!(if \ (-) / \ then \ (-) / \ else \ (-))$ )  
(*Scala true and false and '!* -  
**and** **infixl** 3  $\&\&$  **and** **infixl** 1  $||$   
**and**  $!(if \ ((-)) / \ (-) / \ else \ (-))$ )

**code-reserved** *SML*

*bool true false not*

**code-reserved** *OCaml*

*bool not*

**code-reserved** *Scala*

*Boolean*

using built-in Haskell equality

**code-class** *eq*

*(Haskell Eq)*

**code-const** *eq-class.eq*

*(Haskell infixl 4 ==)*

**code-const** *op =*

*(Haskell infixl 4 ==)*

undefined

**code-const** *undefined*

*(SML !(raise/ Fail/ undefined))*

*(OCaml failwith/ undefined)*

*(Haskell error/ undefined)*

*(Scala !error(undefined))*

### 2.6.6 Evaluation and normalization by evaluation

Avoid some named infixes in evaluation environment

**code-reserved** *Eval oo ooo oooo upto downto orf andf*

**setup**  $\ll$

*Value.add-evaluator (SML, Codegen.eval-term o ProofContext.theory-of)*

$\gg$

**ML**  $\ll$

*structure Eval-Method =*

*struct*

*val eval-ref : (unit -> bool) option Unsynchronized.ref = Unsynchronized.ref NONE;*

*end;*

$\gg$

**oracle** *eval-oracle =  $\ll$  fn ct =>*

*let*

*val thy = Thm.theory-of-cterm ct;*

*val t = Thm.term-of ct;*

*val dummy = @{cprop True};*

*in case try HOLogic.dest-Trueprop t*

*of SOME t' => if Code-Eval.eval NONE*

*(Eval-Method.eval-ref, Eval-Method.eval-ref) (K I) thy t' []*

```

      then Thm.capply (Thm.capply @{cterm op ≡ :: prop ⇒ prop ⇒ prop} ct)
dummy
      else dummy
    | NONE => dummy
  end
>>

```

```

ML <<
fun gen-eval-method conv ctxt = SIMPLE-METHOD'
  (CONVERSION (Conv.params-conv (~1) (K (Conv.concl-conv (~1) conv))
  ctxt)
  THEN' rtac TrueI)
>>

```

```

method-setup eval = << Scan.succeed (gen-eval-method eval-oracle) >>
  solve goal by evaluation

```

```

method-setup evaluation = << Scan.succeed (gen-eval-method Codegen.evaluation-conv)
>>
  solve goal by evaluation

```

```

method-setup normalization = <<
  Scan.succeed (K (SIMPLE-METHOD' (CONVERSION Nbe.norm-conv THEN'
  (fn k => TRY (rtac TrueI k))))))
>> solve goal by normalization

```

## 2.7 Counterexample Search Units

### 2.7.1 Quickcheck

```

quickcheck-params [size = 5, iterations = 50]

```

### 2.7.2 Nitpick setup

```

ML <<
structure Nitpick-Defs = Named-Thms
(
  val name = nitpick-def
  val description = alternative definitions of constants as needed by Nitpick
)
structure Nitpick-Simps = Named-Thms
(
  val name = nitpick-simp
  val description = equational specification of constants as needed by Nitpick
)
structure Nitpick-Psimps = Named-Thms
(
  val name = nitpick-psimp
  val description = partial equational specification of constants as needed by Nitpick
)

```

```

structure Nitpick-Choice-Specs = Named-Thms
(
  val name = nitpick-choice-spec
  val description = choice specification of constants as needed by Nitpick
)
>>

setup <<
  Nitpick-Defs.setup
  #> Nitpick-Simps.setup
  #> Nitpick-Psimps.setup
  #> Nitpick-Choice-Specs.setup
>>

```

## 2.8 Preprocessing for the predicate compiler

```

ML <<
structure Predicate-Compile-Alternative-Defs = Named-Thms
(
  val name = code-pred-def
  val description = alternative definitions of constants for the Predicate Compiler
)
structure Predicate-Compile-Inline-Defs = Named-Thms
(
  val name = code-pred-inline
  val description = inlining definitions for the Predicate Compiler
)
structure Predicate-Compile-Simps = Named-Thms
(
  val name = code-pred-simp
  val description = simplification rules for the optimisations in the Predicate Compiler
)
>>

setup <<
  Predicate-Compile-Alternative-Defs.setup
  #> Predicate-Compile-Inline-Defs.setup
  #> Predicate-Compile-Simps.setup
>>

```

## 2.9 Legacy tactics and ML bindings

```

ML <<
fun strip-tac i = REPEAT (resolve-tac [impI, allI] i);

(* combination of (spec RS spec RS ...(j times) ... spec RS mp) *)
local
  fun wrong-prem (Const (@{const-name All}, -) $ Abs (-, -, t)) = wrong-prem t
    | wrong-prem (Bound -) = true
    | wrong-prem - = false;

```



```

    val filter-right = filter (not o wrong-prem o HOLogic.dest-Trueprop o hd o Thm.premsof);
  in
    fun smp i = funpow i (fn m => filter-right ([spec] RL m)) ([mp]);
    fun smp-tac j = EVERY'[dresolve-tac (smp j), atac];
  end;

  val all-conj-distrib = thm all-conj-distrib;
  val all-simps = thms all-simps;
  val atomize-not = thm atomize-not;
  val case-split = thm case-split;
  val cases-simp = thm cases-simp;
  val choice-eq = thm choice-eq;
  val cong = thm cong;
  val conj-comms = thms conj-comms;
  val conj-cong = thm conj-cong;
  val de-Morgan-conj = thm de-Morgan-conj;
  val de-Morgan-disj = thm de-Morgan-disj;
  val disj-assoc = thm disj-assoc;
  val disj-comms = thms disj-comms;
  val disj-cong = thm disj-cong;
  val eq-ac = thms eq-ac;
  val eq-cong2 = thm eq-cong2;
  val Eq-FalseI = thm Eq-FalseI;
  val Eq-TrueI = thm Eq-TrueI;
  val Ex1-def = thm Ex1-def;
  val ex-disj-distrib = thm ex-disj-distrib;
  val ex-simps = thms ex-simps;
  val if-cancel = thm if-cancel;
  val if-eq-cancel = thm if-eq-cancel;
  val if-False = thm if-False;
  val iff-conv-conj-imp = thm iff-conv-conj-imp;
  val iff = thm iff;
  val if-splits = thms if-splits;
  val if-True = thm if-True;
  val if-weak-cong = thm if-weak-cong;
  val imp-all = thm imp-all;
  val imp-cong = thm imp-cong;
  val imp-conjL = thm imp-conjL;
  val imp-conjR = thm imp-conjR;
  val imp-conv-disj = thm imp-conv-disj;
  val simp-implies-def = thm simp-implies-def;
  val simp-thms = thms simp-thms;
  val split-if = thm split-if;
  val the1-equality = thm the1-equality;
  val theI = thm theI;
  val theI' = thm theI';
  val True-implies-equals = thm True-implies-equals;
  val nnf-conv = Simplifier.rewrite (HOL-basic-ss addsimps simp-thms @ @{thms
    nnf-simps})

```

»

end

### 3 Orderings: Abstract orderings

```
theory Orderings
imports HOL
uses
  ~~/src/Provers/order.ML
  ~~/src/Provers/quasi.ML
begin
```

#### 3.1 Syntactic orders

```
class ord =
  fixes less-eq :: 'a ⇒ 'a ⇒ bool
  and less :: 'a ⇒ 'a ⇒ bool
begin
```

```
notation
  less-eq (op ≤) and
  less-eq ((-/ ≤ -) [51, 51] 50) and
  less (op <) and
  less ((-/ < -) [51, 51] 50)
```

```
notation (xsymbols)
  less-eq (op ≤) and
  less-eq ((-/ ≤ -) [51, 51] 50)
```

```
notation (HTML output)
  less-eq (op ≤) and
  less-eq ((-/ ≤ -) [51, 51] 50)
```

```
abbreviation (input)
  greater-eq (infix ≥ 50) where
  x ≥ y ≡ y ≤ x
```

```
notation (input)
  greater-eq (infix ≥ 50)
```

```
abbreviation (input)
  greater (infix > 50) where
  x > y ≡ y < x
```

end

### 3.2 Quasi orders

```

class preorder = ord +
  assumes less-le-not-le:  $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$ 
  and order-refl [iff]:  $x \leq x$ 
  and order-trans:  $x \leq y \implies y \leq z \implies x \leq z$ 
begin

```

Reflexivity.

```

lemma eq-refl:  $x = y \implies x \leq y$ 
  — This form is useful with the classical reasoner.
by (erule ssubst) (rule order-refl)

```

```

lemma less-irrefl [iff]:  $\neg x < x$ 
by (simp add: less-le-not-le)

```

```

lemma less-imp-le:  $x < y \implies x \leq y$ 
unfolding less-le-not-le by blast

```

Asymmetry.

```

lemma less-not-sym:  $x < y \implies \neg (y < x)$ 
by (simp add: less-le-not-le)

```

```

lemma less-asym:  $x < y \implies (\neg P \implies y < x) \implies P$ 
by (drule less-not-sym, erule contrapos-np) simp

```

Transitivity.

```

lemma less-trans:  $x < y \implies y < z \implies x < z$ 
by (auto simp add: less-le-not-le intro: order-trans)

```

```

lemma le-less-trans:  $x \leq y \implies y < z \implies x < z$ 
by (auto simp add: less-le-not-le intro: order-trans)

```

```

lemma less-le-trans:  $x < y \implies y \leq z \implies x < z$ 
by (auto simp add: less-le-not-le intro: order-trans)

```

Useful for simplification, but too risky to include by default.

```

lemma less-imp-not-less:  $x < y \implies (\neg y < x) \longleftrightarrow \text{True}$ 
by (blast elim: less-asym)

```

```

lemma less-imp-triv:  $x < y \implies (y < x \longrightarrow P) \longleftrightarrow \text{True}$ 
by (blast elim: less-asym)

```

Transitivity rules for calculational reasoning

```

lemma less-asym':  $a < b \implies b < a \implies P$ 
by (rule less-asym)

```

Dual order

```

lemma dual-preorder:
  class.preorder (op  $\geq$ ) (op  $>$ )
proof qed (auto simp add: less-le-not-le intro: order-trans)

end

```

### 3.3 Partial orders

```

class order = preorder +
  assumes antisym:  $x \leq y \implies y \leq x \implies x = y$ 
begin

```

Reflexivity.

```

lemma less-le:  $x < y \iff x \leq y \wedge x \neq y$ 
by (auto simp add: less-le-not-le intro: antisym)

```

```

lemma le-less:  $x \leq y \iff x < y \vee x = y$ 
  — NOT suitable for iff, since it can cause PROOF FAILED.
by (simp add: less-le) blast

```

```

lemma le-imp-less-or-eq:  $x \leq y \implies x < y \vee x = y$ 
unfolding less-le by blast

```

Useful for simplification, but too risky to include by default.

```

lemma less-imp-not-eq:  $x < y \implies (x = y) \iff \text{False}$ 
by auto

```

```

lemma less-imp-not-eq2:  $x < y \implies (y = x) \iff \text{False}$ 
by auto

```

Transitivity rules for calculational reasoning

```

lemma neq-le-trans:  $a \neq b \implies a \leq b \implies a < b$ 
by (simp add: less-le)

```

```

lemma le-neq-trans:  $a \leq b \implies a \neq b \implies a < b$ 
by (simp add: less-le)

```

Asymmetry.

```

lemma eq-iff:  $x = y \iff x \leq y \wedge y \leq x$ 
by (blast intro: antisym)

```

```

lemma antisym-conv:  $y \leq x \implies x \leq y \iff x = y$ 
by (blast intro: antisym)

```

```

lemma less-imp-neq:  $x < y \implies x \neq y$ 
by (erule contrapos-pn, erule subst, rule less-irrefl)

```

Least value operator

**definition** (in *ord*)

*Least* :: ( $'a \Rightarrow \text{bool}$ )  $\Rightarrow$   $'a$  (**binder** *LEAST* 10) **where**

*Least*  $P = (\text{THE } x. P\ x \wedge (\forall y. P\ y \longrightarrow x \leq y))$

**lemma** *Least-equality*:

**assumes**  $P\ x$

**and**  $\bigwedge y. P\ y \Longrightarrow x \leq y$

**shows**  $\text{Least } P = x$

**unfolding** *Least-def* **by** (rule *the-equality*)

(blast intro: *assms antisym*)+

**lemma** *LeastI2-order*:

**assumes**  $P\ x$

**and**  $\bigwedge y. P\ y \Longrightarrow x \leq y$

**and**  $\bigwedge x. P\ x \Longrightarrow \forall y. P\ y \longrightarrow x \leq y \Longrightarrow Q\ x$

**shows**  $Q\ (\text{Least } P)$

**unfolding** *Least-def* **by** (rule *theI2*)

(blast intro: *assms antisym*)+

Dual order

**lemma** *dual-order*:

*class.order* ( $op \geq$ ) ( $op >$ )

**by** (*intro-locales*, rule *dual-preorder*) (*unfold-locales*, rule *antisym*)

**end**

### 3.4 Linear (total) orders

**class** *linorder* = *order* +

**assumes** *linear*:  $x \leq y \vee y \leq x$

**begin**

**lemma** *less-linear*:  $x < y \vee x = y \vee y < x$

**unfolding** *less-le* **using** *less-le linear* **by** *blast*

**lemma** *le-less-linear*:  $x \leq y \vee y < x$

**by** (*simp add: le-less less-linear*)

**lemma** *le-cases* [*case-names le ge*]:

$(x \leq y \Longrightarrow P) \Longrightarrow (y \leq x \Longrightarrow P) \Longrightarrow P$

**using** *linear* **by** *blast*

**lemma** *linorder-cases* [*case-names less equal greater*]:

$(x < y \Longrightarrow P) \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow (y < x \Longrightarrow P) \Longrightarrow P$

**using** *less-linear* **by** *blast*

**lemma** *not-less*:  $\neg x < y \longleftrightarrow y \leq x$

**apply** (*simp add: less-le*)

**using** *linear* **apply** (*blast intro: antisym*)

done

**lemma** *not-less-iff-gr-or-eq*:  
 $\neg(x < y) \longleftrightarrow (x > y \mid x = y)$   
**apply** (*simp add: not-less le-less*)  
**apply** *blast*  
**done**

**lemma** *not-le*:  $\neg x \leq y \longleftrightarrow y < x$   
**apply** (*simp add: less-le*)  
**using** *linear* **apply** (*blast intro: antisym*)  
**done**

**lemma** *neq-iff*:  $x \neq y \longleftrightarrow x < y \vee y < x$   
**by** (*cut-tac x = x and y = y in less-linear, auto*)

**lemma** *neqE*:  $x \neq y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$   
**by** (*simp add: neq-iff*) *blast*

**lemma** *antisym-conv1*:  $\neg x < y \Longrightarrow x \leq y \longleftrightarrow x = y$   
**by** (*blast intro: antisym dest: not-less [THEN iffD1]*)

**lemma** *antisym-conv2*:  $x \leq y \Longrightarrow \neg x < y \longleftrightarrow x = y$   
**by** (*blast intro: antisym dest: not-less [THEN iffD1]*)

**lemma** *antisym-conv3*:  $\neg y < x \Longrightarrow \neg x < y \longleftrightarrow x = y$   
**by** (*blast intro: antisym dest: not-less [THEN iffD1]*)

**lemma** *leI*:  $\neg x < y \Longrightarrow y \leq x$   
**unfolding** *not-less* .

**lemma** *leD*:  $y \leq x \Longrightarrow \neg x < y$   
**unfolding** *not-less* .

**lemma** *not-leE*:  $\neg y \leq x \Longrightarrow x < y$   
**unfolding** *not-le* .

Dual order

**lemma** *dual-linorder*:  
*class.linorder* (*op*  $\geq$ ) (*op*  $>$ )  
**by** (*rule class.linorder.intro, rule dual-order*) (*unfold-locales, rule linear*)

min/max

**definition** (*in ord*) *min* ::  $'a \Rightarrow 'a \Rightarrow 'a$  **where**  
 $[code\ del]: min\ a\ b = (if\ a \leq b\ then\ a\ else\ b)$

**definition** (*in ord*) *max* ::  $'a \Rightarrow 'a \Rightarrow 'a$  **where**  
 $[code\ del]: max\ a\ b = (if\ a \leq b\ then\ b\ else\ a)$

**lemma** *min-le-iff-disj*:

$$\min x y \leq z \iff x \leq z \vee y \leq z$$

**unfolding** *min-def* **using** *linear* **by** (*auto intro: order-trans*)

**lemma** *le-max-iff-disj*:

$$z \leq \max x y \iff z \leq x \vee z \leq y$$

**unfolding** *max-def* **using** *linear* **by** (*auto intro: order-trans*)

**lemma** *min-less-iff-disj*:

$$\min x y < z \iff x < z \vee y < z$$

**unfolding** *min-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *less-max-iff-disj*:

$$z < \max x y \iff z < x \vee z < y$$

**unfolding** *max-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *min-less-iff-conj* [*simp*]:

$$z < \min x y \iff z < x \wedge z < y$$

**unfolding** *min-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *max-less-iff-conj* [*simp*]:

$$\max x y < z \iff x < z \wedge y < z$$

**unfolding** *max-def le-less* **using** *less-linear* **by** (*auto intro: less-trans*)

**lemma** *split-min* [*no-atp*]:

$$P (\min i j) \iff (i \leq j \longrightarrow P i) \wedge (\neg i \leq j \longrightarrow P j)$$

**by** (*simp add: min-def*)

**lemma** *split-max* [*no-atp*]:

$$P (\max i j) \iff (i \leq j \longrightarrow P j) \wedge (\neg i \leq j \longrightarrow P i)$$

**by** (*simp add: max-def*)

**end**

Explicit dictionaries for code generation

**lemma** *min-ord-min* [*code, code-unfold, code-inline del*]:

$$\min = \text{ord.min } (op \leq)$$

**by** (*rule ext*)+ (*simp add: min-def ord.min-def*)

**declare** *ord.min-def* [*code*]

**lemma** *max-ord-max* [*code, code-unfold, code-inline del*]:

$$\max = \text{ord.max } (op \leq)$$

**by** (*rule ext*)+ (*simp add: max-def ord.max-def*)

**declare** *ord.max-def* [*code*]

### 3.5 Reasoning tools setup

ML  $\ll$

```
signature ORDERS =
sig
  val print-structures: Proof.context -> unit
  val setup: theory -> theory
  val order-tac: Proof.context -> thm list -> int -> tactic
end;

structure Orders: ORDERS =
struct

(** Theory and context data **)

fun struct-eq ((s1: string, ts1), (s2, ts2)) =
  (s1 = s2) andalso eq-list (op aconv) (ts1, ts2);

structure Data = Generic-Data
(
  type T = ((string * term list) * Order-Tac.less-arith) list;
  (* Order structures:
     identifier of the structure, list of operations and record of theorems
     needed to set up the transitivity reasoner,
     identifier and operations identify the structure uniquely. *)
  val empty = [];
  val extend = I;
  fun merge data = AList.join struct-eq (K fst) data;
);

fun print-structures ctxt =
let
  val structs = Data.get (Context.Proof ctxt);
  fun pretty-term t = Pretty.block
    [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
     Pretty.str ::, Pretty.brk 1,
     Pretty.quote (Syntax.pretty-typ ctxt (type-of t))];
  fun pretty-struct ((s, ts), _) = Pretty.block
    [Pretty.str s, Pretty.str :, Pretty.brk 1,
     Pretty.enclose ( ) (Pretty.breaks (map pretty-term ts))];
in
  Pretty.writeln (Pretty.big-list Order structures: (map pretty-struct structs))
end;

(** Method **)

fun struct-tac ((s, [eq, le, less]), thms) ctxt prems =
let
```



```

fun decomp thy (@{const Trueprop} $ t) =
  let
    fun excluded t =
      (* exclude numeric types: linear arithmetic subsumes transitivity *)
      let val T = type-of t
      in
        T = HOLogic.natT orelse T = HOLogic.intT orelse T = HOLogic.realT
      end;
    fun rel (bin-op $ t1 $ t2) =
      if excluded t1 then NONE
      else if Pattern.matches thy (eq, bin-op) then SOME (t1, =, t2)
      else if Pattern.matches thy (le, bin-op) then SOME (t1, <=, t2)
      else if Pattern.matches thy (less, bin-op) then SOME (t1, <, t2)
      else NONE
      | rel - = NONE;
    fun dec (Const (@{const-name Not}, -) $ t) = (case rel t
      of NONE => NONE
      | SOME (t1, rel, t2) => SOME (t1, ~ ^ rel, t2))
      | dec x = rel x;
    in dec t end
  | decomp thy - = NONE;
  in
    case s of
      order => Order-Tac.partial-tac decomp thms ctxt prems
    | linorder => Order-Tac.linear-tac decomp thms ctxt prems
    | - => error (Unknown kind of order ' ^ s ^ ' encountered in transitivity
      reasoner.)
    end

fun order-tac ctxt prems =
  FIRST' (map (fn s => CHANGED o struct-tac s ctxt prems) (Data.get (Context.Proof
    ctxt)));

(** Attribute **)

fun add-struct-thm s tag =
  Thm.declaration-attribute
    (fn thm => Data.map (AList.map-default struct-eq (s, Order-Tac.empty TrueI)
      (Order-Tac.update tag thm)));
fun del-struct s =
  Thm.declaration-attribute
    (fn - => Data.map (AList.delete struct-eq s));

val attrib-setup =
  Attrib.setup @{binding order}
    (Scan.lift ((Args.add -- Args.name >> (fn (-, s) => SOME s) || Args.del
    >> K NONE) --|
    Args.colon (* FIXME || Scan.succeed true *) ) -- Scan.lift Args.name --

```

```

    Scan.repeat Args.term
    >> (fn ((SOME tag, n), ts) => add-struct-thm (n, ts) tag
        | ((NONE, n), ts) => del-struct (n, ts)))
    theorems controlling transitivity reasoner;

(** Diagnostic command **)

val - =
  Outer-Syntax.improper-command print-orders
  print order structures available to transitivity reasoner Keyword.diag
  (Scan.succeed (Toplevel.no-timing o Toplevel.unknown-context o
    Toplevel.keep (print-structures o Toplevel.context-of)));

(** Setup **)

val setup =
  Method.setup @{binding order} (Scan.succeed (fn ctxt => SIMPLE-METHOD'
    (order-tac ctxt [])))
  transitivity reasoner #>
  attrib-setup;

end;

>>

setup Orders.setup

Declarations to set up transitivity reasoner of partial and linear orders.

context order
begin

declare less-irrefl [THEN notE, order add less-reflE: order op = :: 'a ⇒ 'a ⇒
  bool op <= op <]

declare order-refl [order add le-refl: order op = :: 'a => 'a => bool op <= op
  <]

declare less-imp-le [order add less-imp-le: order op = :: 'a => 'a => bool op <=
  op <]

declare antisym [order add eqI: order op = :: 'a => 'a => bool op <= op <]

declare eq-refl [order add eqD1: order op = :: 'a => 'a => bool op <= op <]

declare sym [THEN eq-refl, order add eqD2: order op = :: 'a => 'a => bool op

```

$\leq op <$ ]

**declare** *less-trans* [order add *less-trans*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *less-le-trans* [order add *less-le-trans*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *le-less-trans* [order add *le-less-trans*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *order-trans* [order add *le-trans*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *le-neq-trans* [order add *le-neq-trans*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *neq-le-trans* [order add *neq-le-trans*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *less-imp-neq* [order add *less-imp-neq*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *eq-neq-eq-imp-neq* [order add *eq-neq-eq-imp-neq*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *not-sym* [order add *not-sym*: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**end**

**context** *linorder*

**begin**

**declare** [[order del: order  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]]

**declare** *less-irrefl* [THEN *notE*, order add *less-reflE*: *linorder*  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *order-refl* [order add *le-refl*: *linorder*  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *less-imp-le* [order add *less-imp-le*: *linorder*  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *not-less* [THEN *iffD2*, order add *not-lessI*: *linorder*  $op = :: 'a \Rightarrow 'a \Rightarrow bool\ op \leq op <$ ]

**declare** *not-le* [THEN *iffD2*, order add *not-leI*: *linorder*  $op = :: 'a \Rightarrow 'a \Rightarrow bool$

*op* <= *op* <]

**declare** *not-less* [THEN *iffD1*, order add *not-lessD*: *linorder op* = :: '*a* => '*a* =>  
bool *op* <= *op* <]

**declare** *not-le* [THEN *iffD1*, order add *not-leD*: *linorder op* = :: '*a* => '*a* =>  
bool *op* <= *op* <]

**declare** *antisym* [order add *eqI*: *linorder op* = :: '*a* => '*a* => bool *op* <= *op* <]

**declare** *eq-refl* [order add *eqD1*: *linorder op* = :: '*a* => '*a* => bool *op* <= *op* <]

**declare** *sym* [THEN *eq-refl*, order add *eqD2*: *linorder op* = :: '*a* => '*a* => bool  
*op* <= *op* <]

**declare** *less-trans* [order add *less-trans*: *linorder op* = :: '*a* => '*a* => bool *op* <=  
*op* <]

**declare** *less-le-trans* [order add *less-le-trans*: *linorder op* = :: '*a* => '*a* => bool  
*op* <= *op* <]

**declare** *le-less-trans* [order add *le-less-trans*: *linorder op* = :: '*a* => '*a* => bool  
*op* <= *op* <]

**declare** *order-trans* [order add *le-trans*: *linorder op* = :: '*a* => '*a* => bool *op* <=  
*op* <]

**declare** *le-neq-trans* [order add *le-neq-trans*: *linorder op* = :: '*a* => '*a* => bool *op*  
<= *op* <]

**declare** *neq-le-trans* [order add *neq-le-trans*: *linorder op* = :: '*a* => '*a* => bool *op*  
<= *op* <]

**declare** *less-imp-neq* [order add *less-imp-neq*: *linorder op* = :: '*a* => '*a* => bool  
*op* <= *op* <]

**declare** *eq-neq-eq-imp-neq* [order add *eq-neq-eq-imp-neq*: *linorder op* = :: '*a* => '*a*  
=> bool *op* <= *op* <]

**declare** *not-sym* [order add *not-sym*: *linorder op* = :: '*a* => '*a* => bool *op* <=  
*op* <]

**end**

**setup** <<  
*let*

*fun* *prp t thm* = (#*prop* (*rep-thm thm*) = *t*);

```

fun prove-antisym-le sg ss ((le as Const(-,T)) $ r $ s) =
  let val prems = prems-of-ss ss;
      val less = Const (@{const-name less}, T);
      val t = HOLogic.mk-Trueprop(le $ s $ r);
  in case find-first (prp t) prems of
      NONE =>
        let val t = HOLogic.mk-Trueprop(HOLogic.Not $ (less $ r $ s))
        in case find-first (prp t) prems of
            NONE => NONE
          | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv1}))
        end
      | SOME thm => SOME(mk-meta-eq(thm RS @{thm order-class.antisym-conv}))
  end
end
handle THM - => NONE;

fun prove-antisym-less sg ss (NotC $ ((less as Const(-,T)) $ r $ s)) =
  let val prems = prems-of-ss ss;
      val le = Const (@{const-name less-eq}, T);
      val t = HOLogic.mk-Trueprop(le $ r $ s);
  in case find-first (prp t) prems of
      NONE =>
        let val t = HOLogic.mk-Trueprop(NotC $ (less $ s $ r))
        in case find-first (prp t) prems of
            NONE => NONE
          | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv3}))
        end
      | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv2}))
  end
end
handle THM - => NONE;

fun add-simprocs procs thy =
  Simplifier.map-simpset (fn ss => ss
    addsimprocs (map (fn (name, raw-ts, proc) =>
      Simplifier.simproc thy name raw-ts proc) procs)) thy;
fun add-solver name tac =
  Simplifier.map-simpset (fn ss => ss addSolver
    mk-solver' name (fn ss => tac (Simplifier.the-context ss) (Simplifier.prems-of-ss
ss)));

in
  add-simprocs [
    (antisym le, [(x::'a::order) <= y], prove-antisym-le),
    (antisym less, [~ (x::'a::linorder) < y], prove-antisym-less)
  ]
#> add-solver Transitivity Orders.order-tac
(* Adding the transitivity reasoners also as safe solvers showed a slight
speed up, but the reasoning strength appears to be not higher (at least
no breaking of additional proofs in the entire HOL distribution, as

```

of 5 March 2004, was observed). \*)  
end  
»

### 3.6 Bounded quantifiers

#### syntax

-All-less :: [idt, 'a, bool] => bool (( $\exists ALL$  -<./ -) [0, 0, 10] 10)  
-Ex-less :: [idt, 'a, bool] => bool (( $\exists EX$  -<./ -) [0, 0, 10] 10)  
-All-less-eq :: [idt, 'a, bool] => bool (( $\exists ALL$  -<=./ -) [0, 0, 10] 10)  
-Ex-less-eq :: [idt, 'a, bool] => bool (( $\exists EX$  -<=./ -) [0, 0, 10] 10)  
  
-All-greater :: [idt, 'a, bool] => bool (( $\exists ALL$  ->./ -) [0, 0, 10] 10)  
-Ex-greater :: [idt, 'a, bool] => bool (( $\exists EX$  ->./ -) [0, 0, 10] 10)  
-All-greater-eq :: [idt, 'a, bool] => bool (( $\exists ALL$  ->=./ -) [0, 0, 10] 10)  
-Ex-greater-eq :: [idt, 'a, bool] => bool (( $\exists EX$  ->=./ -) [0, 0, 10] 10)

#### syntax (xsymbols)

-All-less :: [idt, 'a, bool] => bool (( $\exists \forall$  -<./ -) [0, 0, 10] 10)  
-Ex-less :: [idt, 'a, bool] => bool (( $\exists \exists$  -<./ -) [0, 0, 10] 10)  
-All-less-eq :: [idt, 'a, bool] => bool (( $\exists \forall$  -<=./ -) [0, 0, 10] 10)  
-Ex-less-eq :: [idt, 'a, bool] => bool (( $\exists \exists$  -<=./ -) [0, 0, 10] 10)  
  
-All-greater :: [idt, 'a, bool] => bool (( $\exists \forall$  ->./ -) [0, 0, 10] 10)  
-Ex-greater :: [idt, 'a, bool] => bool (( $\exists \exists$  ->./ -) [0, 0, 10] 10)  
-All-greater-eq :: [idt, 'a, bool] => bool (( $\exists \forall$  ->=./ -) [0, 0, 10] 10)  
-Ex-greater-eq :: [idt, 'a, bool] => bool (( $\exists \exists$  ->=./ -) [0, 0, 10] 10)

#### syntax (HOL)

-All-less :: [idt, 'a, bool] => bool (( $\exists !$  -<./ -) [0, 0, 10] 10)  
-Ex-less :: [idt, 'a, bool] => bool (( $\exists ?$  -<./ -) [0, 0, 10] 10)  
-All-less-eq :: [idt, 'a, bool] => bool (( $\exists !$  -<=./ -) [0, 0, 10] 10)  
-Ex-less-eq :: [idt, 'a, bool] => bool (( $\exists ?$  -<=./ -) [0, 0, 10] 10)

#### syntax (HTML output)

-All-less :: [idt, 'a, bool] => bool (( $\exists \forall$  -<./ -) [0, 0, 10] 10)  
-Ex-less :: [idt, 'a, bool] => bool (( $\exists \exists$  -<./ -) [0, 0, 10] 10)  
-All-less-eq :: [idt, 'a, bool] => bool (( $\exists \forall$  -<=./ -) [0, 0, 10] 10)  
-Ex-less-eq :: [idt, 'a, bool] => bool (( $\exists \exists$  -<=./ -) [0, 0, 10] 10)  
  
-All-greater :: [idt, 'a, bool] => bool (( $\exists \forall$  ->./ -) [0, 0, 10] 10)  
-Ex-greater :: [idt, 'a, bool] => bool (( $\exists \exists$  ->./ -) [0, 0, 10] 10)  
-All-greater-eq :: [idt, 'a, bool] => bool (( $\exists \forall$  ->=./ -) [0, 0, 10] 10)  
-Ex-greater-eq :: [idt, 'a, bool] => bool (( $\exists \exists$  ->=./ -) [0, 0, 10] 10)

#### translations

$ALL\ x < y. P \Rightarrow ALL\ x. x < y \longrightarrow P$   
 $EX\ x < y. P \Rightarrow EX\ x. x < y \wedge P$   
 $ALL\ x <= y. P \Rightarrow ALL\ x. x <= y \longrightarrow P$

$$\begin{aligned}
EX\ x <= y. P & \Rightarrow EX\ x. x <= y \wedge P \\
ALL\ x > y. P & \Rightarrow ALL\ x. x > y \longrightarrow P \\
EX\ x > y. P & \Rightarrow EX\ x. x > y \wedge P \\
ALL\ x >= y. P & \Rightarrow ALL\ x. x >= y \longrightarrow P \\
EX\ x >= y. P & \Rightarrow EX\ x. x >= y \wedge P
\end{aligned}$$

**print-translation**  $\ll$

*let*

```

val All-binder = Syntax.binder-name @{const-syntax All};
val Ex-binder = Syntax.binder-name @{const-syntax Ex};
val impl = @{const-syntax op -->};
val conj = @{const-syntax op &};
val less = @{const-syntax less};
val less-eq = @{const-syntax less-eq};

```

*val trans =*

```

[((All-binder, impl, less),
  (@{syntax-const -All-less}, @{syntax-const -All-greater})),
 ((All-binder, impl, less-eq),
  (@{syntax-const -All-less-eq}, @{syntax-const -All-greater-eq})),
 ((Ex-binder, conj, less),
  (@{syntax-const -Ex-less}, @{syntax-const -Ex-greater})),
 ((Ex-binder, conj, less-eq),
  (@{syntax-const -Ex-less-eq}, @{syntax-const -Ex-greater-eq}))];

```

*fun matches-bound v t =*

```

(case t of
  Const (@{syntax-const -bound}, -) $ Free (v', -) => v = v'
| - => false);

```

*fun contains-var v = Term.exists-subterm (fn Free (x, -) => x = v | - => false);*

*fun mk v c n P = Syntax.const c \$ Syntax.mark-bound v \$ n \$ P;*

*fun tr' q = (q,*

```

  fn [Const (@{syntax-const -bound}, -) $ Free (v, -),
    Const (c, -) $ (Const (d, -) $ t $ u) $ P] =>
    (case AList.lookup (op =) trans (q, c, d) of
      NONE => raise Match
    | SOME (l, g) =>
      if matches-bound v t andalso not (contains-var v u) then mk v l u P
      else if matches-bound v u andalso not (contains-var v t) then mk v g t P
      else raise Match)
  | - => raise Match);

```

*in [tr' All-binder, tr' Ex-binder] end*

$\gg$

### 3.7 Transitivity reasoning

**context** *ord*

**begin**

**lemma** *ord-le-eq-trans*:  $a \leq b \implies b = c \implies a \leq c$   
**by** (*rule subst*)

**lemma** *ord-eq-le-trans*:  $a = b \implies b \leq c \implies a \leq c$   
**by** (*rule ssubst*)

**lemma** *ord-less-eq-trans*:  $a < b \implies b = c \implies a < c$   
**by** (*rule subst*)

**lemma** *ord-eq-less-trans*:  $a = b \implies b < c \implies a < c$   
**by** (*rule ssubst*)

**end**

**lemma** *order-less-subst2*:  $(a::'a::order) < b \implies f\ b < (c::'c::order) \implies$   
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies f\ a < c$   
**proof** –  
 assume  $r: !!x\ y. x < y \implies f\ x < f\ y$   
 assume  $a < b$  **hence**  $f\ a < f\ b$  **by** (*rule r*)  
 also assume  $f\ b < c$   
 finally (*less-trans*) **show** *?thesis* .  
**qed**

**lemma** *order-less-subst1*:  $(a::'a::order) < f\ b \implies (b::'b::order) < c \implies$   
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies a < f\ c$   
**proof** –  
 assume  $r: !!x\ y. x < y \implies f\ x < f\ y$   
 assume  $a < f\ b$   
 also assume  $b < c$  **hence**  $f\ b < f\ c$  **by** (*rule r*)  
 finally (*less-trans*) **show** *?thesis* .  
**qed**

**lemma** *order-le-less-subst2*:  $(a::'a::order) <= b \implies f\ b < (c::'c::order) \implies$   
 $(!!x\ y. x <= y \implies f\ x <= f\ y) \implies f\ a < c$   
**proof** –  
 assume  $r: !!x\ y. x <= y \implies f\ x <= f\ y$   
 assume  $a <= b$  **hence**  $f\ a <= f\ b$  **by** (*rule r*)  
 also assume  $f\ b < c$   
 finally (*le-less-trans*) **show** *?thesis* .  
**qed**

**lemma** *order-le-less-subst1*:  $(a::'a::order) <= f\ b \implies (b::'b::order) < c \implies$   
 $(!!x\ y. x <= y \implies f\ x <= f\ y) \implies a < f\ c$   
**proof** –  
 assume  $r: !!x\ y. x <= y \implies f\ x <= f\ y$   
 assume  $a <= f\ b$   
 also assume  $b < c$  **hence**  $f\ b < f\ c$  **by** (*rule r*)  
 finally (*le-less-trans*) **show** *?thesis* .



qed

**lemma** *order-less-le-subst2*:  $(a::'a::order) < b ==> f\ b <= (c::'c::order) ==>$   
 $(!!x\ y. x < y ==> f\ x < f\ y) ==> f\ a < c$

**proof** –

assume  $r: !!x\ y. x < y ==> f\ x < f\ y$   
 assume  $a < b$  hence  $f\ a < f\ b$  by (rule  $r$ )  
 also assume  $f\ b <= c$   
 finally (*less-le-trans*) show ?thesis .

qed

**lemma** *order-less-le-subst1*:  $(a::'a::order) < f\ b ==> (b::'b::order) <= c ==>$   
 $(!!x\ y. x <= y ==> f\ x <= f\ y) ==> a < f\ c$

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$   
 assume  $a < f\ b$   
 also assume  $b <= c$  hence  $f\ b <= f\ c$  by (rule  $r$ )  
 finally (*less-le-trans*) show ?thesis .

qed

**lemma** *order-subst1*:  $(a::'a::order) <= f\ b ==> (b::'b::order) <= c ==>$   
 $(!!x\ y. x <= y ==> f\ x <= f\ y) ==> a <= f\ c$

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$   
 assume  $a <= f\ b$   
 also assume  $b <= c$  hence  $f\ b <= f\ c$  by (rule  $r$ )  
 finally (*order-trans*) show ?thesis .

qed

**lemma** *order-subst2*:  $(a::'a::order) <= b ==> f\ b <= (c::'c::order) ==>$   
 $(!!x\ y. x <= y ==> f\ x <= f\ y) ==> f\ a <= c$

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$   
 assume  $a <= b$  hence  $f\ a <= f\ b$  by (rule  $r$ )  
 also assume  $f\ b <= c$   
 finally (*order-trans*) show ?thesis .

qed

**lemma** *ord-le-eq-subst*:  $a <= b ==> f\ b = c ==>$   
 $(!!x\ y. x <= y ==> f\ x <= f\ y) ==> f\ a <= c$

**proof** –

assume  $r: !!x\ y. x <= y ==> f\ x <= f\ y$   
 assume  $a <= b$  hence  $f\ a <= f\ b$  by (rule  $r$ )  
 also assume  $f\ b = c$   
 finally (*ord-le-eq-trans*) show ?thesis .

qed

**lemma** *ord-eq-le-subst*:  $a = f\ b ==> b <= c ==>$   
 $(!!x\ y. x <= y ==> f\ x <= f\ y) ==> a <= f\ c$

**proof** –

**assume**  $r: !!x\ y. x \leq y \implies f\ x \leq f\ y$   
  **assume**  $a = f\ b$   
  **also assume**  $b \leq c$  **hence**  $f\ b \leq f\ c$  **by** (rule  $r$ )  
  **finally** (ord-eq-le-trans) **show** ?thesis .

**qed**

**lemma** ord-less-eq-subst:  $a < b \implies f\ b = c \implies$   
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies f\ a < c$

**proof** –

**assume**  $r: !!x\ y. x < y \implies f\ x < f\ y$   
  **assume**  $a < b$  **hence**  $f\ a < f\ b$  **by** (rule  $r$ )  
  **also assume**  $f\ b = c$   
  **finally** (ord-less-eq-trans) **show** ?thesis .

**qed**

**lemma** ord-eq-less-subst:  $a = f\ b \implies b < c \implies$   
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies a < f\ c$

**proof** –

**assume**  $r: !!x\ y. x < y \implies f\ x < f\ y$   
  **assume**  $a = f\ b$   
  **also assume**  $b < c$  **hence**  $f\ b < f\ c$  **by** (rule  $r$ )  
  **finally** (ord-eq-less-trans) **show** ?thesis .

**qed**

Note that this list of rules is in reverse order of priorities.

**lemmas** [trans] =  
  order-less-subst2  
  order-less-subst1  
  order-le-less-subst2  
  order-le-less-subst1  
  order-less-le-subst2  
  order-less-le-subst1  
  order-subst2  
  order-subst1  
  ord-le-eq-subst  
  ord-eq-le-subst  
  ord-less-eq-subst  
  ord-eq-less-subst  
  forw-subst  
  back-subst  
  rev-mp  
  mp

**lemmas** (in order) [trans] =  
  neq-le-trans  
  le-neq-trans

**lemmas** (in preorder) [trans] =

*less-trans*  
*less-asym'*  
*le-less-trans*  
*less-le-trans*  
*order-trans*

**lemmas** (in *order*) [*trans*] =  
*antisym*

**lemmas** (in *ord*) [*trans*] =  
*ord-le-eq-trans*  
*ord-eq-le-trans*  
*ord-less-eq-trans*  
*ord-eq-less-trans*

**lemmas** [*trans*] =  
*trans*

**lemmas** *order-trans-rules* =  
*order-less-subst2*  
*order-less-subst1*  
*order-le-less-subst2*  
*order-le-less-subst1*  
*order-less-le-subst2*  
*order-less-le-subst1*  
*order-subst2*  
*order-subst1*  
*ord-le-eq-subst*  
*ord-eq-le-subst*  
*ord-less-eq-subst*  
*ord-eq-less-subst*  
*forw-subst*  
*back-subst*  
*rev-mp*  
*mp*  
*neq-le-trans*  
*le-neq-trans*  
*less-trans*  
*less-asym'*  
*le-less-trans*  
*less-le-trans*  
*order-trans*  
*antisym*  
*ord-le-eq-trans*  
*ord-eq-le-trans*  
*ord-less-eq-trans*  
*ord-eq-less-trans*  
*trans*

These support proving chains of decreasing inequalities  $a \leq b \leq c \dots$  in

Isar proofs.

**lemma** *xt1*:

```

  a = b ==> b > c ==> a > c
  a > b ==> b = c ==> a > c
  a = b ==> b >= c ==> a >= c
  a >= b ==> b = c ==> a >= c
  (x::'a::order) >= y ==> y >= x ==> x = y
  (x::'a::order) >= y ==> y >= z ==> x >= z
  (x::'a::order) > y ==> y >= z ==> x > z
  (x::'a::order) >= y ==> y > z ==> x > z
  (a::'a::order) > b ==> b > a ==> P
  (x::'a::order) > y ==> y > z ==> x > z
  (a::'a::order) >= b ==> a ~ b ==> a > b
  (a::'a::order) ~ b ==> a >= b ==> a > b
  a = f b ==> b > c ==> (!!x y. x > y ==> f x > f y) ==> a > f c
  a > b ==> f b = c ==> (!!x y. x > y ==> f x > f y) ==> f a > c
  a = f b ==> b >= c ==> (!!x y. x >= y ==> f x >= f y) ==> a >= f c
  a >= b ==> f b = c ==> (!!x y. x >= y ==> f x >= f y) ==> f a >= c
by auto

```

**lemma** *xt2*:

```

  (a::'a::order) >= f b ==> b >= c ==> (!!x y. x >= y ==> f x >= f y) ==>
  a >= f c
by (subgoal-tac f b >= f c, force, force)

```

**lemma** *xt3*: (a::'a::order) >= b ==> (f b::'b::order) >= c ==>

```

  (!!x y. x >= y ==> f x >= f y) ==> f a >= c
by (subgoal-tac f a >= f b, force, force)

```

**lemma** *xt4*: (a::'a::order) > f b ==> (b::'b::order) >= c ==>

```

  (!!x y. x >= y ==> f x >= f y) ==> a > f c
by (subgoal-tac f b >= f c, force, force)

```

**lemma** *xt5*: (a::'a::order) > b ==> (f b::'b::order) >= c ==>

```

  (!!x y. x > y ==> f x > f y) ==> f a > c
by (subgoal-tac f a > f b, force, force)

```

**lemma** *xt6*: (a::'a::order) >= f b ==> b > c ==>

```

  (!!x y. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f b > f c, force, force)

```

**lemma** *xt7*: (a::'a::order) >= b ==> (f b::'b::order) > c ==>

```

  (!!x y. x >= y ==> f x >= f y) ==> f a > c
by (subgoal-tac f a >= f b, force, force)

```

**lemma** *xt8*: (a::'a::order) > f b ==> (b::'b::order) > c ==>

```

  (!!x y. x > y ==> f x > f y) ==> a > f c
by (subgoal-tac f b > f c, force, force)

```

**lemma** *xt9*:  $(a :: 'a :: \text{order}) > b ==> (f b :: 'b :: \text{order}) > c ==>$   
 $(!!x y. x > y ==> f x > f y) ==> f a > c$   
**by** (*subgoal-tac*  $f a > f b$ , *force*, *force*)

**lemmas** *xtrans* = *xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9*

### 3.8 Monotonicity, least value operator and min/max

**context** *order*  
**begin**

**definition** *mono* ::  $('a \Rightarrow 'b :: \text{order}) \Rightarrow \text{bool}$  **where**  
 $\text{mono } f \longleftrightarrow (\forall x y. x \leq y \longrightarrow f x \leq f y)$

**lemma** *monoI* [*intro?*]:  
**fixes**  $f :: 'a \Rightarrow 'b :: \text{order}$   
**shows**  $(\bigwedge x y. x \leq y \Longrightarrow f x \leq f y) \Longrightarrow \text{mono } f$   
**unfolding** *mono-def* **by** *iprover*

**lemma** *monoD* [*dest?*]:  
**fixes**  $f :: 'a \Rightarrow 'b :: \text{order}$   
**shows**  $\text{mono } f \Longrightarrow x \leq y \Longrightarrow f x \leq f y$   
**unfolding** *mono-def* **by** *iprover*

**definition** *strict-mono* ::  $('a \Rightarrow 'b :: \text{order}) \Rightarrow \text{bool}$  **where**  
 $\text{strict-mono } f \longleftrightarrow (\forall x y. x < y \longrightarrow f x < f y)$

**lemma** *strict-monoI* [*intro?*]:  
**assumes**  $\bigwedge x y. x < y \Longrightarrow f x < f y$   
**shows** *strict-mono*  $f$   
**using** *assms* **unfolding** *strict-mono-def* **by** *auto*

**lemma** *strict-monoD* [*dest?*]:  
 $\text{strict-mono } f \Longrightarrow x < y \Longrightarrow f x < f y$   
**unfolding** *strict-mono-def* **by** *auto*

**lemma** *strict-mono-mono* [*dest?*]:

**assumes** *strict-mono*  $f$   
**shows** *mono*  $f$

**proof** (*rule monoI*)

**fix**  $x y$

**assume**  $x \leq y$

**show**  $f x \leq f y$

**proof** (*cases*  $x = y$ )

**case** *True* **then show** *?thesis* **by** *simp*

**next**

**case** *False* **with**  $\langle x \leq y \rangle$  **have**  $x < y$  **by** *simp*

**with** *assms* *strict-monoD* **have**  $f x < f y$  **by** *auto*

**then show** *?thesis* **by** *simp*

qed  
qed

end

context *linorder*  
begin

lemma *strict-mono-eq*:  
  assumes *strict-mono* *f*  
  shows  $f\ x = f\ y \longleftrightarrow x = y$   
proof  
  assume  $f\ x = f\ y$   
  show  $x = y$  **proof** (*cases*  $x\ y$  rule: *linorder-cases*)  
    case *less* with *assms* *strict-monoD* have  $f\ x < f\ y$  **by** *auto*  
    with  $\langle f\ x = f\ y \rangle$  **show** *?thesis* **by** *simp*  
  next  
    case *equal* **then** **show** *?thesis* .  
  next  
    case *greater* with *assms* *strict-monoD* have  $f\ y < f\ x$  **by** *auto*  
    with  $\langle f\ x = f\ y \rangle$  **show** *?thesis* **by** *simp*  
  qed  
qed *simp*

lemma *strict-mono-less-eq*:  
  assumes *strict-mono* *f*  
  shows  $f\ x \leq f\ y \longleftrightarrow x \leq y$   
proof  
  assume  $x \leq y$   
  with *assms* *strict-mono-mono* *monoD* **show**  $f\ x \leq f\ y$  **by** *auto*  
next  
  assume  $f\ x \leq f\ y$   
  show  $x \leq y$  **proof** (*rule* *ccontr*)  
    assume  $\neg x \leq y$  **then** have  $y < x$  **by** *simp*  
    with *assms* *strict-monoD* have  $f\ y < f\ x$  **by** *auto*  
    with  $\langle f\ x \leq f\ y \rangle$  **show** *False* **by** *simp*  
  qed  
qed

lemma *strict-mono-less*:  
  assumes *strict-mono* *f*  
  shows  $f\ x < f\ y \longleftrightarrow x < y$   
  using *assms*  
  **by** (*auto simp add: less-le Orderings.less-le strict-mono-eq strict-mono-less-eq*)

lemma *min-of-mono*:  
  fixes  $f :: 'a \Rightarrow 'b::linorder$   
  shows  $mono\ f \implies min\ (f\ m)\ (f\ n) = f\ (min\ m\ n)$   
  **by** (*auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym*)

```

lemma max-of-mono:
  fixes  $f :: 'a \Rightarrow 'b::linorder$ 
  shows  $mono\ f \implies \max\ (f\ m)\ (f\ n) = f\ (\max\ m\ n)$ 
  by (auto simp: mono-def Orderings.max-def max-def intro: Orderings.antisym)

end

```

```

lemma min-leastL:  $(!!x. least\ \leq\ x) \implies \min\ least\ x = least$ 
by (simp add: min-def)

```

```

lemma max-leastL:  $(!!x. least\ \leq\ x) \implies \max\ least\ x = x$ 
by (simp add: max-def)

```

```

lemma min-leastR:  $(\bigwedge x::'a::order. least\ \leq\ x) \implies \min\ x\ least = least$ 
apply (simp add: min-def)
apply (blast intro: antisym)
done

```

```

lemma max-leastR:  $(\bigwedge x::'a::order. least\ \leq\ x) \implies \max\ x\ least = x$ 
apply (simp add: max-def)
apply (blast intro: antisym)
done

```

### 3.9 Top and bottom elements

```

class top = preorder +
  fixes  $top :: 'a$ 
  assumes top-greatest [simp]:  $x \leq top$ 

```

```

class bot = preorder +
  fixes  $bot :: 'a$ 
  assumes bot-least [simp]:  $bot \leq x$ 

```

### 3.10 Dense orders

```

class dense-linorder = linorder +
  assumes gt-ex:  $\exists y. x < y$ 
  and lt-ex:  $\exists y. y < x$ 
  and dense:  $x < y \implies (\exists z. x < z \wedge z < y)$ 
begin

```

```

lemma dense-le:
  fixes  $y\ z :: 'a$ 
  assumes  $\bigwedge x. x < y \implies x \leq z$ 
  shows  $y \leq z$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  hence  $z < y$  by simp
  from dense[OF this]

```

obtain  $x$  where  $x < y$  and  $z < x$  by *safe*  
 moreover have  $x \leq z$  using *assms*[*OF*  $\langle x < y \rangle$ ] .  
 ultimately show *False* by *auto*  
 qed

lemma *dense-le-bounded*:

fixes  $x\ y\ z :: 'a$   
 assumes  $x < y$   
 assumes \*:  $\bigwedge w. \llbracket x < w ; w < y \rrbracket \implies w \leq z$   
 shows  $y \leq z$   
 proof (rule *dense-le*)  
 fix  $w$  assume  $w < y$   
 from *dense*[*OF*  $\langle x < y \rangle$ ] obtain  $u$  where  $x < u < y$  by *safe*  
 from *linear*[*of*  $u\ w$ ]  
 show  $w \leq z$   
 proof (rule *disjE*)  
 assume  $u \leq w$   
 from *less-le-trans*[*OF*  $\langle x < u \rangle \langle u \leq w \rangle$ ]  $\langle w < y \rangle$   
 show  $w \leq z$  by (rule \*)  
 next  
 assume  $w \leq u$   
 from  $\langle w \leq u \rangle$  \*[*OF*  $\langle x < u \rangle \langle u < y \rangle$ ]  
 show  $w \leq z$  by (rule *order-trans*)  
 qed  
 qed  
 end

### 3.11 Wellorders

class *wellorder* = *linorder* +  
 assumes *less-induct* [*case-names less*]:  $(\bigwedge x. (\bigwedge y. y < x \implies P\ y) \implies P\ x) \implies P\ a$   
 begin

lemma *wellorder-Least-lemma*:

fixes  $k :: 'a$   
 assumes  $P\ k$   
 shows *LeastI*:  $P\ (\text{LEAST } x. P\ x)$  and *Least-le*:  $(\text{LEAST } x. P\ x) \leq k$   
 proof –  
 have  $P\ (\text{LEAST } x. P\ x) \wedge (\text{LEAST } x. P\ x) \leq k$   
 using *assms* proof (induct  $k$  rule: *less-induct*)  
 case (*less x*) then have  $P\ x$  by *simp*  
 show ?case proof (rule *classical*)  
 assume *assm*:  $\neg (P\ (\text{LEAST } a. P\ a) \wedge (\text{LEAST } a. P\ a) \leq x)$   
 have  $\bigwedge y. P\ y \implies x \leq y$   
 proof (rule *classical*)  
 fix  $y$   
 assume  $P\ y$  and  $\neg x \leq y$



```

    with less have P (LEAST a. P a) and (LEAST a. P a) ≤ y
      by (auto simp add: not-le)
    with assm have x < (LEAST a. P a) and (LEAST a. P a) ≤ y
      by auto
    then show x ≤ y by auto
  qed
  with ⟨P x⟩ have Least: (LEAST a. P a) = x
    by (rule Least-equality)
  with ⟨P x⟩ show ?thesis by simp
  qed
  qed
  then show P (LEAST x. P x) and (LEAST x. P x) ≤ k by auto
  qed

```

— The following 3 lemmas are due to Brian Huffman

**lemma** *LeastI-ex*:  $\exists x. P x \implies P (\text{Least } P)$   
 by (erule exE) (erule LeastI)

**lemma** *LeastI2*:  
 $P a \implies (\bigwedge x. P x \implies Q x) \implies Q (\text{Least } P)$   
 by (blast intro: LeastI)

**lemma** *LeastI2-ex*:  
 $\exists a. P a \implies (\bigwedge x. P x \implies Q x) \implies Q (\text{Least } P)$   
 by (blast intro: LeastI-ex)

**lemma** *not-less-Least*:  $k < (\text{LEAST } x. P x) \implies \neg P k$   
 apply (simp (no-asm-use) add: not-le [symmetric])  
 apply (erule contrapos-nn)  
 apply (erule Least-le)  
 done

end

### 3.12 Order on bool

**instantiation** *bool* :: {*order*, *top*, *bot*}  
**begin**

**definition**  
*le-bool-def* [code del]:  $P \leq Q \longleftrightarrow P \longrightarrow Q$

**definition**  
*less-bool-def* [code del]:  $(P::\text{bool}) < Q \longleftrightarrow \neg P \wedge Q$

**definition**  
*top-bool-eq*:  $\text{top} = \text{True}$

**definition**

*bot-bool-eq*:  $bot = False$

**instance proof**

**qed** (*auto simp add: bot-bool-eq top-bool-eq less-bool-def, auto simp add: le-bool-def*)

**end**

**lemma** *le-boolI*:  $(P \implies Q) \implies P \leq Q$   
**by** (*simp add: le-bool-def*)

**lemma** *le-boolI'*:  $P \longrightarrow Q \implies P \leq Q$   
**by** (*simp add: le-bool-def*)

**lemma** *le-boolE*:  $P \leq Q \implies P \implies (Q \implies R) \implies R$   
**by** (*simp add: le-bool-def*)

**lemma** *le-boolD*:  $P \leq Q \implies P \longrightarrow Q$   
**by** (*simp add: le-bool-def*)

**lemma** *bot-boolE*:  $bot \implies P$   
**by** (*simp add: bot-bool-eq*)

**lemma** *top-boolI*: *top*  
**by** (*simp add: top-bool-eq*)

**lemma** [*code*]:  
 $False \leq b \longleftrightarrow True$   
 $True \leq b \longleftrightarrow b$   
 $False < b \longleftrightarrow b$   
 $True < b \longleftrightarrow False$   
**unfolding** *le-bool-def less-bool-def* **by** *simp-all*

### 3.13 Order on functions

**instantiation** *fun* :: (*type*, *ord*) *ord*  
**begin**

**definition**

*le-fun-def* [*code del*]:  $f \leq g \longleftrightarrow (\forall x. f\ x \leq g\ x)$

**definition**

*less-fun-def* [*code del*]:  $(f :: 'a \Rightarrow 'b) < g \longleftrightarrow f \leq g \wedge \neg (g \leq f)$

**instance ..**

**end**

**instance** *fun* :: (*type*, *preorder*) *preorder* **proof**  
**qed** (*auto simp add: le-fun-def less-fun-def*)

```

    intro: order-trans antisym intro!: ext)

instance fun :: (type, order) order proof
qed (auto simp add: le-fun-def intro: antisym ext)

instantiation fun :: (type, top) top
begin

definition
  top-fun-eq: top = ( $\lambda x.$  top)

instance proof
qed (simp add: top-fun-eq le-fun-def)

end

instantiation fun :: (type, bot) bot
begin

definition
  bot-fun-eq: bot = ( $\lambda x.$  bot)

instance proof
qed (simp add: bot-fun-eq le-fun-def)

end

lemma le-funI: ( $\bigwedge x. f\ x \leq g\ x$ )  $\implies f \leq g$ 
  unfolding le-fun-def by simp

lemma le-funE:  $f \leq g \implies (f\ x \leq g\ x \implies P) \implies P$ 
  unfolding le-fun-def by simp

lemma le-funD:  $f \leq g \implies f\ x \leq g\ x$ 
  unfolding le-fun-def by simp

```

### 3.14 Name duplicates

```

lemmas order-eq-refl = preorder-class.eq-refl
lemmas order-less-irrefl = preorder-class.less-irrefl
lemmas order-less-imp-le = preorder-class.less-imp-le
lemmas order-less-not-sym = preorder-class.less-not-sym
lemmas order-less-asym = preorder-class.less-asym
lemmas order-less-trans = preorder-class.less-trans
lemmas order-le-less-trans = preorder-class.le-less-trans
lemmas order-less-le-trans = preorder-class.less-le-trans
lemmas order-less-imp-not-less = preorder-class.less-imp-not-less
lemmas order-less-imp-triv = preorder-class.less-imp-triv
lemmas order-less-asym' = preorder-class.less-asym'

```

```

lemmas order-less-le = order-class.less-le
lemmas order-le-less = order-class.le-less
lemmas order-le-imp-less-or-eq = order-class.le-imp-less-or-eq
lemmas order-less-imp-not-eq = order-class.less-imp-not-eq
lemmas order-less-imp-not-eq2 = order-class.less-imp-not-eq2
lemmas order-neq-le-trans = order-class.neq-le-trans
lemmas order-le-neq-trans = order-class.le-neq-trans
lemmas order-antisym = order-class.antisym
lemmas order-eq-iff = order-class.eq-iff
lemmas order-antisym-conv = order-class.antisym-conv

lemmas linorder-linear = linorder-class.linear
lemmas linorder-less-linear = linorder-class.less-linear
lemmas linorder-le-less-linear = linorder-class.le-less-linear
lemmas linorder-le-cases = linorder-class.le-cases
lemmas linorder-not-less = linorder-class.not-less
lemmas linorder-not-le = linorder-class.not-le
lemmas linorder-neq-iff = linorder-class.neq-iff
lemmas linorder-neqE = linorder-class.neqE
lemmas linorder-antisym-conv1 = linorder-class.antisym-conv1
lemmas linorder-antisym-conv2 = linorder-class.antisym-conv2
lemmas linorder-antisym-conv3 = linorder-class.antisym-conv3

end

```

## 4 Groups: Groups, also combined with orderings

```

theory Groups
imports Orderings
uses (~~/src/Provers/Arith/abel-cancel.ML)
begin

```

### 4.1 Fact collections

```

ML <<
  structure Ac-Simps = Named-Thms(
    val name = ac-simps
    val description = associativity and commutativity simplification rules
  )
  >>

```

```

setup Ac-Simps.setup

```

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring

equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by *field-simps*.

```
ML <<
structure Algebra-Simps = Named-Thms(
  val name = algebra-simps
  val description = algebra simplification rules
)
>>
```

```
setup Algebra-Simps.setup
```

Lemmas *field-simps* multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequalities). Can be too aggressive and is therefore separate from the more benign *algebra-simps*.

```
ML <<
structure Field-Simps = Named-Thms(
  val name = field-simps
  val description = algebra simplification rules for fields
)
>>
```

```
setup Field-Simps.setup
```

## 4.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

```
locale semigroup =
  fixes f :: 'a ⇒ 'a ⇒ 'a (infixl * 70)
  assumes assoc [ac-simps]: a * b * c = a * (b * c)
```

```
locale abel-semigroup = semigroup +
  assumes commute [ac-simps]: a * b = b * a
begin
```

```
lemma left-commute [ac-simps]:
  b * (a * c) = a * (b * c)
proof -
  have (b * a) * c = (a * b) * c
    by (simp only: commute)
  then show ?thesis
    by (simp only: assoc)
qed
```

**end**

**locale** *monoid* = *semigroup* +  
**fixes** *z* :: 'a (1)  
**assumes** *left-neutral* [*simp*]:  $1 * a = a$   
**assumes** *right-neutral* [*simp*]:  $a * 1 = a$

**locale** *comm-monoid* = *abel-semigroup* +  
**fixes** *z* :: 'a (1)  
**assumes** *comm-neutral*:  $a * 1 = a$

**sublocale** *comm-monoid* < *monoid* **proof**  
**qed** (*simp-all add: commute comm-neutral*)

### 4.3 Generic operations

**class** *zero* =  
**fixes** *zero* :: 'a (0)

**class** *one* =  
**fixes** *one* :: 'a (1)

**hide-const** (**open**) *zero one*

**syntax**  
*-index1* :: *index* (1)

**translations**  
 $(index)_1 \Rightarrow (index) \diamond$

**lemma** *Let-0* [*simp*]:  $Let\ 0\ f = f\ 0$   
**unfolding** *Let-def* ..

**lemma** *Let-1* [*simp*]:  $Let\ 1\ f = f\ 1$   
**unfolding** *Let-def* ..

**setup** <<  
*Reorient-Proc.add*  
 (*fn* *Const*(@{*const-name* *Groups.zero*}, -) => *true*  
   | *Const*(@{*const-name* *Groups.one*}, -) => *true*  
   | - => *false*)  
 >>

**simproc-setup** *reorient-zero* ( $0 = x$ ) = *Reorient-Proc.proc*  
**simproc-setup** *reorient-one* ( $1 = x$ ) = *Reorient-Proc.proc*

**typed-print-translation** <<  
*let*  
*fun* *tr'* *c* = (*c*, *fn* *show-sorts* => *fn* *T* => *fn* *ts* =>

*if* (*not o null*) *ts* *orelse* *T* = *dummyT*

```

    orelse not (! show-types) andalso can Term.dest-Type T
  then raise Match
    else Syntax.const Syntax.constrainC $ Syntax.const c $ Syntax.term-of-typ
show-sorts T);
in map tr' [@{const-syntax Groups.one}, @{const-syntax Groups.zero}] end;
>> — show types that are presumably too general

```

```

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a (infixl + 65)

```

```

class minus =
  fixes minus :: 'a ⇒ 'a ⇒ 'a (infixl − 65)

```

```

class uminus =
  fixes uminus :: 'a ⇒ 'a (− - [81] 80)

```

```

class times =
  fixes times :: 'a ⇒ 'a ⇒ 'a (infixl * 70)

```

```

use ~~/src/Provers/Arith/abel-cancel.ML

```

#### 4.4 Semigroups and Monoids

```

class semigroup-add = plus +
  assumes add-assoc [algebra-simps, field-simps]: (a + b) + c = a + (b + c)

```

```

sublocale semigroup-add < add!: semigroup plus proof
qed (fact add-assoc)

```

```

class ab-semigroup-add = semigroup-add +
  assumes add-commute [algebra-simps, field-simps]: a + b = b + a

```

```

sublocale ab-semigroup-add < add!: ab-semigroup plus proof
qed (fact add-commute)

```

```

context ab-semigroup-add
begin

```

```

lemmas add-left-commute [algebra-simps, field-simps] = add.left-commute

```

```

theorems add-ac = add-assoc add-commute add-left-commute

```

```

end

```

```

theorems add-ac = add-assoc add-commute add-left-commute

```

```

class semigroup-mult = times +
  assumes mult-assoc [algebra-simps, field-simps]: (a * b) * c = a * (b * c)

```

**sublocale** *semigroup-mult* < *mult!*: *semigroup times* **proof**  
**qed** (*fact mult-assoc*)

**class** *ab-semigroup-mult* = *semigroup-mult* +  
**assumes** *mult-commute* [*algebra-simps*, *field-simps*]:  $a * b = b * a$

**sublocale** *ab-semigroup-mult* < *mult!*: *abel-semigroup times* **proof**  
**qed** (*fact mult-commute*)

**context** *ab-semigroup-mult*  
**begin**

**lemmas** *mult-left-commute* [*algebra-simps*, *field-simps*] = *mult.left-commute*

**theorems** *mult-ac* = *mult-assoc mult-commute mult-left-commute*

**end**

**theorems** *mult-ac* = *mult-assoc mult-commute mult-left-commute*

**class** *monoid-add* = *zero* + *semigroup-add* +  
**assumes** *add-0-left*:  $0 + a = a$   
**and** *add-0-right*:  $a + 0 = a$

**sublocale** *monoid-add* < *add!*: *monoid plus 0* **proof**  
**qed** (*fact add-0-left add-0-right*) +

**lemma** *zero-reorient*:  $0 = x \longleftrightarrow x = 0$   
**by** (*rule eq-commute*)

**class** *comm-monoid-add* = *zero* + *ab-semigroup-add* +  
**assumes** *add-0*:  $0 + a = a$

**sublocale** *comm-monoid-add* < *add!*: *comm-monoid plus 0* **proof**  
**qed** (*insert add-0, simp add: ac-simps*)

**subclass** (**in** *comm-monoid-add*) *monoid-add* **proof**  
**qed** (*fact add.left-neutral add.right-neutral*) +

**class** *monoid-mult* = *one* + *semigroup-mult* +  
**assumes** *mult-1-left*:  $1 * a = a$   
**and** *mult-1-right*:  $a * 1 = a$

**sublocale** *monoid-mult* < *mult!*: *monoid times 1* **proof**  
**qed** (*fact mult-1-left mult-1-right*) +

**lemma** *one-reorient*:  $1 = x \longleftrightarrow x = 1$   
**by** (*rule eq-commute*)



```
class comm-monoid-mult = one + ab-semigroup-mult +
  assumes mult-1:  $1 * a = a$ 
```

```
sublocale comm-monoid-mult < mult!: comm-monoid times 1 proof
qed (insert mult-1, simp add: ac-simps)
```

```
subclass (in comm-monoid-mult) monoid-mult proof
qed (fact mult.left-neutral mult.right-neutral)+
```

```
class cancel-semigroup-add = semigroup-add +
  assumes add-left-imp-eq:  $a + b = a + c \implies b = c$ 
  assumes add-right-imp-eq:  $b + a = c + a \implies b = c$ 
begin
```

```
lemma add-left-cancel [simp]:
 $a + b = a + c \longleftrightarrow b = c$ 
by (blast dest: add-left-imp-eq)
```

```
lemma add-right-cancel [simp]:
 $b + a = c + a \longleftrightarrow b = c$ 
by (blast dest: add-right-imp-eq)
```

```
end
```

```
class cancel-ab-semigroup-add = ab-semigroup-add +
  assumes add-imp-eq:  $a + b = a + c \implies b = c$ 
begin
```

```
subclass cancel-semigroup-add
proof
  fix a b c :: 'a
  assume a + b = a + c
  then show b = c by (rule add-imp-eq)
next
  fix a b c :: 'a
  assume b + a = c + a
  then have a + b = a + c by (simp only: add-commute)
  then show b = c by (rule add-imp-eq)
qed
```

```
end
```

```
class cancel-comm-monoid-add = cancel-ab-semigroup-add + comm-monoid-add
```

## 4.5 Groups

```
class group-add = minus + uminus + monoid-add +
  assumes left-minus [simp]:  $- a + a = 0$ 
  assumes diff-minus:  $a - b = a + (- b)$ 
```

**begin**

**lemma** *minus-unique*:

**assumes**  $a + b = 0$  **shows**  $- a = b$

**proof**  $-$

**have**  $- a = - a + (a + b)$  **using** *assms* **by** *simp*

**also have**  $\dots = b$  **by** (*simp add: add-assoc [symmetric]*)

**finally show** *?thesis* .

**qed**

**lemmas** *equals-zero-I = minus-unique*

**lemma** *minus-zero [simp]*:  $- 0 = 0$

**proof**  $-$

**have**  $0 + 0 = 0$  **by** (*rule add-0-right*)

**thus**  $- 0 = 0$  **by** (*rule minus-unique*)

**qed**

**lemma** *minus-minus [simp]*:  $- (- a) = a$

**proof**  $-$

**have**  $- a + a = 0$  **by** (*rule left-minus*)

**thus**  $- (- a) = a$  **by** (*rule minus-unique*)

**qed**

**lemma** *right-minus [simp]*:  $a + - a = 0$

**proof**  $-$

**have**  $a + - a = - (- a) + - a$  **by** *simp*

**also have**  $\dots = 0$  **by** (*rule left-minus*)

**finally show** *?thesis* .

**qed**

**lemma** *minus-add-cancel*:  $- a + (a + b) = b$

**by** (*simp add: add-assoc [symmetric]*)

**lemma** *add-minus-cancel*:  $a + (- a + b) = b$

**by** (*simp add: add-assoc [symmetric]*)

**lemma** *minus-add*:  $- (a + b) = - b + - a$

**proof**  $-$

**have**  $(a + b) + (- b + - a) = 0$

**by** (*simp add: add-assoc add-minus-cancel*)

**thus**  $- (a + b) = - b + - a$

**by** (*rule minus-unique*)

**qed**

**lemma** *right-minus-eq*:  $a - b = 0 \longleftrightarrow a = b$

**proof**

**assume**  $a - b = 0$

**have**  $a = (a - b) + b$  **by** (*simp add: diff-minus add-assoc*)

also have  $\dots = b$  using  $\langle a - b = 0 \rangle$  by *simp*  
 finally show  $a = b$  .  
 next  
 assume  $a = b$  thus  $a - b = 0$  by (*simp add: diff-minus*)  
 qed

**lemma** *diff-self* [*simp*]:  $a - a = 0$   
 by (*simp add: diff-minus*)

**lemma** *diff-0* [*simp*]:  $0 - a = - a$   
 by (*simp add: diff-minus*)

**lemma** *diff-0-right* [*simp*]:  $a - 0 = a$   
 by (*simp add: diff-minus*)

**lemma** *diff-minus-eq-add* [*simp*]:  $a - - b = a + b$   
 by (*simp add: diff-minus*)

**lemma** *neg-equal-iff-equal* [*simp*]:  
 $- a = - b \longleftrightarrow a = b$

**proof**  
 assume  $- a = - b$   
 hence  $- (- a) = - (- b)$  by *simp*  
 thus  $a = b$  by *simp*  
 next  
 assume  $a = b$   
 thus  $- a = - b$  by *simp*  
 qed

**lemma** *neg-equal-0-iff-equal* [*simp*]:  
 $- a = 0 \longleftrightarrow a = 0$   
 by (*subst neg-equal-iff-equal [symmetric], simp*)

**lemma** *neg-0-equal-iff-equal* [*simp*]:  
 $0 = - a \longleftrightarrow 0 = a$   
 by (*subst neg-equal-iff-equal [symmetric], simp*)

The next two equations can make the simplifier loop!

**lemma** *equation-minus-iff*:  
 $a = - b \longleftrightarrow b = - a$   
**proof** -  
 have  $- (- a) = - b \longleftrightarrow - a = b$  by (*rule neg-equal-iff-equal*)  
 thus ?thesis by (*simp add: eq-commute*)  
 qed

**lemma** *minus-equation-iff*:  
 $- a = b \longleftrightarrow - b = a$   
**proof** -  
 have  $- a = - (- b) \longleftrightarrow a = - b$  by (*rule neg-equal-iff-equal*)

```

    thus ?thesis by (simp add: eq-commute)
qed

lemma diff-add-cancel:  $a - b + b = a$ 
by (simp add: diff-minus add-assoc)

lemma add-diff-cancel:  $a + b - b = a$ 
by (simp add: diff-minus add-assoc)

declare diff-minus[symmetric, algebra-simps, field-simps]

lemma eq-neg-iff-add-eq-0:  $a = -b \longleftrightarrow a + b = 0$ 
proof
  assume  $a = -b$  then show  $a + b = 0$  by simp
next
  assume  $a + b = 0$ 
  moreover have  $a + (b + -b) = (a + b) + -b$ 
    by (simp only: add-assoc)
  ultimately show  $a = -b$  by simp
qed

end

class ab-group-add = minus + uminus + comm-monoid-add +
  assumes ab-left-minus:  $-a + a = 0$ 
  assumes ab-diff-minus:  $a - b = a + (-b)$ 
begin

subclass group-add
  proof qed (simp-all add: ab-left-minus ab-diff-minus)

subclass cancel-comm-monoid-add
proof
  fix  $a\ b\ c :: 'a$ 
  assume  $a + b = a + c$ 
  then have  $-a + a + b = -a + a + c$ 
    unfolding add-assoc by simp
  then show  $b = c$  by simp
qed

lemma uminus-add-conv-diff[algebra-simps, field-simps]:
   $-a + b = b - a$ 
by (simp add: diff-minus add-commute)

lemma minus-add-distrib [simp]:
   $-(a + b) = -a + -b$ 
by (rule minus-unique) (simp add: add-ac)

lemma minus-diff-eq [simp]:

```

$-(a - b) = b - a$   
**by** (*simp add: diff-minus add-commute*)

**lemma** *add-diff-eq*[*algebra-simps, field-simps*]:  $a + (b - c) = (a + b) - c$   
**by** (*simp add: diff-minus add-ac*)

**lemma** *diff-add-eq*[*algebra-simps, field-simps*]:  $(a - b) + c = (a + c) - b$   
**by** (*simp add: diff-minus add-ac*)

**lemma** *diff-eq-eq*[*algebra-simps, field-simps*]:  $a - b = c \longleftrightarrow a = c + b$   
**by** (*auto simp add: diff-minus add-assoc*)

**lemma** *eq-diff-eq*[*algebra-simps, field-simps*]:  $a = c - b \longleftrightarrow a + b = c$   
**by** (*auto simp add: diff-minus add-assoc*)

**lemma** *diff-diff-eq*[*algebra-simps, field-simps*]:  $(a - b) - c = a - (b + c)$   
**by** (*simp add: diff-minus add-ac*)

**lemma** *diff-diff-eq2*[*algebra-simps, field-simps*]:  $a - (b - c) = (a + c) - b$   
**by** (*simp add: diff-minus add-ac*)

**lemma** *eq-iff-diff-eq-0*:  $a = b \longleftrightarrow a - b = 0$   
**by** (*simp add: algebra-simps*)

**lemma** *diff-eq-0-iff-eq* [*simp, no-atp*]:  $a - b = 0 \longleftrightarrow a = b$   
**by** (*simp add: algebra-simps*)

**end**

## 4.6 (Partially) Ordered Groups

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- <http://www.mathworld.com> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

**class** *ordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +  
**assumes** *add-left-mono*:  $a \leq b \implies c + a \leq c + b$   
**begin**

**lemma** *add-right-mono*:

$$a \leq b \implies a + c \leq b + c$$

**by** (*simp* *add*: *add-commute* [*of* - *c*] *add-left-mono*)

non-strict, in both arguments

**lemma** *add-mono*:

$$a \leq b \implies c \leq d \implies a + c \leq b + d$$

**apply** (*erule* *add-right-mono* [*THEN* *order-trans*])

**apply** (*simp* *add*: *add-commute* *add-left-mono*)

**done**

**end**

**class** *ordered-cancel-ab-semigroup-add* =

*ordered-ab-semigroup-add* + *cancel-ab-semigroup-add*

**begin**

**lemma** *add-strict-left-mono*:

$$a < b \implies c + a < c + b$$

**by** (*auto* *simp* *add*: *less-le* *add-left-mono*)

**lemma** *add-strict-right-mono*:

$$a < b \implies a + c < b + c$$

**by** (*simp* *add*: *add-commute* [*of* - *c*] *add-strict-left-mono*)

Strict monotonicity in both arguments

**lemma** *add-strict-mono*:

$$a < b \implies c < d \implies a + c < b + d$$

**apply** (*erule* *add-strict-right-mono* [*THEN* *less-trans*])

**apply** (*erule* *add-strict-left-mono*)

**done**

**lemma** *add-less-le-mono*:

$$a < b \implies c \leq d \implies a + c < b + d$$

**apply** (*erule* *add-strict-right-mono* [*THEN* *less-le-trans*])

**apply** (*erule* *add-left-mono*)

**done**

**lemma** *add-le-less-mono*:

$$a \leq b \implies c < d \implies a + c < b + d$$

**apply** (*erule* *add-right-mono* [*THEN* *le-less-trans*])

**apply** (*erule* *add-strict-left-mono*)

**done**

**end**

**class** *ordered-ab-semigroup-add-imp-le* =

*ordered-cancel-ab-semigroup-add* +

**assumes** *add-le-imp-le-left*:  $c + a \leq c + b \implies a \leq b$   
**begin**

**lemma** *add-less-imp-less-left*:  
**assumes** *less*:  $c + a < c + b$  **shows**  $a < b$   
**proof** –  
**from** *less* **have** *le*:  $c + a \leq c + b$  **by** (*simp add: order-le-less*)  
**have**  $a \leq b$   
**apply** (*insert le*)  
**apply** (*drule add-le-imp-le-left*)  
**by** (*insert le, drule add-le-imp-le-left, assumption*)  
**moreover** **have**  $a \neq b$   
**proof** (*rule ccontr*)  
**assume**  $\sim(a \neq b)$   
**then** **have**  $a = b$  **by** *simp*  
**then** **have**  $c + a = c + b$  **by** *simp*  
**with** *less* **show** *False* **by** *simp*  
**qed**  
**ultimately** **show**  $a < b$  **by** (*simp add: order-le-less*)  
**qed**

**lemma** *add-less-imp-less-right*:  
 $a + c < b + c \implies a < b$   
**apply** (*rule add-less-imp-less-left [of c]*)  
**apply** (*simp add: add-commute*)  
**done**

**lemma** *add-less-cancel-left* [*simp*]:  
 $c + a < c + b \iff a < b$   
**by** (*blast intro: add-less-imp-less-left add-strict-left-mono*)

**lemma** *add-less-cancel-right* [*simp*]:  
 $a + c < b + c \iff a < b$   
**by** (*blast intro: add-less-imp-less-right add-strict-right-mono*)

**lemma** *add-le-cancel-left* [*simp*]:  
 $c + a \leq c + b \iff a \leq b$   
**by** (*auto, drule add-le-imp-le-left, simp-all add: add-left-mono*)

**lemma** *add-le-cancel-right* [*simp*]:  
 $a + c \leq b + c \iff a \leq b$   
**by** (*simp add: add-commute [of a c] add-commute [of b c]*)

**lemma** *add-le-imp-le-right*:  
 $a + c \leq b + c \implies a \leq b$   
**by** *simp*

**lemma** *max-add-distrib-left*:  
 $\max x y + z = \max (x + z) (y + z)$

```

unfolding max-def by auto

lemma min-add-distrib-left:
   $\min x y + z = \min (x + z) (y + z)$ 
  unfolding min-def by auto

end

### 4.7 Support for reasoning about signs

class ordered-comm-monoid-add =
  ordered-cancel-ab-semigroup-add + comm-monoid-add
begin

lemma add-pos-nonneg:
  assumes  $0 < a$  and  $0 \leq b$  shows  $0 < a + b$ 
proof –
  have  $0 + 0 < a + b$ 
  using assms by (rule add-less-le-mono)
  then show ?thesis by simp
qed

lemma add-pos-pos:
  assumes  $0 < a$  and  $0 < b$  shows  $0 < a + b$ 
by (rule add-pos-nonneg) (insert assms, auto)

lemma add-nonneg-pos:
  assumes  $0 \leq a$  and  $0 < b$  shows  $0 < a + b$ 
proof –
  have  $0 + 0 < a + b$ 
  using assms by (rule add-le-less-mono)
  then show ?thesis by simp
qed

lemma add-nonneg-nonneg [simp]:
  assumes  $0 \leq a$  and  $0 \leq b$  shows  $0 \leq a + b$ 
proof –
  have  $0 + 0 \leq a + b$ 
  using assms by (rule add-mono)
  then show ?thesis by simp
qed

lemma add-neg-nonpos:
  assumes  $a < 0$  and  $b \leq 0$  shows  $a + b < 0$ 
proof –
  have  $a + b < 0 + 0$ 
  using assms by (rule add-less-le-mono)
  then show ?thesis by simp
qed

```



```

lemma add-neg-neg:
  assumes  $a < 0$  and  $b < 0$  shows  $a + b < 0$ 
by (rule add-neg-nonpos) (insert assms, auto)

lemma add-nonpos-neg:
  assumes  $a \leq 0$  and  $b < 0$  shows  $a + b < 0$ 
proof -
  have  $a + b < 0 + 0$ 
  using assms by (rule add-le-less-mono)
  then show ?thesis by simp
qed

lemma add-nonpos-nonpos:
  assumes  $a \leq 0$  and  $b \leq 0$  shows  $a + b \leq 0$ 
proof -
  have  $a + b \leq 0 + 0$ 
  using assms by (rule add-mono)
  then show ?thesis by simp
qed

lemmas add-sign-intros =
  add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
  add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

lemma add-nonneg-eq-0-iff:
  assumes  $x: 0 \leq x$  and  $y: 0 \leq y$ 
  shows  $x + y = 0 \iff x = 0 \wedge y = 0$ 
proof (intro iffI conjI)
  have  $x = x + 0$  by simp
  also have  $x + 0 \leq x + y$  using  $y$  by (rule add-left-mono)
  also assume  $x + y = 0$ 
  also have  $0 \leq x$  using  $x$  .
  finally show  $x = 0$  .
next
  have  $y = 0 + y$  by simp
  also have  $0 + y \leq x + y$  using  $x$  by (rule add-right-mono)
  also assume  $x + y = 0$ 
  also have  $0 \leq y$  using  $y$  .
  finally show  $y = 0$  .
next
  assume  $x = 0 \wedge y = 0$ 
  then show  $x + y = 0$  by simp
qed

end

class ordered-ab-group-add =
  ab-group-add + ordered-ab-semigroup-add

```

**begin**

**subclass** *ordered-cancel-ab-semigroup-add* ..

**subclass** *ordered-ab-semigroup-add-imp-le*

**proof**

**fix**  $a\ b\ c :: 'a$

**assume**  $c + a \leq c + b$

**hence**  $(-c) + (c + a) \leq (-c) + (c + b)$  **by** (*rule add-left-mono*)

**hence**  $((-c) + c) + a \leq ((-c) + c) + b$  **by** (*simp only: add-assoc*)

**thus**  $a \leq b$  **by** *simp*

**qed**

**subclass** *ordered-comm-monoid-add* ..

**lemma** *max-diff-distrib-left*:

**shows**  $\max\ x\ y - z = \max\ (x - z)\ (y - z)$

**by** (*simp add: diff-minus, rule max-add-distrib-left*)

**lemma** *min-diff-distrib-left*:

**shows**  $\min\ x\ y - z = \min\ (x - z)\ (y - z)$

**by** (*simp add: diff-minus, rule min-add-distrib-left*)

**lemma** *le-imp-neg-le*:

**assumes**  $a \leq b$  **shows**  $-b \leq -a$

**proof** -

**have**  $-a + a \leq -a + b$  **using**  $a \leq b$  **by** (*rule add-left-mono*)

**hence**  $0 \leq -a + b$  **by** *simp*

**hence**  $0 + (-b) \leq (-a + b) + (-b)$  **by** (*rule add-right-mono*)

**thus** *?thesis* **by** (*simp add: add-assoc*)

**qed**

**lemma** *neg-le-iff-le* [*simp*]:  $-b \leq -a \longleftrightarrow a \leq b$

**proof**

**assume**  $-b \leq -a$

**hence**  $-(-a) \leq -(-b)$  **by** (*rule le-imp-neg-le*)

**thus**  $a \leq b$  **by** *simp*

**next**

**assume**  $a \leq b$

**thus**  $-b \leq -a$  **by** (*rule le-imp-neg-le*)

**qed**

**lemma** *neg-le-0-iff-le* [*simp*]:  $-a \leq 0 \longleftrightarrow 0 \leq a$

**by** (*subst neg-le-iff-le [symmetric], simp*)

**lemma** *neg-0-le-iff-le* [*simp*]:  $0 \leq -a \longleftrightarrow a \leq 0$

**by** (*subst neg-le-iff-le [symmetric], simp*)

**lemma** *neg-less-iff-less* [*simp*]:  $-b < -a \longleftrightarrow a < b$

**by** (*force simp add: less-le*)

**lemma** *neg-less-0-iff-less* [*simp*]:  $- a < 0 \longleftrightarrow 0 < a$   
**by** (*subst neg-less-iff-less [symmetric], simp*)

**lemma** *neg-0-less-iff-less* [*simp*]:  $0 < - a \longleftrightarrow a < 0$   
**by** (*subst neg-less-iff-less [symmetric], simp*)

The next several equations can make the simplifier loop!

**lemma** *less-minus-iff*:  $a < - b \longleftrightarrow b < - a$   
**proof** –  
**have**  $(- (-a) < - b) = (b < - a)$  **by** (*rule neg-less-iff-less*)  
**thus** *?thesis* **by** *simp*  
**qed**

**lemma** *minus-less-iff*:  $- a < b \longleftrightarrow - b < a$   
**proof** –  
**have**  $(- a < - (-b)) = (- b < a)$  **by** (*rule neg-less-iff-less*)  
**thus** *?thesis* **by** *simp*  
**qed**

**lemma** *le-minus-iff*:  $a \leq - b \longleftrightarrow b \leq - a$   
**proof** –  
**have** *mm*:  $!! a (b::'a). (-(-a)) < -b \implies -(-b) < -a$  **by** (*simp only: minus-less-iff*)  
**have**  $(- (- a) \leq -b) = (b \leq - a)$   
**apply** (*auto simp only: le-less*)  
**apply** (*drule mm*)  
**apply** (*simp-all*)  
**apply** (*drule mm[simplified], assumption*)  
**done**  
**then show** *?thesis* **by** *simp*  
**qed**

**lemma** *minus-le-iff*:  $- a \leq b \longleftrightarrow - b \leq a$   
**by** (*auto simp add: le-less minus-less-iff*)

**lemma** *less-iff-diff-less-0*:  $a < b \longleftrightarrow a - b < 0$   
**proof** –  
**have**  $(a < b) = (a + (- b) < b + (-b))$   
**by** (*simp only: add-less-cancel-right*)  
**also have**  $\dots = (a - b < 0)$  **by** (*simp add: diff-minus*)  
**finally show** *?thesis* .  
**qed**

**lemma** *diff-less-eq[algebra-simps, field-simps]*:  $a - b < c \longleftrightarrow a < c + b$   
**apply** (*subst less-iff-diff-less-0 [of a]*)  
**apply** (*rule less-iff-diff-less-0 [of - c, THEN ssubst]*)  
**apply** (*simp add: diff-minus add-ac*)  
**done**

```

lemma less-diff-eq[algebra-simps, field-simps]:  $a < c - b \iff a + b < c$ 
apply (subst less-iff-diff-less-0 [of  $a + b$ ])
apply (subst less-iff-diff-less-0 [of  $a$ ])
apply (simp add: diff-minus add-ac)
done

```

```

lemma diff-le-eq[algebra-simps, field-simps]:  $a - b \leq c \iff a \leq c + b$ 
by (auto simp add: le-less diff-less-eq diff-add-cancel add-diff-cancel)

```

```

lemma le-diff-eq[algebra-simps, field-simps]:  $a \leq c - b \iff a + b \leq c$ 
by (auto simp add: le-less less-diff-eq diff-add-cancel add-diff-cancel)

```

```

lemma le-iff-diff-le-0:  $a \leq b \iff a - b \leq 0$ 
by (simp add: algebra-simps)

```

**end**

```

class linordered-ab-semigroup-add =
  linorder + ordered-ab-semigroup-add

```

```

class linordered-cancel-ab-semigroup-add =
  linorder + ordered-cancel-ab-semigroup-add
begin

```

```

subclass linordered-ab-semigroup-add ..

```

```

subclass ordered-ab-semigroup-add-imp-le
proof
  fix  $a\ b\ c :: 'a$ 
  assume le:  $c + a \leq c + b$ 
  show  $a \leq b$ 
  proof (rule ccontr)
    assume  $w: \sim a \leq b$ 
    hence  $b \leq a$  by (simp add: linorder-not-le)
    hence le2:  $c + b \leq c + a$  by (rule add-left-mono)
    have  $a = b$ 
    apply (insert le)
    apply (insert le2)
    apply (drule antisym, simp-all)
    done
  with  $w$  show False
  by (simp add: linorder-not-le [symmetric])
qed
qed

```

**end**

```

class linordered-ab-group-add = linorder + ordered-ab-group-add

```

**begin**

**subclass** *linordered-cancel-ab-semigroup-add* ..

**lemma** *neg-less-eq-nonneg* [*simp*]:

$$- a \leq a \iff 0 \leq a$$

**proof**

**assume**  $A: - a \leq a$  **show**  $0 \leq a$

**proof** (*rule classical*)

**assume**  $\neg 0 \leq a$

**then have**  $a < 0$  **by** *auto*

**with**  $A$  **have**  $- a < 0$  **by** (*rule le-less-trans*)

**then show** *?thesis* **by** *auto*

**qed**

**next**

**assume**  $A: 0 \leq a$  **show**  $- a \leq a$

**proof** (*rule order-trans*)

**show**  $- a \leq 0$  **using**  $A$  **by** (*simp add: minus-le-iff*)

**next**

**show**  $0 \leq a$  **using**  $A$  .

**qed**

**qed**

**lemma** *neg-less-nonneg* [*simp*]:

$$- a < a \iff 0 < a$$

**proof**

**assume**  $A: - a < a$  **show**  $0 < a$

**proof** (*rule classical*)

**assume**  $\neg 0 < a$

**then have**  $a \leq 0$  **by** *auto*

**with**  $A$  **have**  $- a < 0$  **by** (*rule less-le-trans*)

**then show** *?thesis* **by** *auto*

**qed**

**next**

**assume**  $A: 0 < a$  **show**  $- a < a$

**proof** (*rule less-trans*)

**show**  $- a < 0$  **using**  $A$  **by** (*simp add: minus-le-iff*)

**next**

**show**  $0 < a$  **using**  $A$  .

**qed**

**qed**

**lemma** *less-eq-neg-nonpos* [*simp*]:

$$a \leq - a \iff a \leq 0$$

**proof**

**assume**  $A: a \leq - a$  **show**  $a \leq 0$

**proof** (*rule classical*)

**assume**  $\neg a \leq 0$

**then have**  $0 < a$  **by** *auto*

```

    then have  $0 < - a$  using  $A$  by (rule less-le-trans)
    then show ?thesis by auto
  qed
next
  assume  $A: a \leq 0$  show  $a \leq - a$ 
  proof (rule order-trans)
    show  $0 \leq - a$  using  $A$  by (simp add: minus-le-iff)
  next
    show  $a \leq 0$  using  $A$  .
  qed
qed

```

```

lemma equal-neg-zero [simp]:
   $a = - a \longleftrightarrow a = 0$ 
proof
  assume  $a = 0$  then show  $a = - a$  by simp
next
  assume  $A: a = - a$  show  $a = 0$ 
  proof (cases  $0 \leq a$ )
    case True with  $A$  have  $0 \leq - a$  by auto
    with le-minus-iff have  $a \leq 0$  by simp
    with  $A$  show ?thesis by (auto intro: order-trans)
  next
    case False then have  $B: a \leq 0$  by auto
    with  $A$  have  $- a \leq 0$  by auto
    with  $B$  show ?thesis by (auto intro: order-trans)
  qed
qed

```

```

lemma neg-equal-zero [simp]:
   $- a = a \longleftrightarrow a = 0$ 
  by (auto dest: sym)

```

```

lemma double-zero [simp]:
   $a + a = 0 \longleftrightarrow a = 0$ 
proof
  assume  $assm: a + a = 0$ 
  then have  $a: - a = a$  by (rule minus-unique)
  then show  $a = 0$  by (simp only: neg-equal-zero)
qed simp

```

```

lemma double-zero-sym [simp]:
   $0 = a + a \longleftrightarrow a = 0$ 
  by (rule, drule sym) simp-all

```

```

lemma zero-less-double-add-iff-zero-less-single-add [simp]:
   $0 < a + a \longleftrightarrow 0 < a$ 
proof
  assume  $0 < a + a$ 

```

```

then have  $0 - a < a$  by (simp only: diff-less-eq)
then have  $-a < a$  by simp
then show  $0 < a$  by (simp only: neg-less-nonneg)
next
  assume  $0 < a$ 
  with this have  $0 + 0 < a + a$ 
    by (rule add-strict-mono)
  then show  $0 < a + a$  by simp
qed

```

```

lemma zero-le-double-add-iff-zero-le-single-add [simp]:
   $0 \leq a + a \longleftrightarrow 0 \leq a$ 
  by (auto simp add: le-less)

```

```

lemma double-add-less-zero-iff-single-add-less-zero [simp]:
   $a + a < 0 \longleftrightarrow a < 0$ 
proof -
  have  $\neg a + a < 0 \longleftrightarrow \neg a < 0$ 
    by (simp add: not-less)
  then show ?thesis by simp
qed

```

```

lemma double-add-le-zero-iff-single-add-le-zero [simp]:
   $a + a \leq 0 \longleftrightarrow a \leq 0$ 
proof -
  have  $\neg a + a \leq 0 \longleftrightarrow \neg a \leq 0$ 
    by (simp add: not-le)
  then show ?thesis by simp
qed

```

```

lemma le-minus-self-iff:
   $a \leq -a \longleftrightarrow a \leq 0$ 
proof -
  from add-le-cancel-left [of  $-a$   $a + a$   $0$ ]
  have  $a \leq -a \longleftrightarrow a + a \leq 0$ 
    by (simp add: add-assoc [symmetric])
  thus ?thesis by simp
qed

```

```

lemma minus-le-self-iff:
   $-a \leq a \longleftrightarrow 0 \leq a$ 
proof -
  from add-le-cancel-left [of  $-a$   $0$   $a + a$ ]
  have  $-a \leq a \longleftrightarrow 0 \leq a + a$ 
    by (simp add: add-assoc [symmetric])
  thus ?thesis by simp
qed

```

```

lemma minus-max-eq-min:

```

```

  -  $\max x y = \min (-x) (-y)$ 
  by (auto simp add: max-def min-def)

lemma minus-min-eq-max:
  -  $\min x y = \max (-x) (-y)$ 
  by (auto simp add: max-def min-def)

end

context ordered-comm-monoid-add
begin

lemma add-increasing:
   $0 \leq a \implies b \leq c \implies b \leq a + c$ 
  by (insert add-mono [of 0 a b c], simp)

lemma add-increasing2:
   $0 \leq c \implies b \leq a \implies b \leq a + c$ 
  by (simp add: add-increasing add-commute [of a])

lemma add-strict-increasing:
   $0 < a \implies b \leq c \implies b < a + c$ 
  by (insert add-less-le-mono [of 0 a b c], simp)

lemma add-strict-increasing2:
   $0 \leq a \implies b < c \implies b < a + c$ 
  by (insert add-le-less-mono [of 0 a b c], simp)

end

class abs =
  fixes abs :: 'a  $\Rightarrow$  'a
begin

notation (xsymbols)
  abs (|·|)

notation (HTML output)
  abs (|·|)

end

class sgn =
  fixes sgn :: 'a  $\Rightarrow$  'a

class abs-if = minus + uminus + ord + zero + abs +
  assumes abs-if:  $|a| = (\text{if } a < 0 \text{ then } -a \text{ else } a)$ 

class sgn-if = minus + uminus + zero + one + ord + sgn +

```



**assumes** *sgn-if*:  $\text{sgn } x = (\text{if } x = 0 \text{ then } 0 \text{ else if } 0 < x \text{ then } 1 \text{ else } -1)$   
**begin**

**lemma** *sgn0* [*simp*]:  $\text{sgn } 0 = 0$   
**by** (*simp add:sgn-if*)

**end**

**class** *ordered-ab-group-add-abs* = *ordered-ab-group-add* + *abs* +  
**assumes** *abs-ge-zero* [*simp*]:  $|a| \geq 0$   
**and** *abs-ge-self*:  $a \leq |a|$   
**and** *abs-leI*:  $a \leq b \implies -a \leq b \implies |a| \leq b$   
**and** *abs-minus-cancel* [*simp*]:  $|-a| = |a|$   
**and** *abs-triangle-ineq*:  $|a + b| \leq |a| + |b|$   
**begin**

**lemma** *abs-minus-le-zero*:  $-|a| \leq 0$   
**unfolding** *neg-le-0-iff-le* **by** *simp*

**lemma** *abs-of-nonneg* [*simp*]:  
**assumes** *nonneg*:  $0 \leq a$  **shows**  $|a| = a$   
**proof** (*rule antisym*)  
**from** *nonneg le-imp-neg-le* **have**  $-a \leq 0$  **by** *simp*  
**from** *this nonneg* **have**  $-a \leq a$  **by** (*rule order-trans*)  
**then show**  $|a| \leq a$  **by** (*auto intro: abs-leI*)  
**qed** (*rule abs-ge-self*)

**lemma** *abs-idempotent* [*simp*]:  $||a|| = |a|$   
**by** (*rule antisym*)  
*(auto intro!: abs-ge-self abs-leI order-trans [of - |a| 0 |a|])*

**lemma** *abs-eq-0* [*simp*]:  $|a| = 0 \iff a = 0$   
**proof** –  
**have**  $|a| = 0 \implies a = 0$   
**proof** (*rule antisym*)  
**assume** *zero*:  $|a| = 0$   
**with** *abs-ge-self* **show**  $a \leq 0$  **by** *auto*  
**from** *zero* **have**  $|-a| = 0$  **by** *simp*  
**with** *abs-ge-self* [*of - a*] **have**  $-a \leq 0$  **by** *auto*  
**with** *neg-le-0-iff-le* **show**  $0 \leq a$  **by** *auto*  
**qed**  
**then show** *?thesis* **by** *auto*  
**qed**

**lemma** *abs-zero* [*simp*]:  $|0| = 0$   
**by** *simp*

**lemma** *abs-0-eq* [*simp*, *no-atp*]:  $0 = |a| \iff a = 0$   
**proof** –

have  $0 = |a| \iff |a| = 0$  **by** (*simp only: eq-ac*)  
 thus *?thesis* **by** *simp*  
**qed**

**lemma** *abs-le-zero-iff* [*simp*]:  $|a| \leq 0 \iff a = 0$   
**proof** –  
 assume  $|a| \leq 0$   
 then have  $|a| = 0$  **by** (*rule antisym*) *simp*  
 thus  $a = 0$  **by** *simp*  
**next**  
 assume  $a = 0$   
 thus  $|a| \leq 0$  **by** *simp*  
**qed**

**lemma** *zero-less-abs-iff* [*simp*]:  $0 < |a| \iff a \neq 0$   
**by** (*simp add: less-le*)

**lemma** *abs-not-less-zero* [*simp*]:  $\neg |a| < 0$   
**proof** –  
 have  $a: \bigwedge x y. x \leq y \implies \neg y < x$  **by** *auto*  
 show *?thesis* **by** (*simp add: a*)  
**qed**

**lemma** *abs-ge-minus-self*:  $- a \leq |a|$   
**proof** –  
 have  $- a \leq |-a|$  **by** (*rule abs-ge-self*)  
 then show *?thesis* **by** *simp*  
**qed**

**lemma** *abs-minus-commute*:  
 $|a - b| = |b - a|$   
**proof** –  
 have  $|a - b| = |- (a - b)|$  **by** (*simp only: abs-minus-cancel*)  
 also have  $\dots = |b - a|$  **by** *simp*  
 finally show *?thesis* .  
**qed**

**lemma** *abs-of-pos*:  $0 < a \implies |a| = a$   
**by** (*rule abs-of-nonneg, rule less-imp-le*)

**lemma** *abs-of-nonpos* [*simp*]:  
 assumes  $a \leq 0$  shows  $|a| = - a$   
**proof** –  
 let  $?b = - a$   
 have  $- ?b \leq 0 \implies |- ?b| = - (- ?b)$   
 unfolding *abs-minus-cancel* [of  $?b$ ]  
 unfolding *neg-le-0-iff-le* [of  $?b$ ]  
 unfolding *minus-minus* **by** (*erule abs-of-nonneg*)  
 then show *?thesis* **using** *assms* **by** *auto*

qed

**lemma** *abs-of-neg*:  $a < 0 \implies |a| = -a$   
**by** (*rule abs-of-nonpos*, *rule less-imp-le*)

**lemma** *abs-le-D1*:  $|a| \leq b \implies a \leq b$   
**by** (*insert abs-ge-self*, *blast intro: order-trans*)

**lemma** *abs-le-D2*:  $|a| \leq b \implies -a \leq b$   
**by** (*insert abs-le-D1 [of - a]*, *simp*)

**lemma** *abs-le-iff*:  $|a| \leq b \iff a \leq b \wedge -a \leq b$   
**by** (*blast intro: abs-leI dest: abs-le-D1 abs-le-D2*)

**lemma** *abs-triangle-ineq2*:  $|a| - |b| \leq |a - b|$   
**proof** –  
 have  $|a| = |b + (a - b)|$   
   **by** (*simp add: algebra-simps add-diff-cancel*)  
 then have  $|a| \leq |b| + |a - b|$   
   **by** (*simp add: abs-triangle-ineq*)  
 then show *?thesis*  
   **by** (*simp add: algebra-simps*)

qed

**lemma** *abs-triangle-ineq2-sym*:  $|a| - |b| \leq |b - a|$   
**by** (*simp only: abs-minus-commute [of b] abs-triangle-ineq2*)

**lemma** *abs-triangle-ineq3*:  $||a| - |b|| \leq |a - b|$   
**by** (*simp add: abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym*)

**lemma** *abs-triangle-ineq4*:  $|a - b| \leq |a| + |b|$   
**proof** –  
 have  $|a - b| = |a + -b|$  **by** (*subst diff-minus*, *rule refl*)  
 also have  $\dots \leq |a| + |-b|$  **by** (*rule abs-triangle-ineq*)  
 finally show *?thesis* **by** *simp*

qed

**lemma** *abs-diff-triangle-ineq*:  $|a + b - (c + d)| \leq |a - c| + |b - d|$   
**proof** –  
 have  $|a + b - (c + d)| = |(a - c) + (b - d)|$  **by** (*simp add: diff-minus add-ac*)  
 also have  $\dots \leq |a - c| + |b - d|$  **by** (*rule abs-triangle-ineq*)  
 finally show *?thesis* .

qed

**lemma** *abs-add-abs [simp]*:  
 $||a| + |b|| = |a| + |b|$  (**is** *?L = ?R*)  
**proof** (*rule antisym*)  
 show  $?L \geq ?R$  **by** (*rule abs-ge-self*)  
**next**

```

have ?L ≤ ||a|| + ||b|| by (rule abs-triangle-ineq)
also have ... = ?R by simp
finally show ?L ≤ ?R .
qed

```

```

end

```

Needed for abelian cancellation simprocs:

```

lemma add-cancel-21: ((x::'a::ab-group-add) + (y + z) = y + u) = (x + z = u)
apply (subst add-left-commute)
apply (subst add-left-cancel)
apply simp
done

```

```

lemma add-cancel-end: (x + (y + z) = y) = (x = - (z::'a::ab-group-add))
apply (subst add-cancel-21 [of - - 0, simplified])
apply (simp add: add-right-cancel [symmetric, of x - z z, simplified])
done

```

```

lemma less-eqI: (x::'a::ordered-ab-group-add) - y = x' - y' ⇒ (x < y) = (x'
< y')
by (simp add: less-iff-diff-less-0 [of x y] less-iff-diff-less-0 [of x' y'])

```

```

lemma le-eqI: (x::'a::ordered-ab-group-add) - y = x' - y' ⇒ (y <= x) = (y'
<= x')
apply (simp add: le-iff-diff-le-0 [of y x] le-iff-diff-le-0 [of y' x'])
apply (simp add: neg-le-iff-le [symmetric, of y-x 0] neg-le-iff-le [symmetric, of
y'-x' 0])
done

```

```

lemma eq-eqI: (x::'a::ab-group-add) - y = x' - y' ⇒ (x = y) = (x' = y')
by (simp only: eq-iff-diff-eq-0 [of x y] eq-iff-diff-eq-0 [of x' y'])

```

```

lemma diff-def: (x::'a::ab-group-add) - y == x + (-y)
by (simp add: diff-minus)

```

```

lemma le-add-right-mono:
  assumes
    a <= b + (c::'a::ordered-ab-group-add)
    c <= d
  shows a <= b + d
  apply (rule-tac order-trans [where y = b+c])
  apply (simp-all add: prems)
  done

```

## 4.8 Tools setup

```

lemma add-mono-thms-linordered-semiring [no-atp]:
  fixes i j k :: 'a::ordered-ab-semigroup-add

```

shows  $i \leq j \wedge k \leq l \implies i + k \leq j + l$   
 and  $i = j \wedge k \leq l \implies i + k \leq j + l$   
 and  $i \leq j \wedge k = l \implies i + k \leq j + l$   
 and  $i = j \wedge k = l \implies i + k = j + l$   
 by (rule add-mono, clarify+)+

lemma add-mono-thms-linordered-field [no-atp]:  
 fixes  $i\ j\ k :: 'a::\text{ordered-cancel-ab-semigroup-add}$   
 shows  $i < j \wedge k = l \implies i + k < j + l$   
 and  $i = j \wedge k < l \implies i + k < j + l$   
 and  $i < j \wedge k \leq l \implies i + k < j + l$   
 and  $i \leq j \wedge k < l \implies i + k < j + l$   
 and  $i < j \wedge k < l \implies i + k < j + l$   
 by (auto intro: add-strict-right-mono add-strict-left-mono  
 add-less-le-mono add-le-less-mono add-strict-mono)

Simplification of  $x - y < (0::'a)$ , etc.

lemmas diff-less-0-iff-less [simp, no-atp] = less-iff-diff-less-0 [symmetric]  
 lemmas diff-le-0-iff-le [simp, no-atp] = le-iff-diff-le-0 [symmetric]

ML  $\ll$   
 structure ab-group-add-cancel = Abel-Cancel  
 (

(\* term order for abelian groups \*)

fun agrp-ord (Const (a, -)) = find-index (fn a' => a = a')  
 [@{const-name Groups.zero}, @{const-name Groups.plus},  
 @{const-name Groups.uminus}, @{const-name Groups.minus}]  
 | agrp-ord - = ~1;

fun termless-agrp (a, b) = (Term-Ord.term-lpo agrp-ord (a, b) = LESS);

local  
 val ac1 = mk-meta-eq @{thm add-associative};  
 val ac2 = mk-meta-eq @{thm add-commute};  
 val ac3 = mk-meta-eq @{thm add-left-commute};  
 fun solve-add-ac thy - (- \$ (Const (@{const-name Groups.plus},-) \$ - \$ -) \$ -) =  
 SOME ac1  
 | solve-add-ac thy - (- \$ x \$ (Const (@{const-name Groups.plus},-) \$ y \$ z)) =  
 if termless-agrp (y, x) then SOME ac3 else NONE  
 | solve-add-ac thy - (- \$ x \$ y) =  
 if termless-agrp (y, x) then SOME ac2 else NONE  
 | solve-add-ac thy - - = NONE  
in  
 val add-ac-proc = Simplifier.simproc @{theory}  
 add-ac-proc [x + y::'a::ab-semigroup-add] solve-add-ac;  
end;

```

val eq-reflection = @{thm eq-reflection};

val T = @{typ 'a::ab-group-add};

val cancel-ss = HOL-basic-ss settermless termless-agrp
  addsimprocs [add-ac-proc] addsimps
  [@{thm add-0-left}, @{thm add-0-right}, @{thm diff-def},
   @{thm minus-add-distrib}, @{thm minus-minus}, @{thm minus-zero},
   @{thm right-minus}, @{thm left-minus}, @{thm add-minus-cancel},
   @{thm minus-add-cancel}];

val sum-pats = [@{cterm x + y::'a::ab-group-add}, @{cterm x - y::'a::ab-group-add}];

val eqI-rules = [@{thm less-eqI}, @{thm le-eqI}, @{thm eq-eqI}];

val dest-eqI =
  fst o HOLogic.dest-bin @{const-name op =} HOLogic.boolT o HOLogic.dest-Trueprop
  o concl-of;

);
⟩⟩

ML ⟨⟨
  Addsimprocs [ab-group-add-cancel.sum-conv, ab-group-add-cancel.rel-conv];
⟩⟩

code-modulename SML
  Groups Arith

code-modulename OCaml
  Groups Arith

code-modulename Haskell
  Groups Arith

end

```

## 5 Lattices: Abstract lattices

```

theory Lattices
imports Orderings Groups
begin

```

### 5.1 Abstract semilattice

This locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may

occur due to interpretation.

```
locale semilattice = abel-semigroup +
  assumes idem [simp]:  $f\ a\ a = a$ 
begin
```

```
lemma left-idem [simp]:
   $f\ a\ (f\ a\ b) = f\ a\ b$ 
by (simp add: assoc [symmetric])
```

```
end
```

## 5.2 Idempotent semigroup

```
class ab-semigroup-idem-mult = ab-semigroup-mult +
  assumes mult-idem:  $x * x = x$ 
```

```
sublocale ab-semigroup-idem-mult < times!: semilattice times proof
qed (fact mult-idem)
```

```
context ab-semigroup-idem-mult
begin
```

```
lemmas mult-left-idem = times.left-idem
```

```
end
```

## 5.3 Concrete lattices

```
notation
```

```
  less-eq (infix  $\sqsubseteq$  50) and
  less (infix  $\sqsubset$  50) and
  top ( $\top$ ) and
  bot ( $\perp$ )
```

```
class semilattice-inf = order +
  fixes inf :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl  $\sqcap$  70)
  assumes inf-le1 [simp]:  $x \sqcap y \sqsubseteq x$ 
  and inf-le2 [simp]:  $x \sqcap y \sqsubseteq y$ 
  and inf-greatest:  $x \sqsubseteq y \Longrightarrow x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcap z$ 
```

```
class semilattice-sup = order +
  fixes sup :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl  $\sqcup$  65)
  assumes sup-ge1 [simp]:  $x \sqsubseteq x \sqcup y$ 
  and sup-ge2 [simp]:  $y \sqsubseteq x \sqcup y$ 
  and sup-least:  $y \sqsubseteq x \Longrightarrow z \sqsubseteq x \Longrightarrow y \sqcup z \sqsubseteq x$ 
begin
```

Dual lattice

```
lemma dual-semilattice:
```

```

class.semilattice-inf (op ≥) (op >) sup
by (rule class.semilattice-inf.intro, rule dual-order)
  (unfold-locales, simp-all add: sup-least)

end

class lattice = semilattice-inf + semilattice-sup

```

### 5.3.1 Intro and elim rules

```

context semilattice-inf
begin

```

```

lemma le-infI1:
  a ⊆ x ⟹ a ⊓ b ⊆ x
  by (rule order-trans) auto

```

```

lemma le-infI2:
  b ⊆ x ⟹ a ⊓ b ⊆ x
  by (rule order-trans) auto

```

```

lemma le-infI: x ⊆ a ⟹ x ⊆ b ⟹ x ⊆ a ⊓ b
  by (rule inf-greatest)

```

```

lemma le-infE: x ⊆ a ⊓ b ⟹ (x ⊆ a ⟹ x ⊆ b ⟹ P) ⟹ P
  by (blast intro: order-trans inf-le1 inf-le2)

```

```

lemma le-inf-iff [simp]:
  x ⊆ y ⊓ z ⟷ x ⊆ y ∧ x ⊆ z
  by (blast intro: le-infI elim: le-infE)

```

```

lemma le-iff-inf:
  x ⊆ y ⟷ x ⊓ y = x
  by (auto intro: le-infI1 antisym dest: eq-iff [THEN iffD1])

```

```

lemma inf-mono: a ⊆ c ⟹ b ≤ d ⟹ a ⊓ b ⊆ c ⊓ d
  by (fast intro: inf-greatest le-infI1 le-infI2)

```

```

lemma mono-inf:
  fixes f :: 'a ⇒ 'b::semilattice-inf
  shows mono f ⟹ f (A ⊓ B) ⊆ f A ⊓ f B
  by (auto simp add: mono-def intro: Lattices.inf-greatest)

```

```

end

```

```

context semilattice-sup
begin

```

```

lemma le-supI1:

```



```

 $x \sqsubseteq a \implies x \sqsubseteq a \sqcup b$ 
by (rule order-trans) auto

lemma le-supI2:
 $x \sqsubseteq b \implies x \sqsubseteq a \sqcup b$ 
by (rule order-trans) auto

lemma le-supI:
 $a \sqsubseteq x \implies b \sqsubseteq x \implies a \sqcup b \sqsubseteq x$ 
by (rule sup-least)

lemma le-supE:
 $a \sqcup b \sqsubseteq x \implies (a \sqsubseteq x \implies b \sqsubseteq x \implies P) \implies P$ 
by (blast intro: order-trans sup-ge1 sup-ge2)

lemma le-sup-iff [simp]:
 $x \sqcup y \sqsubseteq z \iff x \sqsubseteq z \wedge y \sqsubseteq z$ 
by (blast intro: le-supI elim: le-supE)

lemma le-iff-sup:
 $x \sqsubseteq y \iff x \sqcup y = y$ 
by (auto intro: le-supI2 antisym dest: eq-iff [THEN iffD1])

lemma sup-mono:  $a \sqsubseteq c \implies b \leq d \implies a \sqcup b \sqsubseteq c \sqcup d$ 
by (fast intro: sup-least le-supI1 le-supI2)

lemma mono-sup:
  fixes f :: 'a  $\Rightarrow$  'b::semilattice-sup
  shows mono f  $\implies f A \sqcup f B \sqsubseteq f (A \sqcup B)$ 
  by (auto simp add: mono-def intro: Lattices.sup-least)

end

5.3.2 Equational laws

sublocale semilattice-inf < inf!: semilattice inf
proof
  fix a b c
  show  $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$ 
    by (rule antisym) (auto intro: le-infI1 le-infI2)
  show  $a \sqcap b = b \sqcap a$ 
    by (rule antisym) auto
  show  $a \sqcap a = a$ 
    by (rule antisym) auto
qed

context semilattice-inf
begin

```

**lemma** *inf-assoc*:  $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$   
**by** (*fact inf.assoc*)

**lemma** *inf-commute*:  $(x \sqcap y) = (y \sqcap x)$   
**by** (*fact inf.commute*)

**lemma** *inf-left-commute*:  $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$   
**by** (*fact inf.left-commute*)

**lemma** *inf-idem*:  $x \sqcap x = x$   
**by** (*fact inf.idem*)

**lemma** *inf-left-idem*:  $x \sqcap (x \sqcap y) = x \sqcap y$   
**by** (*fact inf.left-idem*)

**lemma** *inf-absorb1*:  $x \sqsubseteq y \implies x \sqcap y = x$   
**by** (*rule antisym*) *auto*

**lemma** *inf-absorb2*:  $y \sqsubseteq x \implies x \sqcap y = y$   
**by** (*rule antisym*) *auto*

**lemmas** *inf-aci* = *inf-commute inf-assoc inf-left-commute inf-left-idem*

**end**

**sublocale** *semilattice-sup* < *sup!*: *semilattice sup*

**proof**

**fix** *a b c*

**show**  $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

**by** (*rule antisym*) (*auto intro: le-supI1 le-supI2*)

**show**  $a \sqcup b = b \sqcup a$

**by** (*rule antisym*) *auto*

**show**  $a \sqcup a = a$

**by** (*rule antisym*) *auto*

**qed**

**context** *semilattice-sup*

**begin**

**lemma** *sup-assoc*:  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$   
**by** (*fact sup.assoc*)

**lemma** *sup-commute*:  $(x \sqcup y) = (y \sqcup x)$   
**by** (*fact sup.commute*)

**lemma** *sup-left-commute*:  $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$   
**by** (*fact sup.left-commute*)

**lemma** *sup-idem*:  $x \sqcup x = x$

```

by (fact sup.idem)

lemma sup-left-idem:  $x \sqcup (x \sqcap y) = x \sqcup y$ 
by (fact sup.left-idem)

lemma sup-absorb1:  $y \sqsubseteq x \implies x \sqcup y = x$ 
by (rule antisym) auto

lemma sup-absorb2:  $x \sqsubseteq y \implies x \sqcup y = y$ 
by (rule antisym) auto

lemmas sup-aci = sup-commute sup-assoc sup-left-commute sup-left-idem

end

context lattice
begin

lemma dual-lattice:
  class.lattice (op  $\geq$ ) (op  $>$ ) sup inf
  by (rule class.lattice.intro, rule dual-semilattice, rule class.semilattice-sup.intro,
    rule dual-order)
    (unfold-locales, auto)

lemma inf-sup-absorb:  $x \sqcap (x \sqcup y) = x$ 
by (blast intro: antisym inf-le1 inf-greatest sup-ge1)

lemma sup-inf-absorb:  $x \sqcup (x \sqcap y) = x$ 
by (blast intro: antisym sup-ge1 sup-least inf-le1)

lemmas inf-sup-aci = inf-aci sup-aci

lemmas inf-sup-ord = inf-le1 inf-le2 sup-ge1 sup-ge2

Towards distributivity

lemma distrib-sup-le:  $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$ 
by (auto intro: le-infI1 le-infI2 le-supI1 le-supI2)

lemma distrib-inf-le:  $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$ 
by (auto intro: le-infI1 le-infI2 le-supI1 le-supI2)

If you have one of them, you have them all.

lemma distrib-imp1:
assumes D:  $\forall x y z. x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ 
shows  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ 
proof–
  have  $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$  by(simp add:sup-inf-absorb)
  also have  $\dots = x \sqcup (z \sqcap (x \sqcup y))$  by(simp add:D inf-commute sup-assoc)
  also have  $\dots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$ 

```

```

    by(simp add:inf-sup-absorb inf-commute)
    also have ... = (x  $\sqcup$  y)  $\sqcap$  (x  $\sqcup$  z) by(simp add:D)
    finally show ?thesis .
qed

```

**lemma** *distrib-imp2*:

**assumes** *D*:  $\forall x y z. x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

**shows**  $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$

**proof** –

have  $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$  by(simp add:inf-sup-absorb)

also have ... =  $x \sqcap (z \sqcup (x \sqcap y))$  by(simp add:D sup-commute inf-assoc)

also have ... =  $((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$

by(simp add:sup-inf-absorb sup-commute)

also have ... =  $(x \sqcap y) \sqcup (x \sqcap z)$  by(simp add:D)

finally show ?thesis .

qed

end

### 5.3.3 Strict order

**context** *semilattice-inf*

**begin**

**lemma** *less-infI1*:

$a \sqsubset x \implies a \sqcap b \sqsubset x$

by (auto simp add: less-le inf-absorb1 intro: le-infI1)

**lemma** *less-infI2*:

$b \sqsubset x \implies a \sqcap b \sqsubset x$

by (auto simp add: less-le inf-absorb2 intro: le-infI2)

end

**context** *semilattice-sup*

**begin**

**lemma** *less-supI1*:

$x \sqsubset a \implies x \sqsubset a \sqcup b$

**proof** –

**interpret** *dual*: *semilattice-inf*  $op \geq op > sup$

by (fact dual-semilattice)

**assume**  $x \sqsubset a$

**then show**  $x \sqsubset a \sqcup b$

by (fact dual.less-infI1)

qed

**lemma** *less-supI2*:

$x \sqsubset b \implies x \sqsubset a \sqcup b$

```

proof –
  interpret dual: semilattice-inf op ≥ op > sup
    by (fact dual-semilattice)
  assume  $x \sqsubseteq b$ 
  then show  $x \sqsubseteq a \sqcup b$ 
    by (fact dual.less-infI2)
qed

end

```

## 5.4 Distributive lattices

```

class distrib-lattice = lattice +
  assumes sup-inf-distrib1:  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ 

```

```

context distrib-lattice
begin

```

```

lemma sup-inf-distrib2:
   $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$ 
by(simp add: inf-sup-aci sup-inf-distrib1)

```

```

lemma inf-sup-distrib1:
   $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ 
by(rule distrib-imp2[OF sup-inf-distrib1])

```

```

lemma inf-sup-distrib2:
   $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$ 
by(simp add: inf-sup-aci inf-sup-distrib1)

```

```

lemma dual-distrib-lattice:
  class.distrib-lattice (op ≥) (op >) sup inf
  by (rule class.distrib-lattice.intro, rule dual-lattice)
    (unfold-locals, fact inf-sup-distrib1)

```

```

lemmas sup-inf-distrib =
  sup-inf-distrib1 sup-inf-distrib2

```

```

lemmas inf-sup-distrib =
  inf-sup-distrib1 inf-sup-distrib2

```

```

lemmas distrib =
  sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

```

```

end

```

## 5.5 Bounded lattices and boolean algebras

```

class bounded-lattice-bot = lattice + bot
begin

```

```

lemma inf-bot-left [simp]:
   $\perp \sqcap x = \perp$ 
  by (rule inf-absorb1) simp

lemma inf-bot-right [simp]:
   $x \sqcap \perp = \perp$ 
  by (rule inf-absorb2) simp

lemma sup-bot-left [simp]:
   $\perp \sqcup x = x$ 
  by (rule sup-absorb2) simp

lemma sup-bot-right [simp]:
   $x \sqcup \perp = x$ 
  by (rule sup-absorb1) simp

lemma sup-eq-bot-iff [simp]:
   $x \sqcup y = \perp \iff x = \perp \wedge y = \perp$ 
  by (simp add: eq-iff)

end

class bounded-lattice-top = lattice + top
begin

lemma sup-top-left [simp]:
   $\top \sqcup x = \top$ 
  by (rule sup-absorb1) simp

lemma sup-top-right [simp]:
   $x \sqcup \top = \top$ 
  by (rule sup-absorb2) simp

lemma inf-top-left [simp]:
   $\top \sqcap x = x$ 
  by (rule inf-absorb2) simp

lemma inf-top-right [simp]:
   $x \sqcap \top = x$ 
  by (rule inf-absorb1) simp

lemma inf-eq-top-iff [simp]:
   $x \sqcap y = \top \iff x = \top \wedge y = \top$ 
  by (simp add: eq-iff)

end

class bounded-lattice = bounded-lattice-bot + bounded-lattice-top

```

**begin**

**lemma** *dual-bounded-lattice*:

*class.bounded-lattice* (*op*  $\geq$ ) (*op*  $>$ ) (*op*  $\sqcup$ ) (*op*  $\sqcap$ )  $\top$   $\perp$   
**by** *unfold-locales* (*auto simp add: less-le-not-le*)

**end**

**class** *boolean-algebra* = *distrib-lattice* + *bounded-lattice* + *minus* + *uminus* +

**assumes** *inf-compl-bot*:  $x \sqcap -x = \perp$

**and** *sup-compl-top*:  $x \sqcup -x = \top$

**assumes** *diff-eq*:  $x - y = x \sqcap -y$

**begin**

**lemma** *dual-boolean-algebra*:

*class.boolean-algebra* ( $\lambda x y. x \sqcup -y$ ) *uminus* (*op*  $\geq$ ) (*op*  $>$ ) (*op*  $\sqcup$ ) (*op*  $\sqcap$ )  $\top$   $\perp$

**by** (*rule class.boolean-algebra.intro*, *rule dual-bounded-lattice*, *rule dual-distrib-lattice*)  
(*unfold-locales*, *auto simp add: inf-compl-bot sup-compl-top diff-eq*)

**lemma** *compl-inf-bot*:

$-x \sqcap x = \perp$

**by** (*simp add: inf-commute inf-compl-bot*)

**lemma** *compl-sup-top*:

$-x \sqcup x = \top$

**by** (*simp add: sup-commute sup-compl-top*)

**lemma** *compl-unique*:

**assumes**  $x \sqcap y = \perp$

**and**  $x \sqcup y = \top$

**shows**  $-x = y$

**proof** –

**have**  $(x \sqcap -x) \sqcup (-x \sqcap y) = (x \sqcap y) \sqcup (-x \sqcap y)$

**using** *inf-compl-bot* *assms(1)* **by** *simp*

**then have**  $(-x \sqcap x) \sqcup (-x \sqcap y) = (y \sqcap x) \sqcup (y \sqcap -x)$

**by** (*simp add: inf-commute*)

**then have**  $-x \sqcap (x \sqcup y) = y \sqcap (x \sqcup -x)$

**by** (*simp add: inf-sup-distrib1*)

**then have**  $-x \sqcap \top = y \sqcap \top$

**using** *sup-compl-top* *assms(2)* **by** *simp*

**then show**  $-x = y$  **by** *simp*

**qed**

**lemma** *double-compl* [*simp*]:

$-(-x) = x$

**using** *compl-inf-bot* *compl-sup-top* **by** (*rule compl-unique*)

**lemma** *compl-eq-compl-iff* [*simp*]:

$-x = -y \longleftrightarrow x = y$

```

proof
  assume  $\neg x = \neg y$ 
  then have  $\neg(\neg x) = \neg(\neg y)$  by (rule arg-cong)
  then show  $x = y$  by simp
next
  assume  $x = y$ 
  then show  $\neg x = \neg y$  by simp
qed

lemma compl-bot-eq [simp]:
   $\neg \perp = \top$ 
proof  $\neg$ 
  from sup-compl-top have  $\perp \sqcup \neg \perp = \top$  .
  then show ?thesis by simp
qed

lemma compl-top-eq [simp]:
   $\neg \top = \perp$ 
proof  $\neg$ 
  from inf-compl-bot have  $\top \sqcap \neg \top = \perp$  .
  then show ?thesis by simp
qed

lemma compl-inf [simp]:
   $\neg(x \sqcap y) = \neg x \sqcup \neg y$ 
proof (rule compl-unique)
  have  $(x \sqcap y) \sqcap (\neg x \sqcup \neg y) = (y \sqcap (x \sqcap \neg x)) \sqcup (x \sqcap (y \sqcap \neg y))$ 
    by (simp only: inf-sup-distrib inf-aci)
  then show  $(x \sqcap y) \sqcap (\neg x \sqcup \neg y) = \perp$ 
    by (simp add: inf-compl-bot)
next
  have  $(x \sqcap y) \sqcup (\neg x \sqcup \neg y) = (\neg y \sqcup (x \sqcup \neg x)) \sqcap (\neg x \sqcup (y \sqcup \neg y))$ 
    by (simp only: sup-inf-distrib sup-aci)
  then show  $(x \sqcap y) \sqcup (\neg x \sqcup \neg y) = \top$ 
    by (simp add: sup-compl-top)
qed

lemma compl-sup [simp]:
   $\neg(x \sqcup y) = \neg x \sqcap \neg y$ 
proof  $\neg$ 
  interpret boolean-algebra  $\lambda x y. x \sqcup \neg y$  uminus op  $\geq$  op  $>$  op  $\sqcup$  op  $\sqcap$   $\top$   $\perp$ 
    by (rule dual-boolean-algebra)
  then show ?thesis by simp
qed

lemma compl-mono:
   $x \sqsubseteq y \implies \neg y \sqsubseteq \neg x$ 
proof  $\neg$ 
  assume  $x \sqsubseteq y$ 

```



```

then have  $x \sqcup y = y$  by (simp only: le-iff-sup)
then have  $\neg (x \sqcup y) = \neg y$  by simp
then have  $\neg x \sqcap \neg y = \neg y$  by simp
then have  $\neg y \sqcap \neg x = \neg y$  by (simp only: inf-commute)
then show  $\neg y \sqsubseteq \neg x$  by (simp only: le-iff-inf)
qed

```

```

lemma compl-le-compl-iff:
   $\neg x \leq \neg y \iff y \leq x$ 
by (auto dest: compl-mono)

```

```

end

```

## 5.6 Uniqueness of inf and sup

```

lemma (in semilattice-inf) inf-unique:
  fixes f (infixl  $\triangle$  70)
  assumes le1:  $\bigwedge x y. x \triangle y \sqsubseteq x$  and le2:  $\bigwedge x y. x \triangle y \sqsubseteq y$ 
  and greatest:  $\bigwedge x y z. x \sqsubseteq y \implies x \sqsubseteq z \implies x \sqsubseteq y \triangle z$ 
  shows  $x \sqcap y = x \triangle y$ 
proof (rule antisym)
  show  $x \triangle y \sqsubseteq x \sqcap y$  by (rule le-infI) (rule le1, rule le2)
next
  have leI:  $\bigwedge x y z. x \sqsubseteq y \implies x \sqsubseteq z \implies x \sqsubseteq y \triangle z$  by (blast intro: greatest)
  show  $x \sqcap y \sqsubseteq x \triangle y$  by (rule leI) simp-all
qed

```

```

lemma (in semilattice-sup) sup-unique:
  fixes f (infixl  $\nabla$  70)
  assumes ge1 [simp]:  $\bigwedge x y. x \sqsubseteq x \nabla y$  and ge2:  $\bigwedge x y. y \sqsubseteq x \nabla y$ 
  and least:  $\bigwedge x y z. y \sqsubseteq x \implies z \sqsubseteq x \implies y \nabla z \sqsubseteq x$ 
  shows  $x \sqcup y = x \nabla y$ 
proof (rule antisym)
  show  $x \sqcup y \sqsubseteq x \nabla y$  by (rule le-supI) (rule ge1, rule ge2)
next
  have leI:  $\bigwedge x y z. x \sqsubseteq z \implies y \sqsubseteq z \implies x \nabla y \sqsubseteq z$  by (blast intro: least)
  show  $x \nabla y \sqsubseteq x \sqcup y$  by (rule leI) simp-all
qed

```

## 5.7 min/max on linear orders as special case of $op \sqcap / op \sqcup$

```

sublocale linorder < min-max!: distrib-lattice less-eq less min max

```

```

proof

```

```

  fix x y z
  show  $\max x (\min y z) = \min (\max x y) (\max x z)$ 
  by (auto simp add: min-def max-def)
qed (auto simp add: min-def max-def not-le less-imp-le)

```

```

lemma inf-min:  $\inf = (\min :: 'a :: \{\text{semilattice-inf, linorder}\} \Rightarrow 'a \Rightarrow 'a)$ 
  by (rule ext)+ (auto intro: antisym)

```

**lemma** *sup-max*:  $\text{sup} = (\text{max} :: 'a :: \{\text{semilattice-sup}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$   
**by** (*rule ext*) + (*auto intro: antisym*)

**lemmas** *le-maxI1* = *min-max.sup-ge1*

**lemmas** *le-maxI2* = *min-max.sup-ge2*

**lemmas** *min-ac* = *min-max.inf-assoc min-max.inf-commute*  
*min-max.inf.left-commute*

**lemmas** *max-ac* = *min-max.sup-assoc min-max.sup-commute*  
*min-max.sup.left-commute*

## 5.8 Bool as lattice

**instantiation** *bool* :: *boolean-algebra*  
**begin**

**definition**  
*bool-Compl-def*:  $\text{uminus} = \text{Not}$

**definition**  
*bool-diff-def*:  $A - B \longleftrightarrow A \wedge \neg B$

**definition**  
*inf-bool-eq*:  $P \sqcap Q \longleftrightarrow P \wedge Q$

**definition**  
*sup-bool-eq*:  $P \sqcup Q \longleftrightarrow P \vee Q$

**instance proof**  
**qed** (*simp-all add: inf-bool-eq sup-bool-eq le-bool-def*  
*bot-bool-eq top-bool-eq bool-Compl-def bool-diff-def, auto*)

**end**

**lemma** *sup-boolI1*:  
 $P \Longrightarrow P \sqcup Q$   
**by** (*simp add: sup-bool-eq*)

**lemma** *sup-boolI2*:  
 $Q \Longrightarrow P \sqcup Q$   
**by** (*simp add: sup-bool-eq*)

**lemma** *sup-boolE*:  
 $P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$   
**by** (*auto simp add: sup-bool-eq*)

## 5.9 Fun as lattice

**instantiation** *fun* :: (*type*, *lattice*) *lattice*  
**begin**

**definition**

*inf-fun-eq* [*code del*]:  $f \sqcap g = (\lambda x. f\ x \sqcap g\ x)$

**definition**

*sup-fun-eq* [*code del*]:  $f \sqcup g = (\lambda x. f\ x \sqcup g\ x)$

**instance proof**

**qed** (*simp-all add: le-fun-def inf-fun-eq sup-fun-eq*)

**end**

**instance** *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*

**proof**

**qed** (*simp-all add: inf-fun-eq sup-fun-eq sup-inf-distrib1*)

**instance** *fun* :: (*type*, *bounded-lattice*) *bounded-lattice* ..

**instantiation** *fun* :: (*type*, *uminus*) *uminus*

**begin**

**definition**

*fun-Compl-def*:  $-\ A = (\lambda x. -\ A\ x)$

**instance** ..

**end**

**instantiation** *fun* :: (*type*, *minus*) *minus*

**begin**

**definition**

*fun-diff-def*:  $A - B = (\lambda x. A\ x - B\ x)$

**instance** ..

**end**

**instance** *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*

**proof**

**qed** (*simp-all add: inf-fun-eq sup-fun-eq bot-fun-eq top-fun-eq fun-Compl-def fun-diff-def  
inf-compl-bot sup-compl-top diff-eq*)

**no-notation**

*less-eq* (**infix**  $\sqsubseteq$  50) **and**

```

less (infix  $\sqsubset$  50) and
inf (infixl  $\sqcap$  70) and
sup (infixl  $\sqcup$  65) and
top ( $\top$ ) and
bot ( $\perp$ )

end

```

## 6 Set: Set theory for higher-order logic

```

theory Set
imports Lattices
begin

```

### 6.1 Sets as predicates

```
global
```

```
types 'a set = 'a => bool
```

```
consts
```

```

Collect      :: ('a => bool) => 'a set      — comprehension
op :         :: 'a => 'a set => bool        — membership

```

```
local
```

```
notation
```

```

op : (op :) and
op : ((-/ : -) [50, 51] 50)

```

```
defs
```

```

mem-def [code]: x : S == S x
Collect-def [code]: Collect P == P

```

```
abbreviation
```

```
not-mem x A ==  $\sim$  (x : A) — non-membership
```

```
notation
```

```

not-mem (op  $\sim$ :) and
not-mem ((-/  $\sim$ : -) [50, 51] 50)

```

```
notation (xsymbols)
```

```

op : (op  $\in$ ) and
op : ((-/  $\in$  -) [50, 51] 50) and
not-mem (op  $\notin$ ) and
not-mem ((-/  $\notin$  -) [50, 51] 50)

```

```
notation (HTML output)
```

```

op : (op ∈) and
op : ((-/ ∈ -) [50, 51] 50) and
not-mem (op ∉) and
not-mem ((-/ ∉ -) [50, 51] 50)

```

Set comprehensions

**syntax**

```
-Coll :: ptn => bool => 'a set  ((1{-./ -})
```

**translations**

```
{x. P} == CONST Collect (%x. P)
```

**syntax**

```
-Collect :: idt => 'a set => bool => 'a set  ((1{- ./ -/ -})
```

**syntax** (*xsymbols*)

```
-Collect :: idt => 'a set => bool => 'a set  ((1{- ∈/ -/ -})
```

**translations**

```
{x:A. P} => {x. x:A & P}
```

**lemma** *mem-Collect-eq [iff]*:  $(a : \{x. P(x)\}) = P(a)$

**by** (*simp add: Collect-def mem-def*)

**lemma** *Collect-mem-eq [simp]*:  $\{x. x:A\} = A$

**by** (*simp add: Collect-def mem-def*)

**lemma** *CollectI*:  $P(a) ==> a : \{x. P(x)\}$

**by** *simp*

**lemma** *CollectD*:  $a : \{x. P(x)\} ==> P(a)$

**by** *simp*

**lemma** *Collect-cong*:  $(!!x. P\ x = Q\ x) ==> \{x. P(x)\} = \{x. Q(x)\}$

**by** *simp*

Simproc for pulling  $x=t$  in  $\{x. \dots \& x=t \& \dots\}$  to the front (and similarly for  $t=x$ ):

**setup**  $\ll$

*let*

```

val Coll-perm-tac = rtac @{thm Collect-cong} 1 THEN rtac @{thm iffI} 1 THEN
  ALLGOALS(EVERY'[REPEAT-DETERM o (etac @{thm conjE}),
    DEPTH-SOLVE-1 o (ares-tac [@{thm conjI}])])

```

```
val defColl-regroup = Simplifier.simproc @{theory}
```

```
  defined Collect [{x. P x & Q x}]
```

```
  (Quantifier1.rearrange-Coll Coll-perm-tac)
```

*in*

```
Simplifier.map-simpset (fn ss => ss addsimprocs [defColl-regroup])
```

*end*

$\gg$

**lemmas** *CollectE* = *CollectD* [*elim-format*]

Set enumerations

**abbreviation** *empty* :: 'a set ({} ) **where**  
 {}  $\equiv$  bot

**definition** *insert* :: 'a  $\Rightarrow$  'a set  $\Rightarrow$  'a set **where**  
*insert-compr*: *insert* a B = {x. x = a  $\vee$  x  $\in$  B}

**syntax**

-*Finset* :: args  $\Rightarrow$  'a set ({}(-))

**translations**

{x, xs} == CONST *insert* x {xs}  
 {x} == CONST *insert* x {}

## 6.2 Subsets and bounded quantifiers

**abbreviation**

*subset* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*subset*  $\equiv$  less

**abbreviation**

*subset-eq* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*subset-eq*  $\equiv$  less-eq

**notation (output)**

*subset* (op <) **and**  
*subset* ((-/ < -) [50, 51] 50) **and**  
*subset-eq* (op <=) **and**  
*subset-eq* ((-/ <= -) [50, 51] 50)

**notation (xsymbols)**

*subset* (op  $\subset$ ) **and**  
*subset* ((-/  $\subset$  -) [50, 51] 50) **and**  
*subset-eq* (op  $\subseteq$ ) **and**  
*subset-eq* ((-/  $\subseteq$  -) [50, 51] 50)

**notation (HTML output)**

*subset* (op  $\subset$ ) **and**  
*subset* ((-/  $\subset$  -) [50, 51] 50) **and**  
*subset-eq* (op  $\subseteq$ ) **and**  
*subset-eq* ((-/  $\subseteq$  -) [50, 51] 50)

**abbreviation (input)**

*supset* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*supset*  $\equiv$  greater

**abbreviation (input)**

*supset-eq* :: 'a set  $\Rightarrow$  'a set  $\Rightarrow$  bool **where**  
*supset-eq*  $\equiv$  greater-eq

**notation** (*xsymbols*)

*supset* (*op*  $\supset$ ) **and**  
*supset* ((*-*  $\supset$  *-*) [50, 51] 50) **and**  
*supset-eq* (*op*  $\supseteq$ ) **and**  
*supset-eq* ((*-*  $\supseteq$  *-*) [50, 51] 50)

**global****consts**

*Ball* :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool — bounded universal quantifiers

*Bex* :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  bool — bounded existential quantifiers

**local****defs**

*Ball-def*: *Ball* A P == *ALL* x. x:A  $\longrightarrow$  P(x)  
*Bex-def*: *Bex* A P == *EX* x. x:A & P(x)

**syntax**

-*Ball* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3ALL*  $\vdash$  *-* *-*) [0, 0, 10] 10)  
-*Bex* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3EX*  $\vdash$  *-* *-*) [0, 0, 10] 10)  
-*Bex1* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3EX!*  $\vdash$  *-* *-*) [0, 0, 10] 10)  
-*Bleat* :: *id*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  'a ((*3LEAST*  $\vdash$  *-* *-*) [0, 0, 10] 10)

**syntax** (*HOL*)

-*Ball* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3!*  $\vdash$  *-* *-*) [0, 0, 10] 10)  
-*Bex* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3?*  $\vdash$  *-* *-*) [0, 0, 10] 10)  
-*Bex1* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3?!*  $\vdash$  *-* *-*) [0, 0, 10] 10)

**syntax** (*xsymbols*)

-*Ball* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3 $\forall$*   $\dashv$  *-* *-*) [0, 0, 10] 10)  
-*Bex* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3 $\exists$*   $\dashv$  *-* *-*) [0, 0, 10] 10)  
-*Bex1* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3 $\exists!$*   $\dashv$  *-* *-*) [0, 0, 10] 10)  
-*Bleat* :: *id*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  'a ((*3LEAST*  $\dashv$  *-* *-*) [0, 0, 10] 10)

**syntax** (*HTML output*)

-*Ball* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3 $\forall$*   $\dashv$  *-* *-*) [0, 0, 10] 10)  
-*Bex* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3 $\exists$*   $\dashv$  *-* *-*) [0, 0, 10] 10)  
-*Bex1* :: *pttrn*  $\Rightarrow$  'a set  $\Rightarrow$  bool  $\Rightarrow$  bool ((*3 $\exists!$*   $\dashv$  *-* *-*) [0, 0, 10] 10)

**translations**

*ALL* x:A. P == *CONST* *Ball* A (%x. P)  
*EX* x:A. P == *CONST* *Bex* A (%x. P)  
*EX!* x:A. P  $\Rightarrow$  *EX!* x. x:A & P  
*LEAST* x:A. P  $\Rightarrow$  *LEAST* x. x:A & P

**syntax (output)**

```

-setlessAll :: [idt, 'a, bool] => bool ((3ALL -<-./ -) [0, 0, 10] 10)
-setlessEx  :: [idt, 'a, bool] => bool ((3EX -<-./ -) [0, 0, 10] 10)
-settleAll  :: [idt, 'a, bool] => bool ((3ALL -<=.-./ -) [0, 0, 10] 10)
-settleEx   :: [idt, 'a, bool] => bool ((3EX -<=.-./ -) [0, 0, 10] 10)
-settleEx1  :: [idt, 'a, bool] => bool ((3EX! -<=.-./ -) [0, 0, 10] 10)

```

**syntax (xsymbols)**

```

-setlessAll :: [idt, 'a, bool] => bool ((3∀ -C-./ -) [0, 0, 10] 10)
-setlessEx  :: [idt, 'a, bool] => bool ((3∃ -C-./ -) [0, 0, 10] 10)
-settleAll  :: [idt, 'a, bool] => bool ((3∀ -⊆-./ -) [0, 0, 10] 10)
-settleEx   :: [idt, 'a, bool] => bool ((3∃ -⊆-./ -) [0, 0, 10] 10)
-settleEx1  :: [idt, 'a, bool] => bool ((3∃! -⊆-./ -) [0, 0, 10] 10)

```

**syntax (HOL output)**

```

-setlessAll :: [idt, 'a, bool] => bool ((3! -<-./ -) [0, 0, 10] 10)
-setlessEx  :: [idt, 'a, bool] => bool ((3? -<-./ -) [0, 0, 10] 10)
-settleAll  :: [idt, 'a, bool] => bool ((3! -<=.-./ -) [0, 0, 10] 10)
-settleEx   :: [idt, 'a, bool] => bool ((3? -<=.-./ -) [0, 0, 10] 10)
-settleEx1  :: [idt, 'a, bool] => bool ((3?! -<=.-./ -) [0, 0, 10] 10)

```

**syntax (HTML output)**

```

-setlessAll :: [idt, 'a, bool] => bool ((3∀ -C-./ -) [0, 0, 10] 10)
-setlessEx  :: [idt, 'a, bool] => bool ((3∃ -C-./ -) [0, 0, 10] 10)
-settleAll  :: [idt, 'a, bool] => bool ((3∀ -⊆-./ -) [0, 0, 10] 10)
-settleEx   :: [idt, 'a, bool] => bool ((3∃ -⊆-./ -) [0, 0, 10] 10)
-settleEx1  :: [idt, 'a, bool] => bool ((3∃! -⊆-./ -) [0, 0, 10] 10)

```

**translations**

```

∀ A ⊂ B. P  =>  ALL A. A ⊂ B --> P
∃ A ⊂ B. P  =>  EX A. A ⊂ B & P
∀ A ⊆ B. P  =>  ALL A. A ⊆ B --> P
∃ A ⊆ B. P  =>  EX A. A ⊆ B & P
∃! A ⊆ B. P =>  EX! A. A ⊆ B & P

```

**print-translation <<**

let

```

val Type (set-type, -) = @{typ 'a set}; (* FIXME 'a => bool (!?) *)
val All-binder = Syntax.binder-name @{const-syntax All};
val Ex-binder  = Syntax.binder-name @{const-syntax Ex};
val impl = @{const-syntax op -->};
val conj = @{const-syntax op &};
val sbset = @{const-syntax subset};
val sbset-eq = @{const-syntax subset-eq};

```

val trans =

```

(((All-binder, impl, sbset), @{syntax-const -setlessAll}),
 ((All-binder, impl, sbset-eq), @{syntax-const -settleAll}),

```



```

((Ex-binder, conj, sbset), @{syntax-const -setlessEx}),
((Ex-binder, conj, sbset-eq), @{syntax-const -setleEx}]);

fun mk v v' c n P =
  if v = v' andalso not (Term.exists-subterm (fn Free (x, -) => x = v | - =>
false) n)
  then Syntax.const c $ Syntax.mark-bound v' $ n $ P else raise Match;

fun tr' q = (q,
  fn [Const (@{syntax-const -bound}, -) $ Free (v, Type (T, -)),
    Const (c, -) $
      (Const (d, -) $ (Const (@{syntax-const -bound}, -) $ Free (v', -)) $ n)
  $ P] =>
  if T = set-type then
    (case AList.lookup (op =) trans (q, c, d) of
      NONE => raise Match
    | SOME l => mk v v' l n P)
    else raise Match
  | - => raise Match);
in
  [tr' All-binder, tr' Ex-binder]
end
>>

```

Translate between  $\{e \mid x1...xn. P\}$  and  $\{u. EX\ x1..xn. u = e \ \& \ P\}$ ;  $\{y. EX\ x1..xn. y = e \ \& \ P\}$  is only translated if  $[0..n]$  subset  $bvs(e)$ .

#### **syntax**

-Setcompr :: 'a => idts => bool => 'a set ((1{- |./-/-})

#### **parse-translation** <<

```

let
  val ex-tr = snd (mk-binder-tr (EX , @{const-syntax Ex}));

  fun nvars (Const (@{syntax-const -idts}, -) $ - $ idts) = nvars idts + 1
    | nvars - = 1;

  fun setcompr-tr [e, idts, b] =
    let
      val eq = Syntax.const @{const-syntax op =} $ Bound (nvars idts) $ e;
      val P = Syntax.const @{const-syntax op &} $ eq $ b;
      val exP = ex-tr [idts, P];
    in Syntax.const @{const-syntax Collect} $ Term.absdummy (dummyT, exP)
    end;

  in [(@{syntax-const -Setcompr}, setcompr-tr)] end;
>>

```

#### **print-translation** <<

[Syntax.preserve-binder-abs2-tr' @{const-syntax Ball} @{syntax-const -Ball},

*Syntax.preserve-binder-abs2-tr' @ {const-syntax Bex} @ {syntax-const -Bex}*]  
 >> — to avoid eta-contraction of body

**print-translation** <<

let

val *ex-tr'* = *snd* (*mk-binder-tr'* (@ {const-syntax *Ex*}, *DUMMY*));

fun *setcompr-tr'* [*Abs* (*abs as* (-, -, *P*))] =

let

fun *check* (*Const* (@ {const-syntax *Ex*}, -) \$ *Abs* (-, -, *P*), *n*) = *check* (*P*, *n* + 1)

| *check* (*Const* (@ {const-syntax *op* &}, -) \$  
     (*Const* (@ {const-syntax *op* =}, -) \$ *Bound m* \$ *e*) \$ *P*, *n*) =  
     *n* > 0 andalso *m* = *n* andalso not (*loose-bvar1* (*P*, *n*)) andalso  
     subset (*op* =) (0 upto (*n* - 1), *add-loose-bnos* (*e*, 0, []))  
 | *check* - = false;

fun *tr'* (- \$ *abs*) =

let val - \$ *idts* \$ (- \$ (- \$ - \$ *e*) \$ *Q*) = *ex-tr'* [*abs*]

in *Syntax.const* @ {syntax-const -Setcompr} \$ *e* \$ *idts* \$ *Q* end;

in

if *check* (*P*, 0) then *tr' P*

else

let

val (*x as* - \$ *Free*(*xN*, -), *t*) = *atomic-abs-tr'* *abs*;

val *M* = *Syntax.const* @ {syntax-const -Coll} \$ *x* \$ *t*;

in

case *t* of

*Const* (@ {const-syntax *op* &}, -) \$

(*Const* (@ {const-syntax *op* :}, -) \$

(*Const* (@ {syntax-const -bound}, -) \$ *Free* (*yN*, -)) \$ *A*) \$ *P* =>

if *xN* = *yN* then *Syntax.const* @ {syntax-const -Collect} \$ *x* \$ *A* \$ *P* else

*M*

| - => *M*

end

end;

in [(@ {const-syntax *Collect*}, *setcompr-tr'*)] end;

>>

**setup** <<

let

val *unfold-bex-tac* = *unfold-tac* @ {thms *Bex-def*};

fun *prove-bex-tac ss* = *unfold-bex-tac ss THEN Quantifier1.prove-one-point-ex-tac*;

val *rearrange-bex* = *Quantifier1.rearrange-bex prove-bex-tac*;

val *unfold-ball-tac* = *unfold-tac* @ {thms *Ball-def*};

fun *prove-ball-tac ss* = *unfold-ball-tac ss THEN Quantifier1.prove-one-point-all-tac*;

val *rearrange-ball* = *Quantifier1.rearrange-ball prove-ball-tac*;

val *defBEX-regroup* = *Simplifier.simproc* @ {theory}

defined *BEX* [*EX x:A. P x* & *Q x*] *rearrange-bex*;

```

    val defBALL-regroup = Simplifier.simproc @{theory}
      defined BALL [ALL x:A. P x ==> Q x] rearrange-ball;
  in
    Simplifier.map-simpset (fn ss => ss addsimprocs [defBALL-regroup, defBEX-regroup])
  end
  >>

```

```

lemma ballI [intro!]: (!x. x:A ==> P x) ==> ALL x:A. P x
  by (simp add: Ball-def)

```

```

lemmas strip = impI allI ballI

```

```

lemma bspec [dest?]: ALL x:A. P x ==> x:A ==> P x
  by (simp add: Ball-def)

```

Gives better instantiation for bound:

```

declaration << fn - ==>
  Classical.map-cs (fn cs => cs addbefore (bspec, datac @{thm bspec} 1))
  >>

```

```

ML <<
  structure Simpdata =
  struct

```

```

    open Simpdata;

```

```

    val mksimps-pairs = [(@{const-name Ball}, @{thms bspec})] @ mksimps-pairs;

```

```

    end;

```

```

    open Simpdata;
  >>

```

```

declaration << fn - ==>
  Simplifier.map-ss (fn ss => ss setmksimps (mksimps mksimps-pairs))
  >>

```

```

lemma ballE [elim]: ALL x:A. P x ==> (P x ==> Q) ==> (x ~: A ==> Q)
  ==> Q
  by (unfold Ball-def) blast

```

```

lemma bexI [intro]: P x ==> x:A ==> EX x:A. P x
  — Normally the best argument order: P x constrains the choice of x ∈ A.
  by (unfold Bex-def) blast

```

```

lemma rev-bexI [intro?]: x:A ==> P x ==> EX x:A. P x
  — The best argument order when there is only one x ∈ A.
  by (unfold Bex-def) blast

```

**lemma** *bexCI*:  $(\text{ALL } x:A. \sim P x \implies P a) \implies a:A \implies \text{EX } x:A. P x$   
**by** (*unfold Bex-def*) *blast*

**lemma** *bexE* [*elim!*]:  $\text{EX } x:A. P x \implies (!x. x:A \implies P x \implies Q) \implies Q$   
**by** (*unfold Bex-def*) *blast*

**lemma** *ball-triv* [*simp*]:  $(\text{ALL } x:A. P) = ((\text{EX } x. x:A) \dashv\vdash P)$   
 — Trivial rewrite rule.  
**by** (*simp add: Ball-def*)

**lemma** *bex-triv* [*simp*]:  $(\text{EX } x:A. P) = ((\text{EX } x. x:A) \& P)$   
 — Dual form for existentials.  
**by** (*simp add: Bex-def*)

**lemma** *bex-triv-one-point1* [*simp*]:  $(\text{EX } x:A. x = a) = (a:A)$   
**by** *blast*

**lemma** *bex-triv-one-point2* [*simp*]:  $(\text{EX } x:A. a = x) = (a:A)$   
**by** *blast*

**lemma** *bex-one-point1* [*simp*]:  $(\text{EX } x:A. x = a \& P x) = (a:A \& P a)$   
**by** *blast*

**lemma** *bex-one-point2* [*simp*]:  $(\text{EX } x:A. a = x \& P x) = (a:A \& P a)$   
**by** *blast*

**lemma** *ball-one-point1* [*simp*]:  $(\text{ALL } x:A. x = a \dashv\vdash P x) = (a:A \dashv\vdash P a)$   
**by** *blast*

**lemma** *ball-one-point2* [*simp*]:  $(\text{ALL } x:A. a = x \dashv\vdash P x) = (a:A \dashv\vdash P a)$   
**by** *blast*

Congruence rules

**lemma** *ball-cong*:  
 $A = B \implies (!x. x:B \implies P x = Q x) \implies$   
 $(\text{ALL } x:A. P x) = (\text{ALL } x:B. Q x)$   
**by** (*simp add: Ball-def*)

**lemma** *strong-ball-cong* [*cong*]:  
 $A = B \implies (!x. x:B \implies P x = Q x) \implies$   
 $(\text{ALL } x:A. P x) = (\text{ALL } x:B. Q x)$   
**by** (*simp add: simp-implies-def Ball-def*)

**lemma** *bex-cong*:  
 $A = B \implies (!x. x:B \implies P x = Q x) \implies$   
 $(\text{EX } x:A. P x) = (\text{EX } x:B. Q x)$   
**by** (*simp add: Bex-def cong: conj-cong*)

**lemma** *strong-bex-cong* [*cong*]:

$A = B ==> (!!x. x:B =simp==> P\ x = Q\ x) ==>$   
 $(EX\ x:A. P\ x) = (EX\ x:B. Q\ x)$   
**by** (*simp add: simp-implies-def Bex-def cong: conj-cong*)

## 6.3 Basic operations

### 6.3.1 Subsets

**lemma** *subsetI* [*intro!*]:  $(\bigwedge x. x \in A \implies x \in B) \implies A \subseteq B$   
**unfolding** *mem-def* **by** (*rule le-funI, rule le-boolI*)

Map the type '*a set* => *anything*' to just '*a*'; for overloading constants whose first argument has type '*a set*'.

**lemma** *subsetD* [*elim, intro?*]:  $A \subseteq B ==> c \in A ==> c \in B$   
**unfolding** *mem-def* **by** (*erule le-funE, erule le-boolE*)  
 — Rule in Modus Ponens style.

**lemma** *rev-subsetD* [*no-atp,intro?*]:  $c \in A ==> A \subseteq B ==> c \in B$   
 — The same, with reversed premises for use with *erule* – cf *rev-mp*.  
**by** (*rule subsetD*)

Converts  $A \subseteq B$  to  $x \in A \implies x \in B$ .

**lemma** *subsetCE* [*no-atp,elim*]:  $A \subseteq B ==> (c \notin A ==> P) ==> (c \in B ==> P) ==> P$   
 — Classical elimination rule.  
**unfolding** *mem-def* **by** (*blast dest: le-funE le-boolE*)

**lemma** *subset-eq* [*no-atp*]:  $A \leq B = (\forall x \in A. x \in B)$  **by** *blast*

**lemma** *contra-subsetD* [*no-atp*]:  $A \subseteq B ==> c \notin B ==> c \notin A$   
**by** *blast*

**lemma** *subset-refl* [*simp*]:  $A \subseteq A$   
**by** (*fact order-refl*)

**lemma** *subset-trans*:  $A \subseteq B ==> B \subseteq C ==> A \subseteq C$   
**by** (*fact order-trans*)

**lemma** *set-rev-mp*:  $x:A ==> A \subseteq B ==> x:B$   
**by** (*rule subsetD*)

**lemma** *set-mp*:  $A \subseteq B ==> x:A ==> x:B$   
**by** (*rule subsetD*)

**lemma** *eq-mem-trans*:  $a=b ==> b \in A ==> a \in A$   
**by** *simp*

**lemmas** *basic-trans-rules* [*trans*] =  
*order-trans-rules set-rev-mp set-mp eq-mem-trans*

### 6.3.2 Equality

**lemma** *set-ext*: **assumes** *prem*:  $(!!x. (x:A) = (x:B))$  **shows**  $A = B$   
**apply** (*rule* *prem* [*THEN ext*, *THEN arg-cong*, *THEN box-equals*])  
**apply** (*rule Collect-mem-eq*)  
**apply** (*rule Collect-mem-eq*)  
**done**

**lemma** *expand-set-eq*:  $(A = B) = (ALL\ x. (x:A) = (x:B))$   
**by**(*auto intro:set-ext*)

**lemma** *subset-antisym* [*intro!*]:  $A \subseteq B ==> B \subseteq A ==> A = B$   
— Anti-symmetry of the subset relation.  
**by** (*iprover intro: set-ext subsetD*)

Equality rules from ZF set theory – are they appropriate here?

**lemma** *equalityD1*:  $A = B ==> A \subseteq B$   
**by** *simp*

**lemma** *equalityD2*:  $A = B ==> B \subseteq A$   
**by** *simp*

Be careful when adding this to the claset as *subset-empty* is in the simpset:  
 $A = \{\}$  goes to  $\{\} \subseteq A$  and  $A \subseteq \{\}$  and then back to  $A = \{\}$ !

**lemma** *equalityE*:  $A = B ==> (A \subseteq B ==> B \subseteq A ==> P) ==> P$   
**by** *simp*

**lemma** *equalityCE* [*elim*]:  
 $A = B ==> (c \in A ==> c \in B ==> P) ==> (c \notin A ==> c \notin B ==> P)$   
 $==> P$   
**by** *blast*

**lemma** *eqset-imp-iff*:  $A = B ==> (x : A) = (x : B)$   
**by** *simp*

**lemma** *equelem-imp-iff*:  $x = y ==> (x : A) = (y : A)$   
**by** *simp*

### 6.3.3 The universal set – UNIV

**abbreviation** *UNIV* :: 'a set **where**  
 $UNIV \equiv top$

**lemma** *UNIV-def*:  
 $UNIV = \{x. True\}$   
**by** (*simp add: top-fun-eq top-bool-eq Collect-def*)

**lemma** *UNIV-I* [*simp*]:  $x : UNIV$

**by** (*simp add: UNIV-def*)

**declare** *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

**lemma** *UNIV-witness* [*intro?*]:  $EX\ x. x : UNIV$   
**by** *simp*

**lemma** *subset-UNIV* [*simp*]:  $A \subseteq UNIV$   
**by** (*rule subsetI*) (*rule UNIV-I*)

Eta-contracting these two rules (to remove  $P$ ) causes them to be ignored because of their interaction with congruence rules.

**lemma** *ball-UNIV* [*simp*]:  $Ball\ UNIV\ P = All\ P$   
**by** (*simp add: Ball-def*)

**lemma** *bex-UNIV* [*simp*]:  $Bex\ UNIV\ P = Ex\ P$   
**by** (*simp add: Bex-def*)

**lemma** *UNIV-eq-I*:  $(\bigwedge x. x \in A) \implies UNIV = A$   
**by** *auto*

#### 6.3.4 The empty set

**lemma** *empty-def*:  
 $\{\} = \{x. False\}$   
**by** (*simp add: bot-fun-eq bot-bool-eq Collect-def*)

**lemma** *empty-iff* [*simp*]:  $(c : \{\}) = False$   
**by** (*simp add: empty-def*)

**lemma** *emptyE* [*elim!*]:  $a : \{\} \implies P$   
**by** *simp*

**lemma** *empty-subsetI* [*iff*]:  $\{\} \subseteq A$   
 — One effect is to delete the ASSUMPTION  $\{\} \subseteq A$   
**by** *blast*

**lemma** *equals0I*:  $(!!y. y \in A \implies False) \implies A = \{\}$   
**by** *blast*

**lemma** *equals0D*:  $A = \{\} \implies a \notin A$   
 — Use for reasoning about disjointness:  $A\ Int\ B = \{\}$   
**by** *blast*

**lemma** *ball-empty* [*simp*]:  $Ball\ \{\}\ P = True$   
**by** (*simp add: Ball-def*)

**lemma** *bex-empty* [*simp*]:  $Bex\ \{\}\ P = False$   
**by** (*simp add: Bex-def*)

**lemma** *UNIV-not-empty* [iff]:  $UNIV \sim = \{\}$   
**by** (*blast elim: equalityE*)

### 6.3.5 The Powerset operator – Pow

**definition** *Pow* :: 'a set => 'a set set **where**  
*Pow-def*:  $Pow\ A = \{B. B \leq A\}$

**lemma** *Pow-iff* [iff]:  $(A \in Pow\ B) = (A \subseteq B)$   
**by** (*simp add: Pow-def*)

**lemma** *PowI*:  $A \subseteq B ==> A \in Pow\ B$   
**by** (*simp add: Pow-def*)

**lemma** *PowD*:  $A \in Pow\ B ==> A \subseteq B$   
**by** (*simp add: Pow-def*)

**lemma** *Pow-bottom*:  $\{\} \in Pow\ B$   
**by** *simp*

**lemma** *Pow-top*:  $A \in Pow\ A$   
**by** *simp*

### 6.3.6 Set complement

**lemma** *Compl-iff* [simp]:  $(c \in -A) = (c \notin A)$   
**by** (*simp add: mem-def fun-Compl-def bool-Compl-def*)

**lemma** *ComplI* [intro!]:  $(c \in A ==> False) ==> c \in -A$   
**by** (*unfold mem-def fun-Compl-def bool-Compl-def*) *blast*

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...

**lemma** *ComplD* [dest!]:  $c : -A ==> c \sim A$   
**by** (*simp add: mem-def fun-Compl-def bool-Compl-def*)

**lemmas** *ComplE* = *ComplD* [elim-format]

**lemma** *Compl-eq*:  $-A = \{x. \sim x : A\}$  **by** *blast*

### 6.3.7 Binary union – Un

**abbreviation** *union* :: 'a set => 'a set => 'a set (**infixl** *Un* 65) **where**  
*op Un*  $\equiv sup$

**notation** (*xsymbols*)  
*union* (**infixl**  $\cup$  65)



**notation** (*HTML output*)

*union* (**infixl**  $\cup$  65)

**lemma** *Un-def*:

$A \cup B = \{x. x \in A \vee x \in B\}$

**by** (*simp add: sup-fun-eq sup-bool-eq Collect-def mem-def*)

**lemma** *Un-iff* [*simp*]:  $(c : A \text{ Un } B) = (c:A \mid c:B)$

**by** (*unfold Un-def*) *blast*

**lemma** *UnI1* [*elim?*]:  $c:A \implies c : A \text{ Un } B$

**by** *simp*

**lemma** *UnI2* [*elim?*]:  $c:B \implies c : A \text{ Un } B$

**by** *simp*

Classical introduction rule: no commitment to  $A$  vs  $B$ .

**lemma** *UnCI* [*intro!*]:  $(c\sim:B \implies c:A) \implies c : A \text{ Un } B$

**by** *auto*

**lemma** *UnE* [*elim!*]:  $c : A \text{ Un } B \implies (c:A \implies P) \implies (c:B \implies P) \implies P$

**by** (*unfold Un-def*) *blast*

**lemma** *insert-def*:  $\text{insert } a \ B = \{x. x = a\} \cup B$

**by** (*simp add: Collect-def mem-def insert-compr Un-def*)

**lemma** *mono-Un*:  $\text{mono } f \implies f \ A \cup f \ B \subseteq f \ (A \cup B)$

**by** (*fact mono-sup*)

### 6.3.8 Binary intersection – Int

**abbreviation** *inter* ::  $'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$  (**infixl** *Int* 70) **where**

*op Int*  $\equiv$  *inf*

**notation** (*xsymbols*)

*inter* (**infixl**  $\cap$  70)

**notation** (*HTML output*)

*inter* (**infixl**  $\cap$  70)

**lemma** *Int-def*:

$A \cap B = \{x. x \in A \wedge x \in B\}$

**by** (*simp add: inf-fun-eq inf-bool-eq Collect-def mem-def*)

**lemma** *Int-iff* [*simp*]:  $(c : A \text{ Int } B) = (c:A \ \& \ c:B)$

**by** (*unfold Int-def*) *blast*

**lemma** *IntI* [*intro!*]:  $c:A \implies c:B \implies c : A \text{ Int } B$   
**by** *simp*

**lemma** *IntD1*:  $c : A \text{ Int } B \implies c:A$   
**by** *simp*

**lemma** *IntD2*:  $c : A \text{ Int } B \implies c:B$   
**by** *simp*

**lemma** *IntE* [*elim!*]:  $c : A \text{ Int } B \implies (c:A \implies c:B \implies P) \implies P$   
**by** *simp*

**lemma** *mono-Int*:  $\text{mono } f \implies f (A \cap B) \subseteq f A \cap f B$   
**by** (*fact mono-inf*)

### 6.3.9 Set difference

**lemma** *Diff-iff* [*simp*]:  $(c : A - B) = (c:A \ \& \ c\sim:B)$   
**by** (*simp add: mem-def fun-diff-def bool-diff-def*)

**lemma** *DiffI* [*intro!*]:  $c : A \implies c \sim : B \implies c : A - B$   
**by** *simp*

**lemma** *DiffD1*:  $c : A - B \implies c : A$   
**by** *simp*

**lemma** *DiffD2*:  $c : A - B \implies c : B \implies P$   
**by** *simp*

**lemma** *DiffE* [*elim!*]:  $c : A - B \implies (c:A \implies c\sim:B \implies P) \implies P$   
**by** *simp*

**lemma** *set-diff-eq*:  $A - B = \{x. x : A \ \& \ \sim x : B\}$  **by** *blast*

**lemma** *Compl-eq-Diff-UNIV*:  $\neg A = (UNIV - A)$   
**by** *blast*

### 6.3.10 Augmenting a set – insert

**lemma** *insert-iff* [*simp*]:  $(a : \text{insert } b \ A) = (a = b \mid a:A)$   
**by** (*unfold insert-def*) *blast*

**lemma** *insertI1*:  $a : \text{insert } a \ B$   
**by** *simp*

**lemma** *insertI2*:  $a : B \implies a : \text{insert } b \ B$   
**by** *simp*

**lemma** *insertE* [*elim!*]:  $a : \text{insert } b \ A \implies (a = b \implies P) \implies (a:A \implies P) \implies P$

**by** (*unfold insert-def*) *blast*

**lemma** *insertCI* [*intro!*]:  $(a \sim B \implies a = b) \implies a : \text{insert } b \ B$   
 — Classical introduction rule.  
**by** *auto*

**lemma** *subset-insert-iff*:  $(A \subseteq \text{insert } x \ B) = (\text{if } x:A \text{ then } A - \{x\} \subseteq B \text{ else } A \subseteq B)$   
**by** *auto*

**lemma** *set-insert*:  
 assumes  $x \in A$   
 obtains  $B$  where  $A = \text{insert } x \ B$  and  $x \notin B$   
**proof**  
 from *assms* show  $A = \text{insert } x \ (A - \{x\})$  **by** *blast*  
**next**  
 show  $x \notin A - \{x\}$  **by** *blast*  
**qed**

**lemma** *insert-ident*:  $x \sim A \implies x \sim B \implies (\text{insert } x \ A = \text{insert } x \ B) = (A = B)$   
**by** *auto*

### 6.3.11 Singletons, using insert

**lemma** *singletonI* [*intro!,no-atp*]:  $a : \{a\}$   
 — Redundant? But unlike *insertCI*, it proves the subgoal immediately!  
**by** (*rule insertI1*)

**lemma** *singletonD* [*dest!,no-atp*]:  $b : \{a\} \implies b = a$   
**by** *blast*

**lemmas** *singletonE* = *singletonD* [*elim-format*]

**lemma** *singleton-iff*:  $(b : \{a\}) = (b = a)$   
**by** *blast*

**lemma** *singleton-inject* [*dest!*]:  $\{a\} = \{b\} \implies a = b$   
**by** *blast*

**lemma** *singleton-insert-inj-eq* [*iff,no-atp*]:  
 $(\{b\} = \text{insert } a \ A) = (a = b \ \& \ A \subseteq \{b\})$   
**by** *blast*

**lemma** *singleton-insert-inj-eq'* [*iff,no-atp*]:  
 $(\text{insert } a \ A = \{b\}) = (a = b \ \& \ A \subseteq \{b\})$   
**by** *blast*

**lemma** *subset-singletonD*:  $A \subseteq \{x\} \implies A = \{\} \mid A = \{x\}$

by *fast*

**lemma** *singleton-conv* [*simp*]:  $\{x. x = a\} = \{a\}$   
by *blast*

**lemma** *singleton-conv2* [*simp*]:  $\{x. a = x\} = \{a\}$   
by *blast*

**lemma** *diff-single-insert*:  $A - \{x\} \subseteq B \implies x \in A \implies A \subseteq \text{insert } x B$   
by *blast*

**lemma** *doubleton-eq-iff*:  $(\{a, b\} = \{c, d\}) = (a=c \ \& \ b=d \mid a=d \ \& \ b=c)$   
by (*blast elim: equalityE*)

### 6.3.12 Image of a set under a function

Frequently  $b$  does not have the syntactic form of  $f x$ .

**definition** *image* ::  $('a \Rightarrow 'b) \Rightarrow 'a \text{ set} \Rightarrow 'b \text{ set}$  (*infixr* ‘90) **where**  
*image-def* [*no-atp*]:  $f \text{ ‘ } A = \{y. \exists x:A. y = f(x)\}$

**abbreviation**

*range* ::  $('a \Rightarrow 'b) \Rightarrow 'b \text{ set}$  **where** — of function  
*range*  $f == f \text{ ‘ } \text{UNIV}$

**lemma** *image-eqI* [*simp, intro*]:  $b = f x \implies x:A \implies b : f \text{ ‘ } A$   
by (*unfold image-def*) *blast*

**lemma** *imageI*:  $x : A \implies f x : f \text{ ‘ } A$   
by (*rule image-eqI*) (*rule refl*)

**lemma** *rev-image-eqI*:  $x:A \implies b = f x \implies b : f \text{ ‘ } A$   
— This version’s more effective when we already have the required  $x$ .  
by (*unfold image-def*) *blast*

**lemma** *imageE* [*elim!*]:  
 $b : (\%x. f x) \text{ ‘ } A \implies (!x. b = f x \implies x:A \implies P) \implies P$   
— The eta-expansion gives variable-name preservation.  
by (*unfold image-def*) *blast*

**lemma** *image-Un*:  $f \text{ ‘ } (A \text{ Un } B) = f \text{ ‘ } A \text{ Un } f \text{ ‘ } B$   
by *blast*

**lemma** *image-iff*:  $(z : f \text{ ‘ } A) = (\exists x:A. z = f x)$   
by *blast*

**lemma** *image-subset-iff*:  $(f \text{ ‘ } A \subseteq B) = (\forall x \in A. f x \in B)$   
— This rewrite rule would confuse users if made default.  
by *blast*

**lemma** *subset-image-iff*:  $(B \subseteq f^*A) = (EX\ AA.\ AA \subseteq A \ \&\ B = f^*AA)$   
**apply** *safe*  
**prefer** 2 **apply** *fast*  
**apply** (*rule-tac*  $x = \{a.\ a : A \ \&\ f\ a : B\}$  **in** *exI*, *fast*)  
**done**

**lemma** *image-subsetI*:  $(!!x.\ x \in A ==> f\ x \in B) ==> f^*A \subseteq B$   
 — Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.  
**by** *blast*

Range of a function – just a translation for image!

**lemma** *range-eqI*:  $b = f\ x ==> b \in \text{range}\ f$   
**by** *simp*

**lemma** *rangeI*:  $f\ x \in \text{range}\ f$   
**by** *simp*

**lemma** *rangeE* [*elim?*]:  $b \in \text{range}\ (\lambda x.\ f\ x) ==> (!!x.\ b = f\ x ==> P) ==> P$   
**by** *blast*

### 6.3.13 Some rules with *if*

Elimination of  $\{x.\ \dots \ \&\ x=t \ \&\ \dots\}$ .

**lemma** *Collect-conv-if*:  $\{x.\ x=a \ \&\ P\ x\} = (\text{if}\ P\ a\ \text{then}\ \{a\}\ \text{else}\ \{\})$   
**by** *auto*

**lemma** *Collect-conv-if2*:  $\{x.\ a=x \ \&\ P\ x\} = (\text{if}\ P\ a\ \text{then}\ \{a\}\ \text{else}\ \{\})$   
**by** *auto*

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

**lemma** *split-if-eq1*:  $((\text{if}\ Q\ \text{then}\ x\ \text{else}\ y) = b) = ((Q \longrightarrow x = b) \ \&\ (\sim Q \longrightarrow y = b))$   
**by** (*rule split-if*)

**lemma** *split-if-eq2*:  $(a = (\text{if}\ Q\ \text{then}\ x\ \text{else}\ y)) = ((Q \longrightarrow a = x) \ \&\ (\sim Q \longrightarrow a = y))$   
**by** (*rule split-if*)

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

**lemma** *split-if-mem1*:  $((\text{if}\ Q\ \text{then}\ x\ \text{else}\ y) : b) = ((Q \longrightarrow x : b) \ \&\ (\sim Q \longrightarrow y : b))$   
**by** (*rule split-if*)

**lemma** *split-if-mem2*:  $(a : (\text{if}\ Q\ \text{then}\ x\ \text{else}\ y)) = ((Q \longrightarrow a : x) \ \&\ (\sim Q \longrightarrow a : y))$

**by** (*rule split-if* [**where**  $P = \%S. a : S$ ])

**lemmas** *split-ifs* = *if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

## 6.4 Further operations and lemmas

### 6.4.1 The “proper subset” relation

**lemma** *psubsetI* [*intro!,no-atp*]:  $A \subseteq B \implies A \neq B \implies A \subset B$   
**by** (*unfold less-le*) *blast*

**lemma** *psubsetE* [*elim!,no-atp*]:  
 $[|A \subset B; [|A \subseteq B; \sim (B \subseteq A)|] \implies R|] \implies R$   
**by** (*unfold less-le*) *blast*

**lemma** *psubset-insert-iff*:  
 $(A \subset \text{insert } x \ B) = (\text{if } x \in B \text{ then } A \subset B \text{ else if } x \in A \text{ then } A - \{x\} \subset B \text{ else } A \subseteq B)$   
**by** (*auto simp add: less-le subset-insert-iff*)

**lemma** *psubset-eq*:  $(A \subset B) = (A \subseteq B \ \& \ A \neq B)$   
**by** (*simp only: less-le*)

**lemma** *psubset-imp-subset*:  $A \subset B \implies A \subseteq B$   
**by** (*simp add: psubset-eq*)

**lemma** *psubset-trans*:  $[|A \subset B; B \subset C|] \implies A \subset C$   
**apply** (*unfold less-le*)  
**apply** (*auto dest: subset-antisym*)  
**done**

**lemma** *psubsetD*:  $[|A \subset B; c \in A|] \implies c \in B$   
**apply** (*unfold less-le*)  
**apply** (*auto dest: subsetD*)  
**done**

**lemma** *psubset-subset-trans*:  $A \subset B \implies B \subseteq C \implies A \subset C$   
**by** (*auto simp add: psubset-eq*)

**lemma** *subset-psubset-trans*:  $A \subseteq B \implies B \subset C \implies A \subset C$   
**by** (*auto simp add: psubset-eq*)

**lemma** *psubset-imp-ex-mem*:  $A \subset B \implies \exists b. b \in (B - A)$   
**by** (*unfold less-le*) *blast*

**lemma** *atomize-ball*:  
 $(!!x. x \in A \implies P \ x) == \text{Trueprop } (\forall x \in A. P \ x)$   
**by** (*simp only: Ball-def atomize-all atomize-imp*)

**lemmas** [*symmetric, rulify*] = *atomize-ball*

and  $[symmetric, defn] = atomize-ball$

#### 6.4.2 Derived rules involving subsets.

*insert.*

**lemma** *subset-insertI*:  $B \subseteq insert\ a\ B$   
**by** (*rule subsetI*) (*erule insertI2*)

**lemma** *subset-insertI2*:  $A \subseteq B \implies A \subseteq insert\ b\ B$   
**by** *blast*

**lemma** *subset-insert*:  $x \notin A \implies (A \subseteq insert\ x\ B) = (A \subseteq B)$   
**by** *blast*

Finite Union – the least upper bound of two sets.

**lemma** *Un-upper1*:  $A \subseteq A \cup B$   
**by** (*fact sup-ge1*)

**lemma** *Un-upper2*:  $B \subseteq A \cup B$   
**by** (*fact sup-ge2*)

**lemma** *Un-least*:  $A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C$   
**by** (*fact sup-least*)

Finite Intersection – the greatest lower bound of two sets.

**lemma** *Int-lower1*:  $A \cap B \subseteq A$   
**by** (*fact inf-le1*)

**lemma** *Int-lower2*:  $A \cap B \subseteq B$   
**by** (*fact inf-le2*)

**lemma** *Int-greatest*:  $C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B$   
**by** (*fact inf-greatest*)

Set difference.

**lemma** *Diff-subset*:  $A - B \subseteq A$   
**by** *blast*

**lemma** *Diff-subset-conv*:  $(A - B \subseteq C) = (A \subseteq B \cup C)$   
**by** *blast*

#### 6.4.3 Equalities involving union, intersection, inclusion, etc.

$\{\}$ .

**lemma** *Collect-const* [*simp*]:  $\{s. P\} = (if\ P\ then\ UNIV\ else\ \{\})$   
 — supersedes *Collect-False-empty*

**by** *auto*

**lemma** *subset-empty* [*simp*]:  $(A \subseteq \{\}) = (A = \{\})$   
**by** *blast*

**lemma** *not-psubset-empty* [*iff*]:  $\neg (A < \{\})$   
**by** (*unfold less-le*) *blast*

**lemma** *Collect-empty-eq* [*simp*]:  $(\text{Collect } P = \{\}) = (\forall x. \neg P x)$   
**by** *blast*

**lemma** *empty-Collect-eq* [*simp*]:  $(\{\} = \text{Collect } P) = (\forall x. \neg P x)$   
**by** *blast*

**lemma** *Collect-neg-eq*:  $\{x. \neg P x\} = - \{x. P x\}$   
**by** *blast*

**lemma** *Collect-disj-eq*:  $\{x. P x \mid Q x\} = \{x. P x\} \cup \{x. Q x\}$   
**by** *blast*

**lemma** *Collect-imp-eq*:  $\{x. P x \longrightarrow Q x\} = -\{x. P x\} \cup \{x. Q x\}$   
**by** *blast*

**lemma** *Collect-conj-eq*:  $\{x. P x \ \& \ Q x\} = \{x. P x\} \cap \{x. Q x\}$   
**by** *blast*

*insert.*

**lemma** *insert-is-Un*:  $\text{insert } a \ A = \{a\} \ \text{Un } A$   
 — NOT SUITABLE FOR REWRITING since  $\{a\} == \text{insert } a \ \{\}$   
**by** *blast*

**lemma** *insert-not-empty* [*simp*]:  $\text{insert } a \ A \neq \{\}$   
**by** *blast*

**lemmas** *empty-not-insert* = *insert-not-empty* [*symmetric, standard*]  
**declare** *empty-not-insert* [*simp*]

**lemma** *insert-absorb*:  $a \in A \implies \text{insert } a \ A = A$   
 — [*simp*] causes recursive calls when there are nested inserts  
 — with *quadratic* running time  
**by** *blast*

**lemma** *insert-absorb2* [*simp*]:  $\text{insert } x \ (\text{insert } x \ A) = \text{insert } x \ A$   
**by** *blast*

**lemma** *insert-commute*:  $\text{insert } x \ (\text{insert } y \ A) = \text{insert } y \ (\text{insert } x \ A)$   
**by** *blast*

**lemma** *insert-subset* [*simp*]:  $(\text{insert } x \ A \subseteq B) = (x \in B \ \& \ A \subseteq B)$



by *blast*

**lemma** *mk-disjoint-insert*:  $a \in A \implies \exists B. A = \text{insert } a B \ \& \ a \notin B$   
 — use new  $B$  rather than  $A - \{a\}$  to avoid infinite unfolding  
**apply** (*rule-tac*  $x = A - \{a\}$  **in** *exI*, *blast*)  
**done**

**lemma** *insert-Collect*:  $\text{insert } a (\text{Collect } P) = \{u. u \neq a \longrightarrow P u\}$   
**by** *auto*

**lemma** *insert-inter-insert*[*simp*]:  $\text{insert } a A \cap \text{insert } a B = \text{insert } a (A \cap B)$   
**by** *blast*

**lemma** *insert-disjoint* [*simp*,*no-atp*]:  
 $(\text{insert } a A \cap B = \{\}) = (a \notin B \wedge A \cap B = \{\})$   
 $(\{\} = \text{insert } a A \cap B) = (a \notin B \wedge \{\} = A \cap B)$   
**by** *auto*

**lemma** *disjoint-insert* [*simp*,*no-atp*]:  
 $(B \cap \text{insert } a A = \{\}) = (a \notin B \wedge B \cap A = \{\})$   
 $(\{\} = A \cap \text{insert } b B) = (b \notin A \wedge \{\} = A \cap B)$   
**by** *auto*

*image.*

**lemma** *image-empty* [*simp*]:  $f' \{\} = \{\}$   
**by** *blast*

**lemma** *image-insert* [*simp*]:  $f' \text{insert } a B = \text{insert } (f a) (f' B)$   
**by** *blast*

**lemma** *image-constant*:  $x \in A \implies (\lambda x. c)' A = \{c\}$   
**by** *auto*

**lemma** *image-constant-conv*:  $(\%x. c)' A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})$   
**by** *auto*

**lemma** *image-image*:  $f' (g' A) = (\lambda x. f (g x))' A$   
**by** *blast*

**lemma** *insert-image* [*simp*]:  $x \in A \implies \text{insert } (f x) (f' A) = f' A$   
**by** *blast*

**lemma** *image-is-empty* [*iff*]:  $(f' A = \{\}) = (A = \{\})$   
**by** *blast*

**lemma** *empty-is-image*[*iff*]:  $(\{\} = f' A) = (A = \{\})$   
**by** *blast*

**lemma** *image-Collect* [no-atp]:  $f \cdot \{x. P\ x\} = \{f\ x \mid x. P\ x\}$

— NOT suitable as a default simp rule: the RHS isn’t simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.

**by** *blast*

**lemma** *if-image-distrib* [simp]:

$(\lambda x. \text{if } P\ x \text{ then } f\ x \text{ else } g\ x) \cdot S$   
 $= (f \cdot (S \cap \{x. P\ x\})) \cup (g \cdot (S \cap \{x. \neg P\ x\}))$

**by** (*auto simp add: image-def*)

**lemma** *image-cong*:  $M = N \implies (!x. x \in N \implies f\ x = g\ x) \implies f \cdot M = g \cdot N$

**by** (*simp add: image-def*)

*range.*

**lemma** *full-SetCompr-eq* [no-atp]:  $\{u. \exists x. u = f\ x\} = \text{range } f$

**by** *auto*

**lemma** *range-composition*:  $\text{range } (\lambda x. f\ (g\ x)) = f \cdot \text{range } g$

**by** (*subst image-image, simp*)

*Int*

**lemma** *Int-absorb* [simp]:  $A \cap A = A$

**by** (*fact inf-idem*)

**lemma** *Int-left-absorb*:  $A \cap (A \cap B) = A \cap B$

**by** (*fact inf-left-idem*)

**lemma** *Int-commute*:  $A \cap B = B \cap A$

**by** (*fact inf-commute*)

**lemma** *Int-left-commute*:  $A \cap (B \cap C) = B \cap (A \cap C)$

**by** (*fact inf-left-commute*)

**lemma** *Int-assoc*:  $(A \cap B) \cap C = A \cap (B \cap C)$

**by** (*fact inf-assoc*)

**lemmas** *Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute*

— Intersection is an AC-operator

**lemma** *Int-absorb1*:  $B \subseteq A \implies A \cap B = B$

**by** (*fact inf-absorb2*)

**lemma** *Int-absorb2*:  $A \subseteq B \implies A \cap B = A$

**by** (*fact inf-absorb1*)

**lemma** *Int-empty-left* [simp]:  $\{\} \cap B = \{\}$

**by** (*fact inf-bot-left*)

**lemma** *Int-empty-right* [simp]:  $A \cap \{\} = \{\}$   
**by** (*fact inf-bot-right*)

**lemma** *disjoint-eq-subset-Compl*:  $(A \cap B = \{\}) = (A \subseteq -B)$   
**by** *blast*

**lemma** *disjoint-iff-not-equal*:  $(A \cap B = \{\}) = (\forall x \in A. \forall y \in B. x \neq y)$   
**by** *blast*

**lemma** *Int-UNIV-left* [simp]:  $UNIV \cap B = B$   
**by** (*fact inf-top-left*)

**lemma** *Int-UNIV-right* [simp]:  $A \cap UNIV = A$   
**by** (*fact inf-top-right*)

**lemma** *Int-Un-distrib*:  $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$   
**by** (*fact inf-sup-distrib1*)

**lemma** *Int-Un-distrib2*:  $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$   
**by** (*fact inf-sup-distrib2*)

**lemma** *Int-UNIV* [simp,no-atp]:  $(A \cap B = UNIV) = (A = UNIV \ \& \ B = UNIV)$   
**by** (*fact inf-eq-top-iff*)

**lemma** *Int-subset-iff* [simp]:  $(C \subseteq A \cap B) = (C \subseteq A \ \& \ C \subseteq B)$   
**by** (*fact le-inf-iff*)

**lemma** *Int-Collect*:  $(x \in A \cap \{x. P \ x\}) = (x \in A \ \& \ P \ x)$   
**by** *blast*

*Un.*

**lemma** *Un-absorb* [simp]:  $A \cup A = A$   
**by** (*fact sup-idem*)

**lemma** *Un-left-absorb*:  $A \cup (A \cup B) = A \cup B$   
**by** (*fact sup-left-idem*)

**lemma** *Un-commute*:  $A \cup B = B \cup A$   
**by** (*fact sup-commute*)

**lemma** *Un-left-commute*:  $A \cup (B \cup C) = B \cup (A \cup C)$   
**by** (*fact sup-left-commute*)

**lemma** *Un-assoc*:  $(A \cup B) \cup C = A \cup (B \cup C)$   
**by** (*fact sup-assoc*)

**lemmas** *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*  
 — Union is an AC-operator

**lemma** *Un-absorb1*:  $A \subseteq B \implies A \cup B = B$   
**by** (*fact sup-absorb2*)

**lemma** *Un-absorb2*:  $B \subseteq A \implies A \cup B = A$   
**by** (*fact sup-absorb1*)

**lemma** *Un-empty-left [simp]*:  $\{\} \cup B = B$   
**by** (*fact sup-bot-left*)

**lemma** *Un-empty-right [simp]*:  $A \cup \{\} = A$   
**by** (*fact sup-bot-right*)

**lemma** *Un-UNIV-left [simp]*:  $UNIV \cup B = UNIV$   
**by** (*fact sup-top-left*)

**lemma** *Un-UNIV-right [simp]*:  $A \cup UNIV = UNIV$   
**by** (*fact sup-top-right*)

**lemma** *Un-insert-left [simp]*:  $(\text{insert } a \ B) \cup C = \text{insert } a \ (B \cup C)$   
**by** *blast*

**lemma** *Un-insert-right [simp]*:  $A \cup (\text{insert } a \ B) = \text{insert } a \ (A \cup B)$   
**by** *blast*

**lemma** *Int-insert-left*:  
 $(\text{insert } a \ B) \text{ Int } C = (\text{if } a \in C \text{ then } \text{insert } a \ (B \cap C) \text{ else } B \cap C)$   
**by** *auto*

**lemma** *Int-insert-left-if0 [simp]*:  
 $a \notin C \implies (\text{insert } a \ B) \text{ Int } C = B \cap C$   
**by** *auto*

**lemma** *Int-insert-left-if1 [simp]*:  
 $a \in C \implies (\text{insert } a \ B) \text{ Int } C = \text{insert } a \ (B \text{ Int } C)$   
**by** *auto*

**lemma** *Int-insert-right*:  
 $A \cap (\text{insert } a \ B) = (\text{if } a \in A \text{ then } \text{insert } a \ (A \cap B) \text{ else } A \cap B)$   
**by** *auto*

**lemma** *Int-insert-right-if0 [simp]*:  
 $a \notin A \implies A \text{ Int } (\text{insert } a \ B) = A \text{ Int } B$   
**by** *auto*

**lemma** *Int-insert-right-if1 [simp]*:  
 $a \in A \implies A \text{ Int } (\text{insert } a \ B) = \text{insert } a \ (A \text{ Int } B)$   
**by** *auto*

**lemma** *Un-Int-distrib*:  $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$   
**by** (*fact sup-inf-distrib1*)

**lemma** *Un-Int-distrib2*:  $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$   
**by** (*fact sup-inf-distrib2*)

**lemma** *Un-Int-crazy*:  
 $(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$   
**by** *blast*

**lemma** *subset-Un-eq*:  $(A \subseteq B) = (A \cup B = B)$   
**by** (*fact le-iff-sup*)

**lemma** *Un-empty [iff]*:  $(A \cup B = \{\}) = (A = \{\} \ \& \ B = \{\})$   
**by** (*fact sup-eq-bot-iff*)

**lemma** *Un-subset-iff [simp]*:  $(A \cup B \subseteq C) = (A \subseteq C \ \& \ B \subseteq C)$   
**by** (*fact le-sup-iff*)

**lemma** *Un-Diff-Int*:  $(A - B) \cup (A \cap B) = A$   
**by** *blast*

**lemma** *Diff-Int2*:  $A \cap C - B \cap C = A \cap C - B$   
**by** *blast*

Set complement

**lemma** *Compl-disjoint [simp]*:  $A \cap -A = \{\}$   
**by** (*fact inf-compl-bot*)

**lemma** *Compl-disjoint2 [simp]*:  $-A \cap A = \{\}$   
**by** (*fact compl-inf-bot*)

**lemma** *Compl-partition*:  $A \cup -A = UNIV$   
**by** (*fact sup-compl-top*)

**lemma** *Compl-partition2*:  $-A \cup A = UNIV$   
**by** (*fact compl-sup-top*)

**lemma** *double-complement [simp]*:  $-(-A) = (A::'a \text{ set})$   
**by** (*fact double-compl*)

**lemma** *Compl-Un [simp]*:  $-(A \cup B) = (-A) \cap (-B)$   
**by** (*fact compl-sup*)

**lemma** *Compl-Int [simp]*:  $-(A \cap B) = (-A) \cup (-B)$   
**by** (*fact compl-inf*)

**lemma** *subset-Compl-self-eq*:  $(A \subseteq -A) = (A = \{\})$   
**by** *blast*

**lemma** *Un-Int-assoc-eq*:  $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$   
 — Halmos, Naive Set Theory, page 16.  
**by** *blast*

**lemma** *Compl-UNIV-eq* [*simp*]:  $-UNIV = \{\}$   
**by** (*fact compl-top-eq*)

**lemma** *Compl-empty-eq* [*simp*]:  $-\{\} = UNIV$   
**by** (*fact compl-bot-eq*)

**lemma** *Compl-subset-Compl-iff* [*iff*]:  $(-A \subseteq -B) = (B \subseteq A)$   
**by** (*fact compl-le-compl-iff*)

**lemma** *Compl-eq-Compl-iff* [*iff*]:  $(-A = -B) = (A = (B::'a \text{ set}))$   
**by** (*fact compl-eq-compl-iff*)

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

**lemma** *ball-Un*:  $(\forall x \in A \cup B. P x) = ((\forall x \in A. P x) \ \& \ (\forall x \in B. P x))$   
**by** *blast*

**lemma** *bex-Un*:  $(\exists x \in A \cup B. P x) = ((\exists x \in A. P x) \mid (\exists x \in B. P x))$   
**by** *blast*

Set difference.

**lemma** *Diff-eq*:  $A - B = A \cap (-B)$   
**by** *blast*

**lemma** *Diff-eq-empty-iff* [*simp,no-atp*]:  $(A - B = \{\}) = (A \subseteq B)$   
**by** *blast*

**lemma** *Diff-cancel* [*simp*]:  $A - A = \{\}$   
**by** *blast*

**lemma** *Diff-idemp* [*simp*]:  $(A - B) - B = A - (B::'a \text{ set})$   
**by** *blast*

**lemma** *Diff-triv*:  $A \cap B = \{\} ==> A - B = A$   
**by** (*blast elim: equalityE*)

**lemma** *empty-Diff* [*simp*]:  $\{\} - A = \{\}$   
**by** *blast*

**lemma** *Diff-empty* [*simp*]:  $A - \{\} = A$   
**by** *blast*

**lemma** *Diff-UNIV* [simp]:  $A - UNIV = \{\}$   
**by** *blast*

**lemma** *Diff-insert0* [simp,no-atp]:  $x \notin A ==> A - insert\ x\ B = A - B$   
**by** *blast*

**lemma** *Diff-insert*:  $A - insert\ a\ B = A - B - \{a\}$   
 — NOT SUITABLE FOR REWRITING since  $\{a\} == insert\ a\ 0$   
**by** *blast*

**lemma** *Diff-insert2*:  $A - insert\ a\ B = A - \{a\} - B$   
 — NOT SUITABLE FOR REWRITING since  $\{a\} == insert\ a\ 0$   
**by** *blast*

**lemma** *insert-Diff-if*:  $insert\ x\ A - B = (if\ x \in B\ then\ A - B\ else\ insert\ x\ (A - B))$   
**by** *auto*

**lemma** *insert-Diff1* [simp]:  $x \in B ==> insert\ x\ A - B = A - B$   
**by** *blast*

**lemma** *insert-Diff-single*[simp]:  $insert\ a\ (A - \{a\}) = insert\ a\ A$   
**by** *blast*

**lemma** *insert-Diff*:  $a \in A ==> insert\ a\ (A - \{a\}) = A$   
**by** *blast*

**lemma** *Diff-insert-absorb*:  $x \notin A ==> (insert\ x\ A) - \{x\} = A$   
**by** *auto*

**lemma** *Diff-disjoint* [simp]:  $A \cap (B - A) = \{\}$   
**by** *blast*

**lemma** *Diff-partition*:  $A \subseteq B ==> A \cup (B - A) = B$   
**by** *blast*

**lemma** *double-diff*:  $A \subseteq B ==> B \subseteq C ==> B - (C - A) = A$   
**by** *blast*

**lemma** *Un-Diff-cancel* [simp]:  $A \cup (B - A) = A \cup B$   
**by** *blast*

**lemma** *Un-Diff-cancel2* [simp]:  $(B - A) \cup A = B \cup A$   
**by** *blast*

**lemma** *Diff-Un*:  $A - (B \cup C) = (A - B) \cap (A - C)$   
**by** *blast*

**lemma** *Diff-Int*:  $A - (B \cap C) = (A - B) \cup (A - C)$

by *blast*

**lemma** *Un-Diff*:  $(A \cup B) - C = (A - C) \cup (B - C)$   
by *blast*

**lemma** *Int-Diff*:  $(A \cap B) - C = A \cap (B - C)$   
by *blast*

**lemma** *Diff-Int-distrib*:  $C \cap (A - B) = (C \cap A) - (C \cap B)$   
by *blast*

**lemma** *Diff-Int-distrib2*:  $(A - B) \cap C = (A \cap C) - (B \cap C)$   
by *blast*

**lemma** *Diff-Compl [simp]*:  $A - (\neg B) = A \cap B$   
by *auto*

**lemma** *Compl-Diff-eq [simp]*:  $\neg (A - B) = \neg A \cup B$   
by *blast*

Quantification over type *bool*.

**lemma** *bool-induct*:  $P \text{ True} \implies P \text{ False} \implies P x$   
by (*cases x*) *auto*

**lemma** *all-bool-eq*:  $(\forall b. P b) \longleftrightarrow P \text{ True} \wedge P \text{ False}$   
by (*auto intro: bool-induct*)

**lemma** *bool-contrapos*:  $P x \implies \neg P \text{ False} \implies P \text{ True}$   
by (*cases x*) *auto*

**lemma** *ex-bool-eq*:  $(\exists b. P b) \longleftrightarrow P \text{ True} \vee P \text{ False}$   
by (*auto intro: bool-contrapos*)

*Pow*

**lemma** *Pow-empty [simp]*:  $\text{Pow } \{\} = \{\{\}\}$   
by (*auto simp add: Pow-def*)

**lemma** *Pow-insert*:  $\text{Pow } (\text{insert } a \ A) = \text{Pow } A \cup (\text{insert } a \ ' \text{Pow } A)$   
by (*blast intro: image-eqI [where ?x = u - \{a\}, standard]*)

**lemma** *Pow-Compl*:  $\text{Pow } (\neg A) = \{-B \mid B. A \in \text{Pow } B\}$   
by (*blast intro: exI [where ?x = - u, standard]*)

**lemma** *Pow-UNIV [simp]*:  $\text{Pow } \text{UNIV} = \text{UNIV}$   
by *blast*

**lemma** *Un-Pow-subset*:  $\text{Pow } A \cup \text{Pow } B \subseteq \text{Pow } (A \cup B)$   
by *blast*



**lemma** *Pow-Int-eq* [*simp*]:  $\text{Pow } (A \cap B) = \text{Pow } A \cap \text{Pow } B$   
**by** *blast*

Miscellany.

**lemma** *set-eq-subset*:  $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$   
**by** *blast*

**lemma** *subset-iff*:  $(A \subseteq B) = (\forall t. t \in A \longrightarrow t \in B)$   
**by** *blast*

**lemma** *subset-iff-psubset-eq*:  $(A \subseteq B) = ((A \subset B) \mid (A = B))$   
**by** (*unfold less-le*) *blast*

**lemma** *all-not-in-conv* [*simp*]:  $(\forall x. x \notin A) = (A = \{\})$   
**by** *blast*

**lemma** *ex-in-conv*:  $(\exists x. x \in A) = (A \neq \{\})$   
**by** *blast*

**lemma** *distinct-lemma*:  $f\ x \neq f\ y \implies x \neq y$   
**by** *iprover*

#### 6.4.4 Monotonicity of various operations

**lemma** *image-mono*:  $A \subseteq B \implies f^*A \subseteq f^*B$   
**by** *blast*

**lemma** *Pow-mono*:  $A \subseteq B \implies \text{Pow } A \subseteq \text{Pow } B$   
**by** *blast*

**lemma** *insert-mono*:  $C \subseteq D \implies \text{insert } a\ C \subseteq \text{insert } a\ D$   
**by** *blast*

**lemma** *Un-mono*:  $A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D$   
**by** (*fact sup-mono*)

**lemma** *Int-mono*:  $A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D$   
**by** (*fact inf-mono*)

**lemma** *Diff-mono*:  $A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D$   
**by** *blast*

**lemma** *Compl-anti-mono*:  $A \subseteq B \implies -B \subseteq -A$   
**by** (*fact compl-mono*)

Monotonicity of implications.

**lemma** *in-mono*:  $A \subseteq B \implies x \in A \longrightarrow x \in B$

```

apply (rule impI)
apply (erule subsetD, assumption)
done

```

```

lemma conj-mono:  $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \ \& \ P2) \longrightarrow (Q1 \ \& \ Q2)$ 
by iprover

```

```

lemma disj-mono:  $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \ | \ P2) \longrightarrow (Q1 \ | \ Q2)$ 
by iprover

```

```

lemma imp-mono:  $Q1 \longrightarrow P1 \implies P2 \longrightarrow Q2 \implies (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$ 
by iprover

```

```

lemma imp-refl:  $P \longrightarrow P \ ..$ 

```

```

lemma not-mono:  $Q \longrightarrow P \implies \sim P \longrightarrow \sim Q$ 
by iprover

```

```

lemma ex-mono:  $(!!x. P \ x \longrightarrow Q \ x) \implies (EX \ x. P \ x) \longrightarrow (EX \ x. Q \ x)$ 
by iprover

```

```

lemma all-mono:  $(!!x. P \ x \longrightarrow Q \ x) \implies (ALL \ x. P \ x) \longrightarrow (ALL \ x. Q \ x)$ 
by iprover

```

```

lemma Collect-mono:  $(!!x. P \ x \longrightarrow Q \ x) \implies Collect \ P \subseteq Collect \ Q$ 
by blast

```

```

lemma Int-Collect-mono:
   $A \subseteq B \implies (!!x. x \in A \implies P \ x \longrightarrow Q \ x) \implies A \cap Collect \ P \subseteq B \cap Collect \ Q$ 
by blast

```

```

lemmas basic-monos =
  subset-refl imp-refl disj-mono conj-mono
  ex-mono Collect-mono in-mono

```

```

lemma eq-to-mono:  $a = b \implies c = d \implies b \longrightarrow d \implies a \longrightarrow c$ 
by iprover

```

### 6.4.5 Inverse image of a function

```

definition vimage :: ('a => 'b) => 'b set => 'a set (infixr -' 90) where
  [code del]:  $f \ -' \ B == \{x. f \ x : B\}$ 

```

```

lemma vimage-eq [simp]:  $(a : f \ -' \ B) = (f \ a : B)$ 
by (unfold vimage-def) blast

```

**lemma** *vimage-singleton-eq*:  $(a : f -' \{b\}) = (f a = b)$   
**by** *simp*

**lemma** *vimageI* [*intro*]:  $f a = b ==> b:B ==> a : f -' B$   
**by** (*unfold vimage-def*) *blast*

**lemma** *vimageI2*:  $f a : A ==> a : f -' A$   
**by** (*unfold vimage-def*) *fast*

**lemma** *vimageE* [*elim!*]:  $a: f -' B ==> (!!x. f a = x ==> x:B ==> P) ==> P$   
**by** (*unfold vimage-def*) *blast*

**lemma** *vimageD*:  $a : f -' A ==> f a : A$   
**by** (*unfold vimage-def*) *fast*

**lemma** *vimage-empty* [*simp*]:  $f -' \{\} = \{\}$   
**by** *blast*

**lemma** *vimage-Compl*:  $f -' (-A) = -(f -' A)$   
**by** *blast*

**lemma** *vimage-Un* [*simp*]:  $f -' (A \text{ Un } B) = (f -' A) \text{ Un } (f -' B)$   
**by** *blast*

**lemma** *vimage-Int* [*simp*]:  $f -' (A \text{ Int } B) = (f -' A) \text{ Int } (f -' B)$   
**by** *fast*

**lemma** *vimage-Collect-eq* [*simp*]:  $f -' \text{Collect } P = \{y. P (f y)\}$   
**by** *blast*

**lemma** *vimage-Collect*:  $(!!x. P (f x) = Q x) ==> f -' (\text{Collect } P) = \text{Collect } Q$   
**by** *blast*

**lemma** *vimage-insert*:  $f -' (\text{insert } a B) = (f -' \{a\}) \text{ Un } (f -' B)$   
 — NOT suitable for rewriting because of the recurrence of  $\{a\}$ .  
**by** *blast*

**lemma** *vimage-Diff*:  $f -' (A - B) = (f -' A) - (f -' B)$   
**by** *blast*

**lemma** *vimage-UNIV* [*simp*]:  $f -' \text{UNIV} = \text{UNIV}$   
**by** *blast*

**lemma** *vimage-mono*:  $A \subseteq B ==> f -' A \subseteq f -' B$   
 — monotonicity  
**by** *blast*

**lemma** *vimage-image-eq* [*no-atp*]:  $f -' (f -' A) = \{y. \text{EX } x:A. f\ x = f\ y\}$   
**by** (*blast intro: sym*)

**lemma** *image-vimage-subset*:  $f -' (f -' A) \leq A$   
**by** *blast*

**lemma** *image-vimage-eq* [*simp*]:  $f -' (f -' A) = A \text{ Int range } f$   
**by** *blast*

**lemma** *vimage-const* [*simp*]:  $((\lambda x. c) -' A) = (\text{if } c \in A \text{ then UNIV else } \{\})$   
**by** *auto*

**lemma** *vimage-if* [*simp*]:  $((\lambda x. \text{if } x \in B \text{ then } c \text{ else } d) -' A) =$   
 $(\text{if } c \in A \text{ then } (\text{if } d \in A \text{ then UNIV else } B)$   
 $\text{else if } d \in A \text{ then } -B \text{ else } \{\})$   
**by** (*auto simp add: vimage-def*)

**lemma** *vimage-inter-cong*:  
 $(\bigwedge w. w \in S \implies f\ w = g\ w) \implies f -' y \cap S = g -' y \cap S$   
**by** *auto*

**lemma** *image-Int-subset*:  $f'(A \text{ Int } B) \leq f'A \text{ Int } f'B$   
**by** *blast*

**lemma** *image-diff-subset*:  $f'A - f'B \leq f'(A - B)$   
**by** *blast*

#### 6.4.6 Getting the Contents of a Singleton Set

**definition** *contents* :: 'a set  $\Rightarrow$  'a **where**  
 $[\text{code del}]: \text{contents } X = (\text{THE } x. X = \{x\})$

**lemma** *contents-eq* [*simp*]:  $\text{contents } \{x\} = x$   
**by** (*simp add: contents-def*)

#### 6.4.7 Least value operator

**lemma** *Least-mono*:  
 $\text{mono } (f::'a::\text{order} \Rightarrow 'b::\text{order}) \implies \text{EX } x:S. \text{ALL } y:S. x \leq y$   
 $\implies (\text{LEAST } y. y : f -' S) = f (\text{LEAST } x. x : S)$   
 — Courtesy of Stephan Merz  
**apply** *clarify*  
**apply** (*erule-tac P = %x. x : S in LeastI2-order, fast*)  
**apply** (*rule LeastI2-order*)  
**apply** (*auto elim: monoD intro!: order-antisym*)  
**done**

### 6.5 Misc

Rudimentary code generation

**lemma** *insert-code* [code]:  $\text{insert } y \ A \ x \longleftrightarrow y = x \vee A \ x$   
**by** (*auto simp add: insert-compr Collect-def mem-def*)

**lemma** *vimage-code* [code]:  $(f - 'A) \ x = A \ (f \ x)$   
**by** (*simp add: vimage-def Collect-def mem-def*)

Misc theorem and ML bindings

**lemmas** *equalityI = subset-antisym*

**ML**  $\ll$   
*val Ball-def* = @{thm *Ball-def*}  
*val Bex-def* = @{thm *Bex-def*}  
*val CollectD* = @{thm *CollectD*}  
*val CollectE* = @{thm *CollectE*}  
*val CollectI* = @{thm *CollectI*}  
*val Collect-conj-eq* = @{thm *Collect-conj-eq*}  
*val Collect-mem-eq* = @{thm *Collect-mem-eq*}  
*val IntD1* = @{thm *IntD1*}  
*val IntD2* = @{thm *IntD2*}  
*val IntE* = @{thm *IntE*}  
*val IntI* = @{thm *IntI*}  
*val Int-Collect* = @{thm *Int-Collect*}  
*val UNIV-I* = @{thm *UNIV-I*}  
*val UNIV-witness* = @{thm *UNIV-witness*}  
*val UnE* = @{thm *UnE*}  
*val UnI1* = @{thm *UnI1*}  
*val UnI2* = @{thm *UnI2*}  
*val ballE* = @{thm *ballE*}  
*val ballI* = @{thm *ballI*}  
*val bexCI* = @{thm *bexCI*}  
*val bexE* = @{thm *bexE*}  
*val bexI* = @{thm *bexI*}  
*val bex-triv* = @{thm *bex-triv*}  
*val bspec* = @{thm *bspec*}  
*val contra-subsetD* = @{thm *contra-subsetD*}  
*val distinct-lemma* = @{thm *distinct-lemma*}  
*val eq-to-mono* = @{thm *eq-to-mono*}  
*val equalityCE* = @{thm *equalityCE*}  
*val equalityD1* = @{thm *equalityD1*}  
*val equalityD2* = @{thm *equalityD2*}  
*val equalityE* = @{thm *equalityE*}  
*val equalityI* = @{thm *equalityI*}  
*val imageE* = @{thm *imageE*}  
*val imageI* = @{thm *imageI*}  
*val image-Un* = @{thm *image-Un*}  
*val image-insert* = @{thm *image-insert*}  
*val insert-commute* = @{thm *insert-commute*}  
*val insert-iff* = @{thm *insert-iff*}  
*val mem-Collect-eq* = @{thm *mem-Collect-eq*}

```

val rangeE = @{thm rangeE}
val rangeI = @{thm rangeI}
val range-eqI = @{thm range-eqI}
val subsetCE = @{thm subsetCE}
val subsetD = @{thm subsetD}
val subsetI = @{thm subsetI}
val subset-refl = @{thm subset-refl}
val subset-trans = @{thm subset-trans}
val vimageD = @{thm vimageD}
val vimageE = @{thm vimageE}
val vimageI = @{thm vimageI}
val vimageI2 = @{thm vimageI2}
val vimage-Collect = @{thm vimage-Collect}
val vimage-Int = @{thm vimage-Int}
val vimage-Un = @{thm vimage-Un}
>>

```

**end**

## 7 Typedef: HOL type definitions

```

theory Typedef
imports Set
uses
  (Tools/typedef.ML)
  (Tools/typecopy.ML)
  (Tools/typedef-codegen.ML)
begin

ML <<
  structure HOL = struct val thy = theory HOL end;
  >> — belongs to theory HOL

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep x ∈ A
    and Rep-inverse: Abs (Rep x) = x
    and Abs-inverse: y ∈ A ==> Rep (Abs y) = y
  — This will be axiomatized for each typedef!
begin

lemma Rep-inject:
  (Rep x = Rep y) = (x = y)
proof
  assume Rep x = Rep y
  then have Abs (Rep x) = Abs (Rep y) by (simp only:)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  moreover have Abs (Rep y) = y by (rule Rep-inverse)

```

```

    ultimately show  $x = y$  by simp
next
  assume  $x = y$ 
  thus  $\text{Rep } x = \text{Rep } y$  by (simp only:)
qed

```

```

lemma Abs-inject:
  assumes  $x: x \in A$  and  $y: y \in A$ 
  shows  $(\text{Abs } x = \text{Abs } y) = (x = y)$ 
proof
  assume  $\text{Abs } x = \text{Abs } y$ 
  then have  $\text{Rep } (\text{Abs } x) = \text{Rep } (\text{Abs } y)$  by (simp only:)
  moreover from  $x$  have  $\text{Rep } (\text{Abs } x) = x$  by (rule Abs-inverse)
  moreover from  $y$  have  $\text{Rep } (\text{Abs } y) = y$  by (rule Abs-inverse)
  ultimately show  $x = y$  by simp
next
  assume  $x = y$ 
  thus  $\text{Abs } x = \text{Abs } y$  by (simp only:)
qed

```

```

lemma Rep-cases [cases set]:
  assumes  $y: y \in A$ 
  and hyp:  $!!x. y = \text{Rep } x \implies P$ 
  shows  $P$ 
proof (rule hyp)
  from  $y$  have  $\text{Rep } (\text{Abs } y) = y$  by (rule Abs-inverse)
  thus  $y = \text{Rep } (\text{Abs } y)$  ..
qed

```

```

lemma Abs-cases [cases type]:
  assumes  $r: !!y. x = \text{Abs } y \implies y \in A \implies P$ 
  shows  $P$ 
proof (rule r)
  have  $\text{Abs } (\text{Rep } x) = x$  by (rule Rep-inverse)
  thus  $x = \text{Abs } (\text{Rep } x)$  ..
  show  $\text{Rep } x \in A$  by (rule Rep)
qed

```

```

lemma Rep-induct [induct set]:
  assumes  $y: y \in A$ 
  and hyp:  $!!x. P (\text{Rep } x)$ 
  shows  $P y$ 
proof -
  have  $P (\text{Rep } (\text{Abs } y))$  by (rule hyp)
  moreover from  $y$  have  $\text{Rep } (\text{Abs } y) = y$  by (rule Abs-inverse)
  ultimately show  $P y$  by simp
qed

```

```

lemma Abs-induct [induct type]:

```

```

    assumes  $r: !!y. y \in A ==> P (Abs\ y)$ 
    shows  $P\ x$ 
  proof -
    have  $Rep\ x \in A$  by (rule Rep)
    then have  $P (Abs (Rep\ x))$  by (rule r)
    moreover have  $Abs (Rep\ x) = x$  by (rule Rep-inverse)
    ultimately show  $P\ x$  by simp
  qed

  lemma Rep-range:  $range\ Rep = A$ 
  proof
    show  $range\ Rep \leq A$  using Rep by (auto simp add: image-def)
    show  $A \leq range\ Rep$ 
  proof
    fix  $x$  assume  $x : A$ 
    hence  $x = Rep (Abs\ x)$  by (rule Abs-inverse [symmetric])
    thus  $x : range\ Rep$  by (rule range-eqI)
  qed
  qed

  lemma Abs-image:  $Abs\ 'A = UNIV$ 
  proof
    show  $Abs\ 'A \leq UNIV$  by (rule subset-UNIV)
  next
    show  $UNIV \leq Abs\ 'A$ 
  proof
    fix  $x$ 
    have  $x = Abs (Rep\ x)$  by (rule Rep-inverse [symmetric])
    moreover have  $Rep\ x : A$  by (rule Rep)
    ultimately show  $x : Abs\ 'A$  by (rule image-eqI)
  qed
  qed

  end

  use Tools/typedef.ML setup Typedef.setup
  use Tools/typecopy.ML setup Typecopy.setup
  use Tools/typedef-codegen.ML setup TypedefCodegen.setup

  end

```

## 8 Complete-Lattice: Complete lattices, with special focus on sets

```

theory Complete-Lattice
imports Set
begin

```



**notation**

*less-eq* (**infix**  $\sqsubseteq$  50) **and**  
*less* (**infix**  $\sqsubset$  50) **and**  
*inf* (**infixl**  $\sqcap$  70) **and**  
*sup* (**infixl**  $\sqcup$  65) **and**  
*top* ( $\top$ ) **and**  
*bot* ( $\perp$ )

**8.1 Syntactic infimum and supremum operations**

**class** *Inf* =  
 fixes *Inf* :: 'a set  $\Rightarrow$  'a ( $\sqcap$  - [900] 900)

**class** *Sup* =  
 fixes *Sup* :: 'a set  $\Rightarrow$  'a ( $\sqcup$  - [900] 900)

**8.2 Abstract complete lattices**

**class** *complete-lattice* = *bounded-lattice* + *Inf* + *Sup* +  
 assumes *Inf-lower*:  $x \in A \Rightarrow \sqcap A \sqsubseteq x$   
 and *Inf-greatest*:  $(\bigwedge x. x \in A \Rightarrow z \sqsubseteq x) \Rightarrow z \sqsubseteq \sqcap A$   
 assumes *Sup-upper*:  $x \in A \Rightarrow x \sqsubseteq \sqcup A$   
 and *Sup-least*:  $(\bigwedge x. x \in A \Rightarrow x \sqsubseteq z) \Rightarrow \sqcup A \sqsubseteq z$   
**begin**

**lemma** *dual-complete-lattice*:

class.complete-lattice *Sup* (*op*  $\geq$ ) (*op*  $>$ ) (*op*  $\sqcup$ ) (*op*  $\sqcap$ )  $\top$   $\perp$   
**by** (*auto intro*: class.complete-lattice.intro dual-bounded-lattice)  
 (unfold-locales, (fact bot-least top-greatest  
 Sup-upper Sup-least Inf-lower Inf-greatest)+)

**lemma** *Inf-Sup*:  $\sqcap A = \sqcup \{b. \forall a \in A. b \sqsubseteq a\}$   
**by** (*auto intro*: antisym Inf-lower Inf-greatest Sup-upper Sup-least)

**lemma** *Sup-Inf*:  $\sqcup A = \sqcap \{b. \forall a \in A. a \sqsubseteq b\}$   
**by** (*auto intro*: antisym Inf-lower Inf-greatest Sup-upper Sup-least)

**lemma** *Inf-empty*:

$\sqcap \{\} = \top$   
**by** (*auto intro*: antisym Inf-greatest)

**lemma** *Sup-empty*:

$\sqcup \{\} = \perp$   
**by** (*auto intro*: antisym Sup-least)

**lemma** *Inf-insert*:  $\sqcap \text{insert } a \ A = a \sqcap \sqcap A$   
**by** (*auto intro*: le-infI le-infI1 le-infI2 antisym Inf-greatest Inf-lower)

**lemma** *Sup-insert*:  $\sqcup \text{insert } a \ A = a \sqcup \sqcup A$

**by** (*auto intro: le-supI le-supI1 le-supI2 antisym Sup-least Sup-upper*)

**lemma** *Inf-singleton* [*simp*]:

$\sqcap \{a\} = a$

**by** (*auto intro: antisym Inf-lower Inf-greatest*)

**lemma** *Sup-singleton* [*simp*]:

$\sqcup \{a\} = a$

**by** (*auto intro: antisym Sup-upper Sup-least*)

**lemma** *Inf-binary*:

$\sqcap \{a, b\} = a \sqcap b$

**by** (*simp add: Inf-empty Inf-insert*)

**lemma** *Sup-binary*:

$\sqcup \{a, b\} = a \sqcup b$

**by** (*simp add: Sup-empty Sup-insert*)

**lemma** *Inf-UNIV*:

$\sqcap UNIV = bot$

**by** (*simp add: Sup-Inf Sup-empty [symmetric]*)

**lemma** *Sup-UNIV*:

$\sqcup UNIV = top$

**by** (*simp add: Inf-Sup Inf-empty [symmetric]*)

**lemma** *Sup-le-iff*:  $Sup\ A \sqsubseteq b \longleftrightarrow (\forall a \in A. a \sqsubseteq b)$

**by** (*auto intro: Sup-least dest: Sup-upper*)

**lemma** *le-Inf-iff*:  $b \sqsubseteq Inf\ A \longleftrightarrow (\forall a \in A. b \sqsubseteq a)$

**by** (*auto intro: Inf-greatest dest: Inf-lower*)

**definition** *SUPR* ::  $'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$  **where**

$SUPR\ A\ f = \sqcup (f \, 'A)$

**definition** *INFI* ::  $'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$  **where**

$INFI\ A\ f = \sqcap (f \, 'A)$

**end**

**syntax**

-*SUP1*    ::  $pttrns \Rightarrow 'b \Rightarrow 'b$                      $((\exists SUP \ -./ \ -) [0, 10] 10)$   
 -*SUP*    ::  $pttrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b$              $((\exists SUP \ -./ \ -) [0, 0, 10] 10)$   
 -*INF1*    ::  $pttrns \Rightarrow 'b \Rightarrow 'b$                      $((\exists INF \ -./ \ -) [0, 10] 10)$   
 -*INF*    ::  $pttrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b$              $((\exists INF \ -./ \ -) [0, 0, 10] 10)$

**translations**

$SUP\ x\ y. B \quad == \quad SUP\ x. SUP\ y. B$   
 $SUP\ x. B \quad == \quad CONST\ SUPR\ CONST\ UNIV\ (\%x. B)$

$$\begin{aligned}
SUP\ x.\ B &== SUP\ x:CONST\ UNIV.\ B \\
SUP\ x:A.\ B &== CONST\ SUPR\ A\ (\%x.\ B) \\
INF\ x\ y.\ B &== INF\ x.\ INF\ y.\ B \\
INF\ x.\ B &== CONST\ INFI\ CONST\ UNIV\ (\%x.\ B) \\
INF\ x.\ B &== INF\ x:CONST\ UNIV.\ B \\
INF\ x:A.\ B &== CONST\ INFI\ A\ (\%x.\ B)
\end{aligned}$$

**print-translation**  $\ll$   
 $[Syntax.preserve-binder-abs2-tr' @ \{const-syntax\ SUPR\} @ \{syntax-const -SUP\},$   
 $Syntax.preserve-binder-abs2-tr' @ \{const-syntax\ INFI\} @ \{syntax-const -INF\}]$   
 $\gg$  — to avoid eta-contraction of body

**context** *complete-lattice*  
**begin**

**lemma** *le-SUPI*:  $i : A \implies M\ i \sqsubseteq (SUP\ i:A.\ M\ i)$   
**by** (*auto simp add: SUPR-def intro: Sup-upper*)

**lemma** *SUP-leI*:  $(\bigwedge i.\ i : A \implies M\ i \sqsubseteq u) \implies (SUP\ i:A.\ M\ i) \sqsubseteq u$   
**by** (*auto simp add: SUPR-def intro: Sup-least*)

**lemma** *INF-leI*:  $i : A \implies (INF\ i:A.\ M\ i) \sqsubseteq M\ i$   
**by** (*auto simp add: INFI-def intro: Inf-lower*)

**lemma** *le-INFI*:  $(\bigwedge i.\ i : A \implies u \sqsubseteq M\ i) \implies u \sqsubseteq (INF\ i:A.\ M\ i)$   
**by** (*auto simp add: INFI-def intro: Inf-greatest*)

**lemma** *SUP-le-iff*:  $(SUP\ i:A.\ M\ i) \sqsubseteq u \longleftrightarrow (\forall i \in A.\ M\ i \sqsubseteq u)$   
**unfolding** *SUPR-def* **by** (*auto simp add: Sup-le-iff*)

**lemma** *le-INF-iff*:  $u \sqsubseteq (INF\ i:A.\ M\ i) \longleftrightarrow (\forall i \in A.\ u \sqsubseteq M\ i)$   
**unfolding** *INFI-def* **by** (*auto simp add: le-Inf-iff*)

**lemma** *SUP-const[simp]*:  $A \neq \{\} \implies (SUP\ i:A.\ M) = M$   
**by** (*auto intro: antisym SUP-leI le-SUPI*)

**lemma** *INF-const[simp]*:  $A \neq \{\} \implies (INF\ i:A.\ M) = M$   
**by** (*auto intro: antisym INF-leI le-INFI*)

**end**

### 8.3 *bool* and $- \Rightarrow -$ as complete lattice

**instantiation** *bool* :: *complete-lattice*  
**begin**

**definition**  
*Inf-bool-def*:  $\bigcap A \longleftrightarrow (\forall x \in A.\ x)$

**definition**

*Sup-bool-def*:  $\sqcup A \longleftrightarrow (\exists x \in A. x)$

**instance proof**

**qed** (*auto simp add: Inf-bool-def Sup-bool-def le-bool-def*)

**end**

**lemma** *Inf-empty-bool* [*simp*]:

$\sqcap \{\}$

**unfolding** *Inf-bool-def* **by** *auto*

**lemma** *not-Sup-empty-bool* [*simp*]:

$\neg \sqcup \{\}$

**unfolding** *Sup-bool-def* **by** *auto*

**lemma** *INFI-bool-eq*:

*INFI* = *Ball*

**proof** (*rule ext*)+

**fix** *A* :: 'a set

**fix** *P* :: 'a  $\Rightarrow$  bool

**show**  $(\text{INF } x:A. P\ x) \longleftrightarrow (\forall x \in A. P\ x)$

**by** (*auto simp add: Ball-def INFI-def Inf-bool-def*)

**qed**

**lemma** *SUPR-bool-eq*:

*SUPR* = *Bex*

**proof** (*rule ext*)+

**fix** *A* :: 'a set

**fix** *P* :: 'a  $\Rightarrow$  bool

**show**  $(\text{SUP } x:A. P\ x) \longleftrightarrow (\exists x \in A. P\ x)$

**by** (*auto simp add: Bex-def SUPR-def Sup-bool-def*)

**qed**

**instantiation** *fun* :: (*type*, *complete-lattice*) *complete-lattice*

**begin**

**definition**

*Inf-fun-def* [*code del*]:  $\sqcap A = (\lambda x. \sqcap \{y. \exists f \in A. y = f\ x\})$

**definition**

*Sup-fun-def* [*code del*]:  $\sqcup A = (\lambda x. \sqcup \{y. \exists f \in A. y = f\ x\})$

**instance proof**

**qed** (*auto simp add: Inf-fun-def Sup-fun-def le-fun-def*

*intro: Inf-lower Sup-upper Inf-greatest Sup-least*)

**end**

**lemma** *Inf-empty-fun*:

$$\sqcap \{\} = (\lambda \cdot. \sqcap \{\})$$

**by** (*simp add: Inf-fun-def*)

**lemma** *Sup-empty-fun*:

$$\sqcup \{\} = (\lambda \cdot. \sqcup \{\})$$

**by** (*simp add: Sup-fun-def*)

## 8.4 Union

**abbreviation** *Union* :: 'a set set  $\Rightarrow$  'a set **where**

$$Union\ S \equiv \sqcup S$$

**notation** (*xsymbols*)

$$Union\ (\bigcup - [90] 90)$$

**lemma** *Union-eq*:

$$\bigcup A = \{x. \exists B \in A. x \in B\}$$

**proof** (*rule set-ext*)

**fix** *x*

$$\text{have } (\exists Q \in \{P. \exists B \in A. P \longleftrightarrow x \in B\}. Q) \longleftrightarrow (\exists B \in A. x \in B)$$

**by** *auto*

$$\text{then show } x \in \bigcup A \longleftrightarrow x \in \{x. \exists B \in A. x \in B\}$$

**by** (*simp add: Sup-fun-def Sup-bool-def*) (*simp add: mem-def*)

**qed**

**lemma** *Union-iff* [*simp, no-atp*]:

$$A \in \bigcup C \longleftrightarrow (\exists X \in C. A \in X)$$

**by** (*unfold Union-eq blast*)

**lemma** *UnionI* [*intro*]:

$$X \in C \Longrightarrow A \in X \Longrightarrow A \in \bigcup C$$

— The order of the premises presupposes that *C* is rigid; *A* may be flexible.

**by** *auto*

**lemma** *UnionE* [*elim!*]:

$$A \in \bigcup C \Longrightarrow (\bigwedge X. A \in X \Longrightarrow X \in C \Longrightarrow R) \Longrightarrow R$$

**by** *auto*

**lemma** *Union-upper*:  $B \in A \Longrightarrow B \subseteq Union\ A$

**by** (*iprover intro: subsetI UnionI*)

**lemma** *Union-least*:  $(!!X. X \in A \Longrightarrow X \subseteq C) \Longrightarrow Union\ A \subseteq C$

**by** (*iprover intro: subsetI elim: UnionE dest: subsetD*)

**lemma** *Un-eq-Union*:  $A \cup B = \bigcup \{A, B\}$

**by** *blast*

**lemma** *Union-empty* [*simp*]:  $Union(\{\}) = \{\}$

by *blast*

**lemma** *Union-UNIV* [simp]: *Union UNIV = UNIV*  
by *blast*

**lemma** *Union-insert* [simp]: *Union (insert a B) = a  $\cup$   $\bigcup B$*   
by *blast*

**lemma** *Union-Un-distrib* [simp]:  *$\bigcup (A \text{ Un } B) = \bigcup A \cup \bigcup B$*   
by *blast*

**lemma** *Union-Int-subset*:  *$\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$*   
by *blast*

**lemma** *Union-empty-conv* [simp,no-atp]:  *$(\bigcup A = \{\}) = (\forall x \in A. x = \{\})$*   
by *blast*

**lemma** *empty-Union-conv* [simp,no-atp]:  *$(\{\} = \bigcup A) = (\forall x \in A. x = \{\})$*   
by *blast*

**lemma** *Union-disjoint*:  *$(\bigcup C \cap A = \{\}) = (\forall B \in C. B \cap A = \{\})$*   
by *blast*

**lemma** *subset-Pow-Union*:  *$A \subseteq \text{Pow } (\bigcup A)$*   
by *blast*

**lemma** *Union-Pow-eq* [simp]:  *$\bigcup (\text{Pow } A) = A$*   
by *blast*

**lemma** *Union-mono*:  *$A \subseteq B \implies \bigcup A \subseteq \bigcup B$*   
by *blast*

## 8.5 Unions of families

**abbreviation** *UNION* :: *'a set  $\Rightarrow$  ('a  $\Rightarrow$  'b set)  $\Rightarrow$  'b set* **where**  
*UNION  $\equiv$  SUPR*

**syntax**

-UNION1    :: *pttrns  $\Rightarrow$  'b set  $\Rightarrow$  'b set*    *(( $\exists UN$  -./ -) [0, 10] 10)*  
-UNION    :: *pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  'b set* *(( $\exists UN$  -./ -) [0, 0, 10] 10)*

**syntax** (*xsymbols*)

-UNION1    :: *pttrns  $\Rightarrow$  'b set  $\Rightarrow$  'b set*    *(( $\exists \bigcup$  -./ -) [0, 10] 10)*  
-UNION    :: *pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  'b set* *(( $\exists \bigcup$  - $\in$ -./ -) [0, 0, 10] 10)*

**syntax** (*latex output*)

-UNION1    :: *pttrns  $\Rightarrow$  'b set  $\Rightarrow$  'b set*    *(( $\exists \bigcup (00.)$  / -) [0, 10] 10)*  
-UNION    :: *pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  'b set* *(( $\exists \bigcup (00.\in)$  / -) [0, 0, 10] 10)*

10)

**translations**

$$\begin{aligned} UN\ x\ y.\ B &== UN\ x.\ UN\ y.\ B \\ UN\ x.\ B &== CONST\ UNION\ CONST\ UNIV\ (\%x.\ B) \\ UN\ x.\ B &== UN\ x:CONST\ UNIV.\ B \\ UN\ x:A.\ B &== CONST\ UNION\ A\ (\%x.\ B) \end{aligned}$$

Note the difference between ordinary xsymbol syntax of indexed unions and intersections (e.g.  $\bigcup_{a_1 \in A_1} B$ ) and their L<sup>A</sup>T<sub>E</sub>X rendition:  $\bigcup_{a_1 \in A_1} B$ . The former does not make the index expression a subscript of the union/intersection symbol because this leads to problems with nested subscripts in Proof General.

**print-translation**  $\ll$ 

$$[Syntax.preserve-binder-abs2-tr' @ \{const-syntax\ UNION\} @ \{syntax-const-UNION\}]$$

$\gg$  — to avoid eta-contraction of body

**lemma** *UNION-eq-Union-image*:
$$(\bigcup_{x \in A} B\ x) = \bigcup (B'A)$$

**by** (*fact SUPR-def*)

**lemma** *Union-def*:
$$\bigcup S = (\bigcup_{x \in S} x)$$

**by** (*simp add: UNION-eq-Union-image image-def*)

**lemma** *UNION-def* [*no-atp*]:
$$(\bigcup_{x \in A} B\ x) = \{y. \exists x \in A. y \in B\ x\}$$

**by** (*auto simp add: UNION-eq-Union-image Union-eq*)

**lemma** *Union-image-eq* [*simp*]:
$$\bigcup (B'A) = (\bigcup_{x \in A} B\ x)$$

**by** (*rule sym*) (*fact UNION-eq-Union-image*)

**lemma** *UN-iff* [*simp*]: ( $b: (UN\ x:A.\ B\ x)$ ) = ( $EX\ x:A.\ b: B\ x$ )

**by** (*unfold UNION-def*) *blast*

**lemma** *UN-I* [*intro*]:  $a:A ==> b: B\ a ==> b: (UN\ x:A.\ B\ x)$ 

— The order of the premises presupposes that  $A$  is rigid;  $b$  may be flexible.

**by** *auto*

**lemma** *UN-E* [*elim!*]:  $b : (UN\ x:A.\ B\ x) ==> (!!x. x:A ==> b: B\ x ==> R) ==> R$ 

**by** (*unfold UNION-def*) *blast*

**lemma** *UN-cong* [*cong*]:
$$A = B ==> (!!x. x:B ==> C\ x = D\ x) ==> (UN\ x:A.\ C\ x) = (UN\ x:B.\ D\ x)$$

**by** (*simp add: UNION-def*)

**lemma** *strong-UN-cong*:

$A = B \implies (!x. x:B \text{ simp} \implies C\ x = D\ x) \implies (\bigcup x:A. C\ x) = (\bigcup x:B. D\ x)$   
**by** (*simp add: UNION-def simp-implies-def*)

**lemma** *image-eq-UN*:  $f^{\ast}A = (\bigcup x:A. \{f\ x\})$

**by** *blast*

**lemma** *UN-upper*:  $a \in A \implies B\ a \subseteq (\bigcup x \in A. B\ x)$

**by** (*fact le-SUPI*)

**lemma** *UN-least*:  $(!x. x \in A \implies B\ x \subseteq C) \implies (\bigcup x \in A. B\ x) \subseteq C$

**by** (*iprover intro: subsetI elim: UN-E dest: subsetD*)

**lemma** *Collect-bex-eq* [*no-atp*]:  $\{x. \exists y \in A. P\ x\ y\} = (\bigcup y \in A. \{x. P\ x\ y\})$

**by** *blast*

**lemma** *UN-insert-distrib*:  $u \in A \implies (\bigcup x \in A. \text{insert}\ a\ (B\ x)) = \text{insert}\ a\ (\bigcup x \in A. B\ x)$

**by** *blast*

**lemma** *UN-empty* [*simp,no-atp*]:  $(\bigcup x \in \{\}. B\ x) = \{\}$

**by** *blast*

**lemma** *UN-empty2* [*simp*]:  $(\bigcup x \in A. \{\}) = \{\}$

**by** *blast*

**lemma** *UN-singleton* [*simp*]:  $(\bigcup x \in A. \{x\}) = A$

**by** *blast*

**lemma** *UN-absorb*:  $k \in I \implies A\ k \cup (\bigcup i \in I. A\ i) = (\bigcup i \in I. A\ i)$

**by** *auto*

**lemma** *UN-insert* [*simp*]:  $(\bigcup x \in \text{insert}\ a\ A. B\ x) = B\ a \cup \text{UNION}\ A\ B$

**by** *blast*

**lemma** *UN-Un* [*simp*]:  $(\bigcup i \in A \cup B. M\ i) = (\bigcup i \in A. M\ i) \cup (\bigcup i \in B. M\ i)$

**by** *blast*

**lemma** *UN-UN-flatten*:  $(\bigcup x \in (\bigcup y \in A. B\ y). C\ x) = (\bigcup y \in A. \bigcup x \in B\ y. C\ x)$

**by** *blast*

**lemma** *UN-subset-iff*:  $((\bigcup i \in I. A\ i) \subseteq B) = (\forall i \in I. A\ i \subseteq B)$

**by** (*fact SUP-le-iff*)

**lemma** *image-Union*:  $f^{\ast} \bigcup S = (\bigcup x \in S. f^{\ast} x)$

**by** *blast*

**lemma** *UN-constant* [*simp*]:  $(\bigcup y \in A. c) = (\text{if}\ A = \{\}\ \text{then}\ \{\}\ \text{else}\ c)$



by *auto*

**lemma** *UN-eq*:  $(\bigcup x \in A. B\ x) = \bigcup (\{Y. \exists x \in A. Y = B\ x\})$   
by *blast*

**lemma** *UNION-empty-conv*[*simp*]:  
 $(\{\} = (\bigcup x:A. B\ x)) = (\forall x \in A. B\ x = \{\})$   
 $((\bigcup x:A. B\ x) = \{\}) = (\forall x \in A. B\ x = \{\})$   
by *blast+*

**lemma** *Collect-ex-eq* [*no-atp*]:  $\{x. \exists y. P\ x\ y\} = (\bigcup y. \{x. P\ x\ y\})$   
by *blast*

**lemma** *ball-UN*:  $(\forall z \in \text{UNION } A\ B. P\ z) = (\forall x \in A. \forall z \in B\ x. P\ z)$   
by *blast*

**lemma** *bex-UN*:  $(\exists z \in \text{UNION } A\ B. P\ z) = (\exists x \in A. \exists z \in B\ x. P\ z)$   
by *blast*

**lemma** *Un-eq-UN*:  $A \cup B = (\bigcup b. \text{if } b \text{ then } A \text{ else } B)$   
by (*auto simp add: split-if-mem2*)

**lemma** *UN-bool-eq*:  $(\bigcup b::\text{bool}. A\ b) = (A\ \text{True} \cup A\ \text{False})$   
by (*auto intro: bool-contrapos*)

**lemma** *UN-Pow-subset*:  $(\bigcup x \in A. \text{Pow } (B\ x)) \subseteq \text{Pow } (\bigcup x \in A. B\ x)$   
by *blast*

**lemma** *UN-mono*:  
 $A \subseteq B \implies (!x. x \in A \implies f\ x \subseteq g\ x) \implies$   
 $(\bigcup x \in A. f\ x) \subseteq (\bigcup x \in B. g\ x)$   
by (*blast dest: subsetD*)

**lemma** *image-Union*:  $f\ -' (\text{Union } A) = (\bigcup X:A. f\ -' X)$   
by *blast*

**lemma** *image-UN*:  $f\ -' (\bigcup x:A. B\ x) = (\bigcup x:A. f\ -' B\ x)$   
by *blast*

**lemma** *image-eq-UN*:  $f\ -' B = (\bigcup y: B. f\ -' \{y\})$   
— NOT suitable for rewriting  
by *blast*

**lemma** *image-UN*:  $(f\ -' (\text{UNION } A\ B)) = (\bigcup x:A. (f\ -' (B\ x)))$   
by *blast*

## 8.6 Inter

**abbreviation** *Inter* :: 'a set set  $\Rightarrow$  'a set **where**

$Inter\ S \equiv \bigcap S$

**notation** (*xsymbols*)

$Inter\ (\bigcap - [90] 90)$

**lemma** *Inter-eq* [*code del*]:

$\bigcap A = \{x. \forall B \in A. x \in B\}$

**proof** (*rule set-ext*)

**fix**  $x$

**have**  $(\forall Q \in \{P. \exists B \in A. P \longleftrightarrow x \in B\}. Q) \longleftrightarrow (\forall B \in A. x \in B)$

**by** *auto*

**then show**  $x \in \bigcap A \longleftrightarrow x \in \{x. \forall B \in A. x \in B\}$

**by** (*simp add: Inf-fun-def Inf-bool-def*) (*simp add: mem-def*)

**qed**

**lemma** *Inter-iff* [*simp,no-atp*]:  $(A : Inter\ C) = (ALL\ X:C. A:X)$

**by** (*unfold Inter-eq*) *blast*

**lemma** *InterI* [*intro!*]:  $(!!X. X:C ==> A:X) ==> A : Inter\ C$

**by** (*simp add: Inter-eq*)

A “destruct” rule – every  $X$  in  $C$  contains  $A$  as an element, but  $A \in X$  can hold when  $X \in C$  does not! This rule is analogous to *spec*.

**lemma** *InterD* [*elim*]:  $A : Inter\ C ==> X:C ==> A:X$

**by** *auto*

**lemma** *InterE* [*elim*]:  $A : Inter\ C ==> (X\sim:C ==> R) ==> (A:X ==> R) ==> R$

— “Classical” elimination rule – does not require proving  $X \in C$ .

**by** (*unfold Inter-eq*) *blast*

**lemma** *Inter-lower*:  $B \in A ==> Inter\ A \subseteq B$

**by** *blast*

**lemma** *Inter-subset*:

$[! X. X \in A ==> X \subseteq B; A \sim = \{\}] ==> \bigcap A \subseteq B$

**by** *blast*

**lemma** *Inter-greatest*:  $(!!X. X \in A ==> C \subseteq X) ==> C \subseteq Inter\ A$

**by** (*iprover intro: InterI subsetI dest: subsetD*)

**lemma** *Int-eq-Inter*:  $A \cap B = \bigcap \{A, B\}$

**by** *blast*

**lemma** *Inter-empty* [*simp*]:  $\bigcap \{\} = UNIV$

**by** *blast*

**lemma** *Inter-UNIV* [*simp*]:  $\bigcap UNIV = \{\}$

**by** *blast*

**lemma** *Inter-insert* [simp]:  $\bigcap (\text{insert } a \ B) = a \cap \bigcap B$   
**by** blast

**lemma** *Inter-Un-subset*:  $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$   
**by** blast

**lemma** *Inter-Un-distrib*:  $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$   
**by** blast

**lemma** *Inter-UNIV-conv* [simp,no-atp]:  
 $(\bigcap A = \text{UNIV}) = (\forall x \in A. x = \text{UNIV})$   
 $(\text{UNIV} = \bigcap A) = (\forall x \in A. x = \text{UNIV})$   
**by** blast+

**lemma** *Inter-anti-mono*:  $B \subseteq A \implies \bigcap A \subseteq \bigcap B$   
**by** blast

## 8.7 Intersections of families

**abbreviation** *INTER* :: 'a set  $\Rightarrow$  ('a  $\Rightarrow$  'b set)  $\Rightarrow$  'b set **where**  
*INTER*  $\equiv$  *INFI*

**syntax**  
*-INTER1* :: ptrns  $\Rightarrow$  'b set  $\Rightarrow$  'b set  $((\exists \text{INT} \text{ -./ -}) [0, 10] 10)$   
*-INTER* :: pptrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  'b set  $((\exists \text{INT} \text{ :-./ -}) [0, 0, 10] 10)$

**syntax** (xsymbols)  
*-INTER1* :: ptrns  $\Rightarrow$  'b set  $\Rightarrow$  'b set  $((\exists \bigcap \text{ -./ -}) [0, 10] 10)$   
*-INTER* :: pptrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  'b set  $((\exists \bigcap \text{ -}\in\text{ -./ -}) [0, 0, 10] 10)$

**syntax** (latex output)  
*-INTER1* :: ptrns  $\Rightarrow$  'b set  $\Rightarrow$  'b set  $((\exists \bigcap (00\text{ -}) / \text{ -}) [0, 10] 10)$   
*-INTER* :: pptrn  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  'b set  $((\exists \bigcap (00\text{ -}\in\text{ -}) / \text{ -}) [0, 0, 10] 10)$

**translations**  
 $\text{INT } x \ y. B == \text{INT } x. \text{INT } y. B$   
 $\text{INT } x. B == \text{CONST } \text{INTER } \text{CONST } \text{UNIV } (\%x. B)$   
 $\text{INT } x. B == \text{INT } x:\text{CONST } \text{UNIV}. B$   
 $\text{INT } x:A. B == \text{CONST } \text{INTER } A (\%x. B)$

**print-translation**  $\llbracket$   
 $\llbracket \text{Syntax.preserve-binder-abs2-tr}' @ \{ \text{const-syntax } \text{INTER} \} @ \{ \text{syntax-const -INTER} \} \rrbracket$   
 $\rrbracket$  — to avoid eta-contraction of body

**lemma** *INTER-eq-Inter-image*:  
 $(\bigcap x \in A. B \ x) = \bigcap (B \ A)$

**by** (*fact INFI-def*)

**lemma** *Inter-def*:

$\bigcap S = (\bigcap_{x \in S}. x)$

**by** (*simp add: INTER-eq-Inter-image image-def*)

**lemma** *INTER-def*:

$(\bigcap_{x \in A}. B\ x) = \{y. \forall x \in A. y \in B\ x\}$

**by** (*auto simp add: INTER-eq-Inter-image Inter-eq*)

**lemma** *Inter-image-eq* [*simp*]:

$\bigcap (B^i A) = (\bigcap_{x \in A}. B\ x)$

**by** (*rule sym*) (*fact INTER-eq-Inter-image*)

**lemma** *INT-iff* [*simp*]:  $(b: (INT\ x:A. B\ x)) = (ALL\ x:A. b: B\ x)$

**by** (*unfold INTER-def*) *blast*

**lemma** *INT-I* [*intro!*]:  $(!!x. x:A ==> b: B\ x) ==> b: (INT\ x:A. B\ x)$

**by** (*unfold INTER-def*) *blast*

**lemma** *INT-D* [*elim*]:  $b: (INT\ x:A. B\ x) ==> a:A ==> b: B\ a$

**by** *auto*

**lemma** *INT-E* [*elim*]:  $b: (INT\ x:A. B\ x) ==> (b: B\ a ==> R) ==> (a^{\sim}:A ==> R) ==> R$

— “Classical” elimination – by the Excluded Middle on  $a \in A$ .

**by** (*unfold INTER-def*) *blast*

**lemma** *INT-cong* [*cong*]:

$A = B ==> (!!x. x:B ==> C\ x = D\ x) ==> (INT\ x:A. C\ x) = (INT\ x:B. D\ x)$

**by** (*simp add: INTER-def*)

**lemma** *Collect-ball-eq*:  $\{x. \forall y \in A. P\ x\ y\} = (\bigcap_{y \in A}. \{x. P\ x\ y\})$

**by** *blast*

**lemma** *Collect-all-eq*:  $\{x. \forall y. P\ x\ y\} = (\bigcap y. \{x. P\ x\ y\})$

**by** *blast*

**lemma** *INT-lower*:  $a \in A ==> (\bigcap_{x \in A}. B\ x) \subseteq B\ a$

**by** (*fact INF-leI*)

**lemma** *INT-greatest*:  $(!!x. x \in A ==> C \subseteq B\ x) ==> C \subseteq (\bigcap_{x \in A}. B\ x)$

**by** (*fact le-INF*)

**lemma** *INT-empty* [*simp*]:  $(\bigcap_{x \in \{\}}. B\ x) = UNIV$

**by** *blast*

**lemma** *INT-absorb*:  $k \in I ==> A\ k \cap (\bigcap_{i \in I}. A\ i) = (\bigcap_{i \in I}. A\ i)$

by *blast*

**lemma** *INT-subset-iff*:  $(B \subseteq (\bigcap_{i \in I}. A \ i)) = (\forall i \in I. B \subseteq A \ i)$   
by *(fact le-INF-iff)*

**lemma** *INT-insert [simp]*:  $(\bigcap x \in \text{insert } a \ A. B \ x) = B \ a \cap \text{INTER } A \ B$   
by *blast*

**lemma** *INT-Un*:  $(\bigcap i \in A \cup B. M \ i) = (\bigcap i \in A. M \ i) \cap (\bigcap i \in B. M \ i)$   
by *blast*

**lemma** *INT-insert-distrib*:  
 $u \in A ==> (\bigcap x \in A. \text{insert } a \ (B \ x)) = \text{insert } a \ (\bigcap x \in A. B \ x)$   
by *blast*

**lemma** *INT-constant [simp]*:  $(\bigcap y \in A. c) = (\text{if } A = \{\} \text{ then } \text{UNIV} \text{ else } c)$   
by *auto*

**lemma** *INT-eq*:  $(\bigcap x \in A. B \ x) = \bigcap (\{Y. \exists x \in A. Y = B \ x\})$   
— Look: it has an *existential* quantifier  
by *blast*

**lemma** *INTER-UNIV-conv[simp]*:  
 $(\text{UNIV} = (\text{INT } x:A. B \ x)) = (\forall x \in A. B \ x = \text{UNIV})$   
 $((\text{INT } x:A. B \ x) = \text{UNIV}) = (\forall x \in A. B \ x = \text{UNIV})$   
by *blast+*

**lemma** *INT-bool-eq*:  $(\bigcap b::\text{bool}. A \ b) = (A \ \text{True} \cap A \ \text{False})$   
by *(auto intro: bool-induct)*

**lemma** *Pow-INT-eq*:  $\text{Pow } (\bigcap x \in A. B \ x) = (\bigcap x \in A. \text{Pow } (B \ x))$   
by *blast*

**lemma** *INT-anti-mono*:  
 $B \subseteq A ==> (!x. x \in A ==> f \ x \subseteq g \ x) ==>$   
 $(\bigcap x \in A. f \ x) \subseteq (\bigcap x \in A. g \ x)$   
— The last inclusion is POSITIVE!  
by *(blast dest: subsetD)*

**lemma** *image-INT*:  $f - ' (\text{INT } x:A. B \ x) = (\text{INT } x:A. f - ' B \ x)$   
by *blast*

## 8.8 Distributive laws

**lemma** *Int-Union*:  $A \cap \bigcup B = (\bigcup C \in B. A \cap C)$   
by *blast*

**lemma** *Int-Union2*:  $\bigcup B \cap A = (\bigcup C \in B. C \cap A)$   
by *blast*

**lemma** *Un-Union-image*:  $(\bigcup_{x \in C}. A \ x \cup B \ x) = \bigcup (A' C) \cup \bigcup (B' C)$   
 — Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:  
 — Union of a family of unions  
**by** *blast*

**lemma** *UN-Un-distrib*:  $(\bigcup_{i \in I}. A \ i \cup B \ i) = (\bigcup_{i \in I}. A \ i) \cup (\bigcup_{i \in I}. B \ i)$   
 — Equivalent version  
**by** *blast*

**lemma** *Un-Inter*:  $A \cup \bigcap B = (\bigcap_{C \in B}. A \cup C)$   
**by** *blast*

**lemma** *Int-Inter-image*:  $(\bigcap_{x \in C}. A \ x \cap B \ x) = \bigcap (A' C) \cap \bigcap (B' C)$   
**by** *blast*

**lemma** *INT-Int-distrib*:  $(\bigcap_{i \in I}. A \ i \cap B \ i) = (\bigcap_{i \in I}. A \ i) \cap (\bigcap_{i \in I}. B \ i)$   
 — Equivalent version  
**by** *blast*

**lemma** *Int-UN-distrib*:  $B \cap (\bigcup_{i \in I}. A \ i) = (\bigcup_{i \in I}. B \cap A \ i)$   
 — Halmos, Naive Set Theory, page 35.  
**by** *blast*

**lemma** *Un-INT-distrib*:  $B \cup (\bigcap_{i \in I}. A \ i) = (\bigcap_{i \in I}. B \cup A \ i)$   
**by** *blast*

**lemma** *Int-UN-distrib2*:  $(\bigcup_{i \in I}. A \ i) \cap (\bigcup_{j \in J}. B \ j) = (\bigcup_{i \in I}. \bigcup_{j \in J}. A \ i \cap B \ j)$   
**by** *blast*

**lemma** *Un-INT-distrib2*:  $(\bigcap_{i \in I}. A \ i) \cup (\bigcap_{j \in J}. B \ j) = (\bigcap_{i \in I}. \bigcap_{j \in J}. A \ i \cup B \ j)$   
**by** *blast*

## 8.9 Complement

**lemma** *Compl-UN* [*simp*]:  $-(\bigcup_{x \in A}. B \ x) = (\bigcap_{x \in A}. -B \ x)$   
**by** *blast*

**lemma** *Compl-INT* [*simp*]:  $-(\bigcap_{x \in A}. B \ x) = (\bigcup_{x \in A}. -B \ x)$   
**by** *blast*

## 8.10 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

**lemma** *UN-simps* [*simp*]:  
 $!!a \ B \ C. (UN \ x:C. insert \ a \ (B \ x)) = (if \ C=\{\} \ then \ \{\} \ else \ insert \ a \ (UN \ x:C. B \ x))$

$!!A \ B \ C. (UN \ x:C. A \ x \ Un \ B) = ((if \ C=\{\} \ then \ \{\} \ else \ (UN \ x:C. A \ x) \ Un \ B))$   
 $!!A \ B \ C. (UN \ x:C. A \ Un \ B \ x) = ((if \ C=\{\} \ then \ \{\} \ else \ A \ Un \ (UN \ x:C. B \ x)))$   
 $!!A \ B \ C. (UN \ x:C. A \ x \ Int \ B) = ((UN \ x:C. A \ x) \ Int \ B)$   
 $!!A \ B \ C. (UN \ x:C. A \ Int \ B \ x) = (A \ Int \ (UN \ x:C. B \ x))$   
 $!!A \ B \ C. (UN \ x:C. A \ x - B) = ((UN \ x:C. A \ x) - B)$   
 $!!A \ B \ C. (UN \ x:C. A - B \ x) = (A - (INT \ x:C. B \ x))$   
 $!!A \ B. (UN \ x: Union \ A. B \ x) = (UN \ y:A. UN \ x:y. B \ x)$   
 $!!A \ B \ C. (UN \ z: UNION \ A \ B. C \ z) = (UN \ x:A. UN \ z: B(x). C \ z)$   
 $!!A \ B \ f. (UN \ x:f'A. B \ x) = (UN \ a:A. B \ (f \ a))$   
**by auto**

**lemma** *INT-simps* [*simp*]:

$!!A \ B \ C. (INT \ x:C. A \ x \ Int \ B) = (if \ C=\{\} \ then \ UNIV \ else \ (INT \ x:C. A \ x) \ Int \ B)$   
 $!!A \ B \ C. (INT \ x:C. A \ Int \ B \ x) = (if \ C=\{\} \ then \ UNIV \ else \ A \ Int \ (INT \ x:C. B \ x))$   
 $!!A \ B \ C. (INT \ x:C. A \ x - B) = (if \ C=\{\} \ then \ UNIV \ else \ (INT \ x:C. A \ x) - B)$   
 $!!A \ B \ C. (INT \ x:C. A - B \ x) = (if \ C=\{\} \ then \ UNIV \ else \ A - (UN \ x:C. B \ x))$   
 $!!a \ B \ C. (INT \ x:C. insert \ a \ (B \ x)) = insert \ a \ (INT \ x:C. B \ x)$   
 $!!A \ B \ C. (INT \ x:C. A \ x \ Un \ B) = ((INT \ x:C. A \ x) \ Un \ B)$   
 $!!A \ B \ C. (INT \ x:C. A \ Un \ B \ x) = (A \ Un \ (INT \ x:C. B \ x))$   
 $!!A \ B. (INT \ x: Union \ A. B \ x) = (INT \ y:A. INT \ x:y. B \ x)$   
 $!!A \ B \ C. (INT \ z: UNION \ A \ B. C \ z) = (INT \ x:A. INT \ z: B(x). C \ z)$   
 $!!A \ B \ f. (INT \ x:f'A. B \ x) = (INT \ a:A. B \ (f \ a))$   
**by auto**

**lemma** *ball-simps* [*simp,no-atp*]:

$!!A \ P \ Q. (ALL \ x:A. P \ x \mid Q) = ((ALL \ x:A. P \ x) \mid Q)$   
 $!!A \ P \ Q. (ALL \ x:A. P \mid Q \ x) = (P \mid (ALL \ x:A. Q \ x))$   
 $!!A \ P \ Q. (ALL \ x:A. P \ \longrightarrow \ Q \ x) = (P \ \longrightarrow \ (ALL \ x:A. Q \ x))$   
 $!!A \ P \ Q. (ALL \ x:A. P \ x \ \longrightarrow \ Q) = ((EX \ x:A. P \ x) \ \longrightarrow \ Q)$   
 $!!P. (ALL \ x:\{\}. P \ x) = True$   
 $!!P. (ALL \ x:UNIV. P \ x) = (ALL \ x. P \ x)$   
 $!!a \ B \ P. (ALL \ x:insert \ a \ B. P \ x) = (P \ a \ \& \ (ALL \ x:B. P \ x))$   
 $!!A \ P. (ALL \ x:Union \ A. P \ x) = (ALL \ y:A. ALL \ x:y. P \ x)$   
 $!!A \ B \ P. (ALL \ x: UNION \ A \ B. P \ x) = (ALL \ a:A. ALL \ x: B \ a. P \ x)$   
 $!!P \ Q. (ALL \ x:Collect \ Q. P \ x) = (ALL \ x. Q \ x \ \longrightarrow \ P \ x)$   
 $!!A \ P \ f. (ALL \ x:f'A. P \ x) = (ALL \ x:A. P \ (f \ x))$   
 $!!A \ P. (\sim (ALL \ x:A. P \ x)) = (EX \ x:A. \sim P \ x)$   
**by auto**

**lemma** *bex-simps* [*simp,no-atp*]:

$!!A \ P \ Q. (EX \ x:A. P \ x \ \& \ Q) = ((EX \ x:A. P \ x) \ \& \ Q)$   
 $!!A \ P \ Q. (EX \ x:A. P \ \& \ Q \ x) = (P \ \& \ (EX \ x:A. Q \ x))$   
 $!!P. (EX \ x:\{\}. P \ x) = False$

$!!P. (EX\ x:UNIV. P\ x) = (EX\ x. P\ x)$   
 $!!a\ B\ P. (EX\ x:insert\ a\ B. P\ x) = (P(a) \mid (EX\ x:B. P\ x))$   
 $!!A\ P. (EX\ x:Union\ A. P\ x) = (EX\ y:A. EX\ x:y. P\ x)$   
 $!!A\ B\ P. (EX\ x: UNION\ A\ B. P\ x) = (EX\ a:A. EX\ x:B\ a. P\ x)$   
 $!!P\ Q. (EX\ x:Collect\ Q. P\ x) = (EX\ x. Q\ x \ \&\ P\ x)$   
 $!!A\ P\ f. (EX\ x:f^*A. P\ x) = (EX\ x:A. P\ (f\ x))$   
 $!!A\ P. (\sim (EX\ x:A. P\ x)) = (ALL\ x:A. \sim P\ x)$   
**by** *auto*

**lemma** *ball-conj-distrib*:

$(ALL\ x:A. P\ x \ \&\ Q\ x) = ((ALL\ x:A. P\ x) \ \&\ (ALL\ x:A. Q\ x))$   
**by** *blast*

**lemma** *bex-disj-distrib*:

$(EX\ x:A. P\ x \mid Q\ x) = ((EX\ x:A. P\ x) \mid (EX\ x:A. Q\ x))$   
**by** *blast*

Maxiscoping: pulling out big Unions and Intersections.

**lemma** *UN-extend-simps*:

$!!a\ B\ C. insert\ a\ (UN\ x:C. B\ x) = (if\ C=\{\} \ then\ \{a\} \ else\ (UN\ x:C. insert\ a\ (B\ x)))$   
 $!!A\ B\ C. (UN\ x:C. A\ x) \ Un\ B = (if\ C=\{\} \ then\ B \ else\ (UN\ x:C. A\ x \ Un\ B))$   
 $!!A\ B\ C. A \ Un\ (UN\ x:C. B\ x) = (if\ C=\{\} \ then\ A \ else\ (UN\ x:C. A \ Un\ B\ x))$   
 $!!A\ B\ C. ((UN\ x:C. A\ x) \ Int\ B) = (UN\ x:C. A\ x \ Int\ B)$   
 $!!A\ B\ C. (A \ Int\ (UN\ x:C. B\ x)) = (UN\ x:C. A \ Int\ B\ x)$   
 $!!A\ B\ C. ((UN\ x:C. A\ x) - B) = (UN\ x:C. A\ x - B)$   
 $!!A\ B\ C. (A - (INT\ x:C. B\ x)) = (UN\ x:C. A - B\ x)$   
 $!!A\ B. (UN\ y:A. UN\ x:y. B\ x) = (UN\ x: Union\ A. B\ x)$   
 $!!A\ B\ C. (UN\ x:A. UN\ z: B(x). C\ z) = (UN\ z: UNION\ A\ B. C\ z)$   
 $!!A\ B\ f. (UN\ a:A. B\ (f\ a)) = (UN\ x:f^*A. B\ x)$   
**by** *auto*

**lemma** *INT-extend-simps*:

$!!A\ B\ C. (INT\ x:C. A\ x) \ Int\ B = (if\ C=\{\} \ then\ B \ else\ (INT\ x:C. A\ x \ Int\ B))$   
 $!!A\ B\ C. A \ Int\ (INT\ x:C. B\ x) = (if\ C=\{\} \ then\ A \ else\ (INT\ x:C. A \ Int\ B\ x))$   
 $!!A\ B\ C. (INT\ x:C. A\ x) - B = (if\ C=\{\} \ then\ UNIV - B \ else\ (INT\ x:C. A\ x - B))$   
 $!!A\ B\ C. A - (UN\ x:C. B\ x) = (if\ C=\{\} \ then\ A \ else\ (INT\ x:C. A - B\ x))$   
 $!!a\ B\ C. insert\ a\ (INT\ x:C. B\ x) = (INT\ x:C. insert\ a\ (B\ x))$   
 $!!A\ B\ C. ((INT\ x:C. A\ x) \ Un\ B) = (INT\ x:C. A\ x \ Un\ B)$   
 $!!A\ B\ C. A \ Un\ (INT\ x:C. B\ x) = (INT\ x:C. A \ Un\ B\ x)$   
 $!!A\ B. (INT\ y:A. INT\ x:y. B\ x) = (INT\ x: Union\ A. B\ x)$   
 $!!A\ B\ C. (INT\ x:A. INT\ z: B(x). C\ z) = (INT\ z: UNION\ A\ B. C\ z)$   
 $!!A\ B\ f. (INT\ a:A. B\ (f\ a)) = (INT\ x:f^*A. B\ x)$   
**by** *auto*

**no-notation**

*less-eq* (**infix**  $\sqsubseteq$  50) **and**



```

less (infix  $\sqsubset$  50) and
inf (infixl  $\sqcap$  70) and
sup (infixl  $\sqcup$  65) and
Inf ( $\sqcap$  - [900] 900) and
Sup ( $\sqcup$  - [900] 900) and
top ( $\top$ ) and
bot ( $\perp$ )

lemmas mem-simps =
  insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
  mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
  — Each of these has ALREADY been added [simp] above.

end

```

## 9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

```

theory Inductive
imports Complete-Lattice
uses
  (Tools/inductive.ML)
  Tools/dseq.ML
  (Tools/inductive-codegen.ML)
  Tools/Datatype/datatype-aux.ML
  Tools/Datatype/datatype-prop.ML
  Tools/Datatype/datatype-case.ML
  (Tools/Datatype/datatype-abs-proofs.ML)
  (Tools/Datatype/datatype-data.ML)
  (Tools/old-primrec.ML)
  (Tools/primrec.ML)
  (Tools/Datatype/datatype-codegen.ML)
begin

```

### 9.1 Least and greatest fixed points

```

context complete-lattice
begin

```

```

definition
  lfp :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  lfp f = Inf {u. f u  $\leq$  u} — least fixed point

```

```

definition
  gfp :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a where
  gfp f = Sup {u. u  $\leq$  f u} — greatest fixed point

```

## 9.2 Proof of Knaster-Tarski Theorem using *lfp*

*lfp* *f* is the least upper bound of the set  $\{u. f\ u \leq u\}$

**lemma** *lfp-lowerbound*:  $f\ A \leq A \implies lfp\ f \leq A$   
**by** (*auto simp add: lfp-def intro: Inf-lower*)

**lemma** *lfp-greatest*:  $(!!u. f\ u \leq u \implies A \leq u) \implies A \leq lfp\ f$   
**by** (*auto simp add: lfp-def intro: Inf-greatest*)

**end**

**lemma** *lfp-lemma2*:  $mono\ f \implies f\ (lfp\ f) \leq lfp\ f$   
**by** (*iprover intro: lfp-greatest order-trans monoD lfp-lowerbound*)

**lemma** *lfp-lemma3*:  $mono\ f \implies lfp\ f \leq f\ (lfp\ f)$   
**by** (*iprover intro: lfp-lemma2 monoD lfp-lowerbound*)

**lemma** *lfp-unfold*:  $mono\ f \implies lfp\ f = f\ (lfp\ f)$   
**by** (*iprover intro: order-antisym lfp-lemma2 lfp-lemma3*)

**lemma** *lfp-const*:  $lfp\ (\lambda x. t) = t$   
**by** (*rule lfp-unfold*) (*simp add: mono-def*)

## 9.3 General induction rules for least fixed points

**theorem** *lfp-induct*:

**assumes** *mono*:  $mono\ f$  **and** *ind*:  $f\ (inf\ (lfp\ f)\ P) \leq P$   
**shows**  $lfp\ f \leq P$

**proof** –

**have**  $inf\ (lfp\ f)\ P \leq lfp\ f$  **by** (*rule inf-le1*)  
**with** *mono* **have**  $f\ (inf\ (lfp\ f)\ P) \leq f\ (lfp\ f)$  **..**  
**also from** *mono* **have**  $f\ (lfp\ f) = lfp\ f$  **by** (*rule lfp-unfold [symmetric]*)  
**finally have**  $f\ (inf\ (lfp\ f)\ P) \leq lfp\ f$  **.**  
**from this and** *ind* **have**  $f\ (inf\ (lfp\ f)\ P) \leq inf\ (lfp\ f)\ P$  **by** (*rule le-infI*)  
**hence**  $lfp\ f \leq inf\ (lfp\ f)\ P$  **by** (*rule lfp-lowerbound*)  
**also have**  $inf\ (lfp\ f)\ P \leq P$  **by** (*rule inf-le2*)  
**finally show** *?thesis* **.**

**qed**

**lemma** *lfp-induct-set*:

**assumes** *lfp*:  $a: lfp(f)$   
**and** *mono*:  $mono(f)$   
**and** *indhyp*:  $!!x. [| x: f(lfp(f)\ Int\ \{x. P(x)\}) |] \implies P(x)$   
**shows**  $P(a)$   
**by** (*rule lfp-induct [THEN subsetD, THEN CollectD, OF mono - lfp]*)  
*(auto simp: intro: indhyp)*

**lemma** *lfp-ordinal-induct*:

**fixes**  $f :: 'a::complete-lattice \Rightarrow 'a$

```

assumes mono: mono f
and P-f:  $\bigwedge S. P\ S \implies P\ (f\ S)$ 
and P-Union:  $\bigwedge M. \forall S \in M. P\ S \implies P\ (\text{Sup } M)$ 
shows  $P\ (\text{lfp } f)$ 
proof –
  let  $?M = \{S. S \leq \text{lfp } f \wedge P\ S\}$ 
  have  $P\ (\text{Sup } ?M)$  using P-Union by simp
  also have  $\text{Sup } ?M = \text{lfp } f$ 
  proof (rule antisym)
    show  $\text{Sup } ?M \leq \text{lfp } f$  by (blast intro: Sup-least)
    hence  $f\ (\text{Sup } ?M) \leq f\ (\text{lfp } f)$  by (rule mono [THEN monoD])
    hence  $f\ (\text{Sup } ?M) \leq \text{lfp } f$  using mono [THEN lfp-unfold] by simp
    hence  $f\ (\text{Sup } ?M) \in ?M$  using P-f P-Union by simp
    hence  $f\ (\text{Sup } ?M) \leq \text{Sup } ?M$  by (rule Sup-upper)
    thus  $\text{lfp } f \leq \text{Sup } ?M$  by (rule lfp-lowerbound)
  qed
  finally show ?thesis .
qed

```

```

lemma lfp-ordinal-induct-set:
  assumes mono: mono f
  and P-f:  $\forall S. P\ S \implies P\ (f\ S)$ 
  and P-Union:  $\forall M. \forall S \in M. P\ S \implies P\ (\text{Union } M)$ 
  shows  $P\ (\text{lfp } f)$ 
  using assms by (rule lfp-ordinal-induct [where P=P])

```

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

```

lemma def-lfp-unfold:  $\llbracket h == \text{lfp}(f); \text{mono}(f) \rrbracket \implies h = f(h)$ 
by (auto intro!: lfp-unfold)

```

```

lemma def-lfp-induct:
   $\llbracket A == \text{lfp}(f); \text{mono}(f);$ 
     $f\ (\text{inf } A\ P) \leq P$ 
   $\rrbracket \implies A \leq P$ 
by (blast intro: lfp-induct)

```

```

lemma def-lfp-induct-set:
   $\llbracket A == \text{lfp}(f); \text{mono}(f); a:A;$ 
     $\forall x. \llbracket x: f(A\ \text{Int } \{x. P(x)\}) \rrbracket \implies P(x)$ 
   $\rrbracket \implies P(a)$ 
by (blast intro: lfp-induct-set)

```

```

lemma lfp-mono:  $(\forall Z. f\ Z \leq g\ Z) \implies \text{lfp } f \leq \text{lfp } g$ 
by (rule lfp-lowerbound [THEN lfp-greatest], blast intro: order-trans)

```

## 9.4 Proof of Knaster-Tarski Theorem using *gfp*

*gfp f* is the greatest lower bound of the set  $\{u. u \leq f\ u\}$

```

lemma gfp-upperbound:  $X \leq f X \implies X \leq \text{gfp } f$ 
  by (auto simp add: gfp-def intro: Sup-upper)

lemma gfp-least:  $(!!u. u \leq f u \implies u \leq X) \implies \text{gfp } f \leq X$ 
  by (auto simp add: gfp-def intro: Sup-least)

lemma gfp-lemma2:  $\text{mono } f \implies \text{gfp } f \leq f (\text{gfp } f)$ 
  by (iprover intro: gfp-least order-trans monoD gfp-upperbound)

lemma gfp-lemma3:  $\text{mono } f \implies f (\text{gfp } f) \leq \text{gfp } f$ 
  by (iprover intro: gfp-lemma2 monoD gfp-upperbound)

lemma gfp-unfold:  $\text{mono } f \implies \text{gfp } f = f (\text{gfp } f)$ 
  by (iprover intro: order-antisym gfp-lemma2 gfp-lemma3)

```

## 9.5 Coinduction rules for greatest fixed points

weak version

```

lemma weak-coinduct:  $[| a : X; X \subseteq f(X) |] \implies a : \text{gfp}(f)$ 
by (rule gfp-upperbound [THEN subsetD], auto)

lemma weak-coinduct-image:  $!!X. [| a : X; g'X \subseteq f (g'X) |] \implies g a : \text{gfp } f$ 
apply (erule gfp-upperbound [THEN subsetD])
apply (erule imageI)
done

```

```

lemma coinduct-lemma:
   $[| X \leq f (\text{sup } X (\text{gfp } f)); \text{mono } f |] \implies \text{sup } X (\text{gfp } f) \leq f (\text{sup } X (\text{gfp } f))$ 
apply (frule gfp-lemma2)
apply (drule mono-sup)
apply (rule le-supI)
apply assumption
apply (rule order-trans)
apply (rule order-trans)
apply assumption
apply (rule sup-ge2)
apply assumption
done

```

strong version, thanks to Coen and Frost

```

lemma coinduct-set:  $[| \text{mono}(f); a : X; X \subseteq f(X \text{ Un } \text{gfp}(f)) |] \implies a : \text{gfp}(f)$ 
by (blast intro: weak-coinduct [OF - coinduct-lemma])

lemma coinduct:  $[| \text{mono}(f); X \leq f (\text{sup } X (\text{gfp } f)) |] \implies X \leq \text{gfp}(f)$ 
apply (rule order-trans)
apply (rule sup-ge1)
apply (erule gfp-upperbound [OF coinduct-lemma])
apply assumption
done

```

**lemma** *gfp-fun-UnI2*:  $[| \text{mono}(f); a: \text{gfp}(f) |] \implies a: f(X \text{ Un } \text{gfp}(f))$   
**by** (*blast dest: GFP-lemma2 mono-Un*)

## 9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition  $X \subseteq f X$  to one expressed using both *lfp* and *gfp*

**lemma** *coinduct3-mono-lemma*:  $\text{mono}(f) \implies \text{mono}(\%x. f(x) \text{ Un } X \text{ Un } B)$   
**by** (*iprover intro: subset-refl monoI Un-mono monoD*)

**lemma** *coinduct3-lemma*:

$[| X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f))); \text{mono}(f) |]$   
 $\implies \text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)) \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)))$

**apply** (*rule subset-trans*)

**apply** (*erule coinduct3-mono-lemma [THEN lfp-lemma3]*)

**apply** (*rule Un-least [THEN Un-least]*)

**apply** (*rule subset-refl, assumption*)

**apply** (*rule GFP-unfold [THEN equalityD1, THEN subset-trans], assumption*)

**apply** (*rule monoD [where f=f], assumption*)

**apply** (*subst coinduct3-mono-lemma [THEN lfp-unfold], auto*)

**done**

**lemma** *coinduct3*:

$[| \text{mono}(f); a: X; X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f))) |] \implies a: \text{gfp}(f)$

**apply** (*rule coinduct3-lemma [THEN [2] weak-coinduct]*)

**apply** (*rule coinduct3-mono-lemma [THEN lfp-unfold, THEN ssubst], auto*)

**done**

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

**lemma** *def-gfp-unfold*:  $[| A == \text{gfp}(f); \text{mono}(f) |] \implies A = f(A)$

**by** (*auto intro!: GFP-unfold*)

**lemma** *def-coinduct*:

$[| A == \text{gfp}(f); \text{mono}(f); X \leq f(\text{sup } X A) |] \implies X \leq A$

**by** (*iprover intro!: coinduct*)

**lemma** *def-coinduct-set*:

$[| A == \text{gfp}(f); \text{mono}(f); a: X; X \subseteq f(X \text{ Un } A) |] \implies a: A$

**by** (*auto intro!: coinduct-set*)

**lemma** *def-Collect-coinduct*:

$[| A == \text{gfp}(\%w. \text{Collect}(P(w))); \text{mono}(\%w. \text{Collect}(P(w)));$   
 $a: X; !!z. z: X \implies P(X \text{ Un } A) z |] \implies$

$a: A$

**apply** (*erule def-coinduct-set, auto*)

**done**

**lemma** *def-coinduct3*:

$$[| A == \text{gfp}(f); \text{mono}(f); a:X; X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } A)) |] ==> a: A$$
  
**by** (*auto intro!: coinduct3*)

Monotonicity of *gfp*!

**lemma** *gfp-mono*: (!Z.  $f Z \leq g Z$ ) ==>  $\text{gfp } f \leq \text{gfp } g$   
**by** (*rule gfp-upperbound [THEN gfp-least], blast intro: order-trans*)

## 9.7 Inductive predicates and sets

Package setup.

**theorems** *basic-monos* =  
*subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj*  
*Collect-mono in-mono vimage-mono*

**use** *Tools/inductive.ML*  
**setup** *Inductive.setup*

**theorems** [*mono*] =  
*imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj*  
*imp-mono not-mono*  
*Ball-def Bex-def*  
*induct-rulify-fallback*

## 9.8 Inductive datatypes and primitive recursion

Package setup.

**use** *Tools/Datatype/datatype-abs-proofs.ML*  
**use** *Tools/Datatype/datatype-data.ML*  
**setup** *Datatype-Data.setup*

**use** *Tools/Datatype/datatype-codegen.ML*  
**setup** *Datatype-Codegen.setup*

**use** *Tools/old-primrec.ML*  
**use** *Tools/primrec.ML*

**use** *Tools/inductive-codegen.ML*  
**setup** *InductiveCodegen.setup*

Lambda-abstractions with pattern matching:

**syntax**  
 $\text{-lam-pats-syntax} :: \text{cases-syn} \Rightarrow 'a \Rightarrow 'b \quad ((\% -) 10)$   
**syntax** (*xsymbols*)  
 $\text{-lam-pats-syntax} :: \text{cases-syn} \Rightarrow 'a \Rightarrow 'b \quad ((\lambda -) 10)$

**parse-translation** (**advanced**)  $\ll$   
*let*  
 $\text{fun fun-tr ctxt [cs] =}$

```

    let
      val x = Free (Name.variant (Term.add-free-names cs []) x, dummyT);
      val ft = Datatype-Case.case-tr true Datatype-Data.info-of-constr ctxt [x, cs];
    in lambda x ft end
  in [(@{syntax-const -lam-pats-syntax}, fun-tr)] end
>>

end

```

## 10 Fun: Notions about functions

```

theory Fun
imports Complete-Lattice
begin

```

As a simplification rule, it replaces all function equalities by first-order equalities.

```

lemma expand-fun-eq: f = g  $\longleftrightarrow$  ( $\forall x. f\ x = g\ x$ )
apply (rule iffI)
apply (simp (no-asm-simp))
apply (rule ext)
apply (simp (no-asm-simp))
done

```

```

lemma apply-inverse:
  f x = u  $\implies$  ( $\bigwedge x. P\ x \implies g\ (f\ x) = x$ )  $\implies P\ x \implies x = g\ u$ 
  by auto

```

### 10.1 The Identity Function *id*

```

definition
  id :: 'a  $\Rightarrow$  'a
where
  id = ( $\lambda x. x$ )

```

```

lemma id-apply [simp]: id x = x
  by (simp add: id-def)

```

```

lemma image-ident [simp]: (%x. x) ‘ Y = Y
  by blast

```

```

lemma image-id [simp]: id ‘ Y = Y
  by (simp add: id-def)

```

```

lemma vimage-ident [simp]: (%x. x) -‘ Y = Y
  by blast

```

```

lemma vimage-id [simp]: id -‘ A = A

```

by (simp add: id-def)

## 10.2 The Composition Operator $f \circ g$

**definition**

$comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$  (infixl 55)

**where**

$f \circ g = (\lambda x. f (g x))$

**notation** (xsymbols)

$comp$  (infixl 55)

**notation** (HTML output)

$comp$  (infixl 55)

compatibility

**lemmas** o-def = comp-def

**lemma** o-apply [simp]:  $(f \circ g) x = f (g x)$

by (simp add: comp-def)

**lemma** o-assoc:  $f \circ (g \circ h) = f \circ g \circ h$

by (simp add: comp-def)

**lemma** id-o [simp]:  $id \circ g = g$

by (simp add: comp-def)

**lemma** o-id [simp]:  $f \circ id = f$

by (simp add: comp-def)

**lemma** o-eq-dest:

$a \circ b = c \circ d \implies a (b v) = c (d v)$

by (simp only: o-def) (fact fun-cong)

**lemma** o-eq-elim:

$a \circ b = c \circ d \implies ((\bigwedge v. a (b v) = c (d v)) \implies R) \implies R$

by (erule meta-mp) (fact o-eq-dest)

**lemma** image-compose:  $(f \circ g) \text{ ` } r = f \text{ ` } (g \text{ ` } r)$

by (simp add: comp-def, blast)

**lemma** vimage-compose:  $(g \circ f) \text{ - ` } x = f \text{ - ` } (g \text{ - ` } x)$

by auto

**lemma** UN-o:  $UNION A (g \circ f) = UNION (f \text{ ` } A) g$

by (unfold comp-def, blast)

## 10.3 The Forward Composition Operator $fcomp$

**definition**



*fcomp* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  ('b  $\Rightarrow$  'c)  $\Rightarrow$  'a  $\Rightarrow$  'c (**infixl** o> 60)

**where**

*f* o> *g* = ( $\lambda x. g (f x)$ )

**lemma** *fcomp-apply*: (*f* o> *g*) *x* = *g* (*f* *x*)

**by** (*simp add: fcomp-def*)

**lemma** *fcomp-assoc*: (*f* o> *g*) o> *h* = *f* o> (*g* o> *h*)

**by** (*simp add: fcomp-def*)

**lemma** *id-fcomp* [*simp*]: *id* o> *g* = *g*

**by** (*simp add: fcomp-def*)

**lemma** *fcomp-id* [*simp*]: *f* o> *id* = *f*

**by** (*simp add: fcomp-def*)

**code-const** *fcomp*

(*Eval* **infixl** 1 #>)

**no-notation** *fcomp* (**infixl** o> 60)

## 10.4 Injectivity and Surjectivity

**definition**

*inj-on* :: ['a  $\Rightarrow$  'b, 'a set]  $\Rightarrow$  bool **where**

— injective

*inj-on* *f* *A* == ! *x*:*A*. ! *y*:*A*. *f*(*x*)=*f*(*y*)  $\longrightarrow$  *x*=*y*

A common special case: functions injective over the entire domain type.

**abbreviation**

*inj* *f* == *inj-on* *f* *UNIV*

**definition**

*bij-betw* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a set  $\Rightarrow$  'b set  $\Rightarrow$  bool **where** — bijective

[*code del*]: *bij-betw* *f* *A* *B*  $\longleftrightarrow$  *inj-on* *f* *A* & *f* ' *A* = *B*

**definition**

*surj* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  bool **where**

— surjective

*surj* *f* == ! *y*. ? *x*. *y*=*f*(*x*)

**definition**

*bij* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  bool **where**

— bijective

*bij* *f* == *inj* *f* & *surj* *f*

**lemma** *injI*:

**assumes**  $\bigwedge x y. f x = f y \Longrightarrow x = y$

**shows** *inj* *f*

**using** *assms* **unfolding** *inj-on-def* **by** *auto*

For Proofs in *Tools/Datatype/datatype-rep-proofs*

**lemma** *datatype-injI*:

(!! *x. ALL y. f(x) = f(y) --> x=y*) ==> *inj(f)*  
**by** (*simp add: inj-on-def*)

**theorem** *range-ex1-eq*: *inj f ==> b : range f = (EX! x. b = f x)*

**by** (*unfold inj-on-def, blast*)

**lemma** *injD*: [*inj(f); f(x) = f(y)*] ==> *x=y*

**by** (*simp add: inj-on-def*)

**lemma** *inj-on-eq-iff*: *inj-on f A ==> x:A ==> y:A ==> (f(x) = f(y)) = (x=y)*

**by** (*force simp add: inj-on-def*)

**lemma** *inj-eq*: *inj f ==> (f(x) = f(y)) = (x=y)*

**by** (*simp add: inj-on-eq-iff*)

**lemma** *inj-on-id[simp]*: *inj-on id A*

**by** (*simp add: inj-on-def*)

**lemma** *inj-on-id2[simp]*: *inj-on (%x. x) A*

**by** (*simp add: inj-on-def*)

**lemma** *surj-id[simp]*: *surj id*

**by** (*simp add: surj-def*)

**lemma** *bij-id[simp]*: *bij id*

**by** (*simp add: bij-def*)

**lemma** *inj-onI*:

(!! *x y. [ x:A; y:A; f(x) = f(y) ] ==> x=y*) ==> *inj-on f A*  
**by** (*simp add: inj-on-def*)

**lemma** *inj-on-inverseI*: (!!*x. x:A ==> g(f(x)) = x*) ==> *inj-on f A*

**by** (*auto dest: arg-cong [of concl: g] simp add: inj-on-def*)

**lemma** *inj-onD*: [*inj-on f A; f(x)=f(y); x:A; y:A*] ==> *x=y*

**by** (*unfold inj-on-def, blast*)

**lemma** *inj-on-iff*: [*inj-on f A; x:A; y:A*] ==> (*f(x)=f(y)*) = (*x=y*)

**by** (*blast dest!: inj-onD*)

**lemma** *comp-inj-on*:

[*inj-on f A; inj-on g (f'A)*] ==> *inj-on (g o f) A*  
**by** (*simp add: comp-def inj-on-def*)

**lemma** *inj-on-imageI*: *inj-on (g o f) A ==> inj-on g (f ' A)*

```

apply(simp add:inj-on-def image-def)
apply blast
done

```

```

lemma inj-on-image-iff:  $\llbracket \text{ALL } x:A. \text{ ALL } y:A. (g(f\ x) = g(f\ y)) = (g\ x = g\ y);$ 
   $\text{inj-on } f\ A \rrbracket \implies \text{inj-on } g\ (f\ `A) = \text{inj-on } g\ A$ 
apply(unfold inj-on-def)
apply blast
done

```

```

lemma inj-on-contrad:  $\llbracket \text{inj-on } f\ A; \sim x=y; x:A; y:A \rrbracket \implies \sim f(x)=f(y)$ 
by (unfold inj-on-def, blast)

```

```

lemma inj-singleton:  $\text{inj } (\%s. \{s\})$ 
by (simp add: inj-on-def)

```

```

lemma inj-on-empty[iff]:  $\text{inj-on } f\ \{\}$ 
by(simp add: inj-on-def)

```

```

lemma subset-inj-on:  $\llbracket \text{inj-on } f\ B; A \leq B \rrbracket \implies \text{inj-on } f\ A$ 
by (unfold inj-on-def, blast)

```

```

lemma inj-on-Un:
   $\text{inj-on } f\ (A \cup B) =$ 
   $(\text{inj-on } f\ A \ \& \ \text{inj-on } f\ B \ \& \ f\ `(A-B) \ \text{Int } f\ `(B-A) = \{\})$ 
apply(unfold inj-on-def)
apply (blast intro:sym)
done

```

```

lemma inj-on-insert[iff]:
   $\text{inj-on } f\ (\text{insert } a\ A) = (\text{inj-on } f\ A \ \& \ f\ a \sim: f\ `(A-\{a\}))$ 
apply(unfold inj-on-def)
apply (blast intro:sym)
done

```

```

lemma inj-on-diff:  $\text{inj-on } f\ A \implies \text{inj-on } f\ (A-B)$ 
apply(unfold inj-on-def)
apply (blast)
done

```

```

lemma surjI:  $(!!\ x. g(f\ x) = x) \implies \text{surj } g$ 
apply (simp add: surj-def)
apply (blast intro: sym)
done

```

```

lemma surj-range:  $\text{surj } f \implies \text{range } f = \text{UNIV}$ 
by (auto simp add: surj-def)

```

```

lemma surjD:  $\text{surj } f \implies \exists x. y = f\ x$ 

```

**by** (*simp add: surj-def*)

**lemma** *surjE*: *surj f ==> (!x. y = f x ==> C) ==> C*  
**by** (*simp add: surj-def, blast*)

**lemma** *comp-surj*: *[| surj f; surj g |] ==> surj (g o f)*  
**apply** (*simp add: comp-def surj-def, clarify*)  
**apply** (*drule-tac x = y in spec, clarify*)  
**apply** (*drule-tac x = x in spec, blast*)  
**done**

**lemma** *bijI*: *[| inj f; surj f |] ==> bij f*  
**by** (*simp add: bij-def*)

**lemma** *bij-is-inj*: *bij f ==> inj f*  
**by** (*simp add: bij-def*)

**lemma** *bij-is-surj*: *bij f ==> surj f*  
**by** (*simp add: bij-def*)

**lemma** *bij-betw-imp-inj-on*: *bij-betw f A B ==> inj-on f A*  
**by** (*simp add: bij-betw-def*)

**lemma** *bij-comp*: *bij f ==> bij g ==> bij (g o f)*  
**by**(*fastsimp intro: comp-inj-on comp-surj simp: bij-def surj-range*)

**lemma** *bij-betw-trans*:  
*bij-betw f A B ==> bij-betw g B C ==> bij-betw (g o f) A C*  
**by**(*auto simp add: bij-betw-def comp-inj-on*)

**lemma** *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g. bij-betw g B A*  
**proof** –

**have** *i: inj-on f A* **and** *s: f ‘ A = B*

**using** *assms* **by**(*auto simp: bij-betw-def*)

**let** *?P = %b a. a: A ∧ f a = b* **let** *?g = %b. The (?P b)*

**{ fix** *a b* **assume** *P: ?P b a*

**hence** *ex1: ∃ a. ?P b a* **using** *s* **unfolding** *image-def* **by** *blast*

**hence** *uex1: ∃! a. ?P b a* **by**(*blast dest: inj-onD[OF i]*)

**hence** *?g b = a* **using** *the1-equality[OF uex1, OF P]* *P* **by** *simp*

**}** **note** *g = this*

**have** *inj-on ?g B*

**proof**(*rule inj-onI*)

**fix** *x y* **assume** *x: B y: B ?g x = ?g y*

**from** *s ⟨x: B⟩* **obtain** *a1* **where** *a1: ?P x a1* **unfolding** *image-def* **by** *blast*

**from** *s ⟨y: B⟩* **obtain** *a2* **where** *a2: ?P y a2* **unfolding** *image-def* **by** *blast*

**from** *g[OF a1] a1 g[OF a2] a2 ⟨?g x = ?g y⟩* **show** *x = y* **by** *simp*

**qed**

**moreover** **have** *?g ‘ B = A*

**proof**(*auto simp: image-def*)

```

fix b assume b:B
with s obtain a where P: ?P b a unfolding image-def by blast
thus ?g b ∈ A using g[OF P] by auto
next
fix a assume a:A
then obtain b where P: ?P b a using s unfolding image-def by blast
then have b:B using s unfolding image-def by blast
with g[OF P] show ∃ b∈B. a = ?g b by blast
qed
ultimately show ?thesis by(auto simp:bij-betw-def)
qed

```

```

lemma surj-image-vimage-eq: surj f ==> f ‘ (f –‘ A) = A
by (simp add: surj-range)

```

```

lemma inj-vimage-image-eq: inj f ==> f –‘ (f ‘ A) = A
by (simp add: inj-on-def, blast)

```

```

lemma vimage-subsetD: surj f ==> f –‘ B <= A ==> B <= f ‘ A
apply (unfold surj-def)
apply (blast intro: sym)
done

```

```

lemma vimage-subsetI: inj f ==> B <= f ‘ A ==> f –‘ B <= A
by (unfold inj-on-def, blast)

```

```

lemma vimage-subset-eq: bij f ==> (f –‘ B <= A) = (B <= f ‘ A)
apply (unfold bij-def)
apply (blast del: subsetI intro: vimage-subsetI vimage-subsetD)
done

```

```

lemma inj-on-Un-image-eq-iff: inj-on f (A ∪ B) ==> f ‘ A = f ‘ B ⟷ A = B
by(blast dest: inj-onD)

```

```

lemma inj-on-image-Int:
  [| inj-on f C; A<=C; B<=C |] ==> f‘(A Int B) = f‘A Int f‘B
apply (simp add: inj-on-def, blast)
done

```

```

lemma inj-on-image-set-diff:
  [| inj-on f C; A<=C; B<=C |] ==> f‘(A-B) = f‘A - f‘B
apply (simp add: inj-on-def, blast)
done

```

```

lemma image-Int: inj f ==> f‘(A Int B) = f‘A Int f‘B
by (simp add: inj-on-def, blast)

```

```

lemma image-set-diff: inj f ==> f‘(A-B) = f‘A - f‘B
by (simp add: inj-on-def, blast)

```

**lemma** *inj-image-mem-iff*:  $\text{inj } f \implies (f \text{ ` } a : f^{\text{`}} A) = (a : A)$   
**by** (*blast dest: injD*)

**lemma** *inj-image-subset-iff*:  $\text{inj } f \implies (f^{\text{`}} A \leq f^{\text{`}} B) = (A \leq B)$   
**by** (*simp add: inj-on-def, blast*)

**lemma** *inj-image-eq-iff*:  $\text{inj } f \implies (f^{\text{`}} A = f^{\text{`}} B) = (A = B)$   
**by** (*blast dest: injD*)

**lemma** *image-INT*:  
 $\llbracket \text{inj-on } f \text{ ` } C; \text{ ALL } x:A. B \text{ ` } x \leq C; \text{ ` } j:A \rrbracket$   
 $\implies f^{\text{`}} (INTER \ A \ B) = (INT \ x:A. f^{\text{`}} \ B \ x)$   
**apply** (*simp add: inj-on-def, blast*)  
**done**

**lemma** *bij-image-INT*:  $\text{bij } f \implies f^{\text{`}} (INTER \ A \ B) = (INT \ x:A. f^{\text{`}} \ B \ x)$   
**apply** (*simp add: bij-def*)  
**apply** (*simp add: inj-on-def surj-def, blast*)  
**done**

**lemma** *surj-Compl-image-subset*:  $\text{surj } f \implies \neg(f^{\text{`}} A) \leq f^{\text{`}}(\neg A)$   
**by** (*auto simp add: surj-def*)

**lemma** *inj-image-Compl-subset*:  $\text{inj } f \implies f^{\text{`}}(\neg A) \leq \neg(f^{\text{`}} A)$   
**by** (*auto simp add: inj-on-def*)

**lemma** *bij-image-Compl-eq*:  $\text{bij } f \implies f^{\text{`}}(\neg A) = \neg(f^{\text{`}} A)$   
**apply** (*simp add: bij-def*)  
**apply** (*rule equalityI*)  
**apply** (*simp-all (no-asm-simp) add: inj-image-Compl-subset surj-Compl-image-subset*)  
**done**

**lemma** (*in ordered-ab-group-add*) *inj-uminus*[*simp, intro*]: *inj-on uminus A*  
**by** (*auto intro!: inj-onI*)

**lemma** (*in linorder*) *strict-mono-imp-inj-on*: *strict-mono f  $\implies$  inj-on f A*  
**by** (*auto intro!: inj-onI dest: strict-mono-eq*)

## 10.5 Function Updating

### definition

*fun-upd* ::  $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$  **where**  
*fun-upd* *f* *a* *b* == % *x*. if *x*=*a* then *b* else *f* *x*

### nonterminals

*updbinds updbind*

**syntax**

```

-updbind :: ['a, 'a] => updbind      ((2- := / -))
          :: updbind => updbinds      (-)
-updbinds:: [updbind, updbinds] => updbinds (-, / -)
-Update   :: ['a, updbinds] => 'a      (-/'((-)') [1000, 0] 900)

```

**translations**

```

-Update f (-updbinds b bs) == -Update (-Update f b) bs
f(x:=y) == CONST fun-upd f x y

```

**lemma** *fun-upd-idem-iff*:  $(f(x:=y) = f) = (f\ x = y)$

**apply** (*simp add: fun-upd-def, safe*)

**apply** (*erule subst*)

**apply** (*rule-tac [2] ext, auto*)

**done**

**lemmas** *fun-upd-idem = fun-upd-idem-iff [THEN iffD2, standard]*

**lemmas** *fun-upd-triv = refl [THEN fun-upd-idem]*

**declare** *fun-upd-triv [iff]*

**lemma** *fun-upd-apply [simp]*:  $(f(x:=y))z = (if\ z=x\ then\ y\ else\ f\ z)$

**by** (*simp add: fun-upd-def*)

**lemma** *fun-upd-same*:  $(f(x:=y))\ x = y$

**by** *simp*

**lemma** *fun-upd-other*:  $z \sim x ==> (f(x:=y))\ z = f\ z$

**by** *simp*

**lemma** *fun-upd-upd [simp]*:  $f(x:=y, x:=z) = f(x:=z)$

**by** (*simp add: expand-fun-eq*)

**lemma** *fun-upd-twist*:  $a \sim c ==> (m(a:=b))(c:=d) = (m(c:=d))(a:=b)$

**by** (*rule ext, auto*)

**lemma** *inj-on-fun-updI*:  $\llbracket\ inj-on\ f\ A;\ y \notin f'A \rrbracket \implies inj-on\ (f(x:=y))\ A$

**by** (*fastsimp simp: inj-on-def image-def*)

**lemma** *fun-upd-image*:

$f(x:=y) \text{ ' } A = (if\ x \in A\ then\ insert\ y\ (f \text{ ' } (A - \{x\}))\ else\ f \text{ ' } A)$

**by** *auto*

**lemma** *fun-upd-comp*:  $f \circ (g(x := y)) = (f \circ g)(x := f\ y)$

**by** (*auto intro: ext*)

## 10.6 *override-on*

**definition**

*override-on* :: ( $'a \Rightarrow 'b$ )  $\Rightarrow$  ( $'a \Rightarrow 'b$ )  $\Rightarrow$   $'a \text{ set} \Rightarrow 'a \Rightarrow 'b$

**where**

*override-on*  $f\ g\ A = (\lambda a. \text{ if } a \in A \text{ then } g\ a \text{ else } f\ a)$

**lemma** *override-on-emptyset*[*simp*]: *override-on*  $f\ g\ \{\} = f$

**by**(*simp add: override-on-def*)

**lemma** *override-on-apply-notin*[*simp*]:  $a \sim: A \implies (\text{override-on } f\ g\ A)\ a = f\ a$

**by**(*simp add: override-on-def*)

**lemma** *override-on-apply-in*[*simp*]:  $a : A \implies (\text{override-on } f\ g\ A)\ a = g\ a$

**by**(*simp add: override-on-def*)

## 10.7 *swap*

**definition**

*swap* ::  $'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

**where**

*swap*  $a\ b\ f = f\ (a := f\ b, b := f\ a)$

**lemma** *swap-self* [*simp*]: *swap*  $a\ a\ f = f$

**by** (*simp add: swap-def*)

**lemma** *swap-commute*: *swap*  $a\ b\ f = \text{swap } b\ a\ f$

**by** (*rule ext, simp add: fun-upd-def swap-def*)

**lemma** *swap-nilpotent* [*simp*]: *swap*  $a\ b\ (\text{swap } a\ b\ f) = f$

**by** (*rule ext, simp add: fun-upd-def swap-def*)

**lemma** *swap-triple*:

**assumes**  $a \neq c$  **and**  $b \neq c$

**shows** *swap*  $a\ b\ (\text{swap } b\ c\ (\text{swap } a\ b\ f)) = \text{swap } a\ c\ f$

**using** *assms* **by** (*simp add: expand-fun-eq swap-def*)

**lemma** *comp-swap*:  $f \circ \text{swap } a\ b\ g = \text{swap } a\ b\ (f \circ g)$

**by** (*rule ext, simp add: fun-upd-def swap-def*)

**lemma** *inj-on-imp-inj-on-swap*:

$[|\text{inj-on } f\ A; a \in A; b \in A|] \implies \text{inj-on } (\text{swap } a\ b\ f)\ A$

**by** (*simp add: inj-on-def swap-def, blast*)

**lemma** *inj-on-swap-iff* [*simp*]:

**assumes**  $A: a \in A\ b \in A$  **shows**  $\text{inj-on } (\text{swap } a\ b\ f)\ A = \text{inj-on } f\ A$

**proof**

**assume**  $\text{inj-on } (\text{swap } a\ b\ f)\ A$



```

with A have inj-on (swap a b (swap a b f)) A
  by (iprover intro: inj-on-imp-inj-on-swap)
thus inj-on f A by simp
next
  assume inj-on f A
  with A show inj-on (swap a b f) A by (iprover intro: inj-on-imp-inj-on-swap)
qed

```

```

lemma surj-imp-surj-swap: surj f ==> surj (swap a b f)
apply (simp add: surj-def swap-def, clarify)
apply (case-tac y = f b, blast)
apply (case-tac y = f a, auto)
done

```

```

lemma surj-swap-iff [simp]: surj (swap a b f) = surj f
proof
  assume surj (swap a b f)
  hence surj (swap a b (swap a b f)) by (rule surj-imp-surj-swap)
  thus surj f by simp
next
  assume surj f
  thus surj (swap a b f) by (rule surj-imp-surj-swap)
qed

```

```

lemma bij-swap-iff: bij (swap a b f) = bij f
by (simp add: bij-def)

```

```

hide-const (open) swap

```

## 10.8 Inversion of injective functions

**definition** *the-inv-into* :: 'a set => ('a => 'b) => ('b => 'a) **where**  
*the-inv-into* A f == %x. THE y. y : A & f y = x

```

lemma the-inv-into-f-f:
  [| inj-on f A; x : A |] ==> the-inv-into A f (f x) = x
apply (simp add: the-inv-into-def inj-on-def)
apply blast
done

```

```

lemma f-the-inv-into-f:
  inj-on f A ==> y : f'A ==> f (the-inv-into A f y) = y
apply (simp add: the-inv-into-def)
apply (rule the1I2)
  apply(blast dest: inj-onD)
apply blast
done

```

```

lemma the-inv-into-into:

```

```

  [| inj-on f A; x : f ‘ A; A <= B |] ==> the-inv-into A f x : B
apply (simp add: the-inv-into-def)
apply (rule the1I2)
  apply (blast dest: inj-onD)
apply blast
done

```

```

lemma the-inv-into-onto[simp]:
  inj-on f A ==> the-inv-into A f ‘ (f ‘ A) = A
by (fast intro: the-inv-into-into the-inv-into-f-f[symmetric])

```

```

lemma the-inv-into-f-eq:
  [| inj-on f A; f x = y; x : A |] ==> the-inv-into A f y = x
  apply (erule subst)
  apply (erule the-inv-into-f-f, assumption)
done

```

```

lemma the-inv-into-comp:
  [| inj-on f (g ‘ A); inj-on g A; x : f ‘ g ‘ A |] ==>
    the-inv-into A (f o g) x = (the-inv-into A g o the-inv-into (g ‘ A) f) x
  apply (rule the-inv-into-f-eq)
  apply (fast intro: comp-inj-on)
  apply (simp add: f-the-inv-into-f the-inv-into-into)
  apply (simp add: the-inv-into-into)
done

```

```

lemma inj-on-the-inv-into:
  inj-on f A ==> inj-on (the-inv-into A f) (f ‘ A)
by (auto intro: inj-onI simp: image-def the-inv-into-f-f)

```

```

lemma bij-betw-the-inv-into:
  bij-betw f A B ==> bij-betw (the-inv-into A f) B A
by (auto simp add: bij-betw-def inj-on-the-inv-into the-inv-into-into)

```

```

abbreviation the-inv :: ('a => 'b) => ('b => 'a) where
  the-inv f ≡ the-inv-into UNIV f

```

```

lemma the-inv-f-f:
  assumes inj f
  shows the-inv f (f x) = x using assms UNIV-I
  by (rule the-inv-into-f-f)

```

## 10.9 Proof tool setup

simplifies terms of the form  $f(\dots, x := y, \dots, x := z, \dots)$  to  $f(\dots, x := z, \dots)$

```

simproc-setup fun-upd2 (f(v := w, x := y)) = << fn - =>

```

```

let

```

```

  fun gen-fun-upd NONE T - = NONE
  | gen-fun-upd (SOME f) T x y = SOME (Const (@{const-name fun-upd}, T)

```

```

$ f $ x $ y)
fun dest-fun-T1 (Type (-, T :: Ts)) = T
fun find-double (t as Const (@{const-name fun-upd}, T) $ f $ x $ y) =
  let
    fun find (Const (@{const-name fun-upd}, T) $ g $ v $ w) =
      if v aconv x then SOME g else gen-fun-upd (find g) T v w
    | find t = NONE
  in (dest-fun-T1 T, gen-fun-upd (find f) T x y) end

fun proc ss ct =
  let
    val ctxt = Simplifier.the-context ss
    val t = Thm.term-of ct
  in
    case find-double t of
      (T, NONE) => NONE
    | (T, SOME rhs) =>
        SOME (Goal.prove ctxt [] (Logic.mk-equals (t, rhs))
              (fn - =>
                 rtac eq-reflection 1 THEN
                 rtac ext 1 THEN
                 simp-tac (Simplifier.inherit-context ss @ {simpset}) 1))
    end
  in proc end
  >>

```

## 10.10 Code generator setup

### types-code

```

fun ((- ->/ -))
attach (term-of) <<
fun term-of-fun-type - aT - bT - = Free (<function>, aT --> bT);
>>
attach (test) <<
fun gen-fun-type aF aT bG bT i =
  let
    val tab = Unsynchronized.ref [];
    fun mk-upd (x, (-, y)) t = Const (Fun.fun-upd,
      (aT --> bT) --> aT --> bT --> aT --> bT) $ t $ aF x $ y ()
  in
    (fn x =>
      case AList.lookup op = (!tab) x of
        NONE =>
          let val p as (y, -) = bG i
          in (tab := (x, p) :: !tab; y) end
        | SOME (y, -) => y,
      fn () => Basics.fold mk-upd (!tab) (Const (HOL.undefined, aT --> bT)))
    end;
  >>

```

```

code-const op ◦
  (SML infixl 5 o)
  (Haskell infixr 9 .)

```

```

code-const id
  (Haskell id)

```

```

end

```

## 11 Product-Type: Cartesian products

```

theory Product-Type
imports Typedef Inductive Fun
uses
  (Tools/split-rule.ML)
  (Tools/inductive-set.ML)
begin

```

### 11.1 *bool* is a datatype

```

rep-datatype True False by (auto intro: bool-induct)

```

```

declare case-split [cases type: bool]
  — prefer plain propositional version

```

```

lemma
  shows [code]: eq-class.eq False P  $\longleftrightarrow \neg P$ 
    and [code]: eq-class.eq True P  $\longleftrightarrow P$ 
    and [code]: eq-class.eq P False  $\longleftrightarrow \neg P$ 
    and [code]: eq-class.eq P True  $\longleftrightarrow P$ 
    and [code nbe]: eq-class.eq P P  $\longleftrightarrow \text{True}$ 
  by (simp-all add: eq)

```

```

code-const eq-class.eq :: bool  $\Rightarrow$  bool  $\Rightarrow$  bool
  (Haskell infixl 4 ==)

```

```

code-instance bool :: eq
  (Haskell -)

```

### 11.2 The *unit* type

```

typedef unit = {True}
proof
  show True : ?unit ..
qed

```

```

definition

```

```

Unity :: unit    ('())
where
  () = Abs-unit True

```

```

lemma unit-eq [no-atp]: u = ()
  by (induct u) (simp add: unit-def Unity-def)

```

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

```

ML <<
  val unit-eq-proc =
    let val unit-meta-eq = mk-meta-eq @ {thm unit-eq} in
      Simplifier.simproc @ {theory} unit-eq [x::unit]
        (fn - => fn - => fn t => if HOLogic.is-unit t then NONE else SOME
unit-meta-eq)
    end;

  Addsimprocs [unit-eq-proc];
>>

```

```

rep-datatype () by simp

```

```

lemma unit-all-eq1: (!x::unit. PROP P x) == PROP P ()
  by simp

```

```

lemma unit-all-eq2: (!x::unit. PROP P) == PROP P
  by (rule triv-forall-equality)

```

This rewrite counters the effect of *unit-eq-proc* on  $\%u::unit. f u$ , replacing it by  $f$  rather than by  $\%u. f ()$ .

```

lemma unit-abs-eta-conv [simp,no-atp]: (%u::unit. f ()) = f
  by (rule ext) simp

```

```

instantiation unit :: default
begin

```

```

definition default = ()

```

```

instance ..

```

```

end

```

```

lemma [code]:
  eq-class.eq (u::unit) v <=> True unfolding eq unit-eq [of u] unit-eq [of v] by
rule+

```

```

code-type unit
  (SML unit)
  (OCaml unit)

```

(*Haskell* ())  
 (*Scala* *Unit*)

**code-const** *Unity*

(*SML* ())  
 (*OCaml* ())  
 (*Haskell* ())  
 (*Scala* ())

**code-instance** *unit* :: *eq*

(*Haskell*  $-$ )

**code-const** *eq-class.eq* :: *unit*  $\Rightarrow$  *unit*  $\Rightarrow$  *bool*

(*Haskell* **infixl** 4 ==)

**code-reserved** *SML*

*unit*

**code-reserved** *OCaml*

*unit*

**code-reserved** *Scala*

*Unit*

## 11.3 The product type

### 11.3.1 Type definition

**definition** *Pair-Rep* :: '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  *bool* **where**

*Pair-Rep* *a* *b* = ( $\lambda x y. x = a \wedge y = b$ )

**global**

**typedef** (*Prod*)

(*'a*, '*b*) \* (**infixr** \* 20)  
 = {*f*.  $\exists a b. f = \text{Pair-Rep } (a::'a) (b::'b)$ }

**proof**

**fix** *a* *b* **show** *Pair-Rep* *a* *b*  $\in$  ?*Prod*

**by** *rule+*

**qed**

**type-notation** (*xsymbols*)

\* (( $- \times / -$ ) [21, 20] 20)

**type-notation** (*HTML output*)

\* (( $- \times / -$ ) [21, 20] 20)

**consts**

*Pair* :: '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*a*  $\times$  '*b*

**local**

**defs**

*Pair-def*:  $\text{Pair } a \ b == \text{Abs-Prod } (\text{Pair-Rep } a \ b)$

**rep-datatype** (*prod*) *Pair* **proof** –

**fix**  $P :: 'a \times 'b \Rightarrow \text{bool}$  **and**  $p$

**assume**  $\bigwedge a \ b. P (\text{Pair } a \ b)$

**then show**  $P \ p$  **by** (*cases p*) (*auto simp add: Prod-def Pair-def Pair-Rep-def*)

**next**

**fix**  $a \ c :: 'a$  **and**  $b \ d :: 'b$

**have**  $\text{Pair-Rep } a \ b = \text{Pair-Rep } c \ d \longleftrightarrow a = c \wedge b = d$

**by** (*auto simp add: Pair-Rep-def expand-fun-eq*)

**moreover have**  $\text{Pair-Rep } a \ b \in \text{Prod}$  **and**  $\text{Pair-Rep } c \ d \in \text{Prod}$

**by** (*auto simp add: Prod-def*)

**ultimately show**  $\text{Pair } a \ b = \text{Pair } c \ d \longleftrightarrow a = c \wedge b = d$

**by** (*simp add: Pair-def Abs-Prod-inject*)

**qed**

### 11.3.2 Tuple syntax

**global consts**

*split*  $:: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$

**local defs**

*split-prod-case*:  $\text{split} == \text{prod-case}$

Patterns – extends pre-defined type *pttrn* used in abstractions.

**nonterminals**

*tuple-args patterns*

**syntax**

*-tuple*  $:: 'a \Rightarrow \text{tuple-args} \Rightarrow 'a * 'b \quad ((1 \ '(-, / -')))$

*-tuple-arg*  $:: 'a \Rightarrow \text{tuple-args} \quad (-)$

*-tuple-args*  $:: 'a \Rightarrow \text{tuple-args} \Rightarrow \text{tuple-args} \quad (-, / -)$

*-pattern*  $:: [\text{pttrn}, \text{patterns}] \Rightarrow \text{pttrn} \quad ('(-, / -'))$

$:: \text{pttrn} \Rightarrow \text{patterns} \quad (-)$

*-patterns*  $:: [\text{pttrn}, \text{patterns}] \Rightarrow \text{patterns} \quad (-, / -)$

**translations**

$(x, y) == \text{CONST Pair } x \ y$

*-tuple*  $x \ (-\text{tuple-args } y \ z) == -\text{tuple } x \ (-\text{tuple-arg } (-\text{tuple } y \ z))$

$\%(x, y, zs). b == \text{CONST split } (\%x \ (y, zs). b)$

$\%(x, y). b == \text{CONST split } (\%x \ y. b)$

*-abs*  $(\text{CONST Pair } x \ y) \ t \Rightarrow \%(x, y). t$

— The last rule accommodates tuples in ‘case C ... (x,y) ... =<sub>i</sub> ...’ The (x,y) is parsed as ‘Pair x y’ because it is logic, not pttrn

**print-translation**  $\ll$

```

let
  fun split-tr' [Abs (x, T, t as (Abs abs))] =
    (* split (%x y. t) => %(x,y) t *)
    let
      val (y, t') = atomic-abs-tr' abs;
      val (x', t'') = atomic-abs-tr' (x, T, t');
    in
      Syntax.const @{syntax-const -abs} $
        (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
    end
  | split-tr' [Abs (x, T, (s as Const (@{const-syntax split}, -) $ t))] =
    (* split (%x. (split (%y z. t))) => %(x,y,z). t *)
    let
      val Const (@{syntax-const -abs}, -) $
        (Const (@{syntax-const -pattern}, -) $ y $ z) $ t' = split-tr' [t];
      val (x', t'') = atomic-abs-tr' (x, T, t');
    in
      Syntax.const @{syntax-const -abs} $
        (Syntax.const @{syntax-const -pattern} $ x' $
          (Syntax.const @{syntax-const -patterns} $ y $ z)) $ t''
    end
  | split-tr' [Const (@{const-syntax split}, -) $ t] =
    (* split (split (%x y z. t)) => %((x, y), z). t *)
    split-tr' [(split-tr' [t])] (* inner split-tr' creates next pattern *)
  | split-tr' [Const (@{syntax-const -abs}, -) $ x-y $ Abs abs] =
    (* split (%pttrn z. t) => %(pttrn,z). t *)
    let val (z, t) = atomic-abs-tr' abs in
      Syntax.const @{syntax-const -abs} $
        (Syntax.const @{syntax-const -pattern} $ x-y $ z) $ t
    end
  | split-tr' - = raise Match;
in [(@{const-syntax split}, split-tr')] end
>>

```

**typed-print-translation**  $\ll$ 

```

let
  fun split-guess-names-tr' - T [Abs (x, -, Abs -)] = raise Match
  | split-guess-names-tr' - T [Abs (x, xT, t)] =
    (case (head-of t) of
      Const (@{const-syntax split}, -) => raise Match
    | - =>
      let
        val (- :: yT :: -) = binder-types (domain-type T) handle Bind => raise
Match;
        val (y, t') = atomic-abs-tr' (y, yT, incr-boundvars 1 t $ Bound 0);
        val (x', t'') = atomic-abs-tr' (x, xT, t');
      in
        Syntax.const @{syntax-const -abs} $

```



```

      (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
    end)
  | split-guess-names-tr' - T [t] =
    (case head-of t of
      Const (@{const-syntax split}, -) => raise Match
    | - =>
      let
        val (xT :: yT :: -) = binder-types (domain-type T) handle Bind => raise
Match;
        val (y, t') = atomic-abs-tr' (y, yT, incr-boundvars 2 t $ Bound 1 $
Bound 0);
        val (x', t'') = atomic-abs-tr' (x, xT, t');
      in
        Syntax.const @{syntax-const -abs} $
          (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
        end)
    | split-guess-names-tr' - - = raise Match;
in [(@{const-syntax split}, split-guess-names-tr')] end
>>

```

### 11.3.3 Code generator setup

**lemma** *split-case-cert*:

**assumes** *CASE*  $\equiv$  *split f*  
**shows** *CASE* (*a*, *b*)  $\equiv$  *f a b*  
**using** *assms* **by** (*simp add: split-prod-case*)

```

setup <<
  Code.add-case @{thm split-case-cert}
>>

```

**code-type** \*  
 (*SML* **infix** 2 \*)  
 (*OCaml* **infix** 2 \*)  
 (*Haskell* !((-),/ (-)))  
 (*Scala* ((-),/ (-)))

**code-const** *Pair*  
 (*SML* !((-),/ (-)))  
 (*OCaml* !((-),/ (-)))  
 (*Haskell* !((-),/ (-)))  
 (*Scala* !((-),/ (-)))

**code-instance** \* :: *eq*  
 (*Haskell* -)

**code-const** *eq-class.eq* :: '*a*::*eq*  $\times$  '*b*::*eq*  $\Rightarrow$  '*a*  $\times$  '*b*  $\Rightarrow$  *bool*  
 (*Haskell* **infixl** 4 ==)

**types-code**

```

*      ((- */ -))
attach (term-of) ⟨⟨
fun term-of-id-42 aF aT bF bT (x, y) = HOLogic.pair-const aT bT $ aF x $ bF
y;
⟩⟩
attach (test) ⟨⟨
fun gen-id-42 aG aT bG bT i =
  let
    val (x, t) = aG i;
    val (y, u) = bG i
    in ((x, y), fn () => HOLogic.pair-const aT bT $ t () $ u ()) end;
⟩⟩

```

**consts-code**

```

Pair      ((-, / -))

```

**setup** ⟨⟨

```

let

```

```

fun strip-abs-split 0 t = ([], t)
  | strip-abs-split i (Abs (s, T, t)) =
    let
      val s' = Codegen.new-name t s;
      val v = Free (s', T)
      in apfst (cons v) (strip-abs-split (i-1) (subst-bound (v, t))) end
  | strip-abs-split i (u as Const (@{const-name split}, -) $ t) =
    (case strip-abs-split (i+1) t of
      (v :: v' :: vs, u) => (HOLogic.mk-prod (v, v') :: vs, u)
      | - => ([], u))
  | strip-abs-split i t =
    strip-abs-split i (Abs (x, hd (binder-types (fastype-of t)), t $ Bound 0));

```

```

fun let-codegen thy defs dep thynome brack t gr =

```

```

  (case strip-comb t of

```

```

    (t1 as Const (@{const-name Let}, -), t2 :: t3 :: ts) =>

```

```

    let

```

```

      fun dest-let (l as Const (@{const-name Let}, -) $ t $ u) =

```

```

        (case strip-abs-split 1 u of

```

```

          ([p], u') => apfst (cons (p, t)) (dest-let u')

```

```

          | - => ([], l))

```

```

        | dest-let t = ([], t);

```

```

      fun mk-code (l, r) gr =

```

```

        let

```

```

          val (pl, gr1) = Codegen.invoke-codegen thy defs dep thynome false l gr;

```

```

          val (pr, gr2) = Codegen.invoke-codegen thy defs dep thynome false r gr1;

```

```

          in ((pl, pr), gr2) end

```

```

      in case dest-let (t1 $ t2 $ t3) of

```

```

        ([], -) => NONE

```

```

| (ps, u) =>
  let
    val (qs, gr1) = fold-map mk-code ps gr;
    val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
    val (pargs, gr3) = fold-map
      (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
  in
    SOME (Codegen.mk-app brack
      (Pretty.blk (0, [Codegen.str let , Pretty.blk (0, flat
        (separate [Codegen.str ;, Pretty.brk 1] (map (fn (pl, pr) =>
          [Pretty.block [Codegen.str val , pl, Codegen.str =,
            Pretty.brk 1, pr]]) qs))),
        Pretty.brk 1, Codegen.str in , pu,
        Pretty.brk 1, Codegen.str end])) pargs, gr3)
  end
end
| - => NONE);

fun split-codegen thy defs dep thyname brack t gr = (case strip-comb t of
  (t1 as Const (@{const-name split}, -), t2 :: ts) =>
    let
      val ([p], u) = strip-abs-split 1 (t1 $ t2);
      val (q, gr1) = Codegen.invoke-codegen thy defs dep thyname false p gr;
      val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
      val (pargs, gr3) = fold-map
        (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
    in
      SOME (Codegen.mk-app brack
        (Pretty.block [Codegen.str (fn , q, Codegen.str =>,
          Pretty.brk 1, pu, Codegen.str )]) pargs, gr2)
    end
  | - => NONE);

in

  Codegen.add-codegen let-codegen let-codegen
  #> Codegen.add-codegen split-codegen split-codegen

end
>>

```

#### 11.3.4 Fundamental operations and properties

**lemma** *surj-pair* [simp]:  $EX\ x\ y. p = (x, y)$   
 by (cases p) simp

**global consts**

*fst*     ::  $'a \times 'b \Rightarrow 'a$   
*snd*     ::  $'a \times 'b \Rightarrow 'b$

**local defs**

*fst-def*:  $\text{fst } p == \text{case } p \text{ of } (a, b) \Rightarrow a$   
*snd-def*:  $\text{snd } p == \text{case } p \text{ of } (a, b) \Rightarrow b$

**lemma** *fst-conv* [*simp*, *code*]:  $\text{fst } (a, b) = a$   
**unfolding** *fst-def* **by** *simp*

**lemma** *snd-conv* [*simp*, *code*]:  $\text{snd } (a, b) = b$   
**unfolding** *snd-def* **by** *simp*

**code-const** *fst* **and** *snd*  
 (*Haskell fst and snd*)

**lemma** *prod-case-unfold*:  $\text{prod-case} = (\%c \ p. \ c \ (\text{fst } p) \ (\text{snd } p))$   
**by** (*simp add: expand-fun-eq split: prod.split*)

**lemma** *fst-eqD*:  $\text{fst } (x, y) = a ==> x = a$   
**by** *simp*

**lemma** *snd-eqD*:  $\text{snd } (x, y) = a ==> y = a$   
**by** *simp*

**lemma** *pair-collapse* [*simp*]:  $(\text{fst } p, \text{snd } p) = p$   
**by** (*cases p*) *simp*

**lemmas** *surjective-pairing* = *pair-collapse* [*symmetric*]

**lemma** *Pair-fst-snd-eq*:  $s = t \longleftrightarrow \text{fst } s = \text{fst } t \wedge \text{snd } s = \text{snd } t$   
**by** (*cases s, cases t*) *simp*

**lemma** *prod-eqI* [*intro?*]:  $\text{fst } p = \text{fst } q \implies \text{snd } p = \text{snd } q \implies p = q$   
**by** (*simp add: Pair-fst-snd-eq*)

**lemma** *split-conv* [*simp*, *code*]:  $\text{split } f \ (a, b) = f \ a \ b$   
**by** (*simp add: split-prod-case*)

**lemma** *splitI*:  $f \ a \ b \implies \text{split } f \ (a, b)$   
**by** (*rule split-conv [THEN iffD2]*)

**lemma** *splitD*:  $\text{split } f \ (a, b) \implies f \ a \ b$   
**by** (*rule split-conv [THEN iffD1]*)

**lemma** *split-Pair* [*simp*]:  $(\lambda(x, y). (x, y)) = \text{id}$   
**by** (*simp add: split-prod-case expand-fun-eq split: prod.split*)

**lemma** *split-eta*:  $(\lambda(x, y). f \ (x, y)) = f$   
 — Subsumes the old *split-Pair* when *f* is the identity function.  
**by** (*simp add: split-prod-case expand-fun-eq split: prod.split*)

**lemma** *split-comp*:  $\text{split } (f \circ g) \ x = f \ (g \ (\text{fst } x)) \ (\text{snd } x)$   
**by**  $(\text{cases } x) \ \text{simp}$

**lemma** *split-twice*:  $\text{split } f \ (\text{split } g \ p) = \text{split } (\lambda x \ y. \ \text{split } f \ (g \ x \ y)) \ p$   
**by**  $(\text{cases } p) \ \text{simp}$

**lemma** *The-split*:  $\text{The } (\text{split } P) = (\text{THE } xy. \ P \ (\text{fst } xy) \ (\text{snd } xy))$   
**by**  $(\text{simp add: split-prod-case prod-case-unfold})$

**lemma** *split-weak-cong*:  $p = q \implies \text{split } c \ p = \text{split } c \ q$   
— Prevents simplification of  $c$ : much faster  
**by**  $(\text{erule arg-cong})$

**lemma** *cond-split-eta*:  $(!!x \ y. \ f \ x \ y = g \ (x, \ y)) \implies (\% (x, \ y). \ f \ x \ y) = g$   
**by**  $(\text{simp add: split-eta})$

**lemma** *split-paired-all*:  $(!!x. \ \text{PROP } P \ x) == (!!a \ b. \ \text{PROP } P \ (a, \ b))$   
**proof**  
**fix**  $a \ b$   
**assume**  $!!x. \ \text{PROP } P \ x$   
**then show**  $\text{PROP } P \ (a, \ b) \ .$   
**next**  
**fix**  $x$   
**assume**  $!!a \ b. \ \text{PROP } P \ (a, \ b)$   
**from**  $\langle \text{PROP } P \ (\text{fst } x, \ \text{snd } x) \rangle$  **show**  $\text{PROP } P \ x$  **by** *simp*  
**qed**

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congruence rules, where this can lead to premises of the form  $!!a \ b. \ \dots = ?P(a, \ b)$  which cannot be solved by reflexivity.

**lemmas** *split-tupled-all* = *split-paired-all unit-all-eq2*

**ML**  $\ll$   
 $(*$  *replace parameters of product type by individual component parameters*  $*)$   
 $\text{val safe-full-simp-tac} = \text{generic-simp-tac } \text{true} \ (\text{true}, \ \text{false}, \ \text{false});$   
 $\text{local } (*$  *filtering with exists-paired-all is an essential optimization*  $*)$   
 $\text{fun exists-paired-all } (\text{Const } (all, \ -) \ \$ \ \text{Abs } (-, \ T, \ t)) =$   
 $\quad \text{can } \text{HOLogic.dest-prodT } T \ \text{orelse exists-paired-all } t$   
 $\quad | \ \text{exists-paired-all } (t \ \$ \ u) = \text{exists-paired-all } t \ \text{orelse exists-paired-all } u$   
 $\quad | \ \text{exists-paired-all } (\text{Abs } (-, \ -, \ t)) = \text{exists-paired-all } t$   
 $\quad | \ \text{exists-paired-all } - = \text{false};$   
 $\text{val ss} = \text{HOL-basic-ss}$   
 $\text{addsimps } [\text{@}\{\text{thm split-paired-all}\}, \ \text{@}\{\text{thm unit-all-eq2}\}, \ \text{@}\{\text{thm unit-abs-eta-conv}\}]$   
 $\text{addsimprocs } [\text{unit-eq-proc};$   
**in**  
 $\text{val split-all-tac} = \text{SUBGOAL } (\text{fn } (t, \ i) \ =>$   
 $\quad \text{if exists-paired-all } t \ \text{then safe-full-simp-tac } ss \ i \ \text{else no-tac});$   
 $\text{val unsafe-split-all-tac} = \text{SUBGOAL } (\text{fn } (t, \ i) \ =>$

```

    if exists-paired-all t then full-simp-tac ss i else no-tac);
  fun split-all th =
    if exists-paired-all (Thm.prop-of th) then full-simplify ss th else th;
end;
>>

```

```

declaration << fn - =>
  Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac))
>>

```

**lemma** *split-paired-All* [simp]:  $(ALL\ x.\ P\ x) = (ALL\ a\ b.\ P\ (a,\ b))$   
 — [iff] is not a good idea because it makes *blast* loop  
**by** *fast*

**lemma** *split-paired-Ex* [simp]:  $(EX\ x.\ P\ x) = (EX\ a\ b.\ P\ (a,\ b))$   
**by** *fast*

**lemma** *split-paired-The*:  $(THE\ x.\ P\ x) = (THE\ (a,\ b).\ P\ (a,\ b))$   
 — Can’t be added to simpset: loops!  
**by** (*simp add: split-eta*)

Simplification procedure for *cond-split-eta*. Using *split-eta* as a rewrite rule is not general enough, and using *cond-split-eta* directly would render some existing proofs very inefficient; similarly for *split-beta*.

```

ML <<
local
  val cond-split-eta-ss = HOL-basic-ss addsimps @ {thms cond-split-eta};
  fun Pair-pat k 0 (Bound m) = (m = k)
    | Pair-pat k i (Const (@{const-name Pair}, -) $ Bound m $ t) =
      i > 0 andalso m = k + i andalso Pair-pat k (i - 1) t
    | Pair-pat - - - = false;
  fun no-args k i (Abs (-, -, t)) = no-args (k + 1) i t
    | no-args k i (t $ u) = no-args k i t andalso no-args k i u
    | no-args k i (Bound m) = m < k orelse m > k + i
    | no-args - - - = true;
  fun split-pat tp i (Abs (-, -, t)) = if tp 0 i t then SOME (i, t) else NONE
    | split-pat tp i (Const (@{const-name split}, -) $ Abs (-, -, t)) = split-pat tp (i
+ 1) t
    | split-pat tp i - = NONE;
  fun metaeq ss lhs rhs = mk-meta-eq (Goal.prove (Simplifier.the-context ss) [] []
    (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs)))
    (K (simp-tac (Simplifier.inherit-context ss cond-split-eta-ss) 1)));

  fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k + 1) i t
    | beta-term-pat k i (t $ u) =
      Pair-pat k i (t $ u) orelse (beta-term-pat k i t andalso beta-term-pat k i u)
    | beta-term-pat k i t = no-args k i t;
  fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
    | eta-term-pat - - - = false;

```

```

fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
  | subst arg k i (t $ u) =
    if Pair-pat k i (t $ u) then incr-boundvars k arg
    else (subst arg k i t $ subst arg k i u)
  | subst arg k i t = t;
fun beta-proc ss (s as Const (@{const-name split}, -) $ Abs (-, -, t) $ arg) =
  (case split-pat beta-term-pat 1 t of
    SOME (i, f) => SOME (metaeq ss s (subst arg 0 i f))
  | NONE => NONE)
  | beta-proc - - = NONE;
fun eta-proc ss (s as Const (@{const-name split}, -) $ Abs (-, -, t)) =
  (case split-pat eta-term-pat 1 t of
    SOME (-, ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
  | NONE => NONE)
  | eta-proc - - = NONE;
in
  val split-beta-proc = Simplifier.simproc @{theory} split-beta [split f z] (K beta-proc);
  val split-eta-proc = Simplifier.simproc @{theory} split-eta [split f] (K eta-proc);
end;

Addsimprocs [split-beta-proc, split-eta-proc];
>>

```

**lemma** *split-beta* [mono]:  $(\%(x, y). P \ x \ y) \ z = P \ (fst \ z) \ (snd \ z)$   
**by** (*subst surjective-pairing*, *rule split-conv*)

**lemma** *split-split* [no-atp]:  $R(split \ c \ p) = (ALL \ x \ y. p = (x, y) \ \longrightarrow R(c \ x \ y))$   
 — For use with *split* and the Simplifier.  
**by** (*insert surj-pair* [of *p*], *clarify*, *simp*)

*split-split* could be declared as [*split*] done after the Splitter has been speeded up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

**lemma** *split-split-asm* [no-atp]:  $R \ (split \ c \ p) = (\sim (EX \ x \ y. p = (x, y) \ \& \ (\sim R \ (c \ x \ y))))$   
**by** (*subst split-split*, *simp*)

*split* used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

**lemma** *splitI2*:  $!!p. [\![ \![a \ b. p = (a, b) \implies c \ a \ b] \!] \implies split \ c \ p$   
**apply** (*simp only: split-tupled-all*)  
**apply** (*simp (no-asm-simp)*)  
**done**

**lemma** *splitI2'*:  $!!p. [\![ \![a \ b. (a, b) = p \implies c \ a \ b \ x] \!] \implies split \ c \ p \ x$   
**apply** (*simp only: split-tupled-all*)

```

apply (simp (no-asm-simp))
done

lemma splitE: split c p ==> (!!x y. p = (x, y) ==> c x y ==> Q) ==> Q
by (induct p) (auto simp add: split-prod-case)

lemma splitE': split c p z ==> (!!x y. p = (x, y) ==> c x y z ==> Q) ==> Q
by (induct p) (auto simp add: split-prod-case)

lemma splitE2:
  [| Q (split P z); !!x y. [| z = (x, y); Q (P x y) |] ==> R |] ==> R
proof -
  assume q: Q (split P z)
  assume r: !!x y. [| z = (x, y); Q (P x y) |] ==> R
  show R
  apply (rule r surjective-pairing)+
  apply (rule split-beta [THEN subst], rule q)
  done
qed

lemma splitD': split R (a,b) c ==> R a b c
by simp

lemma mem-splitI: z: c a b ==> z: split c (a, b)
by simp

lemma mem-splitI2: !!p. [| !!a b. p = (a, b) ==> z: c a b |] ==> z: split c p
by (simp only: split-tupled-all, simp)

lemma mem-splitE:
  assumes major: z ∈ split c p
  and cases:  $\bigwedge x y. p = (x, y) \implies z \in c x y \implies Q$ 
  shows Q
  by (rule major [unfolded split-prod-case prod-case-unfold] cases surjective-pairing)+

declare mem-splitI2 [intro!] mem-splitI [intro!] splitI2' [intro!] splitI2 [intro!] splitI
  [intro!]
declare mem-splitE [elim!] splitE' [elim!] splitE [elim!]

ML ⟨⟨
  local (* filtering with exists-p-split is an essential optimization *)
    fun exists-p-split (Const (@{const-name split},-) $ - $ (Const (@{const-name
  Pair},-) $ $-)) = true
    | exists-p-split (t $ u) = exists-p-split t orelse exists-p-split u
    | exists-p-split (Abs (_, -, t)) = exists-p-split t
    | exists-p-split - = false;
    val ss = HOL-basic-ss addsimps @{thms split-conv};
  in
    val split-conv-tac = SUBGOAL (fn (t, i) =>

```



```

    if exists-p-split t then safe-full-simp-tac ss i else no-tac);
end;
>>

```

```

declaration << fn - =>
  Classical.map-cs (fn cs => cs addSbefore (split-conv-tac, split-conv-tac))
>>

```

```

lemma split-eta-SetCompr [simp,no-atp]: (%u. EX x y. u = (x, y) & P (x, y)) =
P
  by (rule ext) fast

```

```

lemma split-eta-SetCompr2 [simp,no-atp]: (%u. EX x y. u = (x, y) & P x y) =
split P
  by (rule ext) fast

```

```

lemma split-part [simp]: (%(a,b). P & Q a b) = (%ab. P & split Q ab)
— Allows simplifications of nested splits in case of independent predicates.
by (rule ext) blast

```

```

lemma split-comp-eq:
  fixes f :: 'a => 'b => 'c and g :: 'd => 'a
  shows (%u. f (g (fst u)) (snd u)) = (split (%x. f (g x)))
  by (rule ext) auto

```

```

lemma pair-imageI [intro]: (a, b) : A ==> f a b : (%(a, b). f a b) ‘ A
apply (rule-tac x = (a, b) in image-eqI)
apply auto
done

```

```

lemma The-split-eq [simp]: (THE (x',y'). x = x' & y = y') = (x, y)
by blast

```

Setup of internal *split-rule*.

```

lemmas prod-caseI = prod.cases [THEN iffD2, standard]

```

```

lemma prod-caseI2: !!p. [| !!a b. p = (a, b) ==> c a b |] ==> prod-case c p
by auto

```

```

lemma prod-caseI2': !!p. [| !!a b. (a, b) = p ==> c a b x |] ==> prod-case c p x
by (auto simp: split-tupled-all)

```

```

lemma prod-caseE: prod-case c p ==> (!!x y. p = (x, y) ==> c x y ==> Q)
==> Q
by (induct p) auto

```

```

lemma prod-caseE': prod-case c p z ==> (!!x y. p = (x, y) ==> c x y z ==>

```

$Q) ==> Q$   
**by** (*induct p*) *auto*

**declare** *prod-caseI2'* [*intro!*] *prod-caseI2* [*intro!*] *prod-caseI* [*intro!*]  
**declare** *prod-caseE'* [*elim!*] *prod-caseE* [*elim!*]

**lemma** *prod-case-split*:  
*prod-case* = *split*  
**by** (*auto simp add: expand-fun-eq*)

**lemma** *prod-case-beta*:  
*prod-case f p* = *f (fst p) (snd p)*  
**unfolding** *prod-case-split split-beta ..*

**lemma** *prod-cases3* [*cases type*]:  
**obtains** (*fields*) *a b c* **where** *y* = (*a, b, c*)  
**by** (*cases y, case-tac b*) *blast*

**lemma** *prod-induct3* [*case-names fields, induct type*]:  
 (!!*a b c. P (a, b, c)*) ==> *P x*  
**by** (*cases x*) *blast*

**lemma** *prod-cases4* [*cases type*]:  
**obtains** (*fields*) *a b c d* **where** *y* = (*a, b, c, d*)  
**by** (*cases y, case-tac c*) *blast*

**lemma** *prod-induct4* [*case-names fields, induct type*]:  
 (!!*a b c d. P (a, b, c, d)*) ==> *P x*  
**by** (*cases x*) *blast*

**lemma** *prod-cases5* [*cases type*]:  
**obtains** (*fields*) *a b c d e* **where** *y* = (*a, b, c, d, e*)  
**by** (*cases y, case-tac d*) *blast*

**lemma** *prod-induct5* [*case-names fields, induct type*]:  
 (!!*a b c d e. P (a, b, c, d, e)*) ==> *P x*  
**by** (*cases x*) *blast*

**lemma** *prod-cases6* [*cases type*]:  
**obtains** (*fields*) *a b c d e f* **where** *y* = (*a, b, c, d, e, f*)  
**by** (*cases y, case-tac e*) *blast*

**lemma** *prod-induct6* [*case-names fields, induct type*]:  
 (!!*a b c d e f. P (a, b, c, d, e, f)*) ==> *P x*  
**by** (*cases x*) *blast*

**lemma** *prod-cases7* [*cases type*]:  
**obtains** (*fields*) *a b c d e f g* **where** *y* = (*a, b, c, d, e, f, g*)  
**by** (*cases y, case-tac f*) *blast*

**lemma** *prod-induct7* [*case-names fields, induct type*]:  
 ( $\text{!!}a\ b\ c\ d\ e\ f\ g. P\ (a, b, c, d, e, f, g)$ )  $\implies P\ x$   
 by (*cases x*) *blast*

**lemma** *split-def*:  
 $\text{split} = (\lambda c\ p. c\ (\text{fst}\ p)\ (\text{snd}\ p))$   
 unfolding *split-prod-case prod-case-unfold* ..

**definition** *internal-split* ::  $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$  **where**  
*internal-split* == *split*

**lemma** *internal-split-conv*:  $\text{internal-split}\ c\ (a, b) = c\ a\ b$   
 by (*simp only: internal-split-def split-conv*)

**use** *Tools/split-rule.ML*  
**setup** *Split-Rule.setup*

**hide-const** *internal-split*

### 11.3.5 Derived operations

**global consts**  
 $\text{curry} :: ('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

**local defs**  
 $\text{curry-def: } \text{curry} == (\%c\ x\ y. c\ (\text{Pair}\ x\ y))$

**lemma** *curry-conv* [*simp, code*]:  $\text{curry}\ f\ a\ b = f\ (a, b)$   
 by (*simp add: curry-def*)

**lemma** *curryI* [*intro!*]:  $f\ (a, b) \implies \text{curry}\ f\ a\ b$   
 by (*simp add: curry-def*)

**lemma** *curryD* [*dest!*]:  $\text{curry}\ f\ a\ b \implies f\ (a, b)$   
 by (*simp add: curry-def*)

**lemma** *curryE*:  $\text{curry}\ f\ a\ b \implies (f\ (a, b) \implies Q) \implies Q$   
 by (*simp add: curry-def*)

**lemma** *curry-split* [*simp*]:  $\text{curry}\ (\text{split}\ f) = f$   
 by (*simp add: curry-def split-def*)

**lemma** *split-curry* [*simp*]:  $\text{split}\ (\text{curry}\ f) = f$   
 by (*simp add: curry-def split-def*)

The composition-uncurry combinator.

**notation** *fcomp* (**infixl** *o* > 60)

**definition** *scomp* :: ( $'a \Rightarrow 'b \times 'c$ )  $\Rightarrow$  ( $'b \Rightarrow 'c \Rightarrow 'd$ )  $\Rightarrow$   $'a \Rightarrow 'd$  (**infixl**  $o \rightarrow 60$ )  
**where**

$f \ o \rightarrow \ g = (\lambda x. \text{split } g \ (f \ x))$

**lemma** *scomp-apply*:  $(f \ o \rightarrow \ g) \ x = \text{split } g \ (f \ x)$   
**by** (*simp add: scomp-def*)

**lemma** *Pair-scomp*:  $\text{Pair } x \ o \rightarrow \ f = f \ x$   
**by** (*simp add: expand-fun-eq scomp-apply*)

**lemma** *scomp-Pair*:  $x \ o \rightarrow \ \text{Pair} = x$   
**by** (*simp add: expand-fun-eq scomp-apply*)

**lemma** *scomp-scomp*:  $(f \ o \rightarrow \ g) \ o \rightarrow \ h = f \ o \rightarrow (\lambda x. \ g \ x \ o \rightarrow \ h)$   
**by** (*simp add: expand-fun-eq split-twice scomp-def*)

**lemma** *scomp-fcomp*:  $(f \ o \rightarrow \ g) \ o > \ h = f \ o \rightarrow (\lambda x. \ g \ x \ o > \ h)$   
**by** (*simp add: expand-fun-eq scomp-apply fcomp-def split-def*)

**lemma** *fcomp-scomp*:  $(f \ o > \ g) \ o \rightarrow \ h = f \ o > (g \ o \rightarrow \ h)$   
**by** (*simp add: expand-fun-eq scomp-apply fcomp-apply*)

**code-const** *scomp*  
*(Eval infixl 3 #->)*

**no-notation** *fcomp* (**infixl**  $o > 60$ )

**no-notation** *scomp* (**infixl**  $o \rightarrow 60$ )

*prod-fun* — action of the product functor upon functions.

**definition** *prod-fun* :: ( $'a \Rightarrow 'c$ )  $\Rightarrow$  ( $'b \Rightarrow 'd$ )  $\Rightarrow$   $'a \times 'b \Rightarrow 'c \times 'd$  **where**  
*[code del]:* *prod-fun*  $f \ g = (\lambda(x, y). (f \ x, g \ y))$

**lemma** *prod-fun [simp, code]*: *prod-fun*  $f \ g \ (a, b) = (f \ a, g \ b)$   
**by** (*simp add: prod-fun-def*)

**lemma** *fst-prod-fun[simp]*: *fst* (*prod-fun*  $f \ g \ x$ ) =  $f \ (\text{fst } x)$   
**by** (*cases x, auto*)

**lemma** *snd-prod-fun[simp]*: *snd* (*prod-fun*  $f \ g \ x$ ) =  $g \ (\text{snd } x)$   
**by** (*cases x, auto*)

**lemma** *fst-comp-prod-fun[simp]*: *fst*  $\circ$  *prod-fun*  $f \ g = f \circ$  *fst*  
**by** (*rule ext*) *auto*

**lemma** *snd-comp-prod-fun[simp]*: *snd*  $\circ$  *prod-fun*  $f \ g = g \circ$  *snd*  
**by** (*rule ext*) *auto*

**lemma** *prod-fun-compose*:

```

  prod-fun (f1 o f2) (g1 o g2) = (prod-fun f1 g1 o prod-fun f2 g2)
by (rule ext) auto

```

```

lemma prod-fun-ident [simp]: prod-fun (%x. x) (%y. y) = (%z. z)
  by (rule ext) auto

```

```

lemma prod-fun-imageI [intro]: (a, b) : r ==> (f a, g b) : prod-fun f g ‘ r
  apply (rule image-eqI)
  apply (rule prod-fun [symmetric], assumption)
done

```

```

lemma prod-fun-imageE [elim!]:
  assumes major: c: (prod-fun f g) ‘ r
  and cases: !!x y. [| c=(f(x),g(y)); (x,y):r |] ==> P
  shows P
  apply (rule major [THEN imageE])
  apply (case-tac x)
  apply (rule cases)
  apply (blast intro: prod-fun)
  apply blast
done

```

```

definition apfst :: ('a => 'c) => 'a × 'b => 'c × 'b where
  apfst f = prod-fun f id

```

```

definition apsnd :: ('b => 'c) => 'a × 'b => 'a × 'c where
  apsnd f = prod-fun id f

```

```

lemma apfst-conv [simp, code]:
  apfst f (x, y) = (f x, y)
by (simp add: apfst-def)

```

```

lemma apsnd-conv [simp, code]:
  apsnd f (x, y) = (x, f y)
by (simp add: apsnd-def)

```

```

lemma fst-apfst [simp]:
  fst (apfst f x) = f (fst x)
by (cases x) simp

```

```

lemma fst-apsnd [simp]:
  fst (apsnd f x) = fst x
by (cases x) simp

```

```

lemma snd-apfst [simp]:
  snd (apfst f x) = snd x
by (cases x) simp

```

**lemma** *snd-apsnd [simp]*:  
 $\text{snd } (\text{apsnd } f \ x) = f \ (\text{snd } x)$   
**by** (*cases x simp*)

**lemma** *apfst-compose*:  
 $\text{apfst } f \ (\text{apfst } g \ x) = \text{apfst } (f \circ g) \ x$   
**by** (*cases x simp*)

**lemma** *apsnd-compose*:  
 $\text{apsnd } f \ (\text{apsnd } g \ x) = \text{apsnd } (f \circ g) \ x$   
**by** (*cases x simp*)

**lemma** *apfst-apsnd [simp]*:  
 $\text{apfst } f \ (\text{apsnd } g \ x) = (f \ (\text{fst } x), g \ (\text{snd } x))$   
**by** (*cases x simp*)

**lemma** *apsnd-apfst [simp]*:  
 $\text{apsnd } f \ (\text{apfst } g \ x) = (g \ (\text{fst } x), f \ (\text{snd } x))$   
**by** (*cases x simp*)

**lemma** *apfst-id [simp]* :  
 $\text{apfst } \text{id} = \text{id}$   
**by** (*simp add: expand-fun-eq*)

**lemma** *apsnd-id [simp]* :  
 $\text{apsnd } \text{id} = \text{id}$   
**by** (*simp add: expand-fun-eq*)

**lemma** *apfst-eq-conv [simp]*:  
 $\text{apfst } f \ x = \text{apfst } g \ x \longleftrightarrow f \ (\text{fst } x) = g \ (\text{fst } x)$   
**by** (*cases x simp*)

**lemma** *apsnd-eq-conv [simp]*:  
 $\text{apsnd } f \ x = \text{apsnd } g \ x \longleftrightarrow f \ (\text{snd } x) = g \ (\text{snd } x)$   
**by** (*cases x simp*)

**lemma** *apsnd-apfst-commute*:  
 $\text{apsnd } f \ (\text{apfst } g \ p) = \text{apfst } g \ (\text{apsnd } f \ p)$   
**by** *simp*

Disjoint union of a family of sets – Sigma.

**definition** *Sigma* :: [*'a set, 'a => 'b set*] => (*'a × 'b*) *set* **where**  
*Sigma-def*: *Sigma A B == UN x:A. UN y:B x. {Pair x y}*

**abbreviation**

*Times* :: [*'a set, 'b set*] => (*'a \* 'b*) *set*  
**(infixr** *<\*>* 80) **where**  
*A <\*> B == Sigma A (%-. B)*

**notation** (*xsymbols*)  
*Times* (**infixr**  $\times 80$ )

**notation** (*HTML output*)  
*Times* (**infixr**  $\times 80$ )

**syntax**  
 $\text{-Sigma} :: [\text{pttrn}, 'a \text{ set}, 'b \text{ set}] \Rightarrow ('a * 'b) \text{ set} \ ((3\text{SIGMA} \text{ :-./ } -) [0, 0, 10] 10)$

**translations**  
 $\text{SIGMA } x:A. B == \text{CONST Sigma } A \ (\%x. B)$

**lemma** *SigmaI* [*intro!*]:  $[[ a:A; b:B(a) ]] \Rightarrow (a,b) : \text{Sigma } A B$   
**by** (*unfold Sigma-def*) *blast*

**lemma** *SigmaE* [*elim!*]:  
 $[[ c: \text{Sigma } A B;$   
 $\quad !!x y. [[ x:A; y:B(x); c=(x,y) ]] \Rightarrow P$   
 $]] \Rightarrow P$   
 — The general elimination rule.  
**by** (*unfold Sigma-def*) *blast*

Elimination of  $(a, b) \in A \times B$  – introduces no eigenvariables.

**lemma** *SigmaD1*:  $(a, b) : \text{Sigma } A B \Rightarrow a : A$   
**by** *blast*

**lemma** *SigmaD2*:  $(a, b) : \text{Sigma } A B \Rightarrow b : B a$   
**by** *blast*

**lemma** *SigmaE2*:  
 $[[ (a, b) : \text{Sigma } A B;$   
 $\quad [[ a:A; b:B(a) ]] \Rightarrow P$   
 $]] \Rightarrow P$   
**by** *blast*

**lemma** *Sigma-cong*:  
 $[[ A = B; !!x. x \in B \Rightarrow C x = D x ]]$   
 $\Rightarrow (\text{SIGMA } x: A. C x) = (\text{SIGMA } x: B. D x)$   
**by** *auto*

**lemma** *Sigma-mono*:  $[[ A \leq C; !!x. x:A \Rightarrow B x \leq D x ]] \Rightarrow \text{Sigma } A B$   
 $\leq \text{Sigma } C D$   
**by** *blast*

**lemma** *Sigma-empty1* [*simp*]:  $\text{Sigma } \{\} B = \{\}$   
**by** *blast*

**lemma** *Sigma-empty2* [*simp*]:  $A <*> \{\} = \{\}$   
**by** *blast*

**lemma** *UNIV-Times-UNIV* [simp]:  $UNIV <*> UNIV = UNIV$   
**by** *auto*

**lemma** *Compl-Times-UNIV1* [simp]:  $\neg (UNIV <*> A) = UNIV <*> (\neg A)$   
**by** *auto*

**lemma** *Compl-Times-UNIV2* [simp]:  $\neg (A <*> UNIV) = (\neg A) <*> UNIV$   
**by** *auto*

**lemma** *mem-Sigma-iff* [iff]:  $((a,b): Sigma\ A\ B) = (a:A \ \&\ b:B(a))$   
**by** *blast*

**lemma** *Times-subset-cancel2*:  $x:C \implies (A <*> C \leq B <*> C) = (A \leq B)$   
**by** *blast*

**lemma** *Times-eq-cancel2*:  $x:C \implies (A <*> C = B <*> C) = (A = B)$   
**by** (*blast elim: equalityE*)

**lemma** *SetCompr-Sigma-eq*:  
 $Collect\ (split\ (\%x\ y.\ P\ x\ \&\ Q\ x\ y)) = (SIGMA\ x:Collect\ P.\ Collect\ (Q\ x))$   
**by** *blast*

**lemma** *Collect-split* [simp]:  $\{(a,b).\ P\ a\ \&\ Q\ b\} = Collect\ P\ <*>\ Collect\ Q$   
**by** *blast*

**lemma** *UN-Times-distrib*:  
 $(UN\ (a,b):(A <*> B).\ E\ a <*> F\ b) = (UNION\ A\ E) <*> (UNION\ B\ F)$   
— Suggested by Pierre Chartier  
**by** *blast*

**lemma** *split-paired-Ball-Sigma* [simp,no-atp]:  
 $(ALL\ z: Sigma\ A\ B.\ P\ z) = (ALL\ x:A.\ ALL\ y: B\ x.\ P(x,y))$   
**by** *blast*

**lemma** *split-paired-Bex-Sigma* [simp,no-atp]:  
 $(EX\ z: Sigma\ A\ B.\ P\ z) = (EX\ x:A.\ EX\ y: B\ x.\ P(x,y))$   
**by** *blast*

**lemma** *Sigma-Un-distrib1*:  $(SIGMA\ i:I\ Un\ J.\ C(i)) = (SIGMA\ i:I.\ C(i))\ Un\ (SIGMA\ j:J.\ C(j))$   
**by** *blast*

**lemma** *Sigma-Un-distrib2*:  $(SIGMA\ i:I.\ A(i)\ Un\ B(i)) = (SIGMA\ i:I.\ A(i))\ Un\ (SIGMA\ i:I.\ B(i))$   
**by** *blast*

**lemma** *Sigma-Int-distrib1*:  $(SIGMA\ i:I\ Int\ J.\ C(i)) = (SIGMA\ i:I.\ C(i))\ Int\ (SIGMA\ j:J.\ C(j))$   
**by** *blast*



**lemma** *Sigma-Int-distrib2*:  $(\text{SIGMA } i:I. A(i) \text{ Int } B(i)) = (\text{SIGMA } i:I. A(i)) \text{ Int } (\text{SIGMA } i:I. B(i))$   
**by** *blast*

**lemma** *Sigma-Diff-distrib1*:  $(\text{SIGMA } i:I - J. C(i)) = (\text{SIGMA } i:I. C(i)) - (\text{SIGMA } j:J. C(j))$   
**by** *blast*

**lemma** *Sigma-Diff-distrib2*:  $(\text{SIGMA } i:I. A(i) - B(i)) = (\text{SIGMA } i:I. A(i)) - (\text{SIGMA } i:I. B(i))$   
**by** *blast*

**lemma** *Sigma-Union*:  $\text{Sigma } (\text{Union } X) B = (\text{UN } A:X. \text{Sigma } A B)$   
**by** *blast*

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

**lemma** *Times-Un-distrib1*:  $(A \text{ Un } B) <*> C = (A <*> C) \text{ Un } (B <*> C)$   
**by** *blast*

**lemma** *Times-Int-distrib1*:  $(A \text{ Int } B) <*> C = (A <*> C) \text{ Int } (B <*> C)$   
**by** *blast*

**lemma** *Times-Diff-distrib1*:  $(A - B) <*> C = (A <*> C) - (B <*> C)$   
**by** *blast*

**lemma** *Times-empty[simp]*:  $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$   
**by** *auto*

**lemma** *fst-image-times[simp]*:  $\text{fst } ' (A \times B) = (\text{if } B = \{\} \text{ then } \{\} \text{ else } A)$   
**by**  $(\text{auto intro!}; \text{image-eqI})$

**lemma** *snd-image-times[simp]*:  $\text{snd } ' (A \times B) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } B)$   
**by**  $(\text{auto intro!}; \text{image-eqI})$

**lemma** *insert-times-insert[simp]*:  
 $\text{insert } a A \times \text{insert } b B =$   
 $\text{insert } (a,b) (A \times \text{insert } b B \cup \text{insert } a A \times B)$   
**by** *blast*

**lemma** *vimage-Times*:  $f - ' (A \times B) = ((\text{fst} \circ f) - ' A) \cap ((\text{snd} \circ f) - ' B)$   
**by**  $(\text{auto}, \text{case-tac } f \ x, \text{auto})$

The following *prod-fun* lemmas are due to Joachim Breitner:

**lemma** *prod-fun-inj-on*:  
**assumes** *inj-on*  $f$   $A$  **and** *inj-on*  $g$   $B$   
**shows** *inj-on*  $(\text{prod-fun } f \ g)$   $(A \times B)$   
**proof**  $(\text{rule } \text{inj-onI})$

**fix**  $x :: 'a \times 'c$  **and**  $y :: 'a \times 'c$   
**assume**  $x \in A \times B$  **hence**  $\text{fst } x \in A$  **and**  $\text{snd } x \in B$  **by** *auto*  
**assume**  $y \in A \times B$  **hence**  $\text{fst } y \in A$  **and**  $\text{snd } y \in B$  **by** *auto*  
**assume**  $\text{prod-fun } f \ g \ x = \text{prod-fun } f \ g \ y$   
**hence**  $\text{fst } (\text{prod-fun } f \ g \ x) = \text{fst } (\text{prod-fun } f \ g \ y)$  **by** *(auto)*  
**hence**  $f \ (\text{fst } x) = f \ (\text{fst } y)$  **by** *(cases x, cases y, auto)*  
**with**  $\langle \text{inj-on } f \ A \rangle$  **and**  $\langle \text{fst } x \in A \rangle$  **and**  $\langle \text{fst } y \in A \rangle$   
**have**  $\text{fst } x = \text{fst } y$  **by** *(auto dest:dest:inj-onD)*  
**moreover from**  $\langle \text{prod-fun } f \ g \ x = \text{prod-fun } f \ g \ y \rangle$   
**have**  $\text{snd } (\text{prod-fun } f \ g \ x) = \text{snd } (\text{prod-fun } f \ g \ y)$  **by** *(auto)*  
**hence**  $g \ (\text{snd } x) = g \ (\text{snd } y)$  **by** *(cases x, cases y, auto)*  
**with**  $\langle \text{inj-on } g \ B \rangle$  **and**  $\langle \text{snd } x \in B \rangle$  **and**  $\langle \text{snd } y \in B \rangle$   
**have**  $\text{snd } x = \text{snd } y$  **by** *(auto dest:dest:inj-onD)*  
**ultimately show**  $x = y$  **by** *(rule prod-eqI)*  
**qed**

**lemma** *prod-fun-surj*:  
**assumes** *surj f* **and** *surj g*  
**shows** *surj (prod-fun f g)*  
**unfolding** *surj-def*  
**proof**  
**fix**  $y :: 'b \times 'd$   
**from**  $\langle \text{surj } f \rangle$  **obtain**  $a$  **where**  $\text{fst } y = f \ a$  **by** *(auto elim:surjE)*  
**moreover**  
**from**  $\langle \text{surj } g \rangle$  **obtain**  $b$  **where**  $\text{snd } y = g \ b$  **by** *(auto elim:surjE)*  
**ultimately have**  $(\text{fst } y, \text{snd } y) = \text{prod-fun } f \ g \ (a, b)$  **by** *auto*  
**thus**  $\exists x. y = \text{prod-fun } f \ g \ x$  **by** *auto*  
**qed**

**lemma** *prod-fun-surj-on*:  
**assumes**  $f \ 'A = A'$  **and**  $g \ 'B = B'$   
**shows**  $\text{prod-fun } f \ g \ ' (A \times B) = A' \times B'$   
**unfolding** *image-def*  
**proof** *(rule set-ext, rule iffI)*  
**fix**  $x :: 'a \times 'c$   
**assume**  $x \in \{y :: 'a \times 'c. \exists x :: 'b \times 'd. x \in A \times B. y = \text{prod-fun } f \ g \ x\}$   
**then obtain**  $y$  **where**  $y \in A \times B$  **and**  $x = \text{prod-fun } f \ g \ y$  **by** *blast*  
**from**  $\langle \text{image } f \ A = A' \rangle$  **and**  $\langle y \in A \times B \rangle$  **have**  $f \ (\text{fst } y) \in A'$  **by** *auto*  
**moreover from**  $\langle \text{image } g \ B = B' \rangle$  **and**  $\langle y \in A \times B \rangle$  **have**  $g \ (\text{snd } y) \in B'$  **by** *auto*  
**ultimately have**  $(f \ (\text{fst } y), g \ (\text{snd } y)) \in (A' \times B')$  **by** *auto*  
**with**  $\langle x = \text{prod-fun } f \ g \ y \rangle$  **show**  $x \in A' \times B'$  **by** *(cases y, auto)*  
**next**  
**fix**  $x :: 'a \times 'c$   
**assume**  $x \in A' \times B'$  **hence**  $\text{fst } x \in A'$  **and**  $\text{snd } x \in B'$  **by** *auto*  
**from**  $\langle \text{image } f \ A = A' \rangle$  **and**  $\langle \text{fst } x \in A' \rangle$  **have**  $\text{fst } x \in \text{image } f \ A$  **by** *auto*  
**then obtain**  $a$  **where**  $a \in A$  **and**  $\text{fst } x = f \ a$  **by** *(rule imageE)*  
**moreover from**  $\langle \text{image } g \ B = B' \rangle$  **and**  $\langle \text{snd } x \in B' \rangle$   
**obtain**  $b$  **where**  $b \in B$  **and**  $\text{snd } x = g \ b$  **by** *auto*

ultimately have  $(fst\ x, snd\ x) = prod\_fun\ f\ g\ (a, b)$  by auto  
 moreover from  $\langle a \in A \rangle$  and  $\langle b \in B \rangle$  have  $(a, b) \in A \times B$  by auto  
 ultimately have  $\exists y \in A \times B. x = prod\_fun\ f\ g\ y$  by auto  
 thus  $x \in \{x. \exists y \in A \times B. x = prod\_fun\ f\ g\ y\}$  by auto  
 qed

lemma swap-inj-on:  
 inj-on  $(\lambda(i, j). (j, i))\ A$   
 by (auto intro!: inj-onI)

lemma swap-product:  
 $(\% (i, j). (j, i))\ ' (A \times B) = B \times A$   
 by (simp add: split-def image-def) blast

lemma image-split-eq-Sigma:  
 $(\lambda x. (f\ x, g\ x))\ ' A = Sigma\ (f\ ' A)\ (\lambda x. g\ ' (f\ -'\ \{x\} \cap A))$   
 proof (safe intro!: imageI vimageI)  
 fix a b assume \*:  $a \in A\ b \in A$  and eq:  $f\ a = f\ b$   
 show  $(f\ b, g\ a) \in (\lambda x. (f\ x, g\ x))\ ' A$   
 using \* eq[symmetric] by auto  
 qed simp-all

## 11.4 Inductively defined sets

use Tools/inductive-set.ML  
 setup Inductive-Set.setup

## 11.5 Legacy theorem bindings and duplicates

lemma PairE:  
 obtains  $x\ y$  where  $p = (x, y)$   
 by (fact prod.exhaust)

lemma Pair-inject:  
 assumes  $(a, b) = (a', b')$   
 and  $a = a' ==> b = b' ==> R$   
 shows  $R$   
 using assms by simp

lemmas Pair-eq = prod.inject

lemmas split = split-conv — for backwards compatibility

end

# 12 Sum-Type: The Disjoint Sum of Two Types

theory Sum-Type

**imports** *Typedef Inductive Fun*  
**begin**

## 12.1 Construction of the sum type and its basic abstract operations

**definition** *Inl-Rep* ::  $'a \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool} \Rightarrow \text{bool}$  **where**  
 $\text{Inl-Rep } a \ x \ y \ p \longleftrightarrow x = a \wedge p$

**definition** *Inr-Rep* ::  $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool} \Rightarrow \text{bool}$  **where**  
 $\text{Inr-Rep } b \ x \ y \ p \longleftrightarrow y = b \wedge \neg p$

**global**

**typedef** (*Sum*) ( $'a, 'b$ ) + (**infixr** + 10) =  $\{f. (\exists a. f = \text{Inl-Rep } (a::'a)) \vee (\exists b. f = \text{Inr-Rep } (b::'b))\}$   
**by** *auto*

**local**

**lemma** *Inl-RepI*:  $\text{Inl-Rep } a \in \text{Sum}$   
**by** (*auto simp add: Sum-def*)

**lemma** *Inr-RepI*:  $\text{Inr-Rep } b \in \text{Sum}$   
**by** (*auto simp add: Sum-def*)

**lemma** *inj-on-Abs-Sum*:  $A \subseteq \text{Sum} \implies \text{inj-on } \text{Abs-Sum } A$   
**by** (*rule inj-on-inverseI, rule Abs-Sum-inverse*) *auto*

**lemma** *Inl-Rep-inject*:  $\text{inj-on } \text{Inl-Rep } A$   
**proof** (*rule inj-onI*)  
**show**  $\bigwedge a \ c. \text{Inl-Rep } a = \text{Inl-Rep } c \implies a = c$   
**by** (*auto simp add: Inl-Rep-def expand-fun-eq*)  
**qed**

**lemma** *Inr-Rep-inject*:  $\text{inj-on } \text{Inr-Rep } A$   
**proof** (*rule inj-onI*)  
**show**  $\bigwedge b \ d. \text{Inr-Rep } b = \text{Inr-Rep } d \implies b = d$   
**by** (*auto simp add: Inr-Rep-def expand-fun-eq*)  
**qed**

**lemma** *Inl-Rep-not-Inr-Rep*:  $\text{Inl-Rep } a \neq \text{Inr-Rep } b$   
**by** (*auto simp add: Inl-Rep-def Inr-Rep-def expand-fun-eq*)

**definition** *Inl* ::  $'a \Rightarrow 'a + 'b$  **where**  
 $\text{Inl} = \text{Abs-Sum} \circ \text{Inl-Rep}$

**definition** *Inr* ::  $'b \Rightarrow 'a + 'b$  **where**  
 $\text{Inr} = \text{Abs-Sum} \circ \text{Inr-Rep}$

**lemma** *inj-Inl* [*simp*]: *inj-on Inl A*  
**by** (*auto simp add: Inl-def intro!: comp-inj-on Inl-Rep-inject inj-on-Abs-Sum Inl-RepI*)

**lemma** *Inl-inject*: *Inl x = Inl y  $\implies$  x = y*  
**using** *inj-Inl* **by** (*rule injD*)

**lemma** *inj-Inr* [*simp*]: *inj-on Inr A*  
**by** (*auto simp add: Inr-def intro!: comp-inj-on Inr-Rep-inject inj-on-Abs-Sum Inr-RepI*)

**lemma** *Inr-inject*: *Inr x = Inr y  $\implies$  x = y*  
**using** *inj-Inr* **by** (*rule injD*)

**lemma** *Inl-not-Inr*: *Inl a  $\neq$  Inr b*  
**proof** –  
**from** *Inl-RepI* [*of a*] *Inr-RepI* [*of b*] **have**  $\{Inl\text{-}Rep\ a, Inr\text{-}Rep\ b\} \subseteq Sum$  **by**  
*auto*  
**with** *inj-on-Abs-Sum* **have** *inj-on Abs-Sum*  $\{Inl\text{-}Rep\ a, Inr\text{-}Rep\ b\}$  .  
**with** *Inl-Rep-not-Inr-Rep* [*of a b*] *inj-on-contrad* **have** *Abs-Sum* (*Inl-Rep a*)  $\neq$   
*Abs-Sum* (*Inr-Rep b*) **by** *auto*  
**then show** *?thesis* **by** (*simp add: Inl-def Inr-def*)  
**qed**

**lemma** *Inr-not-Inl*: *Inr b  $\neq$  Inl a*  
**using** *Inl-not-Inr* **by** (*rule not-sym*)

**lemma** *sumE*:  
**assumes**  $\bigwedge x::'a. s = Inl\ x \implies P$   
**and**  $\bigwedge y::'b. s = Inr\ y \implies P$   
**shows** *P*  
**proof** (*rule Abs-Sum-cases* [*of s*])  
**fix** *f*  
**assume** *s = Abs-Sum f* **and** *f  $\in$  Sum*  
**with** *assms* **show** *P* **by** (*auto simp add: Sum-def Inl-def Inr-def*)  
**qed**

**rep-datatype** (*sum*) *Inl Inr*  
**proof** –  
**fix** *P*  
**fix** *s :: 'a + 'b*  
**assume** *x:  $\bigwedge x::'a. P$  (Inl x)* **and** *y:  $\bigwedge y::'b. P$  (Inr y)*  
**then show** *P s* **by** (*auto intro: sumE* [*of s*])  
**qed** (*auto dest: Inl-inject Inr-inject simp add: Inl-not-Inr*)

## 12.2 Projections

**lemma** *sum-case-KK* [*simp*]: *sum-case* ( $\lambda x. a$ ) ( $\lambda x. a$ ) = ( $\lambda x. a$ )  
**by** (*rule ext*) (*simp split: sum.split*)

**lemma** *surjective-sum*:  $\text{sum-case } (\lambda x::'a. f \text{ (Inl } x)) (\lambda y::'b. f \text{ (Inr } y)) = f$

**proof**

fix  $s :: 'a + 'b$

show  $(\text{case } s \text{ of Inl } (x::'a) \Rightarrow f \text{ (Inl } x) \mid \text{Inr } (y::'b) \Rightarrow f \text{ (Inr } y)) = f \text{ } s$

by  $(\text{cases } s) \text{ simp-all}$

**qed**

**lemma** *sum-case-inject*:

assumes  $a$ :  $\text{sum-case } f1 \text{ } f2 = \text{sum-case } g1 \text{ } g2$

assumes  $r$ :  $f1 = g1 \Longrightarrow f2 = g2 \Longrightarrow P$

shows  $P$

**proof** (*rule*  $r$ )

show  $f1 = g1$  **proof**

fix  $x :: 'a$

from  $a$  have  $\text{sum-case } f1 \text{ } f2 \text{ (Inl } x) = \text{sum-case } g1 \text{ } g2 \text{ (Inl } x)$  **by** *simp*

then show  $f1 \text{ } x = g1 \text{ } x$  **by** *simp*

**qed**

show  $f2 = g2$  **proof**

fix  $y :: 'b$

from  $a$  have  $\text{sum-case } f1 \text{ } f2 \text{ (Inr } y) = \text{sum-case } g1 \text{ } g2 \text{ (Inr } y)$  **by** *simp*

then show  $f2 \text{ } y = g2 \text{ } y$  **by** *simp*

**qed**

**qed**

**lemma** *sum-case-weak-cong*:

$s = t \Longrightarrow \text{sum-case } f \text{ } g \text{ } s = \text{sum-case } f \text{ } g \text{ } t$

— Prevents simplification of  $f$  and  $g$ : much faster.

**by** *simp*

**primrec** *Projl* ::  $'a + 'b \Rightarrow 'a$  **where**

*Projl-Inl*:  $\text{Projl } (\text{Inl } x) = x$

**primrec** *Projr* ::  $'a + 'b \Rightarrow 'b$  **where**

*Projr-Inr*:  $\text{Projr } (\text{Inr } x) = x$

**primrec** *Suml* ::  $('a \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$  **where**

*Suml*  $f \text{ (Inl } x) = f \text{ } x$

**primrec** *Sumr* ::  $('b \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$  **where**

*Sumr*  $f \text{ (Inr } x) = f \text{ } x$

**lemma** *Suml-inject*:

assumes  $\text{Suml } f = \text{Suml } g$  **shows**  $f = g$

**proof**

fix  $x :: 'a$

let  $?s = \text{Inl } x :: 'a + 'b$

from *assms* have  $\text{Suml } f \text{ } ?s = \text{Suml } g \text{ } ?s$  **by** *simp*

then show  $f \text{ } x = g \text{ } x$  **by** *simp*

**qed**

```

lemma Sumr-inject:
  assumes  $\text{Sumr } f = \text{Sumr } g$  shows  $f = g$ 
proof
  fix  $x :: 'b$ 
  let  $?s = \text{Inr } x :: 'a + 'b$ 
  from assms have  $\text{Sumr } f ?s = \text{Sumr } g ?s$  by simp
  then show  $f x = g x$  by simp
qed

```

### 12.3 The Disjoint Sum of Sets

**definition**  $\text{Plus} :: 'a \text{ set} \Rightarrow 'b \text{ set} \Rightarrow ('a + 'b) \text{ set}$  (**infixr**  $<+>$  65) **where**  
 $A <+> B = \text{Inl } ` A \cup \text{Inr } ` B$

**lemma** *InlI* [*intro!*]:  $a \in A \Longrightarrow \text{Inl } a \in A <+> B$   
**by** (*simp add: Plus-def*)

**lemma** *InrI* [*intro!*]:  $b \in B \Longrightarrow \text{Inr } b \in A <+> B$   
**by** (*simp add: Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

**lemma** *PlusE* [*elim!*]:  
 $u \in A <+> B \Longrightarrow (\bigwedge x. x \in A \Longrightarrow u = \text{Inl } x \Longrightarrow P) \Longrightarrow (\bigwedge y. y \in B \Longrightarrow u = \text{Inr } y \Longrightarrow P) \Longrightarrow P$   
**by** (*auto simp add: Plus-def*)

**lemma** *Plus-eq-empty-conv* [*simp*]:  $A <+> B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$   
**by** *auto*

**lemma** *UNIV-Plus-UNIV* [*simp*]:  $\text{UNIV} <+> \text{UNIV} = \text{UNIV}$   
**proof** (*rule set-ext*)  
**fix**  $u :: 'a + 'b$   
**show**  $u \in \text{UNIV} <+> \text{UNIV} \longleftrightarrow u \in \text{UNIV}$  **by** (*cases u*) *auto*  
**qed**

**hide-const** (**open**) *Suml Sumr Projl Projr*

**end**

## 13 Rings: Rings

**theory** *Rings*  
**imports** *Groups*  
**begin**

**class** *semiring* = *ab-semigroup-add* + *semigroup-mult* +  
**assumes** *left-distrib*[*algebra-simps*, *field-simps*]:  $(a + b) * c = a * c + b * c$

```

assumes right-distrib[algebra-simps, field-simps]:  $a * (b + c) = a * b + a * c$ 
begin

```

For the *combine-numerals* simproc

```

lemma combine-common-factor:
   $a * e + (b * e + c) = (a + b) * e + c$ 
by (simp add: left-distrib add-ac)

```

```

end

```

```

class mult-zero = times + zero +
  assumes mult-zero-left [simp]:  $0 * a = 0$ 
  assumes mult-zero-right [simp]:  $a * 0 = 0$ 

```

```

class semiring-0 = semiring + comm-monoid-add + mult-zero

```

```

class semiring-0-cancel = semiring + cancel-comm-monoid-add
begin

```

```

subclass semiring-0
proof
  fix  $a :: 'a$ 
  have  $0 * a + 0 * a = 0 * a + 0$  by (simp add: left-distrib [symmetric])
  thus  $0 * a = 0$  by (simp only: add-left-cancel)
next
  fix  $a :: 'a$ 
  have  $a * 0 + a * 0 = a * 0 + 0$  by (simp add: right-distrib [symmetric])
  thus  $a * 0 = 0$  by (simp only: add-left-cancel)
qed

```

```

end

```

```

class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
  assumes distrib:  $(a + b) * c = a * c + b * c$ 
begin

```

```

subclass semiring
proof
  fix  $a b c :: 'a$ 
  show  $(a + b) * c = a * c + b * c$  by (simp add: distrib)
  have  $a * (b + c) = (b + c) * a$  by (simp add: mult-ac)
  also have  $\dots = b * a + c * a$  by (simp only: distrib)
  also have  $\dots = a * b + a * c$  by (simp add: mult-ac)
  finally show  $a * (b + c) = a * b + a * c$  by blast
qed

```

```

end

```

```

class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero

```



```

begin

subclass semiring-0 ..

end

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..

subclass comm-semiring-0 ..

end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]: 0 ≠ 1
begin

lemma one-neq-zero [simp]: 1 ≠ 0
by (rule not-sym) (rule zero-neq-one)

end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult

Abstract divisibility

class dvd = times
begin

definition dvd :: 'a ⇒ 'a ⇒ bool (infixl dvd 50) where
  [code del]: b dvd a ⟷ (∃ k. a = b * k)

lemma dvdI [intro?]: a = b * k ⟹ b dvd a
  unfolding dvd-def ..

lemma dvdE [elim?]: b dvd a ⟹ (∧ k. a = b * k ⟹ P) ⟹ P
  unfolding dvd-def by blast

end

class comm-semiring-1 = zero-neq-one + comm-semiring-0 + comm-monoid-mult
+ dvd

begin

subclass semiring-1 ..

lemma dvd-refl[simp]: a dvd a

```

**proof**  
 show  $a = a * 1$  **by** *simp*  
**qed**

**lemma** *dvd-trans*:  
 assumes  $a \text{ dvd } b$  **and**  $b \text{ dvd } c$   
 shows  $a \text{ dvd } c$   
**proof** –  
 from *assms* **obtain**  $v$  **where**  $b = a * v$  **by** (*auto elim!*: *dvdE*)  
 moreover from *assms* **obtain**  $w$  **where**  $c = b * w$  **by** (*auto elim!*: *dvdE*)  
 ultimately have  $c = a * (v * w)$  **by** (*simp add*: *mult-assoc*)  
 then show *?thesis* ..  
**qed**

**lemma** *dvd-0-left-iff* [*no-atp, simp*]:  $0 \text{ dvd } a \longleftrightarrow a = 0$   
**by** (*auto intro*: *dvd-refl elim!*: *dvdE*)

**lemma** *dvd-0-right* [*iff*]:  $a \text{ dvd } 0$   
**proof**  
 show  $0 = a * 0$  **by** *simp*  
**qed**

**lemma** *one-dvd* [*simp*]:  $1 \text{ dvd } a$   
**by** (*auto intro!*: *dvdI*)

**lemma** *dvd-mult*[*simp*]:  $a \text{ dvd } c \implies a \text{ dvd } (b * c)$   
**by** (*auto intro!*: *mult-left-commute dvdI elim!*: *dvdE*)

**lemma** *dvd-mult2*[*simp*]:  $a \text{ dvd } b \implies a \text{ dvd } (b * c)$   
**apply** (*subst mult-commute*)  
**apply** (*erule dvd-mult*)  
**done**

**lemma** *dvd-triv-right* [*simp*]:  $a \text{ dvd } b * a$   
**by** (*rule dvd-mult*) (*rule dvd-refl*)

**lemma** *dvd-triv-left* [*simp*]:  $a \text{ dvd } a * b$   
**by** (*rule dvd-mult2*) (*rule dvd-refl*)

**lemma** *mult-dvd-mono*:  
 assumes  $a \text{ dvd } b$   
 and  $c \text{ dvd } d$   
 shows  $a * c \text{ dvd } b * d$   
**proof** –  
 from  $\langle a \text{ dvd } b \rangle$  **obtain**  $b'$  **where**  $b = a * b'$  ..  
 moreover from  $\langle c \text{ dvd } d \rangle$  **obtain**  $d'$  **where**  $d = c * d'$  ..  
 ultimately have  $b * d = (a * c) * (b' * d')$  **by** (*simp add*: *mult-ac*)  
 then show *?thesis* ..  
**qed**

**lemma** *dvd-mult-left*:  $a * b \text{ dvd } c \implies a \text{ dvd } c$

**by** (*simp add: dvd-def mult-assoc, blast*)

**lemma** *dvd-mult-right*:  $a * b \text{ dvd } c \implies b \text{ dvd } c$

**unfolding** *mult-ac* [of *a*] **by** (*rule dvd-mult-left*)

**lemma** *dvd-0-left*:  $0 \text{ dvd } a \implies a = 0$

**by** *simp*

**lemma** *dvd-add*[*simp*]:

**assumes**  $a \text{ dvd } b$  **and**  $a \text{ dvd } c$  **shows**  $a \text{ dvd } (b + c)$

**proof** –

**from**  $\langle a \text{ dvd } b \rangle$  **obtain**  $b'$  **where**  $b = a * b'$  ..

**moreover from**  $\langle a \text{ dvd } c \rangle$  **obtain**  $c'$  **where**  $c = a * c'$  ..

**ultimately have**  $b + c = a * (b' + c')$  **by** (*simp add: right-distrib*)

**then show** *?thesis* ..

**qed**

**end**

**class** *no-zero-divisors* = *zero* + *times* +

**assumes** *no-zero-divisors*:  $a \neq 0 \implies b \neq 0 \implies a * b \neq 0$

**begin**

**lemma** *divisors-zero*:

**assumes**  $a * b = 0$

**shows**  $a = 0 \vee b = 0$

**proof** (*rule classical*)

**assume**  $\neg (a = 0 \vee b = 0)$

**then have**  $a \neq 0$  **and**  $b \neq 0$  **by** *auto*

**with** *no-zero-divisors* **have**  $a * b \neq 0$  **by** *blast*

**with** *assms* **show** *?thesis* **by** *simp*

**qed**

**end**

**class** *semiring-1-cancel* = *semiring* + *cancel-comm-monoid-add*

+ *zero-neq-one* + *monoid-mult*

**begin**

**subclass** *semiring-0-cancel* ..

**subclass** *semiring-1* ..

**end**

**class** *comm-semiring-1-cancel* = *comm-semiring* + *cancel-comm-monoid-add*

+ *zero-neq-one* + *comm-monoid-mult*

**begin**

**subclass** *semiring-1-cancel* ..  
**subclass** *comm-semiring-0-cancel* ..  
**subclass** *comm-semiring-1* ..

**end**

**class** *ring* = *semiring* + *ab-group-add*  
**begin**

**subclass** *semiring-0-cancel* ..

Distribution rules

**lemma** *minus-mult-left*:  $-(a * b) = -a * b$   
**by** (*rule minus-unique*) (*simp add: left-distrib [symmetric]*)

**lemma** *minus-mult-right*:  $-(a * b) = a * -b$   
**by** (*rule minus-unique*) (*simp add: right-distrib [symmetric]*)

Extract signs from products

**lemmas** *mult-minus-left* [*simp, no-atp*] = *minus-mult-left* [*symmetric*]  
**lemmas** *mult-minus-right* [*simp, no-atp*] = *minus-mult-right* [*symmetric*]

**lemma** *minus-mult-minus* [*simp*]:  $-a * -b = a * b$   
**by** *simp*

**lemma** *minus-mult-commute*:  $-a * b = a * -b$   
**by** *simp*

**lemma** *right-diff-distrib*[*algebra-simps, field-simps*]:  $a * (b - c) = a * b - a * c$   
**by** (*simp add: right-distrib diff-minus*)

**lemma** *left-diff-distrib*[*algebra-simps, field-simps*]:  $(a - b) * c = a * c - b * c$   
**by** (*simp add: left-distrib diff-minus*)

**lemmas** *ring-distrib*[*no-atp*] =  
*right-distrib left-distrib left-diff-distrib right-diff-distrib*

**lemma** *eq-add-iff1*:  
 $a * e + c = b * e + d \iff (a - b) * e + c = d$   
**by** (*simp add: algebra-simps*)

**lemma** *eq-add-iff2*:  
 $a * e + c = b * e + d \iff c = (b - a) * e + d$   
**by** (*simp add: algebra-simps*)

**end**

```

lemmas ring-distrib[no-atp] =
  right-distrib left-distrib left-diff-distrib right-diff-distrib

class comm-ring = comm-semiring + ab-group-add
begin

subclass ring ..
subclass comm-semiring-0-cancel ..

end

class ring-1 = ring + zero-neq-one + monoid-mult
begin

subclass semiring-1-cancel ..

end

class comm-ring-1 = comm-ring + zero-neq-one + comm-monoid-mult
begin

subclass ring-1 ..
subclass comm-semiring-1-cancel ..

lemma dvd-minus-iff [simp]:  $x \text{ dvd } - y \longleftrightarrow x \text{ dvd } y$ 
proof
  assume  $x \text{ dvd } - y$ 
  then have  $x \text{ dvd } - 1 * - y$  by (rule dvd-mult)
  then show  $x \text{ dvd } y$  by simp
next
  assume  $x \text{ dvd } y$ 
  then have  $x \text{ dvd } - 1 * y$  by (rule dvd-mult)
  then show  $x \text{ dvd } - y$  by simp
qed

lemma minus-dvd-iff [simp]:  $- x \text{ dvd } y \longleftrightarrow x \text{ dvd } y$ 
proof
  assume  $- x \text{ dvd } y$ 
  then obtain  $k$  where  $y = - x * k$  ..
  then have  $y = x * - k$  by simp
  then show  $x \text{ dvd } y$  ..
next
  assume  $x \text{ dvd } y$ 
  then obtain  $k$  where  $y = x * k$  ..
  then have  $y = - x * - k$  by simp
  then show  $- x \text{ dvd } y$  ..
qed

```

**lemma** *dvd-diff* [*simp*]:  $x \text{ dvd } y \implies x \text{ dvd } z \implies x \text{ dvd } (y - z)$   
**by** (*simp only: diff-minus dvd-add dvd-minus-iff*)

**end**

**class** *ring-no-zero-divisors* = *ring* + *no-zero-divisors*  
**begin**

**lemma** *mult-eq-0-iff* [*simp*]:  
**shows**  $a * b = 0 \iff (a = 0 \vee b = 0)$   
**proof** (*cases a = 0 ∨ b = 0*)  
**case** *False* **then have**  $a \neq 0$  **and**  $b \neq 0$  **by** *auto*  
**then show** *?thesis* **using** *no-zero-divisors* **by** *simp*  
**next**  
**case** *True* **then show** *?thesis* **by** *auto*  
**qed**

Cancellation of equalities with a common factor

**lemma** *mult-cancel-right* [*simp, no-atp*]:  
 $a * c = b * c \iff c = 0 \vee a = b$   
**proof** –  
**have**  $(a * c = b * c) = ((a - b) * c = 0)$   
**by** (*simp add: algebra-simps*)  
**thus** *?thesis* **by** (*simp add: disj-commute*)  
**qed**

**lemma** *mult-cancel-left* [*simp, no-atp*]:  
 $c * a = c * b \iff c = 0 \vee a = b$   
**proof** –  
**have**  $(c * a = c * b) = (c * (a - b) = 0)$   
**by** (*simp add: algebra-simps*)  
**thus** *?thesis* **by** *simp*  
**qed**

**end**

**class** *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*  
**begin**

**lemma** *square-eq-1-iff*:  
 $x * x = 1 \iff x = 1 \vee x = -1$   
**proof** –  
**have**  $(x - 1) * (x + 1) = x * x - 1$   
**by** (*simp add: algebra-simps*)  
**hence**  $x * x = 1 \iff (x - 1) * (x + 1) = 0$   
**by** *simp*  
**thus** *?thesis*  
**by** (*simp add: eq-neg-iff-add-eq-0*)  
**qed**

```

lemma mult-cancel-right1 [simp]:
   $c = b * c \longleftrightarrow c = 0 \vee b = 1$ 
by (insert mult-cancel-right [of 1 c b], force)

lemma mult-cancel-right2 [simp]:
   $a * c = c \longleftrightarrow c = 0 \vee a = 1$ 
by (insert mult-cancel-right [of a c 1], simp)

lemma mult-cancel-left1 [simp]:
   $c = c * b \longleftrightarrow c = 0 \vee b = 1$ 
by (insert mult-cancel-left [of c 1 b], force)

lemma mult-cancel-left2 [simp]:
   $c * a = c \longleftrightarrow c = 0 \vee a = 1$ 
by (insert mult-cancel-left [of c a 1], simp)

end

class idom = comm-ring-1 + no-zero-divisors
begin

subclass ring-1-no-zero-divisors ..

lemma square-eq-iff:  $a * a = b * b \longleftrightarrow (a = b \vee a = - b)$ 
proof
  assume  $a * a = b * b$ 
  then have  $(a - b) * (a + b) = 0$ 
    by (simp add: algebra-simps)
  then show  $a = b \vee a = - b$ 
    by (simp add: eq-neg-iff-add-eq-0)
next
  assume  $a = b \vee a = - b$ 
  then show  $a * a = b * b$  by auto
qed

lemma dvd-mult-cancel-right [simp]:
   $a * c \text{ dvd } b * c \longleftrightarrow c = 0 \vee a \text{ dvd } b$ 
proof –
  have  $a * c \text{ dvd } b * c \longleftrightarrow (\exists k. b * c = (a * k) * c)$ 
    unfolding dvd-def by (simp add: mult-ac)
  also have  $(\exists k. b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \text{ dvd } b$ 
    unfolding dvd-def by simp
  finally show ?thesis .
qed

lemma dvd-mult-cancel-left [simp]:
   $c * a \text{ dvd } c * b \longleftrightarrow c = 0 \vee a \text{ dvd } b$ 
proof –

```

```

have  $c * a \text{ dvd } c * b \longleftrightarrow (\exists k. b * c = (a * k) * c)$ 
  unfolding dvd-def by (simp add: mult-ac)
also have  $(\exists k. b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \text{ dvd } b$ 
  unfolding dvd-def by simp
finally show ?thesis .
qed

```

end

```

class inverse =
  fixes inverse :: 'a  $\Rightarrow$  'a
  and divide :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl '/' 70)

class division-ring = ring-1 + inverse +
  assumes left-inverse [simp]:  $a \neq 0 \implies \text{inverse } a * a = 1$ 
  assumes right-inverse [simp]:  $a \neq 0 \implies a * \text{inverse } a = 1$ 
  assumes divide-inverse:  $a / b = a * \text{inverse } b$ 
begin

```

```

subclass ring-1-no-zero-divisors
proof
  fix a b :: 'a
  assume a:  $a \neq 0$  and b:  $b \neq 0$ 
  show  $a * b \neq 0$ 
  proof
    assume ab:  $a * b = 0$ 
    hence  $0 = \text{inverse } a * (a * b) * \text{inverse } b$  by simp
    also have  $\dots = (\text{inverse } a * a) * (b * \text{inverse } b)$ 
      by (simp only: mult-assoc)
    also have  $\dots = 1$  using a b by simp
    finally show False by simp
  qed
qed

```

```

lemma nonzero-imp-inverse-nonzero:
   $a \neq 0 \implies \text{inverse } a \neq 0$ 
proof
  assume ianz:  $\text{inverse } a = 0$ 
  assume a  $\neq 0$ 
  hence  $1 = a * \text{inverse } a$  by simp
  also have  $\dots = 0$  by (simp add: ianz)
  finally have  $1 = 0$  .
  thus False by (simp add: eq-commute)
qed

```

```

lemma inverse-zero-imp-zero:
   $\text{inverse } a = 0 \implies a = 0$ 
apply (rule classical)
apply (drule nonzero-imp-inverse-nonzero)

```



**apply** *auto*  
**done**

**lemma** *inverse-unique*:  
**assumes** *ab*:  $a * b = 1$   
**shows**  $\text{inverse } a = b$   
**proof** –  
**have**  $a \neq 0$  **using** *ab* **by** (*cases*  $a = 0$ ) *simp-all*  
**moreover** **have**  $\text{inverse } a * (a * b) = \text{inverse } a$  **by** (*simp add: ab*)  
**ultimately show** *?thesis* **by** (*simp add: mult-assoc [symmetric]*)  
**qed**

**lemma** *nonzero-inverse-minus-eq*:  
 $a \neq 0 \implies \text{inverse } (-a) = - \text{inverse } a$   
**by** (*rule inverse-unique*) *simp*

**lemma** *nonzero-inverse-inverse-eq*:  
 $a \neq 0 \implies \text{inverse } (\text{inverse } a) = a$   
**by** (*rule inverse-unique*) *simp*

**lemma** *nonzero-inverse-eq-imp-eq*:  
**assumes**  $\text{inverse } a = \text{inverse } b$  **and**  $a \neq 0$  **and**  $b \neq 0$   
**shows**  $a = b$   
**proof** –  
**from**  $\langle \text{inverse } a = \text{inverse } b \rangle$   
**have**  $\text{inverse } (\text{inverse } a) = \text{inverse } (\text{inverse } b)$  **by** (*rule arg-cong*)  
**with**  $\langle a \neq 0 \rangle$  **and**  $\langle b \neq 0 \rangle$  **show**  $a = b$   
**by** (*simp add: nonzero-inverse-inverse-eq*)  
**qed**

**lemma** *inverse-1* [*simp*]:  $\text{inverse } 1 = 1$   
**by** (*rule inverse-unique*) *simp*

**lemma** *nonzero-inverse-mult-distrib*:  
**assumes**  $a \neq 0$  **and**  $b \neq 0$   
**shows**  $\text{inverse } (a * b) = \text{inverse } b * \text{inverse } a$   
**proof** –  
**have**  $a * (b * \text{inverse } b) * \text{inverse } a = 1$  **using** *assms* **by** *simp*  
**hence**  $a * b * (\text{inverse } b * \text{inverse } a) = 1$  **by** (*simp only: mult-assoc*)  
**thus** *?thesis* **by** (*rule inverse-unique*)  
**qed**

**lemma** *division-ring-inverse-add*:  
 $a \neq 0 \implies b \neq 0 \implies \text{inverse } a + \text{inverse } b = \text{inverse } a * (a + b) * \text{inverse } b$   
**by** (*simp add: algebra-simps*)

**lemma** *division-ring-inverse-diff*:  
 $a \neq 0 \implies b \neq 0 \implies \text{inverse } a - \text{inverse } b = \text{inverse } a * (b - a) * \text{inverse } b$   
**by** (*simp add: algebra-simps*)

**lemma** *right-inverse-eq*:  $b \neq 0 \implies a / b = 1 \iff a = b$

**proof**

**assume** *neq*:  $b \neq 0$   
  {  
    **hence**  $a = (a / b) * b$  **by** (*simp add: divide-inverse mult-assoc*)  
    **also assume**  $a / b = 1$   
    **finally show**  $a = b$  **by** *simp*  
  **next**  
    **assume**  $a = b$   
    **with neq show**  $a / b = 1$  **by** (*simp add: divide-inverse*)  
  }  
**qed**

**lemma** *nonzero-inverse-eq-divide*:  $a \neq 0 \implies \text{inverse } a = 1 / a$   
**by** (*simp add: divide-inverse*)

**lemma** *divide-self* [*simp*]:  $a \neq 0 \implies a / a = 1$   
**by** (*simp add: divide-inverse*)

**lemma** *divide-zero-left* [*simp*]:  $0 / a = 0$   
**by** (*simp add: divide-inverse*)

**lemma** *inverse-eq-divide*:  $\text{inverse } a = 1 / a$   
**by** (*simp add: divide-inverse*)

**lemma** *add-divide-distrib*:  $(a+b) / c = a/c + b/c$   
**by** (*simp add: divide-inverse algebra-simps*)

**lemma** *divide-1* [*simp*]:  $a / 1 = a$   
**by** (*simp add: divide-inverse*)

**lemma** *times-divide-eq-right* [*simp*]:  $a * (b / c) = (a * b) / c$   
**by** (*simp add: divide-inverse mult-assoc*)

**lemma** *minus-divide-left*:  $-(a / b) = (-a) / b$   
**by** (*simp add: divide-inverse*)

**lemma** *nonzero-minus-divide-right*:  $b \neq 0 \implies -(a / b) = a / (-b)$   
**by** (*simp add: divide-inverse nonzero-inverse-minus-eq*)

**lemma** *nonzero-minus-divide-divide*:  $b \neq 0 \implies (-a) / (-b) = a / b$   
**by** (*simp add: divide-inverse nonzero-inverse-minus-eq*)

**lemma** *divide-minus-left* [*simp, no-atp*]:  $(-a) / b = -(a / b)$   
**by** (*simp add: divide-inverse*)

**lemma** *diff-divide-distrib*:  $(a - b) / c = a / c - b / c$   
**by** (*simp add: diff-minus add-divide-distrib*)

**lemma** *nonzero-eq-divide-eq* [*field-simps*]:  $c \neq 0 \implies a = b / c \iff a * c = b$

**proof** –

**assume** [*simp*]:  $c \neq 0$

**have**  $a = b / c \iff a * c = (b / c) * c$  **by** *simp*

**also have**  $\dots \iff a * c = b$  **by** (*simp add: divide-inverse mult-assoc*)

**finally show** *?thesis* .

**qed**

**lemma** *nonzero-divide-eq-eq* [*field-simps*]:  $c \neq 0 \implies b / c = a \iff b = a * c$

**proof** –

**assume** [*simp*]:  $c \neq 0$

**have**  $b / c = a \iff (b / c) * c = a * c$  **by** *simp*

**also have**  $\dots \iff b = a * c$  **by** (*simp add: divide-inverse mult-assoc*)

**finally show** *?thesis* .

**qed**

**lemma** *divide-eq-imp*:  $c \neq 0 \implies b = a * c \implies b / c = a$

**by** (*simp add: divide-inverse mult-assoc*)

**lemma** *eq-divide-imp*:  $c \neq 0 \implies a * c = b \implies a = b / c$

**by** (*drule sym*) (*simp add: divide-inverse mult-assoc*)

**end**

**class** *division-ring-inverse-zero* = *division-ring* +

**assumes** *inverse-zero* [*simp*]:  $\text{inverse } 0 = 0$

**begin**

**lemma** *divide-zero* [*simp*]:

$a / 0 = 0$

**by** (*simp add: divide-inverse*)

**lemma** *divide-self-if* [*simp*]:

$a / a = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$

**by** *simp*

**lemma** *inverse-nonzero-iff-nonzero* [*simp*]:

$\text{inverse } a = 0 \iff a = 0$

**by** *rule* (*fact inverse-zero-imp-zero, simp*)

**lemma** *inverse-minus-eq* [*simp*]:

$\text{inverse } (- a) = - \text{inverse } a$

**proof** *cases*

**assume**  $a=0$  **thus** *?thesis* **by** *simp*

**next**

**assume**  $a \neq 0$

**thus** *?thesis* **by** (*simp add: nonzero-inverse-minus-eq*)

**qed**

```

lemma inverse-eq-imp-eq:
  inverse a = inverse b  $\implies$  a = b
apply (cases a=0 | b=0)
  apply (force dest!: inverse-zero-imp-zero
    simp add: eq-commute [of 0::'a])
apply (force dest!: nonzero-inverse-eq-imp-eq)
done

lemma inverse-eq-iff-eq [simp]:
  inverse a = inverse b  $\iff$  a = b
  by (force dest!: inverse-eq-imp-eq)

lemma inverse-inverse-eq [simp]:
  inverse (inverse a) = a
proof cases
  assume a=0 thus ?thesis by simp
next
  assume a $\neq$ 0
  thus ?thesis by (simp add: nonzero-inverse-inverse-eq)
qed

end

class mult-mono = times + zero + ord +
  assumes mult-left-mono: a  $\leq$  b  $\implies$  0  $\leq$  c  $\implies$  c * a  $\leq$  c * b
  assumes mult-right-mono: a  $\leq$  b  $\implies$  0  $\leq$  c  $\implies$  a * c  $\leq$  b * c

```

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- <http://www.mathworld.com> by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

```

class ordered-semiring = mult-mono + semiring-0 + ordered-ab-semigroup-add
begin

lemma mult-mono:
  a  $\leq$  b  $\implies$  c  $\leq$  d  $\implies$  0  $\leq$  b  $\implies$  0  $\leq$  c
   $\implies$  a * c  $\leq$  b * d
apply (erule mult-right-mono [THEN order-trans], assumption)
apply (erule mult-left-mono, assumption)
done

```

```

lemma mult-mono':
   $a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c$ 
   $\implies a * c \leq b * d$ 
apply (rule mult-mono)
apply (fast intro: order-trans)+
done

end

class ordered-cancel-semiring = mult-mono + ordered-ab-semigroup-add
  + semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ..
subclass ordered-semiring ..

lemma mult-nonneg-nonneg:  $0 \leq a \implies 0 \leq b \implies 0 \leq a * b$ 
using mult-left-mono [of 0 b a] by simp

lemma mult-nonneg-nonpos:  $0 \leq a \implies b \leq 0 \implies a * b \leq 0$ 
using mult-left-mono [of b 0 a] by simp

lemma mult-nonpos-nonneg:  $a \leq 0 \implies 0 \leq b \implies a * b \leq 0$ 
using mult-right-mono [of a 0 b] by simp

Legacy - use mult-nonpos-nonneg

lemma mult-nonneg-nonpos2:  $0 \leq a \implies b \leq 0 \implies b * a \leq 0$ 
by (drule mult-right-mono [of b 0], auto)

lemma split-mult-neg-le:  $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \implies a * b \leq 0$ 
by (auto simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)

end

class linordered-semiring = semiring + comm-monoid-add + linordered-cancel-ab-semigroup-add
  + mult-mono
begin

subclass ordered-cancel-semiring ..

subclass ordered-comm-monoid-add ..

lemma mult-left-less-imp-less:
   $c * a < c * b \implies 0 \leq c \implies a < b$ 
by (force simp add: mult-left-mono not-le [symmetric])

lemma mult-right-less-imp-less:
   $a * c < b * c \implies 0 \leq c \implies a < b$ 

```

**by** (force simp add: mult-right-mono not-le [symmetric])

**end**

**class** linordered-semiring-1 = linordered-semiring + semiring-1  
**begin**

**lemma** convex-bound-le:

**assumes**  $x \leq a$   $y \leq a$   $0 \leq u$   $0 \leq v$   $u + v = 1$

**shows**  $u * x + v * y \leq a$

**proof**–

**from** *assms* **have**  $u * x + v * y \leq u * a + v * a$

**by** (simp add: add-mono mult-left-mono)

**thus** ?thesis **using** *assms* **unfolding** left-distrib[symmetric] **by** simp

**qed**

**end**

**class** linordered-semiring-strict = semiring + comm-monoid-add + linordered-cancel-ab-semigroup-add  
+

**assumes** mult-strict-left-mono:  $a < b \implies 0 < c \implies c * a < c * b$

**assumes** mult-strict-right-mono:  $a < b \implies 0 < c \implies a * c < b * c$

**begin**

**subclass** semiring-0-cancel ..

**subclass** linordered-semiring

**proof**

**fix**  $a\ b\ c :: 'a$

**assume**  $A: a \leq b$   $0 \leq c$

**from**  $A$  **show**  $c * a \leq c * b$

**unfolding** le-less

**using** mult-strict-left-mono **by** (cases  $c = 0$ ) auto

**from**  $A$  **show**  $a * c \leq b * c$

**unfolding** le-less

**using** mult-strict-right-mono **by** (cases  $c = 0$ ) auto

**qed**

**lemma** mult-left-le-imp-le:

$c * a \leq c * b \implies 0 < c \implies a \leq b$

**by** (force simp add: mult-strict-left-mono -not-less [symmetric])

**lemma** mult-right-le-imp-le:

$a * c \leq b * c \implies 0 < c \implies a \leq b$

**by** (force simp add: mult-strict-right-mono not-less [symmetric])

**lemma** mult-pos-pos:  $0 < a \implies 0 < b \implies 0 < a * b$

**using** mult-strict-left-mono [of  $0\ b\ a$ ] **by** simp

**lemma** *mult-pos-neg*:  $0 < a \implies b < 0 \implies a * b < 0$   
**using** *mult-strict-left-mono* [of  $b \ 0 \ a$ ] **by** *simp*

**lemma** *mult-neg-pos*:  $a < 0 \implies 0 < b \implies a * b < 0$   
**using** *mult-strict-right-mono* [of  $a \ 0 \ b$ ] **by** *simp*

Legacy - use *mult-neg-pos*

**lemma** *mult-pos-neg2*:  $0 < a \implies b < 0 \implies b * a < 0$   
**by** (*drule mult-strict-right-mono* [of  $b \ 0$ ], *auto*)

**lemma** *zero-less-mult-pos*:  
 $0 < a * b \implies 0 < a \implies 0 < b$   
**apply** (*cases b ≤ 0*)  
**apply** (*auto simp add: le-less not-less*)  
**apply** (*drule-tac mult-pos-neg* [of  $a \ b$ ])  
**apply** (*auto dest: less-not-sym*)  
**done**

**lemma** *zero-less-mult-pos2*:  
 $0 < b * a \implies 0 < a \implies 0 < b$   
**apply** (*cases b ≤ 0*)  
**apply** (*auto simp add: le-less not-less*)  
**apply** (*drule-tac mult-pos-neg2* [of  $a \ b$ ])  
**apply** (*auto dest: less-not-sym*)  
**done**

Strict monotonicity in both arguments

**lemma** *mult-strict-mono*:  
**assumes**  $a < b$  **and**  $c < d$  **and**  $0 < b$  **and**  $0 \leq c$   
**shows**  $a * c < b * d$   
**using** *assms* **apply** (*cases c=0*)  
**apply** (*simp add: mult-pos-pos*)  
**apply** (*erule mult-strict-right-mono* [THEN *less-trans*])  
**apply** (*force simp add: le-less*)  
**apply** (*erule mult-strict-left-mono, assumption*)  
**done**

This weaker variant has more natural premises

**lemma** *mult-strict-mono'*:  
**assumes**  $a < b$  **and**  $c < d$  **and**  $0 \leq a$  **and**  $0 \leq c$   
**shows**  $a * c < b * d$   
**by** (*rule mult-strict-mono*) (*insert assms, auto*)

**lemma** *mult-less-le-imp-less*:  
**assumes**  $a < b$  **and**  $c \leq d$  **and**  $0 \leq a$  **and**  $0 < c$   
**shows**  $a * c < b * d$   
**using** *assms* **apply** (*subgoal-tac a \* c < b \* c*)  
**apply** (*erule less-le-trans*)  
**apply** (*erule mult-left-mono*)

```

apply simp
apply (erule mult-strict-right-mono)
apply assumption
done

lemma mult-le-less-imp-less:
  assumes  $a \leq b$  and  $c < d$  and  $0 < a$  and  $0 \leq c$ 
  shows  $a * c < b * d$ 
  using assms apply (subgoal-tac  $a * c \leq b * c$ )
  apply (erule le-less-trans)
  apply (erule mult-strict-left-mono)
  apply simp
  apply (erule mult-right-mono)
  apply simp
  done

lemma mult-less-imp-less-left:
  assumes less:  $c * a < c * b$  and nonneg:  $0 \leq c$ 
  shows  $a < b$ 
proof (rule ccontr)
  assume  $\neg a < b$ 
  hence  $b \leq a$  by (simp add: linorder-not-less)
  hence  $c * b \leq c * a$  using nonneg by (rule mult-left-mono)
  with this and less show False by (simp add: not-less [symmetric])
qed

lemma mult-less-imp-less-right:
  assumes less:  $a * c < b * c$  and nonneg:  $0 \leq c$ 
  shows  $a < b$ 
proof (rule ccontr)
  assume  $\neg a < b$ 
  hence  $b \leq a$  by (simp add: linorder-not-less)
  hence  $b * c \leq a * c$  using nonneg by (rule mult-right-mono)
  with this and less show False by (simp add: not-less [symmetric])
qed

end

class linordered-semiring-1-strict = linordered-semiring-strict + semiring-1
begin

subclass linordered-semiring-1 ..

lemma convex-bound-lt:
  assumes  $x < a$   $y < a$   $0 \leq u$   $0 \leq v$   $u + v = 1$ 
  shows  $u * x + v * y < a$ 
proof –
  from assms have  $u * x + v * y < u * a + v * a$ 
  by (cases  $u = 0$ )

```



```

      (auto intro!: add-less-le-mono mult-strict-left-mono mult-left-mono)
    thus ?thesis using assms unfolding left-distrib[symmetric] by simp
qed

end

class mult-mono1 = times + zero + ord +
  assumes mult-mono1:  $a \leq b \implies 0 \leq c \implies c * a \leq c * b$ 

class ordered-comm-semiring = comm-semiring-0
  + ordered-ab-semigroup-add + mult-mono1
begin

subclass ordered-semiring
proof
  fix a b c :: 'a
  assume  $a \leq b$   $0 \leq c$ 
  thus  $c * a \leq c * b$  by (rule mult-mono1)
  thus  $a * c \leq b * c$  by (simp only: mult-commute)
qed

end

class ordered-cancel-comm-semiring = comm-semiring-0-cancel
  + ordered-ab-semigroup-add + mult-mono1
begin

subclass ordered-comm-semiring ..
subclass ordered-cancel-semiring ..

end

class linordered-comm-semiring-strict = comm-semiring-0 + linordered-cancel-ab-semigroup-add
+
  assumes mult-strict-left-mono-comm:  $a < b \implies 0 < c \implies c * a < c * b$ 
begin

subclass linordered-semiring-strict
proof
  fix a b c :: 'a
  assume  $a < b$   $0 < c$ 
  thus  $c * a < c * b$  by (rule mult-strict-left-mono-comm)
  thus  $a * c < b * c$  by (simp only: mult-commute)
qed

subclass ordered-cancel-comm-semiring
proof
  fix a b c :: 'a
  assume  $a \leq b$   $0 \leq c$ 

```

```

    thus  $c * a \leq c * b$ 
      unfolding le-less
      using mult-strict-left-mono by (cases  $c = 0$ ) auto
qed

end

class ordered-ring = ring + ordered-cancel-semiring
begin

subclass ordered-ab-group-add ..

lemma less-add-iff1:
   $a * e + c < b * e + d \iff (a - b) * e + c < d$ 
  by (simp add: algebra-simps)

lemma less-add-iff2:
   $a * e + c < b * e + d \iff c < (b - a) * e + d$ 
  by (simp add: algebra-simps)

lemma le-add-iff1:
   $a * e + c \leq b * e + d \iff (a - b) * e + c \leq d$ 
  by (simp add: algebra-simps)

lemma le-add-iff2:
   $a * e + c \leq b * e + d \iff c \leq (b - a) * e + d$ 
  by (simp add: algebra-simps)

lemma mult-left-mono-neg:
   $b \leq a \implies c \leq 0 \implies c * a \leq c * b$ 
  apply (drule mult-left-mono [of - - - c])
  apply simp-all
  done

lemma mult-right-mono-neg:
   $b \leq a \implies c \leq 0 \implies a * c \leq b * c$ 
  apply (drule mult-right-mono [of - - - c])
  apply simp-all
  done

lemma mult-nonpos-nonpos:  $a \leq 0 \implies b \leq 0 \implies 0 \leq a * b$ 
  using mult-right-mono-neg [of a 0 b] by simp

lemma split-mult-pos-le:
   $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \implies 0 \leq a * b$ 
  by (auto simp add: mult-nonneg-nonneg mult-nonpos-nonpos)

end

```

```

class linordered-ring = ring + linordered-semiring + linordered-ab-group-add +
abs-if
begin

subclass ordered-ring ..

subclass ordered-ab-group-add-abs
proof
  fix a b
  show  $|a + b| \leq |a| + |b|$ 
  by (auto simp add: abs-if not-less)
  (auto simp del: minus-add-distrib simp add: minus-add-distrib [symmetric],
    auto intro!: less-imp-le add-neg-neg)
qed (auto simp add: abs-if)

lemma zero-le-square [simp]:  $0 \leq a * a$ 
  using linear [of 0 a]
  by (auto simp add: mult-nonneg-nonneg mult-nonpos-nonpos)

lemma not-square-less-zero [simp]:  $\neg (a * a < 0)$ 
  by (simp add: not-less)

end

class linordered-ring-strict = ring + linordered-semiring-strict
+ ordered-ab-group-add + abs-if
begin

subclass linordered-ring ..

lemma mult-strict-left-mono-neg:  $b < a \implies c < 0 \implies c * a < c * b$ 
  using mult-strict-left-mono [of b a - c] by simp

lemma mult-strict-right-mono-neg:  $b < a \implies c < 0 \implies a * c < b * c$ 
  using mult-strict-right-mono [of b a - c] by simp

lemma mult-neg-neg:  $a < 0 \implies b < 0 \implies 0 < a * b$ 
  using mult-strict-right-mono-neg [of a 0 b] by simp

subclass ring-no-zero-divisors
proof
  fix a b
  assume  $a \neq 0$  then have  $A: a < 0 \vee 0 < a$  by (simp add: neq-iff)
  assume  $b \neq 0$  then have  $B: b < 0 \vee 0 < b$  by (simp add: neq-iff)
  have  $a * b < 0 \vee 0 < a * b$ 
  proof (cases  $a < 0$ )
    case True note  $A' = this$ 
    show ?thesis proof (cases  $b < 0$ )

```

```

    case True with A'
    show ?thesis by (auto dest: mult-neg-neg)
next
    case False with B have 0 < b by auto
    with A' show ?thesis by (auto dest: mult-strict-right-mono)
qed
next
    case False with A have A': 0 < a by auto
    show ?thesis proof (cases b < 0)
      case True with A'
      show ?thesis by (auto dest: mult-strict-right-mono-neg)
    next
      case False with B have 0 < b by auto
      with A' show ?thesis by (auto dest: mult-pos-pos)
    qed
  qed
then show a * b ≠ 0 by (simp add: neq-iff)
qed

lemma zero-less-mult-iff:
  0 < a * b ⟷ 0 < a ∧ 0 < b ∨ a < 0 ∧ b < 0
  apply (auto simp add: mult-pos-pos mult-neg-neg)
  apply (simp-all add: not-less le-less)
  apply (erule disjE) apply assumption defer
  apply (erule disjE) defer apply (drule sym) apply simp
  apply (erule disjE) defer apply (drule sym) apply simp
  apply (erule disjE) apply assumption apply (drule sym) apply simp
  apply (drule sym) apply simp
  apply (blast dest: zero-less-mult-pos)
  apply (blast dest: zero-less-mult-pos2)
done

lemma zero-le-mult-iff:
  0 ≤ a * b ⟷ 0 ≤ a ∧ 0 ≤ b ∨ a ≤ 0 ∧ b ≤ 0
  by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

lemma mult-less-0-iff:
  a * b < 0 ⟷ 0 < a ∧ b < 0 ∨ a < 0 ∧ 0 < b
  apply (insert zero-less-mult-iff [of -a b])
  apply force
done

lemma mult-le-0-iff:
  a * b ≤ 0 ⟷ 0 ≤ a ∧ b ≤ 0 ∨ a ≤ 0 ∧ 0 ≤ b
  apply (insert zero-le-mult-iff [of -a b])
  apply force
done

```

Cancellation laws for  $c * a < c * b$  and  $a * c < b * c$ , also with the relations

$\leq$  and equality.

These “disjunction” versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

**lemma** *mult-less-cancel-right-disj*:  
 $a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$   
**apply** (cases  $c = 0$ )  
**apply** (auto simp add: neq-iff mult-strict-right-mono  
mult-strict-right-mono-neg)  
**apply** (auto simp add: not-less  
not-le [symmetric, of  $a*c$ ]  
not-le [symmetric, of  $a$ ])  
**apply** (erule-tac [!] notE)  
**apply** (auto simp add: less-imp-le mult-right-mono  
mult-right-mono-neg)  
**done**

**lemma** *mult-less-cancel-left-disj*:  
 $c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$   
**apply** (cases  $c = 0$ )  
**apply** (auto simp add: neq-iff mult-strict-left-mono  
mult-strict-left-mono-neg)  
**apply** (auto simp add: not-less  
not-le [symmetric, of  $c*a$ ]  
not-le [symmetric, of  $a$ ])  
**apply** (erule-tac [!] notE)  
**apply** (auto simp add: less-imp-le mult-left-mono  
mult-left-mono-neg)  
**done**

The “conjunction of implication” lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

**lemma** *mult-less-cancel-right*:  
 $a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$   
**using** *mult-less-cancel-right-disj* [of  $a\ c\ b$ ] **by** auto

**lemma** *mult-less-cancel-left*:  
 $c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$   
**using** *mult-less-cancel-left-disj* [of  $c\ a\ b$ ] **by** auto

**lemma** *mult-le-cancel-right*:  
 $a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$   
**by** (simp add: not-less [symmetric] *mult-less-cancel-right-disj*)

**lemma** *mult-le-cancel-left*:  
 $c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$   
**by** (simp add: not-less [symmetric] *mult-less-cancel-left-disj*)

**lemma** *mult-le-cancel-left-pos*:

$0 < c \implies c * a \leq c * b \longleftrightarrow a \leq b$   
**by** (*auto simp: mult-le-cancel-left*)

**lemma** *mult-le-cancel-left-neg*:  
 $c < 0 \implies c * a \leq c * b \longleftrightarrow b \leq a$   
**by** (*auto simp: mult-le-cancel-left*)

**lemma** *mult-less-cancel-left-pos*:  
 $0 < c \implies c * a < c * b \longleftrightarrow a < b$   
**by** (*auto simp: mult-less-cancel-left*)

**lemma** *mult-less-cancel-left-neg*:  
 $c < 0 \implies c * a < c * b \longleftrightarrow b < a$   
**by** (*auto simp: mult-less-cancel-left*)

**end**

**lemmas** *mult-sign-intros* =  
*mult-nonneg-nonneg mult-nonneg-nonpos*  
*mult-nonpos-nonneg mult-nonpos-nonpos*  
*mult-pos-pos mult-pos-neg*  
*mult-neg-pos mult-neg-neg*

**class** *ordered-comm-ring* = *comm-ring* + *ordered-comm-semiring*  
**begin**

**subclass** *ordered-ring* ..  
**subclass** *ordered-cancel-comm-semiring* ..

**end**

**class** *linordered-semidom* = *comm-semiring-1-cancel* + *linordered-comm-semiring-strict*  
 +

**assumes** *zero-less-one* [*simp*]:  $0 < 1$   
**begin**

**lemma** *pos-add-strict*:  
**shows**  $0 < a \implies b < c \implies b < a + c$   
**using** *add-strict-mono* [*of 0 a b c*] **by** *simp*

**lemma** *zero-le-one* [*simp*]:  $0 \leq 1$   
**by** (*rule zero-less-one [THEN less-imp-le]*)

**lemma** *not-one-le-zero* [*simp*]:  $\neg 1 \leq 0$   
**by** (*simp add: not-le*)

**lemma** *not-one-less-zero* [*simp*]:  $\neg 1 < 0$   
**by** (*simp add: not-less*)

```

lemma less-1-mult:
  assumes  $1 < m$  and  $1 < n$ 
  shows  $1 < m * n$ 
  using assms mult-strict-mono [of 1 m 1 n]
  by (simp add: less-trans [OF zero-less-one])

end

class linordered-idom = comm-ring-1 +
  linordered-comm-semiring-strict + ordered-ab-group-add +
  abs-if + sgn-if

begin

subclass linordered-semiring-1-strict ..
subclass linordered-ring-strict ..
subclass ordered-comm-ring ..
subclass idom ..

subclass linordered-semidom
proof
  have  $0 \leq 1 * 1$  by (rule zero-le-square)
  thus  $0 < 1$  by (simp add: le-less)
qed

lemma linorder-neqE-linordered-idom:
  assumes  $x \neq y$  obtains  $x < y \mid y < x$ 
  using assms by (rule neqE)

These cancellation simprules also produce two cases when the comparison
is a goal.

lemma mult-le-cancel-right1:
   $c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$ 
by (insert mult-le-cancel-right [of 1 c b], simp)

lemma mult-le-cancel-right2:
   $a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$ 
by (insert mult-le-cancel-right [of a c 1], simp)

lemma mult-le-cancel-left1:
   $c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$ 
by (insert mult-le-cancel-left [of c 1 b], simp)

lemma mult-le-cancel-left2:
   $c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$ 
by (insert mult-le-cancel-left [of c a 1], simp)

lemma mult-less-cancel-right1:

```

$c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$   
**by** (*insert mult-less-cancel-right [of 1 c b], simp*)

**lemma** *mult-less-cancel-right2*:  
 $a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$   
**by** (*insert mult-less-cancel-right [of a c 1], simp*)

**lemma** *mult-less-cancel-left1*:  
 $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$   
**by** (*insert mult-less-cancel-left [of c 1 b], simp*)

**lemma** *mult-less-cancel-left2*:  
 $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$   
**by** (*insert mult-less-cancel-left [of c a 1], simp*)

**lemma** *sgn-sgn* [*simp*]:  
 $\text{sgn } (\text{sgn } a) = \text{sgn } a$   
**unfolding** *sgn-if* **by** *simp*

**lemma** *sgn-0-0*:  
 $\text{sgn } a = 0 \longleftrightarrow a = 0$   
**unfolding** *sgn-if* **by** *simp*

**lemma** *sgn-1-pos*:  
 $\text{sgn } a = 1 \longleftrightarrow a > 0$   
**unfolding** *sgn-if* **by** *simp*

**lemma** *sgn-1-neg*:  
 $\text{sgn } a = -1 \longleftrightarrow a < 0$   
**unfolding** *sgn-if* **by** *auto*

**lemma** *sgn-pos* [*simp*]:  
 $0 < a \implies \text{sgn } a = 1$   
**unfolding** *sgn-1-pos* .

**lemma** *sgn-neg* [*simp*]:  
 $a < 0 \implies \text{sgn } a = -1$   
**unfolding** *sgn-1-neg* .

**lemma** *sgn-times*:  
 $\text{sgn } (a * b) = \text{sgn } a * \text{sgn } b$   
**by** (*auto simp add: sgn-if zero-less-mult-iff*)

**lemma** *abs-sgn*:  $|k| = k * \text{sgn } k$   
**unfolding** *sgn-if abs-if* **by** *auto*

**lemma** *sgn-greater* [*simp*]:  
 $0 < \text{sgn } a \longleftrightarrow 0 < a$   
**unfolding** *sgn-if* **by** *auto*



```

lemma sgn-less [simp]:
  sgn a < 0  $\longleftrightarrow$  a < 0
  unfolding sgn-if by auto

lemma abs-dvd-iff [simp]: |m| dvd k  $\longleftrightarrow$  m dvd k
  by (simp add: abs-if)

lemma dvd-abs-iff [simp]: m dvd |k|  $\longleftrightarrow$  m dvd k
  by (simp add: abs-if)

lemma dvd-if-abs-eq:
  |l| = |k|  $\implies$  l dvd k
by(subst abs-dvd-iff[symmetric]) simp

end

```

Simprules for comparisons where common factors can be cancelled.

```

lemmas mult-compare-simps[no-atp] =
  mult-le-cancel-right mult-le-cancel-left
  mult-le-cancel-right1 mult-le-cancel-right2
  mult-le-cancel-left1 mult-le-cancel-left2
  mult-less-cancel-right mult-less-cancel-left
  mult-less-cancel-right1 mult-less-cancel-right2
  mult-less-cancel-left1 mult-less-cancel-left2
  mult-cancel-right mult-cancel-left
  mult-cancel-right1 mult-cancel-right2
  mult-cancel-left1 mult-cancel-left2

```

Reasoning about inequalities with division

```

context linordered-semidom
begin

lemma less-add-one: a < a + 1
proof –
  have a + 0 < a + 1
    by (blast intro: zero-less-one add-strict-left-mono)
  thus ?thesis by simp
qed

lemma zero-less-two: 0 < 1 + 1
by (blast intro: less-trans zero-less-one less-add-one)

end

context linordered-idom
begin

lemma mult-right-le-one-le:

```

```

 $0 \leq x \implies 0 \leq y \implies y \leq 1 \implies x * y \leq x$ 
by (auto simp add: mult-le-cancel-left2)

lemma mult-left-le-one-le:
 $0 \leq x \implies 0 \leq y \implies y \leq 1 \implies y * x \leq x$ 
by (auto simp add: mult-le-cancel-right2)

end

Absolute Value

context linordered-idom
begin

lemma mult-sgn-abs:
 $\text{sgn } x * |x| = x$ 
unfolding abs-if sgn-if by auto

lemma abs-one [simp]:
 $|1| = 1$ 
by (simp add: abs-if zero-less-one [THEN less-not-sym])

end

class ordered-ring-abs = ordered-ring + ordered-ab-group-add-abs +
assumes abs-eq-mult:
 $(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \implies |a * b| = |a| * |b|$ 

context linordered-idom
begin

subclass ordered-ring-abs proof
qed (auto simp add: abs-if not-less mult-less-0-iff)

lemma abs-mult:
 $|a * b| = |a| * |b|$ 
by (rule abs-eq-mult) auto

lemma abs-mult-self:
 $|a| * |a| = a * a$ 
by (simp add: abs-if)

lemma abs-mult-less:
 $|a| < c \implies |b| < d \implies |a| * |b| < c * d$ 
proof –
assume ac:  $|a| < c$ 
hence cpos:  $0 < c$  by (blast intro: le-less-trans abs-ge-zero)
assume  $|b| < d$ 
thus ?thesis by (simp add: ac cpos mult-strict-mono)
qed

```

```

lemma less-minus-self-iff:
   $a < - a \iff a < 0$ 
  by (simp only: less-le less-eq-neg-nonpos equal-neg-zero)

```

```

lemma abs-less-iff:
   $|a| < b \iff a < b \wedge - a < b$ 
  by (simp add: less-le abs-le-iff) (auto simp add: abs-if)

```

```

lemma abs-mult-pos:
   $0 \leq x \implies |y| * x = |y * x|$ 
  by (simp add: abs-mult)

```

```

end

```

```

code-modulename SML
  Rings Arith

```

```

code-modulename OCaml
  Rings Arith

```

```

code-modulename Haskell
  Rings Arith

```

```

end

```

## 14 Fields: Fields

```

theory Fields
imports Rings
begin

class field = comm-ring-1 + inverse +
  assumes field-inverse:  $a \neq 0 \implies \text{inverse } a * a = 1$ 
  assumes field-divide-inverse:  $a / b = a * \text{inverse } b$ 
begin

subclass division-ring
proof
  fix  $a :: 'a$ 
  assume  $a \neq 0$ 
  thus  $\text{inverse } a * a = 1$  by (rule field-inverse)
  thus  $a * \text{inverse } a = 1$  by (simp only: mult-commute)
next
  fix  $a b :: 'a$ 
  show  $a / b = a * \text{inverse } b$  by (rule field-divide-inverse)
qed

```

**subclass** *idom* ..

There is no slick version using division by zero.

**lemma** *inverse-add*:

$\llbracket a \neq 0; b \neq 0 \rrbracket$

$\implies \text{inverse } a + \text{inverse } b = (a + b) * \text{inverse } a * \text{inverse } b$

**by** (*simp add: division-ring-inverse-add mult-ac*)

**lemma** *nonzero-mult-divide-mult-cancel-left* [*simp, no-atp*]:

**assumes** [*simp*]:  $b \neq 0$  **and** [*simp*]:  $c \neq 0$  **shows**  $(c*a)/(c*b) = a/b$

**proof** –

**have**  $(c*a)/(c*b) = c * a * (\text{inverse } b * \text{inverse } c)$

**by** (*simp add: divide-inverse nonzero-inverse-mult-distrib*)

**also have**  $\dots = a * \text{inverse } b * (\text{inverse } c * c)$

**by** (*simp only: mult-ac*)

**also have**  $\dots = a * \text{inverse } b$  **by** *simp*

**finally show** *?thesis* **by** (*simp add: divide-inverse*)

**qed**

**lemma** *nonzero-mult-divide-mult-cancel-right* [*simp, no-atp*]:

$\llbracket b \neq 0; c \neq 0 \rrbracket \implies (a * c) / (b * c) = a / b$

**by** (*simp add: mult-commute [of - c]*)

**lemma** *times-divide-eq-left* [*simp*]:  $(b / c) * a = (b * a) / c$

**by** (*simp add: divide-inverse mult-ac*)

These are later declared as simp rules.

**lemmas** *times-divide-eq* [*no-atp*] = *times-divide-eq-right times-divide-eq-left*

**lemma** *add-frac-eq*:

**assumes**  $y \neq 0$  **and**  $z \neq 0$

**shows**  $x / y + w / z = (x * z + w * y) / (y * z)$

**proof** –

**have**  $x / y + w / z = (x * z) / (y * z) + (y * w) / (y * z)$

**using** *assms* **by** *simp*

**also have**  $\dots = (x * z + y * w) / (y * z)$

**by** (*simp only: add-divide-distrib*)

**finally show** *?thesis*

**by** (*simp only: mult-commute*)

**qed**

Special Cancellation Simprules for Division

**lemma** *nonzero-mult-divide-cancel-right* [*simp, no-atp*]:

$b \neq 0 \implies a * b / b = a$

**using** *nonzero-mult-divide-mult-cancel-right [of 1 b a]* **by** *simp*

**lemma** *nonzero-mult-divide-cancel-left* [*simp, no-atp*]:

$a \neq 0 \implies a * b / a = b$

**using** *nonzero-mult-divide-mult-cancel-left [of 1 a b]* **by** *simp*

**lemma** *nonzero-divide-mult-cancel-right* [*simp, no-atp*]:

$$\llbracket a \neq 0; b \neq 0 \rrbracket \Longrightarrow b / (a * b) = 1 / a$$

**using** *nonzero-mult-divide-mult-cancel-right* [*of a b 1*] **by** *simp*

**lemma** *nonzero-divide-mult-cancel-left* [*simp, no-atp*]:

$$\llbracket a \neq 0; b \neq 0 \rrbracket \Longrightarrow a / (a * b) = 1 / b$$

**using** *nonzero-mult-divide-mult-cancel-left* [*of b a 1*] **by** *simp*

**lemma** *nonzero-mult-divide-mult-cancel-left2* [*simp, no-atp*]:

$$\llbracket b \neq 0; c \neq 0 \rrbracket \Longrightarrow (c * a) / (b * c) = a / b$$

**using** *nonzero-mult-divide-mult-cancel-left* [*of b c a*] **by** (*simp add: mult-ac*)

**lemma** *nonzero-mult-divide-mult-cancel-right2* [*simp, no-atp*]:

$$\llbracket b \neq 0; c \neq 0 \rrbracket \Longrightarrow (a * c) / (c * b) = a / b$$

**using** *nonzero-mult-divide-mult-cancel-right* [*of b c a*] **by** (*simp add: mult-ac*)

**lemma** *add-divide-eq-iff* [*field-simps*]:

$$z \neq 0 \Longrightarrow x + y / z = (z * x + y) / z$$

**by** (*simp add: add-divide-distrib*)

**lemma** *divide-add-eq-iff* [*field-simps*]:

$$z \neq 0 \Longrightarrow x / z + y = (x + z * y) / z$$

**by** (*simp add: add-divide-distrib*)

**lemma** *diff-divide-eq-iff* [*field-simps*]:

$$z \neq 0 \Longrightarrow x - y / z = (z * x - y) / z$$

**by** (*simp add: diff-divide-distrib*)

**lemma** *divide-diff-eq-iff* [*field-simps*]:

$$z \neq 0 \Longrightarrow x / z - y = (x - z * y) / z$$

**by** (*simp add: diff-divide-distrib*)

**lemma** *diff-frac-eq*:

$$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x / y - w / z = (x * z - w * y) / (y * z)$$

**by** (*simp add: field-simps*)

**lemma** *frac-eq-eq*:

$$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x / y = w / z) = (x * z = w * y)$$

**by** (*simp add: field-simps*)

**end**

**class** *field-inverse-zero* = *field* +

**assumes** *field-inverse-zero*: *inverse 0 = 0*

**begin**

**subclass** *division-ring-inverse-zero* **proof**

**qed** (*fact field-inverse-zero*)

This version builds in division by zero while also re-orienting the right-hand side.

```

lemma inverse-mult-distrib [simp]:
  inverse (a * b) = inverse a * inverse b
proof cases
  assume a ≠ 0 & b ≠ 0
  thus ?thesis by (simp add: nonzero-inverse-mult-distrib mult-ac)
next
  assume ~ (a ≠ 0 & b ≠ 0)
  thus ?thesis by force
qed

```

```

lemma inverse-divide [simp]:
  inverse (a / b) = b / a
  by (simp add: divide-inverse mult-commute)

```

Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

```

lemma mult-divide-mult-cancel-left:
  c ≠ 0 ⇒ (c * a) / (c * b) = a / b
apply (cases b = 0)
apply simp-all
done

```

```

lemma mult-divide-mult-cancel-right:
  c ≠ 0 ⇒ (a * c) / (b * c) = a / b
apply (cases b = 0)
apply simp-all
done

```

```

lemma divide-divide-eq-right [simp, no-atp]:
  a / (b / c) = (a * c) / b
  by (simp add: divide-inverse mult-ac)

```

```

lemma divide-divide-eq-left [simp, no-atp]:
  (a / b) / c = a / (b * c)
  by (simp add: divide-inverse mult-assoc)

```

Special Cancellation Simprules for Division

```

lemma mult-divide-mult-cancel-left-if [simp, no-atp]:
  shows (c * a) / (c * b) = (if c = 0 then 0 else a / b)
  by (simp add: mult-divide-mult-cancel-left)

```

Division and Unary Minus

```

lemma minus-divide-right:
  - (a / b) = a / - b

```

```

by (simp add: divide-inverse)

lemma divide-minus-right [simp, no-atp]:
   $a / - b = - (a / b)$ 
by (simp add: divide-inverse)

lemma minus-divide-divide:
   $(- a) / (- b) = a / b$ 
apply (cases b=0, simp)
apply (simp add: nonzero-minus-divide-divide)
done

lemma eq-divide-eq:
   $a = b / c \longleftrightarrow (if\ c \neq 0\ then\ a * c = b\ else\ a = 0)$ 
by (simp add: nonzero-eq-divide-eq)

lemma divide-eq-eq:
   $b / c = a \longleftrightarrow (if\ c \neq 0\ then\ b = a * c\ else\ a = 0)$ 
by (force simp add: nonzero-divide-eq-eq)

lemma inverse-eq-1-iff [simp]:
   $inverse\ x = 1 \longleftrightarrow x = 1$ 
by (insert inverse-eq-iff-eq [of x 1], simp)

lemma divide-eq-0-iff [simp, no-atp]:
   $a / b = 0 \longleftrightarrow a = 0 \vee b = 0$ 
by (simp add: divide-inverse)

lemma divide-cancel-right [simp, no-atp]:
   $a / c = b / c \longleftrightarrow c = 0 \vee a = b$ 
apply (cases c=0, simp)
apply (simp add: divide-inverse)
done

lemma divide-cancel-left [simp, no-atp]:
   $c / a = c / b \longleftrightarrow c = 0 \vee a = b$ 
apply (cases c=0, simp)
apply (simp add: divide-inverse)
done

lemma divide-eq-1-iff [simp, no-atp]:
   $a / b = 1 \longleftrightarrow b \neq 0 \wedge a = b$ 
apply (cases b=0, simp)
apply (simp add: right-inverse-eq)
done

lemma one-eq-divide-iff [simp, no-atp]:
   $1 = a / b \longleftrightarrow b \neq 0 \wedge a = b$ 
by (simp add: eq-commute [of 1])

```

**lemma** *times-divide-times-eq*:

$(x / y) * (z / w) = (x * z) / (y * w)$   
**by** *simp*

**lemma** *add-frac-num*:

$y \neq 0 \implies x / y + z = (x + z * y) / y$   
**by** (*simp add: add-divide-distrib*)

**lemma** *add-num-frac*:

$y \neq 0 \implies z + x / y = (x + z * y) / y$   
**by** (*simp add: add-divide-distrib add.commute*)

**end**

Ordered Fields

**class** *linordered-field* = *field* + *linordered-idom*  
**begin**

**lemma** *positive-imp-inverse-positive*:

**assumes** *a-gt-0*:  $0 < a$   
**shows**  $0 < \text{inverse } a$

**proof** –

**have**  $0 < a * \text{inverse } a$   
**by** (*simp add: a-gt-0 [THEN less-imp-not-eq2]*)  
**thus**  $0 < \text{inverse } a$   
**by** (*simp add: a-gt-0 [THEN less-not-sym] zero-less-mult-iff*)

**qed**

**lemma** *negative-imp-inverse-negative*:

$a < 0 \implies \text{inverse } a < 0$   
**by** (*insert positive-imp-inverse-positive [of  $-a$ ],*  
*simp add: nonzero-inverse-minus-eq less-imp-not-eq*)

**lemma** *inverse-le-imp-le*:

**assumes** *invle*:  $\text{inverse } a \leq \text{inverse } b$  **and** *apos*:  $0 < a$   
**shows**  $b \leq a$

**proof** (*rule classical*)

**assume**  $\sim b \leq a$   
**hence**  $a < b$  **by** (*simp add: linorder-not-le*)  
**hence** *bpos*:  $0 < b$  **by** (*blast intro: apos less-trans*)  
**hence**  $a * \text{inverse } a \leq a * \text{inverse } b$   
**by** (*simp add: apos invle less-imp-le mult-left-mono*)  
**hence**  $(a * \text{inverse } a) * b \leq (a * \text{inverse } b) * b$   
**by** (*simp add: bpos less-imp-le mult-right-mono*)  
**thus**  $b \leq a$  **by** (*simp add: mult-assoc apos bpos less-imp-not-eq2*)

**qed**

**lemma** *inverse-positive-imp-positive*:



```

    assumes inv-gt-0:  $0 < \text{inverse } a$  and nz:  $a \neq 0$ 
    shows  $0 < a$ 
  proof -
    have  $0 < \text{inverse } (\text{inverse } a)$ 
      using inv-gt-0 by (rule positive-imp-inverse-positive)
    thus  $0 < a$ 
      using nz by (simp add: nonzero-inverse-inverse-eq)
  qed

```

```

lemma inverse-negative-imp-negative:
  assumes inv-less-0:  $\text{inverse } a < 0$  and nz:  $a \neq 0$ 
  shows  $a < 0$ 
  proof -
    have  $\text{inverse } (\text{inverse } a) < 0$ 
      using inv-less-0 by (rule negative-imp-inverse-negative)
    thus  $a < 0$  using nz by (simp add: nonzero-inverse-inverse-eq)
  qed

```

```

lemma linordered-field-no-lb:
   $\forall x. \exists y. y < x$ 
  proof
    fix  $x::'a$ 
    have  $m1: -(1::'a) < 0$  by simp
    from add-strict-right-mono[OF m1, where  $c=x$ ]
    have  $(-1) + x < x$  by simp
    thus  $\exists y. y < x$  by blast
  qed

```

```

lemma linordered-field-no-ub:
   $\forall x. \exists y. y > x$ 
  proof
    fix  $x::'a$ 
    have  $m1: (1::'a) > 0$  by simp
    from add-strict-right-mono[OF m1, where  $c=x$ ]
    have  $1 + x > x$  by simp
    thus  $\exists y. y > x$  by blast
  qed

```

```

lemma less-imp-inverse-less:
  assumes less:  $a < b$  and apos:  $0 < a$ 
  shows  $\text{inverse } b < \text{inverse } a$ 
  proof (rule ccontr)
    assume  $\sim \text{inverse } b < \text{inverse } a$ 
    hence  $\text{inverse } a \leq \text{inverse } b$  by simp
    hence  $\sim (a < b)$ 
      by (simp add: not-less inverse-le-imp-le [OF - apos])
    thus False by (rule notE [OF - less])
  qed

```

**lemma** *inverse-less-imp-less*:

$inverse\ a < inverse\ b \implies 0 < a \implies b < a$

**apply** (*simp add: less-le [of inverse a] less-le [of b]*)

**apply** (*force dest!: inverse-le-imp-le nonzero-inverse-eq-imp-eq*)

**done**

Both premises are essential. Consider -1 and 1.

**lemma** *inverse-less-iff-less [simp,no-atp]*:

$0 < a \implies 0 < b \implies inverse\ a < inverse\ b \longleftrightarrow b < a$

**by** (*blast intro: less-imp-inverse-less dest: inverse-less-imp-less*)

**lemma** *le-imp-inverse-le*:

$a \leq b \implies 0 < a \implies inverse\ b \leq inverse\ a$

**by** (*force simp add: le-less less-imp-inverse-less*)

**lemma** *inverse-le-iff-le [simp,no-atp]*:

$0 < a \implies 0 < b \implies inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$

**by** (*blast intro: le-imp-inverse-le dest: inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

**lemma** *inverse-le-imp-le-neg*:

$inverse\ a \leq inverse\ b \implies b < 0 \implies b \leq a$

**apply** (*rule classical*)

**apply** (*subgoal-tac a < 0*)

**prefer 2 apply force**

**apply** (*insert inverse-le-imp-le [of -b -a]*)

**apply** (*simp add: nonzero-inverse-minus-eq*)

**done**

**lemma** *less-imp-inverse-less-neg*:

$a < b \implies b < 0 \implies inverse\ b < inverse\ a$

**apply** (*subgoal-tac a < 0*)

**prefer 2 apply** (*blast intro: less-trans*)

**apply** (*insert less-imp-inverse-less [of -b -a]*)

**apply** (*simp add: nonzero-inverse-minus-eq*)

**done**

**lemma** *inverse-less-imp-less-neg*:

$inverse\ a < inverse\ b \implies b < 0 \implies b < a$

**apply** (*rule classical*)

**apply** (*subgoal-tac a < 0*)

**prefer 2**

**apply force**

**apply** (*insert inverse-less-imp-less [of -b -a]*)

**apply** (*simp add: nonzero-inverse-minus-eq*)

**done**

**lemma** *inverse-less-iff-less-neg [simp,no-atp]*:

$a < 0 \implies b < 0 \implies \text{inverse } a < \text{inverse } b \longleftrightarrow b < a$   
**apply** (*insert inverse-less-iff-less* [of  $-b - a$ ])  
**apply** (*simp del: inverse-less-iff-less*  
*add: nonzero-inverse-minus-eq*)  
**done**

**lemma** *le-imp-inverse-le-neg*:  
 $a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a$   
**by** (*force simp add: le-less less-imp-inverse-less-neg*)

**lemma** *inverse-le-iff-le-neg* [*simp,no-atp*]:  
 $a < 0 \implies b < 0 \implies \text{inverse } a \leq \text{inverse } b \longleftrightarrow b \leq a$   
**by** (*blast intro: le-imp-inverse-le-neg dest: inverse-le-imp-le-neg*)

**lemma** *one-less-inverse*:  
 $0 < a \implies a < 1 \implies 1 < \text{inverse } a$   
**using** *less-imp-inverse-less* [of  $a$  1, *unfolded inverse-1*] .

**lemma** *one-le-inverse*:  
 $0 < a \implies a \leq 1 \implies 1 \leq \text{inverse } a$   
**using** *le-imp-inverse-le* [of  $a$  1, *unfolded inverse-1*] .

**lemma** *pos-le-divide-eq* [*field-simps*]:  $0 < c \implies (a \leq b/c) = (a*c \leq b)$   
**proof** –  
**assume** *less: 0 < c*  
**hence**  $(a \leq b/c) = (a*c \leq (b/c)*c)$   
**by** (*simp add: mult-le-cancel-right less-not-sym* [*OF less*] *del: times-divide-eq*)  
**also have**  $\dots = (a*c \leq b)$   
**by** (*simp add: less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)  
**finally show** *?thesis* .  
**qed**

**lemma** *neg-le-divide-eq* [*field-simps*]:  $c < 0 \implies (a \leq b/c) = (b \leq a*c)$   
**proof** –  
**assume** *less: c < 0*  
**hence**  $(a \leq b/c) = ((b/c)*c \leq a*c)$   
**by** (*simp add: mult-le-cancel-right less-not-sym* [*OF less*] *del: times-divide-eq*)  
**also have**  $\dots = (b \leq a*c)$   
**by** (*simp add: less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)  
**finally show** *?thesis* .  
**qed**

**lemma** *pos-less-divide-eq* [*field-simps*]:  
 $0 < c \implies (a < b/c) = (a*c < b)$   
**proof** –  
**assume** *less: 0 < c*  
**hence**  $(a < b/c) = (a*c < (b/c)*c)$   
**by** (*simp add: mult-less-cancel-right-disj less-not-sym* [*OF less*] *del: times-divide-eq*)  
**also have**  $\dots = (a*c < b)$

by (simp add: less-imp-not-eq2 [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

**lemma** neg-less-divide-eq [field-simps]:  
 $c < 0 \implies (a < b/c) = (b < a*c)$   
**proof** –  
 assume less:  $c < 0$   
 hence  $(a < b/c) = ((b/c)*c < a*c)$   
 by (simp add: mult-less-cancel-right-disj less-not-sym [OF less] del: times-divide-eq)  
 also have  $\dots = (b < a*c)$   
 by (simp add: less-imp-not-eq [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

**lemma** pos-divide-less-eq [field-simps]:  
 $0 < c \implies (b/c < a) = (b < a*c)$   
**proof** –  
 assume less:  $0 < c$   
 hence  $(b/c < a) = ((b/c)*c < a*c)$   
 by (simp add: mult-less-cancel-right-disj less-not-sym [OF less] del: times-divide-eq)  
 also have  $\dots = (b < a*c)$   
 by (simp add: less-imp-not-eq2 [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

**lemma** neg-divide-less-eq [field-simps]:  
 $c < 0 \implies (b/c < a) = (a*c < b)$   
**proof** –  
 assume less:  $c < 0$   
 hence  $(b/c < a) = (a*c < (b/c)*c)$   
 by (simp add: mult-less-cancel-right-disj less-not-sym [OF less] del: times-divide-eq)  
 also have  $\dots = (a*c < b)$   
 by (simp add: less-imp-not-eq [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

**lemma** pos-divide-le-eq [field-simps]:  $0 < c \implies (b/c \leq a) = (b \leq a*c)$   
**proof** –  
 assume less:  $0 < c$   
 hence  $(b/c \leq a) = ((b/c)*c \leq a*c)$   
 by (simp add: mult-le-cancel-right less-not-sym [OF less] del: times-divide-eq)  
 also have  $\dots = (b \leq a*c)$   
 by (simp add: less-imp-not-eq2 [OF less] divide-inverse mult-assoc)  
 finally show ?thesis .  
 qed

**lemma** neg-divide-le-eq [field-simps]:  $c < 0 \implies (b/c \leq a) = (a*c \leq b)$   
**proof** –

```

assume less:  $c < 0$ 
hence  $(b/c \leq a) = (a * c \leq (b/c) * c)$ 
  by (simp add: mult-le-cancel-right less-not-sym [OF less] del: times-divide-eq)
also have  $\dots = (a * c \leq b)$ 
  by (simp add: less-imp-not-eq [OF less] divide-inverse mult-assoc)
finally show ?thesis .
qed

```

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

```

lemmas sign-simps [no-atp] = algebra-simps
  zero-less-mult-iff mult-less-0-iff

```

```

lemmas (in -) sign-simps [no-atp] = algebra-simps
  zero-less-mult-iff mult-less-0-iff

```

```

lemma divide-pos-pos:
   $0 < x \implies 0 < y \implies 0 < x / y$ 
by (simp add: field-simps)

```

```

lemma divide-nonneg-pos:
   $0 \leq x \implies 0 < y \implies 0 \leq x / y$ 
by (simp add: field-simps)

```

```

lemma divide-neg-pos:
   $x < 0 \implies 0 < y \implies x / y < 0$ 
by (simp add: field-simps)

```

```

lemma divide-nonpos-pos:
   $x \leq 0 \implies 0 < y \implies x / y \leq 0$ 
by (simp add: field-simps)

```

```

lemma divide-pos-neg:
   $0 < x \implies y < 0 \implies x / y < 0$ 
by (simp add: field-simps)

```

```

lemma divide-nonneg-neg:
   $0 \leq x \implies y < 0 \implies x / y \leq 0$ 
by (simp add: field-simps)

```

```

lemma divide-neg-neg:
   $x < 0 \implies y < 0 \implies 0 < x / y$ 
by (simp add: field-simps)

```

```

lemma divide-nonpos-neg:
   $x \leq 0 \implies y < 0 \implies 0 \leq x / y$ 

```

**by**(*simp add:field-simps*)

**lemma** *divide-strict-right-mono*:

$[[a < b; 0 < c]] \implies a / c < b / c$

**by** (*simp add: less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

**lemma** *divide-strict-right-mono-neg*:

$[[b < a; c < 0]] \implies a / c < b / c$

**apply** (*drule divide-strict-right-mono [of - -c], simp*)

**apply** (*simp add: less-imp-not-eq nonzero-minus-divide-right [symmetric]*)

**done**

The last premise ensures that  $a$  and  $b$  have the same sign

**lemma** *divide-strict-left-mono*:

$[[b < a; 0 < c; 0 < a*b]] \implies c / a < c / b$

**by**(*auto simp: field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono*)

**lemma** *divide-left-mono*:

$[[b \leq a; 0 \leq c; 0 < a*b]] \implies c / a \leq c / b$

**by**(*auto simp: field-simps times-divide-eq zero-less-mult-iff mult-right-mono*)

**lemma** *divide-strict-left-mono-neg*:

$[[a < b; c < 0; 0 < a*b]] \implies c / a < c / b$

**by**(*auto simp: field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg*)

**lemma** *mult-imp-div-pos-le*:  $0 < y \implies x \leq z * y \implies$

$x / y \leq z$

**by** (*subst pos-divide-le-eq, assumption+*)

**lemma** *mult-imp-le-div-pos*:  $0 < y \implies z * y \leq x \implies$

$z \leq x / y$

**by**(*simp add:field-simps*)

**lemma** *mult-imp-div-pos-less*:  $0 < y \implies x < z * y \implies$

$x / y < z$

**by**(*simp add:field-simps*)

**lemma** *mult-imp-less-div-pos*:  $0 < y \implies z * y < x \implies$

$z < x / y$

**by**(*simp add:field-simps*)

**lemma** *frac-le*:  $0 \leq x \implies$

$x \leq y \implies 0 < w \implies w \leq z \implies x / z \leq y / w$

**apply** (*rule mult-imp-div-pos-le*)

**apply** *simp*

**apply** (*subst times-divide-eq-left*)

**apply** (*rule mult-imp-le-div-pos, assumption*)

```

  apply (rule mult-mono)
  apply simp-all
done

```

```

lemma frac-less:  $0 \leq x \implies$ 
   $x < y \implies 0 < w \implies w \leq z \implies x / z < y / w$ 
  apply (rule mult-imp-div-pos-less)
  apply simp
  apply (subst times-divide-eq-left)
  apply (rule mult-imp-less-div-pos, assumption)
  apply (erule mult-less-le-imp-less)
  apply simp-all
done

```

```

lemma frac-less2:  $0 < x \implies$ 
   $x \leq y \implies 0 < w \implies w < z \implies x / z < y / w$ 
  apply (rule mult-imp-div-pos-less)
  apply simp-all
  apply (rule mult-imp-less-div-pos, assumption)
  apply (erule mult-le-less-imp-less)
  apply simp-all
done

```

It’s not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like  $a*b*c / x*y*z$ . The rationale for that is unclear, but many proofs seem to need them.

```

lemma less-half-sum:  $a < b \implies a < (a+b) / (1+1)$ 
by (simp add: field-simps zero-less-two)

```

```

lemma gt-half-sum:  $a < b \implies (a+b)/(1+1) < b$ 
by (simp add: field-simps zero-less-two)

```

```

subclass dense-linorder

```

```

proof

```

```

  fix x y :: 'a

```

```

  from less-add-one show  $\exists y. x < y$  ..

```

```

  from less-add-one have  $x + (-1) < (x + 1) + (-1)$  by (rule add-strict-right-mono)

```

```

  then have  $x - 1 < x + 1 - 1$  by (simp only: diff-minus [symmetric])

```

```

  then have  $x - 1 < x$  by (simp add: algebra-simps)

```

```

  then show  $\exists y. y < x$  ..

```

```

  show  $x < y \implies \exists z > x. z < y$  by (blast intro!: less-half-sum gt-half-sum)

```

```

qed

```

```

lemma nonzero-abs-inverse:

```

```

   $a \neq 0 \implies |\text{inverse } a| = \text{inverse } |a|$ 

```

```

  apply (auto simp add: neq-iff abs-if nonzero-inverse-minus-eq
    negative-imp-inverse-negative)

```

```

  apply (blast intro: positive-imp-inverse-positive elim: less-asm)

```

```

done

```

**lemma** *nonzero-abs-divide*:

$b \neq 0 \implies |a / b| = |a| / |b|$

**by** (*simp add: divide-inverse abs-mult nonzero-abs-inverse*)

**lemma** *field-le-epsilon*:

**assumes**  $e: \bigwedge e. 0 < e \implies x \leq y + e$

**shows**  $x \leq y$

**proof** (*rule dense-le*)

**fix**  $t$  **assume**  $t < x$

**hence**  $0 < x - t$  **by** (*simp add: less-diff-eq*)

**from**  $e$  [*OF this*] **have**  $x + 0 \leq x + (y - t)$  **by** (*simp add: algebra-simps*)

**then have**  $0 \leq y - t$  **by** (*simp only: add-le-cancel-left*)

**then show**  $t \leq y$  **by** (*simp add: algebra-simps*)

**qed**

**end**

**class** *linordered-field-inverse-zero* = *linordered-field* + *field-inverse-zero*

**begin**

**lemma** *le-divide-eq*:

$(a \leq b/c) =$

$(\text{if } 0 < c \text{ then } a*c \leq b$

$\text{else if } c < 0 \text{ then } b \leq a*c$

$\text{else } a \leq 0)$

**apply** (*cases c=0, simp*)

**apply** (*force simp add: pos-le-divide-eq neg-le-divide-eq linorder-neq-iff*)

**done**

**lemma** *inverse-positive-iff-positive* [*simp*]:

$(0 < \text{inverse } a) = (0 < a)$

**apply** (*cases a = 0, simp*)

**apply** (*blast intro: inverse-positive-imp-positive positive-imp-inverse-positive*)

**done**

**lemma** *inverse-negative-iff-negative* [*simp*]:

$(\text{inverse } a < 0) = (a < 0)$

**apply** (*cases a = 0, simp*)

**apply** (*blast intro: inverse-negative-imp-negative negative-imp-inverse-negative*)

**done**

**lemma** *inverse-nonnegative-iff-nonnegative* [*simp*]:

$0 \leq \text{inverse } a \iff 0 \leq a$

**by** (*simp add: not-less [symmetric]*)

**lemma** *inverse-nonpositive-iff-nonpositive* [*simp*]:

$\text{inverse } a \leq 0 \iff a \leq 0$

**by** (*simp add: not-less [symmetric]*)



**lemma** *one-less-inverse-iff*:

$$1 < \text{inverse } x \longleftrightarrow 0 < x \wedge x < 1$$

**proof** *cases*

**assume**  $0 < x$

**with** *inverse-less-iff-less* [*OF zero-less-one, of x*]

**show** *?thesis* **by** *simp*

**next**

**assume** *notless*:  $\sim (0 < x)$

**have**  $\sim (1 < \text{inverse } x)$

**proof**

**assume**  $1 < \text{inverse } x$

**also with** *notless* **have**  $\dots \leq 0$  **by** *simp*

**also have**  $\dots < 1$  **by** (*rule zero-less-one*)

**finally show** *False* **by** *auto*

**qed**

**with** *notless* **show** *?thesis* **by** *simp*

**qed**

**lemma** *one-le-inverse-iff*:

$$1 \leq \text{inverse } x \longleftrightarrow 0 < x \wedge x \leq 1$$

**proof** (*cases x = 1*)

**case** *True* **then show** *?thesis* **by** *simp*

**next**

**case** *False* **then have**  $\text{inverse } x \neq 1$  **by** *simp*

**then have**  $1 \neq \text{inverse } x$  **by** *blast*

**then have**  $1 \leq \text{inverse } x \longleftrightarrow 1 < \text{inverse } x$  **by** (*simp add: le-less*)

**with** *False* **show** *?thesis* **by** (*auto simp add: one-less-inverse-iff*)

**qed**

**lemma** *inverse-less-1-iff*:

$$\text{inverse } x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$$

**by** (*simp add: not-le [symmetric] one-le-inverse-iff*)

**lemma** *inverse-le-1-iff*:

$$\text{inverse } x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$$

**by** (*simp add: not-less [symmetric] one-less-inverse-iff*)

**lemma** *divide-le-eq*:

$$(b/c \leq a) =$$

$$(if\ 0 < c\ then\ b \leq a*c$$

$$else\ if\ c < 0\ then\ a*c \leq b$$

$$else\ 0 \leq a)$$

**apply** (*cases c=0, simp*)

**apply** (*force simp add: pos-divide-le-eq neg-divide-le-eq*)

**done**

**lemma** *less-divide-eq*:

$$(a < b/c) =$$

```

      (if 0 < c then a*c < b
        else if c < 0 then b < a*c
        else a < 0)
apply (cases c=0, simp)
apply (force simp add: pos-less-divide-eq neg-less-divide-eq)
done

```

```

lemma divide-less-eq:
  (b/c < a) =
    (if 0 < c then b < a*c
      else if c < 0 then a*c < b
      else 0 < a)
apply (cases c=0, simp)
apply (force simp add: pos-divide-less-eq neg-divide-less-eq)
done

```

### Division and Signs

```

lemma zero-less-divide-iff:
  (0 < a/b) = (0 < a & 0 < b | a < 0 & b < 0)
by (simp add: divide-inverse zero-less-mult-iff)

```

```

lemma divide-less-0-iff:
  (a/b < 0) =
    (0 < a & b < 0 | a < 0 & 0 < b)
by (simp add: divide-inverse mult-less-0-iff)

```

```

lemma zero-le-divide-iff:
  (0 ≤ a/b) =
    (0 ≤ a & 0 ≤ b | a ≤ 0 & b ≤ 0)
by (simp add: divide-inverse zero-le-mult-iff)

```

```

lemma divide-le-0-iff:
  (a/b ≤ 0) =
    (0 ≤ a & b ≤ 0 | a ≤ 0 & 0 ≤ b)
by (simp add: divide-inverse mult-le-0-iff)

```

### Division and the Number One

Simplify expressions equated with 1

```

lemma zero-eq-1-divide-iff [simp,no-atp]:
  (0 = 1/a) = (a = 0)
apply (cases a=0, simp)
apply (auto simp add: nonzero-eq-divide-eq)
done

```

```

lemma one-divide-eq-0-iff [simp,no-atp]:
  (1/a = 0) = (a = 0)
apply (cases a=0, simp)
apply (insert zero-neq-one [THEN not-sym])

```

**apply** (*auto simp add: nonzero-divide-eq-eq*)  
**done**

Simplify expressions such as  $0 < 1/x$  to  $0 < x$

**lemma** *zero-le-divide-1-iff* [*simp, no-atp*]:  
 $0 \leq 1 / a \longleftrightarrow 0 \leq a$   
**by** (*simp add: zero-le-divide-iff*)

**lemma** *zero-less-divide-1-iff* [*simp, no-atp*]:  
 $0 < 1 / a \longleftrightarrow 0 < a$   
**by** (*simp add: zero-less-divide-iff*)

**lemma** *divide-le-0-1-iff* [*simp, no-atp*]:  
 $1 / a \leq 0 \longleftrightarrow a \leq 0$   
**by** (*simp add: divide-le-0-iff*)

**lemma** *divide-less-0-1-iff* [*simp, no-atp*]:  
 $1 / a < 0 \longleftrightarrow a < 0$   
**by** (*simp add: divide-less-0-iff*)

**lemma** *divide-right-mono*:  
 $[|a \leq b; 0 \leq c|] \implies a/c \leq b/c$   
**by** (*force simp add: divide-strict-right-mono le-less*)

**lemma** *divide-right-mono-neg*:  $a \leq b$   
 $\implies c \leq 0 \implies b / c \leq a / c$   
**apply** (*drule divide-right-mono [of - - - c]*)  
**apply** *auto*  
**done**

**lemma** *divide-left-mono-neg*:  $a \leq b$   
 $\implies c \leq 0 \implies 0 < a * b \implies c / a \leq c / b$   
**apply** (*drule divide-left-mono [of - - - c]*)  
**apply** (*auto simp add: mult-commute*)  
**done**

Simplify quotients that are compared with the value 1.

**lemma** *le-divide-eq-1* [*no-atp*]:  
 $(1 \leq b / a) = ((0 < a \ \& \ a \leq b) \mid (a < 0 \ \& \ b \leq a))$   
**by** (*auto simp add: le-divide-eq*)

**lemma** *divide-le-eq-1* [*no-atp*]:  
 $(b / a \leq 1) = ((0 < a \ \& \ b \leq a) \mid (a < 0 \ \& \ a \leq b) \mid a=0)$   
**by** (*auto simp add: divide-le-eq*)

**lemma** *less-divide-eq-1* [*no-atp*]:  
 $(1 < b / a) = ((0 < a \ \& \ a < b) \mid (a < 0 \ \& \ b < a))$   
**by** (*auto simp add: less-divide-eq*)

**lemma** *divide-less-eq-1* [*no-atp*]:  
 $(b / a < 1) = ((0 < a \ \& \ b < a) \mid (a < 0 \ \& \ a < b) \mid a=0)$   
**by** (*auto simp add: divide-less-eq*)

Conditional Simplification Rules: No Case Splits

**lemma** *le-divide-eq-1-pos* [*simp,no-atp*]:  
 $0 < a \implies (1 \leq b/a) = (a \leq b)$   
**by** (*auto simp add: le-divide-eq*)

**lemma** *le-divide-eq-1-neg* [*simp,no-atp*]:  
 $a < 0 \implies (1 \leq b/a) = (b \leq a)$   
**by** (*auto simp add: le-divide-eq*)

**lemma** *divide-le-eq-1-pos* [*simp,no-atp*]:  
 $0 < a \implies (b/a \leq 1) = (b \leq a)$   
**by** (*auto simp add: divide-le-eq*)

**lemma** *divide-le-eq-1-neg* [*simp,no-atp*]:  
 $a < 0 \implies (b/a \leq 1) = (a \leq b)$   
**by** (*auto simp add: divide-le-eq*)

**lemma** *less-divide-eq-1-pos* [*simp,no-atp*]:  
 $0 < a \implies (1 < b/a) = (a < b)$   
**by** (*auto simp add: less-divide-eq*)

**lemma** *less-divide-eq-1-neg* [*simp,no-atp*]:  
 $a < 0 \implies (1 < b/a) = (b < a)$   
**by** (*auto simp add: less-divide-eq*)

**lemma** *divide-less-eq-1-pos* [*simp,no-atp*]:  
 $0 < a \implies (b/a < 1) = (b < a)$   
**by** (*auto simp add: divide-less-eq*)

**lemma** *divide-less-eq-1-neg* [*simp,no-atp*]:  
 $a < 0 \implies b/a < 1 \iff a < b$   
**by** (*auto simp add: divide-less-eq*)

**lemma** *eq-divide-eq-1* [*simp,no-atp*]:  
 $(1 = b/a) = ((a \neq 0 \ \& \ a = b))$   
**by** (*auto simp add: eq-divide-eq*)

**lemma** *divide-eq-eq-1* [*simp,no-atp*]:  
 $(b/a = 1) = ((a \neq 0 \ \& \ a = b))$   
**by** (*auto simp add: divide-eq-eq*)

**lemma** *abs-inverse* [*simp*]:  
 $|inverse \ a| =$   
 $inverse \ |a|$   
**apply** (*cases a=0, simp*)

```

apply (simp add: nonzero-abs-inverse)
done

```

```

lemma abs-divide [simp]:
   $|a / b| = |a| / |b|$ 
apply (cases b=0, simp)
apply (simp add: nonzero-abs-divide)
done

```

```

lemma abs-div-pos:  $0 < y \implies$ 
   $|x| / y = |x / y|$ 
apply (subst abs-divide)
apply (simp add: order-less-imp-le)
done

```

```

lemma field-le-mult-one-interval:
  assumes *:  $\bigwedge z. [0 < z ; z < 1] \implies z * x \leq y$ 
  shows  $x \leq y$ 
proof (cases  $0 < x$ )
  assume  $0 < x$ 
  thus ?thesis
    using dense-le-bounded[of 0 1 y/x] *
    unfolding le-divide-eq if-P[OF  $\langle 0 < x \rangle$ ] by simp
next
  assume  $\neg 0 < x$  hence  $x \leq 0$  by simp
  obtain s::'a where  $s: 0 < s \wedge s < 1$  using dense[of 0 1::'a] by auto
  hence  $x \leq s * x$  using mult-le-cancel-right[of 1 x s]  $\langle x \leq 0 \rangle$  by auto
  also note *[OF s]
  finally show ?thesis .
qed

end

```

```

code-modulename SML
  Fields Arith

```

```

code-modulename OCaml
  Fields Arith

```

```

code-modulename Haskell
  Fields Arith

```

```

end

```

## 15 Nat: Natural numbers

```

theory Nat
imports Inductive Typedef Fun Fields

```

**uses**

~~/src/Tools/rat.ML  
 ~~/src/Provers/Arith/cancel-sums.ML  
 Tools/arith-data.ML  
 (Tools/nat-arith.ML)  
 ~~/src/Provers/Arith/fast-lin-arith.ML  
 (Tools/lin-arith.ML)

**begin**

## 15.1 Type *ind*

**typedecl** *ind*

**axiomatization**

*Zero-Rep* :: *ind* **and**  
*Suc-Rep* :: *ind* ==> *ind*

**where**

— the axiom of infinity in 2 parts  
*Suc-Rep-inject*: *Suc-Rep* *x* = *Suc-Rep* *y* ==> *x* = *y* **and**  
*Suc-Rep-not-Zero-Rep*: *Suc-Rep* *x* ≠ *Zero-Rep*

## 15.2 Type *nat*

Type definition

**inductive** *Nat* :: *ind* ⇒ *bool*

**where**

*Zero-RepI*: *Nat* *Zero-Rep*  
 | *Suc-RepI*: *Nat* *i* ⇒ *Nat* (*Suc-Rep* *i*)

**global**

**typedef** (**open** *Nat*)

*nat* = *Nat*

**by** (*rule* *exI*, *unfold* *mem-def*, *rule* *Nat.Zero-RepI*)

**definition** *Suc* :: *nat* ==> *nat* **where**

*Suc-def*: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat* *n*)))

**local**

**instantiation** *nat* :: *zero*

**begin**

**definition** *Zero-nat-def* [*code del*]:

*0* = *Abs-Nat* *Zero-Rep*

**instance** ..

**end**

```

lemma Suc-not-Zero:  $Suc\ m \neq 0$ 
  by (simp add: Zero-nat-def Suc-def Abs-Nat-inject [unfolded mem-def]
    Rep-Nat [unfolded mem-def] Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep [unfolded
mem-def])

lemma Zero-not-Suc:  $0 \neq Suc\ m$ 
  by (rule not-sym, rule Suc-not-Zero not-sym)

lemma Suc-Rep-inject':  $Suc-Rep\ x = Suc-Rep\ y \longleftrightarrow x = y$ 
  by (rule iffI, rule Suc-Rep-inject simp-all)

rep-datatype 0 :: nat Suc
  apply (unfold Zero-nat-def Suc-def)
  apply (rule Rep-Nat-inverse [THEN subst]) — types force good instantiation
  apply (erule Rep-Nat [unfolded mem-def, THEN Nat.induct])
  apply (iprover elim: Abs-Nat-inverse [unfolded mem-def, THEN subst])
  apply (simp-all add: Abs-Nat-inject [unfolded mem-def] Rep-Nat [unfolded
mem-def]
    Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep [unfolded mem-def]
    Suc-Rep-not-Zero-Rep [unfolded mem-def, symmetric]
    Suc-Rep-inject' Rep-Nat-inject)
  done

lemma nat-induct [case-names 0 Suc, induct type: nat]:
  — for backward compatibility – names of variables differ
  fixes n
  assumes P 0
  and  $\bigwedge n. P\ n \implies P\ (Suc\ n)$ 
  shows P n
  using assms by (rule nat.induct)

declare nat.exhaust [case-names 0 Suc, cases type: nat]

lemmas nat-rec-0 = nat.recs(1)
  and nat-rec-Suc = nat.recs(2)

lemmas nat-case-0 = nat.cases(1)
  and nat-case-Suc = nat.cases(2)

Injectiveness and distinctness lemmas

lemma inj-Suc[simp]: inj-on Suc N
  by (simp add: inj-on-def)

lemma Suc-neq-Zero:  $Suc\ m = 0 \implies R$ 
  by (rule notE, rule Suc-not-Zero)

lemma Zero-neq-Suc:  $0 = Suc\ m \implies R$ 
  by (rule Suc-neq-Zero, erule sym)

```

**lemma** *Suc-inject*:  $Suc\ x = Suc\ y \implies x = y$   
**by** (rule *inj-Suc* [THEN *injD*])

**lemma** *n-not-Suc-n*:  $n \neq Suc\ n$   
**by** (induct *n*) *simp-all*

**lemma** *Suc-n-not-n*:  $Suc\ n \neq n$   
**by** (rule *not-sym*, rule *n-not-Suc-n*)

A special form of induction for reasoning about  $m < n$  and  $m - n$

**lemma** *diff-induct*:  $(!!x. P\ x\ 0) \implies (!!y. P\ 0\ (Suc\ y)) \implies$   
 $(!!x\ y. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y)) \implies P\ m\ n$   
**apply** (rule-tac  $x = m$  **in** *spec*)  
**apply** (induct *n*)  
**prefer** 2  
**apply** (rule *allI*)  
**apply** (induct-tac *x*, *iprover*+)  
**done**

### 15.3 Arithmetic operators

**instantiation** *nat* :: {*minus*, *comm-monoid-add*}  
**begin**

**primrec** *plus-nat*  
**where**

*add-0*:  $0 + n = (n::nat)$   
| *add-Suc*:  $Suc\ m + n = Suc\ (m + n)$

**lemma** *add-0-right* [*simp*]:  $m + 0 = (m::nat)$   
**by** (induct *m*) *simp-all*

**lemma** *add-Suc-right* [*simp*]:  $m + Suc\ n = Suc\ (m + n)$   
**by** (induct *m*) *simp-all*

**declare** *add-0* [*code*]

**lemma** *add-Suc-shift* [*code*]:  $Suc\ m + n = m + Suc\ n$   
**by** *simp*

**primrec** *minus-nat*  
**where**

*diff-0*:  $m - 0 = (m::nat)$   
| *diff-Suc*:  $m - Suc\ n = (case\ m - n\ of\ 0 \implies 0 \mid Suc\ k \implies k)$

**declare** *diff-Suc* [*simp del*]  
**declare** *diff-0* [*code*]



**lemma** *diff-0-eq-0* [*simp*, *code*]:  $0 - n = (0::nat)$   
**by** (*induct n*) (*simp-all add: diff-Suc*)

**lemma** *diff-Suc-Suc* [*simp*, *code*]:  $Suc\ m - Suc\ n = m - n$   
**by** (*induct n*) (*simp-all add: diff-Suc*)

**instance proof**

**fix**  $n\ m\ q :: nat$   
**show**  $(n + m) + q = n + (m + q)$  **by** (*induct n*) *simp-all*  
**show**  $n + m = m + n$  **by** (*induct n*) *simp-all*  
**show**  $0 + n = n$  **by** *simp*  
**qed**

**end**

**hide-fact** (**open**) *add-0 add-0-right diff-0*

**instantiation**  $nat :: comm-semiring-1-cancel$   
**begin**

**definition**

*One-nat-def* [*simp*]:  $1 = Suc\ 0$

**primrec** *times-nat*

**where**

*mult-0*:  $0 * n = (0::nat)$   
| *mult-Suc*:  $Suc\ m * n = n + (m * n)$

**lemma** *mult-0-right* [*simp*]:  $(m::nat) * 0 = 0$   
**by** (*induct m*) *simp-all*

**lemma** *mult-Suc-right* [*simp*]:  $m * Suc\ n = m + (m * n)$   
**by** (*induct m*) (*simp-all add: add-left-commute*)

**lemma** *add-mult-distrib*:  $(m + n) * k = (m * k) + ((n * k)::nat)$   
**by** (*induct m*) (*simp-all add: add-assoc*)

**instance proof**

**fix**  $n\ m\ q :: nat$   
**show**  $0 \neq (1::nat)$  **unfolding** *One-nat-def* **by** *simp*  
**show**  $1 * n = n$  **unfolding** *One-nat-def* **by** *simp*  
**show**  $n * m = m * n$  **by** (*induct n*) *simp-all*  
**show**  $(n * m) * q = n * (m * q)$  **by** (*induct n*) (*simp-all add: add-mult-distrib*)  
**show**  $(n + m) * q = n * q + m * q$  **by** (*rule add-mult-distrib*)  
**assume**  $n + m = n + q$  **thus**  $m = q$  **by** (*induct n*) *simp-all*  
**qed**

**end**

### 15.3.1 Addition

**lemma** *nat-add-assoc*:  $(m + n) + k = m + ((n + k)::nat)$   
**by** (*rule add-assoc*)

**lemma** *nat-add-commute*:  $m + n = n + (m::nat)$   
**by** (*rule add-commute*)

**lemma** *nat-add-left-commute*:  $x + (y + z) = y + ((x + z)::nat)$   
**by** (*rule add-left-commute*)

**lemma** *nat-add-left-cancel* [*simp*]:  $(k + m = k + n) = (m = (n::nat))$   
**by** (*rule add-left-cancel*)

**lemma** *nat-add-right-cancel* [*simp*]:  $(m + k = n + k) = (m = (n::nat))$   
**by** (*rule add-right-cancel*)

Reasoning about  $m + 0 = 0$ , etc.

**lemma** *add-is-0* [*iff*]:  
**fixes**  $m\ n :: nat$   
**shows**  $(m + n = 0) = (m = 0 \ \& \ n = 0)$   
**by** (*cases m*) *simp-all*

**lemma** *add-is-1*:  
 $(m + n = Suc\ 0) = (m = Suc\ 0 \ \& \ n = 0 \mid m = 0 \ \& \ n = Suc\ 0)$   
**by** (*cases m*) *simp-all*

**lemma** *one-is-add*:  
 $(Suc\ 0 = m + n) = (m = Suc\ 0 \ \& \ n = 0 \mid m = 0 \ \& \ n = Suc\ 0)$   
**by** (*rule trans*, *rule eq-commute*, *rule add-is-1*)

**lemma** *add-eq-self-zero*:  
**fixes**  $m\ n :: nat$   
**shows**  $m + n = m \implies n = 0$   
**by** (*induct m*) *simp-all*

**lemma** *inj-on-add-nat* [*simp*]: *inj-on*  $(\%n::nat. n+k)\ N$   
**apply** (*induct k*)  
**apply** *simp*  
**apply** (*drule comp-inj-on* [*OF* - *inj-Suc*])  
**apply** (*simp add:o-def*)  
**done**

### 15.3.2 Difference

**lemma** *diff-self-eq-0* [*simp*]:  $(m::nat) - m = 0$   
**by** (*induct m*) *simp-all*

**lemma** *diff-diff-left*:  $(i::nat) - j - k = i - (j + k)$   
**by** (*induct i j rule: diff-induct*) *simp-all*

**lemma** *Suc-diff-diff* [simp]:  $(\text{Suc } m - n) - \text{Suc } k = m - n - k$   
**by** (simp add: diff-diff-left)

**lemma** *diff-commute*:  $(i::\text{nat}) - j - k = i - k - j$   
**by** (simp add: diff-diff-left add-commute)

**lemma** *diff-add-inverse*:  $(n + m) - n = (m::\text{nat})$   
**by** (induct n) simp-all

**lemma** *diff-add-inverse2*:  $(m + n) - n = (m::\text{nat})$   
**by** (simp add: diff-add-inverse add-commute [of m n])

**lemma** *diff-cancel*:  $(k + m) - (k + n) = m - (n::\text{nat})$   
**by** (induct k) simp-all

**lemma** *diff-cancel2*:  $(m + k) - (n + k) = m - (n::\text{nat})$   
**by** (simp add: diff-cancel add-commute)

**lemma** *diff-add-0*:  $n - (n + m) = (0::\text{nat})$   
**by** (induct n) simp-all

**lemma** *diff-Suc-1* [simp]:  $\text{Suc } n - 1 = n$   
**unfolding** One-nat-def **by** simp

Difference distributes over multiplication

**lemma** *diff-mult-distrib*:  $((m::\text{nat}) - n) * k = (m * k) - (n * k)$   
**by** (induct m n rule: diff-induct) (simp-all add: diff-cancel)

**lemma** *diff-mult-distrib2*:  $k * ((m::\text{nat}) - n) = (k * m) - (k * n)$   
**by** (simp add: diff-mult-distrib mult-commute [of k])  
 — NOT added as rewrites, since sometimes they are used from right-to-left

### 15.3.3 Multiplication

**lemma** *nat-mult-assoc*:  $(m * n) * k = m * ((n * k)::\text{nat})$   
**by** (rule mult-assoc)

**lemma** *nat-mult-commute*:  $m * n = n * (m::\text{nat})$   
**by** (rule mult-commute)

**lemma** *add-mult-distrib2*:  $k * (m + n) = (k * m) + ((k * n)::\text{nat})$   
**by** (rule right-distrib)

**lemma** *mult-is-0* [simp]:  $((m::\text{nat}) * n = 0) = (m=0 \mid n=0)$   
**by** (induct m) auto

**lemmas** *nat-distrib* =  
 add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

```

lemma mult-eq-1-iff [simp]:  $(m * n = \text{Suc } 0) = (m = \text{Suc } 0 \ \& \ n = \text{Suc } 0)$ 
  apply (induct m)
    apply simp
  apply (induct n)
    apply auto
  done

```

```

lemma one-eq-mult-iff [simp, no-atp]:  $(\text{Suc } 0 = m * n) = (m = \text{Suc } 0 \ \& \ n = \text{Suc } 0)$ 
  apply (rule trans)
  apply (rule-tac [2] mult-eq-1-iff, fastsimp)
  done

```

```

lemma nat-mult-eq-1-iff [simp]:  $m * n = (1::\text{nat}) \longleftrightarrow m = 1 \ \wedge \ n = 1$ 
  unfolding One-nat-def by (rule mult-eq-1-iff)

```

```

lemma nat-1-eq-mult-iff [simp]:  $(1::\text{nat}) = m * n \longleftrightarrow m = 1 \ \wedge \ n = 1$ 
  unfolding One-nat-def by (rule one-eq-mult-iff)

```

```

lemma mult-cancel1 [simp]:  $(k * m = k * n) = (m = n \mid (k = (0::\text{nat})))$ 
proof –
  have  $k \neq 0 \implies k * m = k * n \implies m = n$ 
  proof (induct n arbitrary: m)
    case 0 then show  $m = 0$  by simp
  next
    case (Suc n) then show  $m = \text{Suc } n$ 
      by (cases m) (simp-all add: eq-commute [of 0])
  qed
  then show ?thesis by auto
qed

```

```

lemma mult-cancel2 [simp]:  $(m * k = n * k) = (m = n \mid (k = (0::\text{nat})))$ 
  by (simp add: mult-commute)

```

```

lemma Suc-mult-cancel1:  $(\text{Suc } k * m = \text{Suc } k * n) = (m = n)$ 
  by (subst mult-cancel1) simp

```

## 15.4 Orders on nat

### 15.4.1 Operation definition

```

instantiation nat :: linorder
begin

```

```

primrec less-eq-nat where
   $(0::\text{nat}) \leq n \longleftrightarrow \text{True}$ 
   $\mid \text{Suc } m \leq n \longleftrightarrow (\text{case } n \text{ of } 0 \Rightarrow \text{False} \mid \text{Suc } n \Rightarrow m \leq n)$ 

```

```

declare less-eq-nat.simps [simp del]

```

**lemma** [code]:  $(0::nat) \leq n \longleftrightarrow \text{True}$  **by** (simp add: less-eq-nat.simps)

**lemma** le0 [iff]:  $0 \leq (n::nat)$  **by** (simp add: less-eq-nat.simps)

**definition** less-nat **where**

less-eq-Suc-le:  $n < m \longleftrightarrow \text{Suc } n \leq m$

**lemma** Suc-le-mono [iff]:  $\text{Suc } n \leq \text{Suc } m \longleftrightarrow n \leq m$

**by** (simp add: less-eq-nat.simps(2))

**lemma** Suc-le-eq [code]:  $\text{Suc } m \leq n \longleftrightarrow m < n$

**unfolding** less-eq-Suc-le ..

**lemma** le-0-eq [iff]:  $(n::nat) \leq 0 \longleftrightarrow n = 0$

**by** (induct n) (simp-all add: less-eq-nat.simps(2))

**lemma** not-less0 [iff]:  $\neg n < (0::nat)$

**by** (simp add: less-eq-Suc-le)

**lemma** less-nat-zero-code [code]:  $n < (0::nat) \longleftrightarrow \text{False}$

**by** simp

**lemma** Suc-less-eq [iff]:  $\text{Suc } m < \text{Suc } n \longleftrightarrow m < n$

**by** (simp add: less-eq-Suc-le)

**lemma** less-Suc-eq-le [code]:  $m < \text{Suc } n \longleftrightarrow m \leq n$

**by** (simp add: less-eq-Suc-le)

**lemma** le-SucI:  $m \leq n \implies m \leq \text{Suc } n$

**by** (induct m arbitrary: n)

(simp-all add: less-eq-nat.simps(2) split: nat.splits)

**lemma** Suc-leD:  $\text{Suc } m \leq n \implies m \leq n$

**by** (cases n) (auto intro: le-SucI)

**lemma** less-SucI:  $m < n \implies m < \text{Suc } n$

**by** (simp add: less-eq-Suc-le) (erule Suc-leD)

**lemma** Suc-lessD:  $\text{Suc } m < n \implies m < n$

**by** (simp add: less-eq-Suc-le) (erule Suc-leD)

**instance**

**proof**

**fix** n m :: nat

**show**  $n < m \longleftrightarrow n \leq m \wedge \neg m \leq n$

**proof** (induct n arbitrary: m)

**case** 0 **then show** ?case **by** (cases m) (simp-all add: less-eq-Suc-le)

**next**

**case** (Suc n) **then show** ?case **by** (cases m) (simp-all add: less-eq-Suc-le)

**qed**

```

next
  fix n :: nat show n ≤ n by (induct n) simp-all
next
  fix n m :: nat assume n ≤ m and m ≤ n
  then show n = m
    by (induct n arbitrary: m)
      (simp-all add: less-eq-nat.simps(2) split: nat.splits)
next
  fix n m q :: nat assume n ≤ m and m ≤ q
  then show n ≤ q
  proof (induct n arbitrary: m q)
    case 0 show ?case by simp
  next
    case (Suc n) then show ?case
      by (simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
        simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
        simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits)
  qed
next
  fix n m :: nat show n ≤ m ∨ m ≤ n
  by (induct n arbitrary: m)
    (simp-all add: less-eq-nat.simps(2) split: nat.splits)
qed

end

instantiation nat :: bot
begin

definition bot-nat :: nat where
  bot-nat = 0

instance proof
qed (simp add: bot-nat-def)

end

```

#### 15.4.2 Introduction properties

```

lemma lessI [iff]: n < Suc n
  by (simp add: less-Suc-eq-le)

```

```

lemma zero-less-Suc [iff]: 0 < Suc n
  by (simp add: less-Suc-eq-le)

```

#### 15.4.3 Elimination properties

```

lemma less-not-refl: ~ n < (n::nat)
  by (rule order-less-irrefl)

```

**lemma** *less-not-refl2*:  $n < m \implies m \neq (n::nat)$   
**by** (*rule not-sym*) (*rule less-imp-neq*)

**lemma** *less-not-refl3*:  $(s::nat) < t \implies s \neq t$   
**by** (*rule less-imp-neq*)

**lemma** *less-irrefl-nat*:  $(n::nat) < n \implies R$   
**by** (*rule notE*, *rule less-not-refl*)

**lemma** *less-zeroE*:  $(n::nat) < 0 \implies R$   
**by** (*rule notE*) (*rule not-less0*)

**lemma** *less-Suc-eq*:  $(m < Suc\ n) = (m < n \mid m = n)$   
**unfolding** *less-Suc-eq-le le-less ..*

**lemma** *less-Suc0* [*iff*]:  $(n < Suc\ 0) = (n = 0)$   
**by** (*simp add: less-Suc-eq*)

**lemma** *less-one* [*iff*, *no-atp*]:  $(n < (1::nat)) = (n = 0)$   
**unfolding** *One-nat-def* **by** (*rule less-Suc0*)

**lemma** *Suc-mono*:  $m < n \implies Suc\ m < Suc\ n$   
**by** *simp*

”Less than” is antisymmetric, sort of

**lemma** *less-antisym*:  $\llbracket \neg n < m; n < Suc\ m \rrbracket \implies m = n$   
**unfolding** *not-less less-Suc-eq-le* **by** (*rule antisym*)

**lemma** *nat-neq-iff*:  $((m::nat) \neq n) = (m < n \mid n < m)$   
**by** (*rule linorder-neq-iff*)

**lemma** *nat-less-cases*: **assumes** *major*:  $(m::nat) < n \implies P\ n\ m$   
**and** *eqCase*:  $m = n \implies P\ n\ m$  **and** *lessCase*:  $n < m \implies P\ n\ m$   
**shows**  $P\ n\ m$   
**apply** (*rule less-linear [THEN disjE]*)  
**apply** (*erule-tac [2] disjE*)  
**apply** (*erule lessCase*)  
**apply** (*erule sym [THEN eqCase]*)  
**apply** (*erule major*)  
**done**

#### 15.4.4 Inductive (?) properties

**lemma** *Suc-lessI*:  $m < n \implies Suc\ m \neq n \implies Suc\ m < n$   
**unfolding** *less-eq-Suc-le [of m] le-less* **by** *simp*

**lemma** *lessE*:  
**assumes** *major*:  $i < k$   
**and** *p1*:  $k = Suc\ i \implies P$  **and** *p2*:  $!!j. i < j \implies k = Suc\ j \implies P$

shows  $P$   
**proof** –  
 from *major* **have**  $\exists j. i \leq j \wedge k = \text{Suc } j$   
 unfolding *less-eq-Suc-le* **by** (*induct k*) *simp-all*  
 then **have**  $(\exists j. i < j \wedge k = \text{Suc } j) \vee k = \text{Suc } i$   
**by** (*clarsimp simp add: less-le*)  
 with *p1 p2* **show**  $P$  **by** *auto*  
**qed**

**lemma** *less-SucE*: **assumes** *major*:  $m < \text{Suc } n$   
**and** *less*:  $m < n \implies P$  **and** *eq*:  $m = n \implies P$  **shows**  $P$   
**apply** (*rule major [THEN lessE]*)  
**apply** (*rule eq, blast*)  
**apply** (*rule less, blast*)  
**done**

**lemma** *Suc-lessE*: **assumes** *major*:  $\text{Suc } i < k$   
**and** *minor*:  $\forall j. i < j \implies k = \text{Suc } j \implies P$  **shows**  $P$   
**apply** (*rule major [THEN lessE]*)  
**apply** (*erule lessI [THEN minor]*)  
**apply** (*erule Suc-lessD [THEN minor], assumption*)  
**done**

**lemma** *Suc-less-SucD*:  $\text{Suc } m < \text{Suc } n \implies m < n$   
**by** *simp*

**lemma** *less-trans-Suc*:  
**assumes** *le*:  $i < j$  **shows**  $j < k \implies \text{Suc } i < k$   
**apply** (*induct k, simp-all*)  
**apply** (*insert le*)  
**apply** (*simp add: less-Suc-eq*)  
**apply** (*blast dest: Suc-lessD*)  
**done**

Can be used with *less-Suc-eq* to get  $n = m \vee n < m$

**lemma** *not-less-eq*:  $\neg m < n \longleftrightarrow n < \text{Suc } m$   
**unfolding** *not-less less-Suc-eq-le* ..

**lemma** *not-less-eq-eq*:  $\neg m \leq n \longleftrightarrow \text{Suc } n \leq m$   
**unfolding** *not-le Suc-le-eq* ..

Properties of “less than or equal”

**lemma** *le-imp-less-Suc*:  $m \leq n \implies m < \text{Suc } n$   
**unfolding** *less-Suc-eq-le* .

**lemma** *Suc-n-not-le-n*:  $\sim \text{Suc } n \leq n$   
**unfolding** *not-le less-Suc-eq-le* ..

**lemma** *le-Suc-eq*:  $(m \leq \text{Suc } n) = (m \leq n \mid m = \text{Suc } n)$



**by** (*simp add: less-Suc-eq-le [symmetric] less-Suc-eq*)

**lemma** *le-SucE*:  $m \leq \text{Suc } n \implies (m \leq n \implies R) \implies (m = \text{Suc } n \implies R)$   
 $\implies R$   
**by** (*drule le-Suc-eq [THEN iffD1], iprover+*)

**lemma** *Suc-leI*:  $m < n \implies \text{Suc}(m) \leq n$   
**unfolding** *Suc-le-eq* .

Stronger version of *Suc-leD*

**lemma** *Suc-le-lessD*:  $\text{Suc } m \leq n \implies m < n$   
**unfolding** *Suc-le-eq* .

**lemma** *less-imp-le-nat*:  $m < n \implies m \leq (n::\text{nat})$   
**unfolding** *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance,  $(\text{Suc } m < \text{Suc } n) = (\text{Suc } m \leq n) = (m < n)$

**lemmas** *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of  $m \leq n$  and  $m < n \vee m = n$

**lemma** *less-or-eq-imp-le*:  $m < n \mid m = n \implies m \leq (n::\text{nat})$   
**unfolding** *le-less* .

**lemma** *le-eq-less-or-eq*:  $(m \leq (n::\text{nat})) = (m < n \mid m = n)$   
**by** (*rule le-less*)

Useful with *blast*.

**lemma** *eq-imp-le*:  $(m::\text{nat}) = n \implies m \leq n$   
**by** *auto*

**lemma** *le-refl*:  $n \leq (n::\text{nat})$   
**by** *simp*

**lemma** *le-trans*:  $[[ i \leq j; j \leq k ]] \implies i \leq (k::\text{nat})$   
**by** (*rule order-trans*)

**lemma** *le-antisym*:  $[[ m \leq n; n \leq m ]] \implies m = (n::\text{nat})$   
**by** (*rule antisym*)

**lemma** *nat-less-le*:  $((m::\text{nat}) < n) = (m \leq n \ \& \ m \neq n)$   
**by** (*rule less-le*)

**lemma** *le-neq-implies-less*:  $(m::\text{nat}) \leq n \implies m \neq n \implies m < n$   
**unfolding** *less-le* ..

**lemma** *nat-le-linear*:  $(m::\text{nat}) \leq n \mid n \leq m$   
**by** (*rule linear*)

**lemmas** *linorder-neqE-nat* = *linorder-neqE* [**where** 'a = nat]

**lemma** *le-less-Suc-eq*:  $m \leq n \implies (n < \text{Suc } m) = (n = m)$   
**unfolding** *less-Suc-eq-le* **by** *auto*

**lemma** *not-less-less-Suc-eq*:  $\sim n < m \implies (n < \text{Suc } m) = (n = m)$   
**unfolding** *not-less* **by** (rule *le-less-Suc-eq*)

**lemmas** *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

**lemma** *def-nat-rec-0*:  $(!!n. f \ n == \text{nat-rec } c \ h \ n) \implies f \ 0 = c$   
**by** *simp*

**lemma** *def-nat-rec-Suc*:  $(!!n. f \ n == \text{nat-rec } c \ h \ n) \implies f \ (\text{Suc } n) = h \ n \ (f \ n)$   
**by** *simp*

**lemma** *not0-implies-Suc*:  $n \neq 0 \implies \exists m. n = \text{Suc } m$   
**by** (cases *n*) *simp-all*

**lemma** *gr0-implies-Suc*:  $n > 0 \implies \exists m. n = \text{Suc } m$   
**by** (cases *n*) *simp-all*

**lemma** *gr-implies-not0*: **fixes** *n* :: nat **shows**  $m < n \implies n \neq 0$   
**by** (cases *n*) *simp-all*

**lemma** *neq0-conv*[*iff*]: **fixes** *n* :: nat **shows**  $(n \neq 0) = (0 < n)$   
**by** (cases *n*) *simp-all*

This theorem is useful with *blast*

**lemma** *gr0I*:  $((n :: \text{nat}) = 0 \implies \text{False}) \implies 0 < n$   
**by** (rule *neq0-conv*[*THEN iffD1*], *iprover*)

**lemma** *gr0-conv-Suc*:  $(0 < n) = (\exists m. n = \text{Suc } m)$   
**by** (fast intro: *not0-implies-Suc*)

**lemma** *not-gr0* [*iff, no-atp*]:  $!!n :: \text{nat}. (\sim (0 < n)) = (n = 0)$   
**using** *neq0-conv* **by** *blast*

**lemma** *Suc-le-D*:  $(\text{Suc } n \leq m') \implies (? m. m' = \text{Suc } m)$   
**by** (induct *m'*) *simp-all*

Useful in certain inductive arguments

**lemma** *less-Suc-eq-0-disj*:  $(m < \text{Suc } n) = (m = 0 \mid (\exists j. m = \text{Suc } j \ \& \ j < n))$   
**by** (cases *m*) *simp-all*

#### 15.4.5 min and max

**lemma** *mono-Suc*: *mono Suc*

**by** (rule *monoI*) *simp*

**lemma** *min-0L* [*simp*]:  $\min\ 0\ n = (0::nat)$

**by** (rule *min-leastL*) *simp*

**lemma** *min-0R* [*simp*]:  $\min\ n\ 0 = (0::nat)$

**by** (rule *min-leastR*) *simp*

**lemma** *min-Suc-Suc* [*simp*]:  $\min\ (Suc\ m)\ (Suc\ n) = Suc\ (\min\ m\ n)$

**by** (*simp add: mono-Suc min-of-mono*)

**lemma** *min-Suc1*:

$\min\ (Suc\ n)\ m = (case\ m\ of\ 0 ==> 0 \mid Suc\ m' ==> Suc(\min\ n\ m'))$

**by** (*simp split: nat.split*)

**lemma** *min-Suc2*:

$\min\ m\ (Suc\ n) = (case\ m\ of\ 0 ==> 0 \mid Suc\ m' ==> Suc(\min\ m'\ n))$

**by** (*simp split: nat.split*)

**lemma** *max-0L* [*simp*]:  $\max\ 0\ n = (n::nat)$

**by** (rule *max-leastL*) *simp*

**lemma** *max-0R* [*simp*]:  $\max\ n\ 0 = (n::nat)$

**by** (rule *max-leastR*) *simp*

**lemma** *max-Suc-Suc* [*simp*]:  $\max\ (Suc\ m)\ (Suc\ n) = Suc(\max\ m\ n)$

**by** (*simp add: mono-Suc max-of-mono*)

**lemma** *max-Suc1*:

$\max\ (Suc\ n)\ m = (case\ m\ of\ 0 ==> Suc\ n \mid Suc\ m' ==> Suc(\max\ n\ m'))$

**by** (*simp split: nat.split*)

**lemma** *max-Suc2*:

$\max\ m\ (Suc\ n) = (case\ m\ of\ 0 ==> Suc\ n \mid Suc\ m' ==> Suc(\max\ m'\ n))$

**by** (*simp split: nat.split*)

#### 15.4.6 Monotonicity of Addition

**lemma** *Suc-pred* [*simp*]:  $n > 0 ==> Suc\ (n - Suc\ 0) = n$

**by** (*simp add: diff-Suc split: nat.split*)

**lemma** *Suc-diff-1* [*simp*]:  $0 < n ==> Suc\ (n - 1) = n$

**unfolding** *One-nat-def* **by** (rule *Suc-pred*)

**lemma** *nat-add-left-cancel-le* [*simp*]:  $(k + m \leq k + n) = (m \leq (n::nat))$

**by** (*induct k*) *simp-all*

**lemma** *nat-add-left-cancel-less* [*simp*]:  $(k + m < k + n) = (m < (n::nat))$

**by** (*induct k*) *simp-all*

**lemma** *add-gr-0* [*iff*]:  $!!m::nat. (m + n > 0) = (m > 0 \mid n > 0)$   
**by** (*auto dest:gr0-implies-Suc*)

strict, in 1st argument

**lemma** *add-less-mono1*:  $i < j ==> i + k < j + (k::nat)$   
**by** (*induct k*) *simp-all*

strict, in both arguments

**lemma** *add-less-mono*:  $[[i < j; k < l]] ==> i + k < j + (l::nat)$   
**apply** (*rule add-less-mono1 [THEN less-trans], assumption+*)  
**apply** (*induct j, simp-all*)  
**done**

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

**lemma** *less-imp-Suc-add*:  $m < n ==> (\exists k. n = \text{Suc } (m + k))$   
**apply** (*induct n*)  
**apply** (*simp-all add: order-le-less*)  
**apply** (*blast elim!: less-SucE*  
           *intro!: Nat.add-0-right [symmetric] add-Suc-right [symmetric]*)  
**done**

strict, in 1st argument; proof is by induction on  $k > 0$

**lemma** *mult-less-mono2*:  $(i::nat) < j ==> 0 < k ==> k * i < k * j$   
**apply** (*auto simp: gr0-conv-Suc*)  
**apply** (*induct-tac m*)  
**apply** (*simp-all add: add-less-mono*)  
**done**

The naturals form an ordered *comm-semiring-1-cancel*

**instance** *nat* :: *linordered-semidom*

**proof**

**fix** *i j k* :: *nat*  
  **show**  $0 < (1::nat)$  **by** *simp*  
  **show**  $i \leq j ==> k + i \leq k + j$  **by** *simp*  
  **show**  $i < j ==> 0 < k ==> k * i < k * j$  **by** (*simp add: mult-less-mono2*)  
**qed**

**instance** *nat* :: *no-zero-divisors*

**proof**

**fix** *a::nat* **and** *b::nat* **show**  $a \sim 0 \implies b \sim 0 \implies a * b \sim 0$  **by** *auto*  
**qed**

**lemma** *nat-mult-1*:  $(1::nat) * n = n$   
**by** *simp*

**lemma** *nat-mult-1-right*:  $n * (1::nat) = n$   
**by** *simp*

15.4.7 Additional theorems about  $op \leq$ 

Complete induction, aka course-of-values induction

```

instance nat :: wellorder proof
  fix P and n :: nat
  assume step:  $\bigwedge n::nat. (\bigwedge m. m < n \implies P\ m) \implies P\ n$ 
  have  $\bigwedge q. q \leq n \implies P\ q$ 
  proof (induct n)
    case (0 n)
    have P 0 by (rule step) auto
    thus ?case using 0 by auto
  next
  case (Suc m n)
  then have  $n \leq m \vee n = \text{Suc } m$  by (simp add: le-Suc-eq)
  thus ?case
  proof
    assume  $n \leq m$  thus P n by (rule Suc(1))
  next
    assume n:  $n = \text{Suc } m$ 
    show P n
    by (rule step) (rule Suc(1), simp add: n le-simps)
  qed
qed
then show P n by auto
qed

```

**lemma** Least-Suc:

```

  [| P n; ~ P 0 |] ==> (LEAST n. P n) = Suc (LEAST m. P (Suc m))
  apply (case-tac n, auto)
  apply (frule LeastI)
  apply (drule-tac P = %x. P (Suc x) in LeastI)
  apply (subgoal-tac (LEAST x. P x) ≤ Suc (LEAST x. P (Suc x)))
  apply (erule-tac [2] Least-le)
  apply (case-tac LEAST x. P x, auto)
  apply (drule-tac P = %x. P (Suc x) in Least-le)
  apply (blast intro: order-antisym)
  done

```

**lemma** Least-Suc2:

```

  [| P n; Q m; ~ P 0; !k. P (Suc k) = Q k |] ==> Least P = Suc (Least Q)
  apply (erule (1) Least-Suc [THEN ssubst])
  apply simp
  done

```

**lemma** ex-least-nat-le:  $\neg P(0) \implies P(n::nat) \implies \exists k \leq n. (\forall i < k. \neg P\ i) \ \& \ P(k)$

```

  apply (cases n)
  apply blast
  apply (rule-tac x=LEAST k. P(k) in exI)
  apply (blast intro: Least-le dest: not-less-Least intro: LeastI-ex)

```

done

**lemma** *ex-least-nat-less*:  $\neg P(0) \implies P(n::nat) \implies \exists k < n. (\forall i \leq k. \neg P i) \ \& \ P(k+1)$   
**unfolding** *One-nat-def*  
**apply** (*cases n*)  
**apply** *blast*  
**apply** (*frule (1) ex-least-nat-le*)  
**apply** (*erule exE*)  
**apply** (*case-tac k*)  
**apply** *simp*  
**apply** (*rename-tac k1*)  
**apply** (*rule-tac x=k1 in exI*)  
**apply** (*auto simp add: less-eq-Suc-le*)  
done

**lemma** *nat-less-induct*:  
**assumes**  $!!n. \forall m::nat. m < n \longrightarrow P m \implies P n$  **shows**  $P n$   
**using** *assms less-induct* **by** *blast*

**lemma** *measure-induct-rule* [*case-names less*]:  
**fixes**  $f :: 'a \Rightarrow nat$   
**assumes** *step*:  $\bigwedge x. (\bigwedge y. f y < f x \implies P y) \implies P x$   
**shows**  $P a$   
**by** (*induct m*  $\equiv$  *a arbitrary: a rule: less-induct*) (*auto intro: step*)

old style induction rules:

**lemma** *measure-induct*:  
**fixes**  $f :: 'a \Rightarrow nat$   
**shows**  $(\bigwedge x. \forall y. f y < f x \longrightarrow P y \implies P x) \implies P a$   
**by** (*rule measure-induct-rule [of f P a]*) *iprover*

**lemma** *full-nat-induct*:  
**assumes** *step*:  $(!!n. (ALL m. Suc m \leq n \longrightarrow P m) \implies P n)$   
**shows**  $P n$   
**by** (*rule less-induct*) (*auto intro: step simp:le-simps*)

An induction rule for establishing binary relations

**lemma** *less-Suc-induct*:  
**assumes** *less*:  $i < j$   
**and** *step*:  $!!i. P i (Suc i)$   
**and** *trans*:  $!!i j k. i < j \implies j < k \implies P i j \implies P j k \implies P i k$   
**shows**  $P i j$   
**proof** –  
**from** *less* **obtain**  $k$  **where**  $j = Suc (i + k)$  **by** (*auto dest: less-imp-Suc-add*)  
**have**  $P i (Suc (i + k))$   
**proof** (*induct k*)  
**case** 0  
**show** ?*case* **by** (*simp add: step*)  
**next**

```

case (Suc k)
have 0 + i < Suc k + i by (rule add-less-mono1) simp
hence i < Suc (i + k) by (simp add: add-commute)
from trans[OF this lessI Suc step]
show ?case by simp
qed
thus P i j by (simp add: j)
qed

```

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis.  $P(n)$  is true for all  $n \in \mathbb{N}$  if

- case “0”: given  $n = 0$  prove  $P(n)$ ,
- case “smaller”: given  $n > 0$  and  $\neg P(n)$  prove there exists a smaller integer  $m$  such that  $\neg P(m)$ .

A compact version without explicit base case:

**lemma** *infinite-descent*:  
 $\llbracket !!n::nat. \neg P\ n \implies \exists m < n. \neg P\ m \rrbracket \implies P\ n$   
**by** (induct n rule: less-induct, auto)

**lemma** *infinite-descent0*[*case-names 0 smaller*]:  
 $\llbracket P\ 0; !!n. n > 0 \implies \neg P\ n \implies (\exists m::nat. m < n \wedge \neg P\ m) \rrbracket \implies P\ n$   
**by** (rule infinite-descent) (case-tac n>0, auto)

Infinite descent using a mapping to  $\mathbb{N}$ :  $P(x)$  is true for all  $x \in D$  if there exists a  $V : D \rightarrow \mathbb{N}$  and

- case “0”: given  $V(x) = 0$  prove  $P(x)$ ,
- case “smaller”: given  $V(x) > 0$  and  $\neg P(x)$  prove there exists a  $y \in D$  such that  $V(y) < V(x)$  and  $\neg P(y)$ .

NB: the proof also shows how to use the previous lemma.

**corollary** *infinite-descent0-measure* [*case-names 0 smaller*]:  
**assumes** A0:  $!!x. V\ x = (0::nat) \implies P\ x$   
**and** A1:  $!!x. V\ x > 0 \implies \neg P\ x \implies (\exists y. V\ y < V\ x \wedge \neg P\ y)$   
**shows**  $P\ x$

**proof** –

```

obtain n where n = V x by auto
moreover have  $\bigwedge x. V\ x = n \implies P\ x$ 
proof (induct n rule: infinite-descent0)
  case 0 — i.e.  $V(x) = 0$ 
  with A0 show  $P\ x$  by auto
next — now  $n > 0$  and  $P(x)$  does not hold for some  $x$  with  $V(x) = n$ 
  case (smaller n)
  then obtain x where vxn:  $V\ x = n$  and  $V\ x > 0 \wedge \neg P\ x$  by auto

```

```

    with A1 obtain y where  $V\ y < V\ x \wedge \neg P\ y$  by auto
    with vxn obtain m where  $m = V\ y \wedge m < n \wedge \neg P\ y$  by auto
    then show ?case by auto
  qed
  ultimately show  $P\ x$  by auto
qed

```

Again, without explicit base case:

```

lemma infinite-descent-measure:
assumes !!x.  $\neg P\ x \implies \exists y. (V::'a \Rightarrow nat)\ y < V\ x \wedge \neg P\ y$  shows  $P\ x$ 
proof -
  from assms obtain n where  $n = V\ x$  by auto
  moreover have !!x.  $V\ x = n \implies P\ x$ 
  proof (induct n rule: infinite-descent, auto)
    fix x assume  $\neg P\ x$ 
    with assms show  $\exists m < V\ x. \exists y. V\ y = m \wedge \neg P\ y$  by auto
  qed
  ultimately show  $P\ x$  by auto
qed

```

A [clumsy] way of lifting  $<$  monotonicity to  $\leq$  monotonicity

```

lemma less-mono-imp-le-mono:
  [| !!i j::nat.  $i < j \implies f\ i < f\ j$ ;  $i \leq j$  |]  $\implies f\ i \leq ((f\ j)::nat)$ 
by (simp add: order-le-less) (blast)

```

non-strict, in 1st argument

```

lemma add-le-mono1:  $i \leq j \implies i + k \leq j + (k::nat)$ 
by (rule add-right-mono)

```

non-strict, in both arguments

```

lemma add-le-mono: [|  $i \leq j$ ;  $k \leq l$  |]  $\implies i + k \leq j + (l::nat)$ 
by (rule add-mono)

```

```

lemma le-add2:  $n \leq ((m + n)::nat)$ 
by (insert add-right-mono [of 0 m n], simp)

```

```

lemma le-add1:  $n \leq ((n + m)::nat)$ 
by (simp add: add-commute, rule le-add2)

```

```

lemma less-add-Suc1:  $i < Suc\ (i + m)$ 
by (rule le-less-trans, rule le-add1, rule lessI)

```

```

lemma less-add-Suc2:  $i < Suc\ (m + i)$ 
by (rule le-less-trans, rule le-add2, rule lessI)

```

```

lemma less-iff-Suc-add:  $(m < n) = (\exists k. n = Suc\ (m + k))$ 
by (iprover intro!: less-add-Suc1 less-imp-Suc-add)

```



**lemma** *trans-le-add1*:  $(i::nat) \leq j \implies i \leq j + m$   
**by** (*rule le-trans, assumption, rule le-add1*)

**lemma** *trans-le-add2*:  $(i::nat) \leq j \implies i \leq m + j$   
**by** (*rule le-trans, assumption, rule le-add2*)

**lemma** *trans-less-add1*:  $(i::nat) < j \implies i < j + m$   
**by** (*rule less-le-trans, assumption, rule le-add1*)

**lemma** *trans-less-add2*:  $(i::nat) < j \implies i < m + j$   
**by** (*rule less-le-trans, assumption, rule le-add2*)

**lemma** *add-lessD1*:  $i + j < (k::nat) \implies i < k$   
**apply** (*rule le-less-trans [of - i+j]*)  
**apply** (*simp-all add: le-add1*)  
**done**

**lemma** *not-add-less1* [*iff*]:  $\sim (i + j < (i::nat))$   
**apply** (*rule notI*)  
**apply** (*drule add-lessD1*)  
**apply** (*erule less-irrefl [THEN notE]*)  
**done**

**lemma** *not-add-less2* [*iff*]:  $\sim (j + i < (i::nat))$   
**by** (*simp add: add-commute*)

**lemma** *add-leD1*:  $m + k \leq n \implies m \leq (n::nat)$   
**apply** (*rule order-trans [of - m+k]*)  
**apply** (*simp-all add: le-add1*)  
**done**

**lemma** *add-leD2*:  $m + k \leq n \implies k \leq (n::nat)$   
**apply** (*simp add: add-commute*)  
**apply** (*erule add-leD1*)  
**done**

**lemma** *add-leE*:  $(m::nat) + k \leq n \implies (m \leq n \implies k \leq n \implies R) \implies R$   
**by** (*blast dest: add-leD1 add-leD2*)

needs !!*k* for *add-ac* to work

**lemma** *less-add-eq-less*:  $!!k::nat. k < l \implies m + l = k + n \implies m < n$   
**by** (*force simp del: add-Suc-right*  
*simp add: less-iff-Suc-add add-Suc-right [symmetric] add-ac*)

#### 15.4.8 More results about difference

Addition is the inverse of subtraction: if  $n \leq m$  then  $n + (m - n) = m$ .

**lemma** *add-diff-inverse*:  $\sim m < n \implies n + (m - n) = (m::nat)$   
**by** (*induct m n rule: diff-induct*) *simp-all*

**lemma** *le-add-diff-inverse* [simp]:  $n \leq m \implies n + (m - n) = (m::nat)$   
**by** (simp add: add-diff-inverse linorder-not-less)

**lemma** *le-add-diff-inverse2* [simp]:  $n \leq m \implies (m - n) + n = (m::nat)$   
**by** (simp add: add-commute)

**lemma** *Suc-diff-le*:  $n \leq m \implies \text{Suc } m - n = \text{Suc } (m - n)$   
**by** (induct m n rule: diff-induct) simp-all

**lemma** *diff-less-Suc*:  $m - n < \text{Suc } m$   
**apply** (induct m n rule: diff-induct)  
**apply** (erule-tac [3] less-SucE)  
**apply** (simp-all add: less-Suc-eq)  
**done**

**lemma** *diff-le-self* [simp]:  $m - n \leq (m::nat)$   
**by** (induct m n rule: diff-induct) (simp-all add: le-SucI)

**lemma** *le-iff-add*:  $(m::nat) \leq n = (\exists k. n = m + k)$   
**by** (auto simp: le-add1 dest!: le-add-diff-inverse sym [of - n])

**lemma** *less-imp-diff-less*:  $(j::nat) < k \implies j - n < k$   
**by** (rule le-less-trans, rule diff-le-self)

**lemma** *diff-Suc-less* [simp]:  $0 < n \implies n - \text{Suc } i < n$   
**by** (cases n) (auto simp add: le-simps)

**lemma** *diff-add-assoc*:  $k \leq (j::nat) \implies (i + j) - k = i + (j - k)$   
**by** (induct j k rule: diff-induct) simp-all

**lemma** *diff-add-assoc2*:  $k \leq (j::nat) \implies (j + i) - k = (j - k) + i$   
**by** (simp add: add-commute diff-add-assoc)

**lemma** *le-imp-diff-is-add*:  $i \leq (j::nat) \implies (j - i = k) = (j = k + i)$   
**by** (auto simp add: diff-add-inverse2)

**lemma** *diff-is-0-eq* [simp]:  $((m::nat) - n = 0) = (m \leq n)$   
**by** (induct m n rule: diff-induct) simp-all

**lemma** *diff-is-0-eq'* [simp]:  $m \leq n \implies (m::nat) - n = 0$   
**by** (rule iffD2, rule diff-is-0-eq)

**lemma** *zero-less-diff* [simp]:  $(0 < n - (m::nat)) = (m < n)$   
**by** (induct m n rule: diff-induct) simp-all

**lemma** *less-imp-add-positive*:  
**assumes**  $i < j$   
**shows**  $\exists k::nat. 0 < k \ \& \ i + k = j$

**proof**

**from** *assms* **show**  $0 < j - i \ \& \ i + (j - i) = j$

**by** (*simp add: order-less-imp-le*)

**qed**

a nice rewrite for bounded subtraction

**lemma** *nat-minus-add-max*:

**fixes**  $n \ m :: \text{nat}$

**shows**  $n - m + m = \max \ n \ m$

**by** (*simp add: max-def not-le order-less-imp-le*)

**lemma** *nat-diff-split*:

$P(a - b::\text{nat}) = ((a < b \dashrightarrow P \ 0) \ \& \ (\text{ALL } d. a = b + d \dashrightarrow P \ d))$

— elimination of  $-$  on *nat*

**by** (*cases a < b*)

(*auto simp add: diff-is-0-eq [THEN iffD2] diff-add-inverse*  
*not-less le-less dest!: sym [of a] sym [of b] add-eq-self-zero*)

**lemma** *nat-diff-split-asm*:

$P(a - b::\text{nat}) = (\sim (a < b \ \& \ \sim P \ 0) \mid (\text{EX } d. a = b + d \ \& \ \sim P \ d))$

— elimination of  $-$  on *nat* in assumptions

**by** (*auto split: nat-diff-split*)

#### 15.4.9 Monotonicity of Multiplication

**lemma** *mult-le-mono1*:  $i \leq (j::\text{nat}) \implies i * k \leq j * k$

**by** (*simp add: mult-right-mono*)

**lemma** *mult-le-mono2*:  $i \leq (j::\text{nat}) \implies k * i \leq k * j$

**by** (*simp add: mult-left-mono*)

$\leq$  monotonicity, BOTH arguments

**lemma** *mult-le-mono*:  $i \leq (j::\text{nat}) \implies k \leq l \implies i * k \leq j * l$

**by** (*simp add: mult-mono*)

**lemma** *mult-less-mono1*:  $(i::\text{nat}) < j \implies 0 < k \implies i * k < j * k$

**by** (*simp add: mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

**lemma** *nat-0-less-mult-iff* [*simp*]:  $(0 < (m::\text{nat}) * n) = (0 < m \ \& \ 0 < n)$

**apply** (*induct m*)

**apply** *simp*

**apply** (*case-tac n*)

**apply** *simp-all*

**done**

**lemma** *one-le-mult-iff* [*simp*]:  $(\text{Suc } 0 \leq m * n) = (\text{Suc } 0 \leq m \ \& \ \text{Suc } 0 \leq n)$

**apply** (*induct m*)

```

    apply simp
    apply (case-tac n)
    apply simp-all
  done

lemma mult-less-cancel2 [simp]: ((m::nat) * k < n * k) = (0 < k & m < n)
  apply (safe intro!: mult-less-mono1)
  apply (case-tac k, auto)
  apply (simp del: le-0-eq add: linorder-not-le [symmetric])
  apply (blast intro: mult-le-mono1)
  done

lemma mult-less-cancel1 [simp]: (k * (m::nat) < k * n) = (0 < k & m < n)
  by (simp add: mult-commute [of k])

lemma mult-le-cancel1 [simp]: (k * (m::nat) ≤ k * n) = (0 < k → m ≤ n)
  by (simp add: linorder-not-less [symmetric], auto)

lemma mult-le-cancel2 [simp]: ((m::nat) * k ≤ n * k) = (0 < k → m ≤ n)
  by (simp add: linorder-not-less [symmetric], auto)

lemma Suc-mult-less-cancel1: (Suc k * m < Suc k * n) = (m < n)
  by (subst mult-less-cancel1) simp

lemma Suc-mult-le-cancel1: (Suc k * m ≤ Suc k * n) = (m ≤ n)
  by (subst mult-le-cancel1) simp

lemma le-square: m ≤ m * (m::nat)
  by (cases m) (auto intro: le-add1)

lemma le-cube: (m::nat) ≤ m * (m * m)
  by (cases m) (auto intro: le-add1)

Lemma for gcd

lemma mult-eq-self-implies-10: (m::nat) = m * n ==> n = 1 | m = 0
  apply (drule sym)
  apply (rule disjCI)
  apply (rule nat-less-cases, erule-tac [2] -)
  apply (drule-tac [2] mult-less-mono2)
  apply (auto)
  done

the lattice order on nat

instantiation nat :: distrib-lattice
begin

definition
  (inf :: nat ⇒ nat ⇒ nat) = min

```

**definition**

$$(sup :: nat \Rightarrow nat \Rightarrow nat) = max$$
**instance by** *intro-classes*

$$(auto simp add: inf-nat-def sup-nat-def max-def not-le min-def \\ intro: order-less-imp-le antisym elim!: order-trans order-less-trans)$$
**end****15.5 Natural operation of natural numbers on functions**

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

**consts** *compow* ::  $nat \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$

**abbreviation** *compower* ::  $('a \Rightarrow 'b) \Rightarrow nat \Rightarrow 'a \Rightarrow 'b$  (**infixr**  $^{\wedge}$  80) **where**  
 $f^{\wedge} n \equiv compow\ n\ f$

**notation** (*latex output*)
$$compower\ ((-))\ [1000]\ 1000$$
**notation** (*HTML output*)
$$compower\ ((-))\ [1000]\ 1000$$

$f^{\wedge} n = f \circ \dots \circ f$ , the  $n$ -fold composition of  $f$

**overloading**

$$funpow == compow :: nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$$
**begin**

**primrec** *funpow* ::  $nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$  **where**

$$funpow\ 0\ f = id$$

$$| funpow\ (Suc\ n)\ f = f \circ funpow\ n\ f$$
**end**

for code generation

**definition** *funpow* ::  $nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$  **where**

$$funpow\text{-code-def}\ [code\text{-post}]: funpow = compow$$

**lemmas**  $[code\text{-unfold}] = funpow\text{-code-def}\ [symmetric]$

**lemma**  $[code]:$

$$funpow\ 0\ f = id$$

$$funpow\ (Suc\ n)\ f = f \circ funpow\ n\ f$$

**unfolding** *funpow-code-def* **by** *simp-all*

**hide-const** (**open**) *funpow*

**lemma** *funpow-add*:

$f^{m+n} = f^m \circ f^n$

**by** (*induct m*) *simp-all*

**lemma** *funpow-swap1*:

$f(f^n x) = (f^n)(f x)$

**proof** –

**have**  $f(f^n x) = (f^{n+1}) x$  **by** *simp*

**also have**  $\dots = (f^n o f^1) x$  **by** (*simp only: funpow-add*)

**also have**  $\dots = (f^n)(f x)$  **by** *simp*

**finally show** *?thesis* .

**qed**

## 15.6 Embedding of the Naturals into any *semiring-1*: *of-nat*

**context** *semiring-1*

**begin**

**primrec**

*of-nat* :: *nat*  $\Rightarrow$  *'a*

**where**

*of-nat-0*:  $of\text{-}nat\ 0 = 0$

| *of-nat-Suc*:  $of\text{-}nat\ (Suc\ m) = 1 + of\text{-}nat\ m$

**lemma** *of-nat-1* [*simp*]:  $of\text{-}nat\ 1 = 1$

**unfolding** *One-nat-def* **by** *simp*

**lemma** *of-nat-add* [*simp*]:  $of\text{-}nat\ (m + n) = of\text{-}nat\ m + of\text{-}nat\ n$

**by** (*induct m*) (*simp-all add: add-ac*)

**lemma** *of-nat-mult*:  $of\text{-}nat\ (m * n) = of\text{-}nat\ m * of\text{-}nat\ n$

**by** (*induct m*) (*simp-all add: add-ac left-distrib*)

**primrec** *of-nat-aux* :: (*'a*  $\Rightarrow$  *'a*)  $\Rightarrow$  *nat*  $\Rightarrow$  *'a*  $\Rightarrow$  *'a* **where**

*of-nat-aux inc 0 i* = *i*

| *of-nat-aux inc (Suc n) i* = *of-nat-aux inc n (inc i)* — tail recursive

**lemma** *of-nat-code*:

$of\text{-}nat\ n = of\text{-}nat\text{-}aux\ (\lambda i. i + 1)\ n\ 0$

**proof** (*induct n*)

**case 0** **then show** *?case* **by** *simp*

**next**

**case** (*Suc n*)

**have**  $\bigwedge i. of\text{-}nat\text{-}aux\ (\lambda i. i + 1)\ n\ (i + 1) = of\text{-}nat\text{-}aux\ (\lambda i. i + 1)\ n\ i + 1$

**by** (*induct n*) *simp-all*

**from this** [*of 0*] **have**  $of\text{-}nat\text{-}aux\ (\lambda i. i + 1)\ n\ 1 = of\text{-}nat\text{-}aux\ (\lambda i. i + 1)\ n\ 0 + 1$

**by** *simp*

**with** *Suc* **show** *?case* **by** (*simp add: add-commute*)

qed

end

**declare** *of-nat-code* [*code*, *code-unfold*, *code-inline del*]

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

**class** *semiring-char-0* = *semiring-1* +  
**assumes** *of-nat-eq-iff* [*simp*]: *of-nat m = of-nat n*  $\longleftrightarrow$  *m = n*  
**begin**

Special cases where either operand is zero

**lemma** *of-nat-0-eq-iff* [*simp*, *no-atp*]: *0 = of-nat n*  $\longleftrightarrow$  *0 = n*  
**by** (*rule of-nat-eq-iff* [*of 0 n*, *unfolded of-nat-0*])

**lemma** *of-nat-eq-0-iff* [*simp*, *no-atp*]: *of-nat m = 0*  $\longleftrightarrow$  *m = 0*  
**by** (*rule of-nat-eq-iff* [*of m 0*, *unfolded of-nat-0*])

**lemma** *inj-of-nat*: *inj of-nat*  
**by** (*simp add: inj-on-def*)

end

**context** *linordered-semidom*  
**begin**

**lemma** *zero-le-imp-of-nat*: *0 ≤ of-nat m*  
**by** (*induct m*) *simp-all*

**lemma** *less-imp-of-nat-less*: *m < n*  $\implies$  *of-nat m < of-nat n*  
**apply** (*induct m n rule: diff-induct, simp-all*)  
**apply** (*rule add-pos-nonneg* [*OF zero-less-one zero-le-imp-of-nat*])  
**done**

**lemma** *of-nat-less-imp-less*: *of-nat m < of-nat n*  $\implies$  *m < n*  
**apply** (*induct m n rule: diff-induct, simp-all*)  
**apply** (*insert zero-le-imp-of-nat*)  
**apply** (*force simp add: not-less* [*symmetric*])  
**done**

**lemma** *of-nat-less-iff* [*simp*]: *of-nat m < of-nat n*  $\longleftrightarrow$  *m < n*  
**by** (*blast intro: of-nat-less-imp-less less-imp-of-nat-less*)

**lemma** *of-nat-le-iff* [*simp*]: *of-nat m ≤ of-nat n*  $\longleftrightarrow$  *m ≤ n*  
**by** (*simp add: not-less* [*symmetric*] *linorder-not-less* [*symmetric*])

Every *linordered-semidom* has characteristic zero.

**subclass** *semiring-char-0*

**proof qed** (*simp add: eq-iff order-eq-iff*)

Special cases where either operand is zero

**lemma** *of-nat-0-le-iff* [*simp*]:  $0 \leq \text{of-nat } n$   
**by** (*rule of-nat-le-iff [of 0, simplified]*)

**lemma** *of-nat-le-0-iff* [*simp, no-atp*]:  $\text{of-nat } m \leq 0 \longleftrightarrow m = 0$   
**by** (*rule of-nat-le-iff [of - 0, simplified]*)

**lemma** *of-nat-0-less-iff* [*simp*]:  $0 < \text{of-nat } n \longleftrightarrow 0 < n$   
**by** (*rule of-nat-less-iff [of 0, simplified]*)

**lemma** *of-nat-less-0-iff* [*simp*]:  $\neg \text{of-nat } m < 0$   
**by** (*rule of-nat-less-iff [of - 0, simplified]*)

**end**

**context** *ring-1*  
**begin**

**lemma** *of-nat-diff*:  $n \leq m \implies \text{of-nat } (m - n) = \text{of-nat } m - \text{of-nat } n$   
**by** (*simp add: algebra-simps of-nat-add [symmetric]*)

**end**

**context** *linordered-idom*  
**begin**

**lemma** *abs-of-nat* [*simp*]:  $|\text{of-nat } n| = \text{of-nat } n$   
**unfolding** *abs-if* **by** *auto*

**end**

**lemma** *of-nat-id* [*simp*]:  $\text{of-nat } n = n$   
**by** (*induct n*) *simp-all*

**lemma** *of-nat-eq-id* [*simp*]:  $\text{of-nat} = \text{id}$   
**by** (*auto simp add: expand-fun-eq*)

## 15.7 The Set of Natural Numbers

**context** *semiring-1*  
**begin**

**definition**  
*Nats* :: 'a set **where**  
 [*code del*]: *Nats* = *range of-nat*

**notation** (*xsymbols*)



*Nats* ( $\mathbb{N}$ )

**lemma** *of-nat-in-Nats* [*simp*]: *of-nat*  $n \in \mathbb{N}$   
**by** (*simp add: Nats-def*)

**lemma** *Nats-0* [*simp*]:  $0 \in \mathbb{N}$   
**apply** (*simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-0 [symmetric]*)  
**done**

**lemma** *Nats-1* [*simp*]:  $1 \in \mathbb{N}$   
**apply** (*simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-1 [symmetric]*)  
**done**

**lemma** *Nats-add* [*simp*]:  $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a + b \in \mathbb{N}$   
**apply** (*auto simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-add [symmetric]*)  
**done**

**lemma** *Nats-mult* [*simp*]:  $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a * b \in \mathbb{N}$   
**apply** (*auto simp add: Nats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-nat-mult [symmetric]*)  
**done**

**lemma** *Nats-cases* [*cases set: Nats*]:  
**assumes**  $x \in \mathbb{N}$   
**obtains** (*of-nat*)  $n$  **where**  $x = \text{of-nat } n$   
**unfolding** *Nats-def*  
**proof** –  
**from**  $\langle x \in \mathbb{N} \rangle$  **have**  $x \in \text{range of-nat}$  **unfolding** *Nats-def* .  
**then obtain**  $n$  **where**  $x = \text{of-nat } n$  ..  
**then show** *thesis* ..  
**qed**

**lemma** *Nats-induct* [*case-names of-nat, induct set: Nats*]:  
 $x \in \mathbb{N} \implies (\bigwedge n. P (\text{of-nat } n)) \implies P x$   
**by** (*rule Nats-cases*) *auto*

**end**

## 15.8 Further Arithmetic Facts Concerning the Natural Numbers

**lemma** *subst-equals*:

```

    assumes 1:  $t = s$  and 2:  $u = t$ 
    shows  $u = s$ 
    using 2 1 by (rule trans)

setup Arith-Data.setup

use Tools/nat-arith.ML
declaration  $\ll K \text{ Nat-Arith.setup} \gg$ 

use Tools/lin-arith.ML
setup  $\ll \text{Lin-Arith.global-setup} \gg$ 
declaration  $\ll K \text{ Lin-Arith.setup} \gg$ 

lemmas [arith-split] = nat-diff-split split-min split-max

context order
begin

lemma lift-Suc-mono-le:
  assumes mono:  $!!n. f\ n \leq f(\text{Suc } n)$  and  $n \leq n'$ 
  shows  $f\ n \leq f\ n'$ 
proof (cases  $n < n'$ )
  case True
  thus ?thesis
    by (induct  $n\ n'$  rule: less-Suc-induct[consumes 1]) (auto intro: mono)
qed (insert  $\langle n \leq n' \rangle$ , auto) — trivial for  $n = n'$ 

lemma lift-Suc-mono-less:
  assumes mono:  $!!n. f\ n < f(\text{Suc } n)$  and  $n < n'$ 
  shows  $f\ n < f\ n'$ 
using  $\langle n < n' \rangle$ 
by (induct  $n\ n'$  rule: less-Suc-induct[consumes 1]) (auto intro: mono)

lemma lift-Suc-mono-less-iff:
  ( $!!n. f\ n < f(\text{Suc } n)$ )  $\implies f(n) < f(m) \longleftrightarrow n < m$ 
by (blast intro: less-asym' lift-Suc-mono-less[of f]
    dest: linorder-not-less[THEN iffD1] le-eq-less-or-eq[THEN iffD1])

end

lemma mono-iff-le-Suc:  $\text{mono } f = (\forall n. f\ n \leq f(\text{Suc } n))$ 
unfolding mono-def
by (auto intro: lift-Suc-mono-le[of f])

lemma mono-nat-linear-lb:
  ( $!!m\ n::\text{nat}. m < n \implies f\ m < f\ n$ )  $\implies f(m)+k \leq f(m+k)$ 
apply (induct-tac k)
  apply simp
  apply (erule-tac  $x=m+n$  in meta-allE)

```

**apply**(*erule-tac*  $x = \text{Suc}(m+n)$  **in** *meta-allE*)  
**apply** *simp*  
**done**

Subtraction laws, mostly by Clemens Ballarin

**lemma** *diff-less-mono*:  $[[\ a < (b::\text{nat});\ c \leq a\ ]\ ] \implies a - c < b - c$   
**by** *arith*

**lemma** *less-diff-conv*:  $(i < j - k) = (i + k < (j::\text{nat}))$   
**by** *arith*

**lemma** *le-diff-conv*:  $(j - k \leq (i::\text{nat})) = (j \leq i + k)$   
**by** *arith*

**lemma** *le-diff-conv2*:  $k \leq j \implies (i \leq j - k) = (i + k \leq (j::\text{nat}))$   
**by** *arith*

**lemma** *diff-diff-cancel* [*simp*]:  $i \leq (n::\text{nat}) \implies n - (n - i) = i$   
**by** *arith*

**lemma** *le-add-diff*:  $k \leq (n::\text{nat}) \implies m \leq n + m - k$   
**by** *arith*

**lemma** *diff-less* [*simp*]:  $!!m::\text{nat}.\ [[\ 0 < n;\ 0 < m\ ]\ ] \implies m - n < m$   
**by** *arith*

Simplification of relational expressions involving subtraction

**lemma** *diff-diff-eq*:  $[[\ k \leq m;\ k \leq (n::\text{nat})\ ]\ ] \implies ((m - k) - (n - k)) = (m - n)$   
**by** (*simp split add: nat-diff-split*)

**hide-fact** (**open**) *diff-diff-eq*

**lemma** *eq-diff-iff*:  $[[\ k \leq m;\ k \leq (n::\text{nat})\ ]\ ] \implies (m - k = n - k) = (m = n)$   
**by** (*auto split add: nat-diff-split*)

**lemma** *less-diff-iff*:  $[[\ k \leq m;\ k \leq (n::\text{nat})\ ]\ ] \implies (m - k < n - k) = (m < n)$   
**by** (*auto split add: nat-diff-split*)

**lemma** *le-diff-iff*:  $[[\ k \leq m;\ k \leq (n::\text{nat})\ ]\ ] \implies (m - k \leq n - k) = (m \leq n)$   
**by** (*auto split add: nat-diff-split*)

(Anti)Monotonicity of subtraction – by Stephan Merz

**lemma** *diff-le-mono*:  $m \leq (n::\text{nat}) \implies (m - l) \leq (n - l)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *diff-le-mono2*:  $m \leq (n::\text{nat}) \implies (l - n) \leq (l - m)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *diff-less-mono2*:  $[[\ m < (n::nat); m < l \ ]] \implies (l - n) < (l - m)$   
**by** (*simp split add: nat-diff-split*)

**lemma** *diffs0-imp-equal*:  $!!m::nat. [[\ m - n = 0; n - m = 0 \ ]] \implies m = n$   
**by** (*simp split add: nat-diff-split*)

**lemma** *min-diff*:  $\min (m - (i::nat)) (n - i) = \min m\ n - i$   
**by** *auto*

**lemma** *inj-on-diff-nat*:  
**assumes** *k-le-n*:  $\forall n \in N. k \leq (n::nat)$   
**shows** *inj-on*  $(\lambda n. n - k)$  *N*  
**proof** (*rule inj-onI*)  
**fix** *x y*  
**assume** *a*:  $x \in N\ y \in N\ x - k = y - k$   
**with** *k-le-n* **have**  $x - k + k = y - k + k$  **by** *auto*  
**with** *a k-le-n* **show**  $x = y$  **by** *auto*  
**qed**

Rewriting to pull differences out

**lemma** *diff-diff-right* [*simp*]:  $k \leq j \implies i - (j - k) = i + (k::nat) - j$   
**by** *arith*

**lemma** *diff-Suc-diff-eq1* [*simp*]:  $k \leq j \implies m - \text{Suc } (j - k) = m + k - \text{Suc } j$   
**by** *arith*

**lemma** *diff-Suc-diff-eq2* [*simp*]:  $k \leq j \implies \text{Suc } (j - k) - m = \text{Suc } j - (k + m)$   
**by** *arith*

Lemmas for ex/Factorization

**lemma** *one-less-mult*:  $[[\ \text{Suc } 0 < n; \text{Suc } 0 < m \ ]] \implies \text{Suc } 0 < m * n$   
**by** (*cases m*) *auto*

**lemma** *n-less-m-mult-n*:  $[[\ \text{Suc } 0 < n; \text{Suc } 0 < m \ ]] \implies n < m * n$   
**by** (*cases m*) *auto*

**lemma** *n-less-n-mult-m*:  $[[\ \text{Suc } 0 < n; \text{Suc } 0 < m \ ]] \implies n < n * m$   
**by** (*cases m*) *auto*

Specialized induction principles that work ”backwards”:

**lemma** *inc-induct*[*consumes 1, case-names base step*]:  
**assumes** *less*:  $i \leq j$   
**assumes** *base*:  $P\ j$   
**assumes** *step*:  $!!i. [[\ i < j; P (\text{Suc } i) \ ]] \implies P\ i$   
**shows**  $P\ i$   
**using** *less*  
**proof** (*induct d == j - i arbitrary: i*)  
**case**  $(0\ i)$   
**hence**  $i = j$  **by** *simp*

```

  with base show ?case by simp
next
  case (Suc d i)
  hence  $i < j$  P (Suc i)
  by simp-all
  thus P i by (rule step)
qed

```

**lemma** *strict-inc-induct*[consumes 1, case-names base step]:

```

  assumes less:  $i < j$ 
  assumes base:  $!!i. j = \text{Suc } i \implies P \ i$ 
  assumes step:  $!!i. [| i < j; P \ (\text{Suc } i) |] \implies P \ i$ 
  shows P i
  using less
proof (induct  $d == j - i - 1$  arbitrary: i)
  case (0 i)
  with  $\langle i < j \rangle$  have  $j = \text{Suc } i$  by simp
  with base show ?case by simp
next
  case (Suc d i)
  hence  $i < j$  P (Suc i)
  by simp-all
  thus P i by (rule step)
qed

```

**lemma** *zero-induct-lemma*:  $P \ k \implies (!!n. P \ (\text{Suc } n) \implies P \ n) \implies P \ (k - i)$   
 using *inc-induct*[of  $k - i$  k P, simplified] by blast

**lemma** *zero-induct*:  $P \ k \implies (!!n. P \ (\text{Suc } n) \implies P \ n) \implies P \ 0$   
 using *inc-induct*[of 0 k P] by blast

```

lemmas add-diff-assoc = diff-add-assoc [symmetric]
lemmas add-diff-assoc2 = diff-add-assoc2[symmetric]
declare diff-diff-left [simp] add-diff-assoc [simp] add-diff-assoc2[simp]

```

At present we prove no analogue of *not-less-Least* or *Least-Suc*, since there appears to be no need.

## 15.9 The divides relation on nat

**lemma** *dvd-1-left* [iff]:  $\text{Suc } 0 \text{ dvd } k$   
 unfolding *dvd-def* by simp

**lemma** *dvd-1-iff-1* [simp]:  $(m \text{ dvd } \text{Suc } 0) = (m = \text{Suc } 0)$   
 by (simp add: *dvd-def*)

**lemma** *nat-dvd-1-iff-1* [simp]:  $m \text{ dvd } (1::\text{nat}) \longleftrightarrow m = 1$   
 by (simp add: *dvd-def*)

```

lemma dvd-antisym: [| m dvd n; n dvd m |] ==> m = (n::nat)
  unfolding dvd-def
  by (force dest: mult-eq-self-implies-10 simp add: mult-assoc)

```

*op dvd* is a partial order

```

interpretation dvd: order op dvd λn m :: nat. n dvd m ∧ ¬ m dvd n
  proof qed (auto intro: dvd-refl dvd-trans dvd-antisym)

```

```

lemma dvd-diff-nat[simp]: [| k dvd m; k dvd n |] ==> k dvd (m-n :: nat)
unfolding dvd-def
by (blast intro: diff-mult-distrib2 [symmetric])

```

```

lemma dvd-diffD: [| k dvd m-n; k dvd n; n ≤ m |] ==> k dvd (m::nat)
  apply (erule linorder-not-less [THEN iffD2, THEN add-diff-inverse, THEN
subst])
  apply (blast intro: dvd-add)
  done

```

```

lemma dvd-diffD1: [| k dvd m-n; k dvd m; n ≤ m |] ==> k dvd (n::nat)
by (drule-tac m = m in dvd-diff-nat, auto)

```

```

lemma dvd-reduce: (k dvd n + k) = (k dvd (n::nat))
  apply (rule iffI)
  apply (erule-tac [2] dvd-add)
  apply (rule-tac [2] dvd-refl)
  apply (subgoal-tac n = (n+k) -k)
  prefer 2 apply simp
  apply (erule ssubst)
  apply (erule dvd-diff-nat)
  apply (rule dvd-refl)
  done

```

```

lemma dvd-mult-cancel: !!k::nat. [| k*m dvd k*n; 0 < k |] ==> m dvd n
  unfolding dvd-def
  apply (erule exE)
  apply (simp add: mult-ac)
  done

```

```

lemma dvd-mult-cancel1: 0 < m ==> (m*n dvd m) = (n = (1::nat))
  apply auto
  apply (subgoal-tac m*n dvd m*1)
  apply (drule dvd-mult-cancel, auto)
  done

```

```

lemma dvd-mult-cancel2: 0 < m ==> (n*m dvd m) = (n = (1::nat))
  apply (subst mult-commute)
  apply (erule dvd-mult-cancel1)
  done

```

```
lemma dvd-imp-le: [| k dvd n; 0 < n |] ==> k ≤ (n::nat)
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)
```

```
lemma nat-dvd-not-less:
  fixes m n :: nat
  shows 0 < m ==> m < n ==> ¬ n dvd m
by (auto elim!: dvdE) (auto simp add: gr0-conv-Suc)
```

### 15.10 size of a datatype value

```
class size =
  fixes size :: 'a ⇒ nat — see further theory Wellfounded
```

### 15.11 code module namespace

```
code-modulename SML
  Nat Arith
```

```
code-modulename OCaml
  Nat Arith
```

```
code-modulename Haskell
  Nat Arith
```

```
end
```

## 16 Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

```
theory Datatype
imports Product-Type Sum-Type Nat
uses
  (Tools/Datatype/datatype.ML)
  (Tools/inductive-realizer.ML)
  (Tools/Datatype/datatype-realizer.ML)
begin
```

### 16.1 The datatype universe

```
typedef (Node)
  ('a,'b) node = {p. EX f x k. p = (f::nat=>'b+nat, x::'a+nat) & f k = Inr 0}
  — it is a subtype of (nat=>'b+nat) * ('a+nat)
by auto
```

Datatypes will be represented by sets of type *node*

```
types 'a item = ('a, unit) node set
```

$$('a, 'b) \text{ dtree} = ('a, 'b) \text{ node set}$$

**consts**

$$\text{Push} \quad :: [('b + \text{nat}), \text{nat} \Rightarrow ('b + \text{nat})] \Rightarrow (\text{nat} \Rightarrow ('b + \text{nat}))$$

$$\text{Push-Node} \quad :: [('b + \text{nat}), ('a, 'b) \text{ node}] \Rightarrow ('a, 'b) \text{ node}$$

$$\text{ndepth} \quad :: ('a, 'b) \text{ node} \Rightarrow \text{nat}$$

$$\text{Atom} \quad :: ('a + \text{nat}) \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{Leaf} \quad :: 'a \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{Numb} \quad :: \text{nat} \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{Scons} \quad :: [('a, 'b) \text{ dtree}, ('a, 'b) \text{ dtree}] \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{In0} \quad :: ('a, 'b) \text{ dtree} \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{In1} \quad :: ('a, 'b) \text{ dtree} \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{Lim} \quad :: ('b \Rightarrow ('a, 'b) \text{ dtree}) \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{ntrunc} \quad :: [\text{nat}, ('a, 'b) \text{ dtree}] \Rightarrow ('a, 'b) \text{ dtree}$$

$$\text{uprod} \quad :: [('a, 'b) \text{ dtree set}, ('a, 'b) \text{ dtree set}] \Rightarrow ('a, 'b) \text{ dtree set}$$

$$\text{usum} \quad :: [('a, 'b) \text{ dtree set}, ('a, 'b) \text{ dtree set}] \Rightarrow ('a, 'b) \text{ dtree set}$$

$$\text{Split} \quad :: [[('a, 'b) \text{ dtree}, ('a, 'b) \text{ dtree}] \Rightarrow 'c, ('a, 'b) \text{ dtree}] \Rightarrow 'c$$

$$\text{Case} \quad :: [[('a, 'b) \text{ dtree}] \Rightarrow 'c, [('a, 'b) \text{ dtree}] \Rightarrow 'c, ('a, 'b) \text{ dtree}] \Rightarrow 'c$$

$$\text{dprod} \quad :: [((('a, 'b) \text{ dtree} * ('a, 'b) \text{ dtree}) \text{ set}, ((('a, 'b) \text{ dtree} * ('a, 'b) \text{ dtree}) \text{ set}) \Rightarrow ((('a, 'b) \text{ dtree} * ('a, 'b) \text{ dtree}) \text{ set})$$

$$\text{dsum} \quad :: [((('a, 'b) \text{ dtree} * ('a, 'b) \text{ dtree}) \text{ set}, ((('a, 'b) \text{ dtree} * ('a, 'b) \text{ dtree}) \text{ set}) \Rightarrow ((('a, 'b) \text{ dtree} * ('a, 'b) \text{ dtree}) \text{ set})$$

**defs**

$$\text{Push-Node-def: } \text{Push-Node} == (\%n \ x. \text{Abs-Node} (\text{apfst} (\text{Push} \ n) (\text{Rep-Node} \ x)))$$

$$\text{Push-def: } \text{Push} == (\%b \ h. \text{nat-case} \ b \ h)$$

$$\text{Atom-def: } \text{Atom} == (\%x. \{\text{Abs-Node}((\%k. \text{Inr} \ 0, \ x))\})$$

$$\text{Scons-def: } \text{Scons} \ M \ N == (\text{Push-Node} (\text{Inr} \ 1) \text{ ‘ } M) \ \text{Un} \ (\text{Push-Node} (\text{Inr} \ (\text{Suc} \ 1)) \text{ ‘ } N)$$

$$\text{Leaf-def: } \text{Leaf} == \text{Atom} \ o \ \text{Inl}$$

$$\text{Numb-def: } \text{Numb} == \text{Atom} \ o \ \text{Inr}$$



*In0-def:*  $In0(M) == Scons (Numb\ 0)\ M$

*In1-def:*  $In1(M) == Scons (Numb\ 1)\ M$

*Lim-def:*  $Lim\ f == Union\ \{z.\ ?\ x.\ z = PushNode\ (Inl\ x)\ '\ (f\ x)\}$

*ndepth-def:*  $ndepth(n) == (\%(f,x).\ LEAST\ k.\ f\ k = Inr\ 0)\ (RepNode\ n)$

*ntrunc-def:*  $ntrunc\ k\ N == \{n.\ n:N\ \&\ ndepth(n) < k\}$

*uprod-def:*  $uprod\ A\ B == UN\ x:A.\ UN\ y:B.\ \{Scons\ x\ y\}$

*usum-def:*  $usum\ A\ B == In0'A\ Un\ In1'B$

*Split-def:*  $Split\ c\ M == THE\ u.\ EX\ x\ y.\ M = Scons\ x\ y\ \&\ u = c\ x\ y$

*Case-def:*  $Case\ c\ d\ M == THE\ u.\ (EX\ x.\ M = In0(x)\ \&\ u = c(x))$   
 $\quad\quad\quad | (EX\ y.\ M = In1(y)\ \&\ u = d(y))$

*dprod-def:*  $dprod\ r\ s == UN\ (x,x'):r.\ UN\ (y,y'):s.\ \{(Scons\ x\ y,\ Scons\ x'\ y')\}$

*dsum-def:*  $dsum\ r\ s == (UN\ (x,x'):r.\ \{(In0(x),In0(x'))\})\ Un$   
 $\quad\quad\quad (UN\ (y,y'):s.\ \{(In1(y),In1(y'))\})$

**lemma** *apfst-convE:*

$\llbracket q = apfst\ f\ p;\ !x\ y.\ \llbracket p = (x,y);\ q = (f(x),y)\ \rrbracket ==>\ R$   
 $\llbracket \rrbracket ==>\ R$

**by** (*force simp add: apfst-def*)

**lemma** *Push-inject1:*  $Push\ i\ f = Push\ j\ g ==> i=j$

**apply** (*simp add: Push-def expand-fun-eq*)

**apply** (*drule-tac x=0 in spec, simp*)

**done**

**lemma** *Push-inject2:*  $Push\ i\ f = Push\ j\ g ==> f=g$

**apply** (*auto simp add: Push-def expand-fun-eq*)

**apply** (*drule-tac x=Suc x in spec, simp*)

**done**

**lemma** *Push-inject:*

$\llbracket \text{Push } i \ f = \text{Push } j \ g; \llbracket i=j; \ f=g \rrbracket \implies P \rrbracket \implies P$   
**by** (*blast dest: Push-inject1 Push-inject2*)

**lemma** *Push-neq-K0*:  $\text{Push } (\text{Inr } (\text{Suc } k)) \ f = (\%z. \text{Inr } 0) \implies P$   
**by** (*auto simp add: Push-def expand-fun-eq split: nat.split-asm*)

**lemmas** *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [2] *rev-iffD1, standard*]

**lemma** *Node-K0-I*:  $(\%k. \text{Inr } 0, \ a) : \text{Node}$   
**by** (*simp add: Node-def*)

**lemma** *Node-Push-I*:  $p : \text{Node} \implies \text{apfst } (\text{Push } i) \ p : \text{Node}$   
**apply** (*simp add: Node-def Push-def*)  
**apply** (*fast intro!: apfst-conv nat-case-Suc [THEN trans]*)  
**done**

## 16.2 Freeness: Distinctness of Constructors

**lemma** *Scons-not-Atom* [*iff*]:  $\text{Scons } M \ N \neq \text{Atom}(a)$   
**unfolding** *Atom-def Scons-def Push-Node-def One-nat-def*  
**by** (*blast intro: Node-K0-I Rep-Node [THEN Node-Push-I]*  
*dest!: Abs-Node-inj*  
*elim!: apfst-convE sym [THEN Push-neq-K0]*)

**lemmas** *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym, standard*]

**lemma** *inj-Atom*:  $\text{inj}(\text{Atom})$   
**apply** (*simp add: Atom-def*)  
**apply** (*blast intro!: inj-onI Node-K0-I dest!: Abs-Node-inj*)  
**done**  
**lemmas** *Atom-inject* = *inj-Atom* [*THEN injD, standard*]

**lemma** *Atom-Atom-eq* [*iff*]:  $(\text{Atom}(a) = \text{Atom}(b)) = (a = b)$   
**by** (*blast dest!: Atom-inject*)

**lemma** *inj-Leaf*:  $\text{inj}(\text{Leaf})$   
**apply** (*simp add: Leaf-def o-def*)  
**apply** (*rule inj-onI*)  
**apply** (*erule Atom-inject [THEN Inl-inject]*)  
**done**

**lemmas** *Leaf-inject* [*dest!*] = *inj-Leaf* [*THEN injD, standard*]

**lemma** *inj-Numb*: *inj(Numb)*  
**apply** (*simp add: Numb-def o-def*)  
**apply** (*rule inj-onI*)  
**apply** (*erule Atom-inject [THEN Inr-inject]*)  
**done**

**lemmas** *Numb-inject* [*dest!*] = *inj-Numb* [*THEN injD, standard*]

**lemma** *Push-Node-inject*:  

$$[[ \text{Push-Node } i \ m = \text{Push-Node } j \ n; \ [i=j; \ m=n] ] ==> P]$$
  

$$]] ==> P$$
  
**apply** (*simp add: Push-Node-def*)  
**apply** (*erule Abs-Node-inj [THEN apfst-convE]*)  
**apply** (*rule Rep-Node [THEN Node-Push-I]*) +  
**apply** (*erule sym [THEN apfst-convE]*)  
**apply** (*blast intro: Rep-Node-inject [THEN iffD1] trans sym elim!: Push-inject*)  
**done**

**lemma** *Scons-inject-lemma1*: *Scons M N <= Scons M' N' ==> M <= M'*  
**unfolding** *Scons-def One-nat-def*  
**by** (*blast dest!: Push-Node-inject*)

**lemma** *Scons-inject-lemma2*: *Scons M N <= Scons M' N' ==> N <= N'*  
**unfolding** *Scons-def One-nat-def*  
**by** (*blast dest!: Push-Node-inject*)

**lemma** *Scons-inject1*: *Scons M N = Scons M' N' ==> M = M'*  
**apply** (*erule equalityE*)  
**apply** (*iprover intro: equalityI Scons-inject-lemma1*)  
**done**

**lemma** *Scons-inject2*: *Scons M N = Scons M' N' ==> N = N'*  
**apply** (*erule equalityE*)  
**apply** (*iprover intro: equalityI Scons-inject-lemma2*)  
**done**

**lemma** *Scons-inject*:  

$$[[ \text{Scons } M \ N = \text{Scons } M' \ N'; \ [M=M'; \ N=N'] ] ==> P ] ==> P$$
  
**by** (*iprover dest: Scons-inject1 Scons-inject2*)

**lemma** *Scons-Scons-eq [iff]*:  $(\text{Scons } M \ N = \text{Scons } M' \ N') = (M=M' \ \& \ N=N')$

**by** (*blast elim!*: *Scons-inject*)

**lemma** *Scons-not-Leaf* [*iff*]: *Scons M N*  $\neq$  *Leaf(a)*  
**unfolding** *Leaf-def o-def* **by** (*rule Scons-not-Atom*)

**lemmas** *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym, standard*]

**lemma** *Scons-not-Numb* [*iff*]: *Scons M N*  $\neq$  *Numb(k)*  
**unfolding** *Numb-def o-def* **by** (*rule Scons-not-Atom*)

**lemmas** *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym, standard*]

**lemma** *Leaf-not-Numb* [*iff*]: *Leaf(a)*  $\neq$  *Numb(k)*  
**by** (*simp add: Leaf-def Numb-def*)

**lemmas** *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym, standard*]

**lemma** *ndepth-K0*: *ndepth (Abs-Node(%k. Inr 0, x))* = 0  
**by** (*simp add: ndepth-def Node-K0-I [THEN Abs-Node-inverse] Least-equality*)

**lemma** *ndepth-Push-Node-aux*:  
*nat-case (Inr (Suc i)) f k = Inr 0 --> Suc(LEAST x. f x = Inr 0) <= k*  
**apply** (*induct-tac k, auto*)  
**apply** (*erule Least-le*)  
**done**

**lemma** *ndepth-Push-Node*:  
*ndepth (Push-Node (Inr (Suc i)) n) = Suc(ndepth(n))*  
**apply** (*insert Rep-Node [of n, unfolded Node-def]*)  
**apply** (*auto simp add: ndepth-def Push-Node-def*  
*Rep-Node [THEN Node-Push-I, THEN Abs-Node-inverse]*)  
**apply** (*rule Least-equality*)  
**apply** (*auto simp add: Push-def ndepth-Push-Node-aux*)  
**apply** (*erule LeastI*)  
**done**

**lemma** *ntrunc-0* [*simp*]: *ntrunc* 0 *M* = {}  
**by** (*simp add: ntrunc-def*)

**lemma** *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc* *k*) (*Atom* *a*) = *Atom*(*a*)  
**by** (*auto simp add: Atom-def ntrunc-def ndepth-K0*)

**lemma** *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc* *k*) (*Leaf* *a*) = *Leaf*(*a*)  
**unfolding** *Leaf-def o-def* **by** (*rule ntrunc-Atom*)

**lemma** *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc* *k*) (*Numb* *i*) = *Numb*(*i*)  
**unfolding** *Numb-def o-def* **by** (*rule ntrunc-Atom*)

**lemma** *ntrunc-Scons* [*simp*]:  
*ntrunc* (*Suc* *k*) (*Scons* *M* *N*) = *Scons* (*ntrunc* *k* *M*) (*ntrunc* *k* *N*)  
**unfolding** *Scons-def ntrunc-def One-nat-def*  
**by** (*auto simp add: ndepth-Push-Node*)

**lemma** *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc* 0) (*In0* *M*) = {}  
**apply** (*simp add: In0-def*)  
**apply** (*simp add: Scons-def*)  
**done**

**lemma** *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc* *k*)) (*In0* *M*) = *In0* (*ntrunc* (*Suc* *k*)  
*M*)  
**by** (*simp add: In0-def*)

**lemma** *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc* 0) (*In1* *M*) = {}  
**apply** (*simp add: In1-def*)  
**apply** (*simp add: Scons-def*)  
**done**

**lemma** *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc* *k*)) (*In1* *M*) = *In1* (*ntrunc* (*Suc* *k*)  
*M*)  
**by** (*simp add: In1-def*)

### 16.3 Set Constructions

**lemma** *uprodI* [*intro!*]: [| *M*:*A*; *N*:*B* |] ==> *Scons* *M* *N* : *uprod* *A* *B*  
**by** (*simp add: uprod-def*)

**lemma** *uprodE* [*elim!*]:  
[| *c* : *uprod* *A* *B*;

$$\begin{aligned} & !!x\ y. \llbracket x:A; \ y:B; \ c = Scons\ x\ y \rrbracket ==> P \\ & \llbracket \rrbracket ==> P \\ \text{by } & (auto\ simp\ add: uprod-def) \end{aligned}$$

**lemma** *uprodE2*:  $\llbracket Scons\ M\ N : uprod\ A\ B; \llbracket M:A; \ N:B \rrbracket ==> P \rrbracket ==> P$   
**by** (*auto simp add: uprod-def*)

**lemma** *usum-In0I* [*intro*]:  $M:A ==> In0(M) : usum\ A\ B$   
**by** (*simp add: usum-def*)

**lemma** *usum-In1I* [*intro*]:  $N:B ==> In1(N) : usum\ A\ B$   
**by** (*simp add: usum-def*)

**lemma** *usumE* [*elim!*]:  

$$\begin{aligned} & \llbracket u : usum\ A\ B; \\ & \quad !!x. \llbracket x:A; \ u=In0(x) \rrbracket ==> P; \\ & \quad !!y. \llbracket y:B; \ u=In1(y) \rrbracket ==> P \\ & \rrbracket ==> P \end{aligned}$$
  
**by** (*auto simp add: usum-def*)

**lemma** *In0-not-In1* [*iff*]:  $In0(M) \neq In1(N)$   
**unfolding** *In0-def In1-def One-nat-def* **by** *auto*

**lemmas** *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym, standard*]

**lemma** *In0-inject*:  $In0(M) = In0(N) ==> M=N$   
**by** (*simp add: In0-def*)

**lemma** *In1-inject*:  $In1(M) = In1(N) ==> M=N$   
**by** (*simp add: In1-def*)

**lemma** *In0-eq* [*iff*]:  $(In0\ M = In0\ N) = (M=N)$   
**by** (*blast dest!: In0-inject*)

**lemma** *In1-eq* [*iff*]:  $(In1\ M = In1\ N) = (M=N)$   
**by** (*blast dest!: In1-inject*)

**lemma** *inj-In0*: *inj In0*  
**by** (*blast intro!: inj-onI*)

**lemma** *inj-In1*: *inj In1*

**by** (*blast intro!*: *inj-onI*)

**lemma** *Lim-inject*:  $\text{Lim } f = \text{Lim } g \implies f = g$   
**apply** (*simp add*: *Lim-def*)  
**apply** (*rule ext*)  
**apply** (*blast elim!*: *Push-Node-inject*)  
**done**

**lemma** *ntrunc-subsetI*:  $\text{ntrunc } k \ M \leq M$   
**by** (*auto simp add*: *ntrunc-def*)

**lemma** *ntrunc-subsetD*:  $(!!k. \text{ntrunc } k \ M \leq N) \implies M \leq N$   
**by** (*auto simp add*: *ntrunc-def*)

**lemma** *ntrunc-equality*:  $(!!k. \text{ntrunc } k \ M = \text{ntrunc } k \ N) \implies M = N$   
**apply** (*rule equalityI*)  
**apply** (*rule-tac* [!] *ntrunc-subsetD*)  
**apply** (*rule-tac* [!] *ntrunc-subsetI* [*THEN* [2] *subset-trans*], *auto*)  
**done**

**lemma** *ntrunc-o-equality*:  
 $[!k. (\text{ntrunc}(k) \ o \ h1) = (\text{ntrunc}(k) \ o \ h2)] \implies h1 = h2$   
**apply** (*rule ntrunc-equality* [*THEN ext*])  
**apply** (*simp add*: *expand-fun-eq*)  
**done**

**lemma** *uprod-mono*:  $[! A \leq A'; B \leq B'] \implies \text{uprod } A \ B \leq \text{uprod } A' \ B'$   
**by** (*simp add*: *uprod-def*, *blast*)

**lemma** *usum-mono*:  $[! A \leq A'; B \leq B'] \implies \text{usum } A \ B \leq \text{usum } A' \ B'$   
**by** (*simp add*: *usum-def*, *blast*)

**lemma** *Scons-mono*:  $[! M \leq M'; N \leq N'] \implies \text{Scons } M \ N \leq \text{Scons } M' \ N'$   
**by** (*simp add*: *Scons-def*, *blast*)

**lemma** *In0-mono*:  $M \leq N \implies \text{In0}(M) \leq \text{In0}(N)$   
**by** (*simp add*: *In0-def Scons-mono*)

**lemma** *In1-mono*:  $M \leq N \implies \text{In1}(M) \leq \text{In1}(N)$

**by** (*simp add: In1-def Scons-mono*)

**lemma** *Split* [*simp*]: *Split* *c* (*Scons* *M* *N*) = *c* *M* *N*  
**by** (*simp add: Split-def*)

**lemma** *Case-In0* [*simp*]: *Case* *c* *d* (*In0* *M*) = *c*(*M*)  
**by** (*simp add: Case-def*)

**lemma** *Case-In1* [*simp*]: *Case* *c* *d* (*In1* *N*) = *d*(*N*)  
**by** (*simp add: Case-def*)

**lemma** *ntrunc-UN1*: *ntrunc* *k* (*UN* *x*. *f*(*x*)) = (*UN* *x*. *ntrunc* *k* (*f* *x*))  
**by** (*simp add: ntrunc-def, blast*)

**lemma** *Scons-UN1-x*: *Scons* (*UN* *x*. *f* *x*) *M* = (*UN* *x*. *Scons* (*f* *x*) *M*)  
**by** (*simp add: Scons-def, blast*)

**lemma** *Scons-UN1-y*: *Scons* *M* (*UN* *x*. *f* *x*) = (*UN* *x*. *Scons* *M* (*f* *x*))  
**by** (*simp add: Scons-def, blast*)

**lemma** *In0-UN1*: *In0*(*UN* *x*. *f*(*x*)) = (*UN* *x*. *In0*(*f*(*x*)))  
**by** (*simp add: In0-def Scons-UN1-y*)

**lemma** *In1-UN1*: *In1*(*UN* *x*. *f*(*x*)) = (*UN* *x*. *In1*(*f*(*x*)))  
**by** (*simp add: In1-def Scons-UN1-y*)

**lemma** *dprodI* [*intro!*]:  

$$[[ (M, M'):r; (N, N'):s ]] ==> (Scons\ M\ N, Scons\ M'\ N') : dprod\ r\ s$$
**by** (*auto simp add: dprod-def*)

**lemma** *dprodE* [*elim!*]:  

$$[[ c : dprod\ r\ s; \\ !!x\ y\ x'\ y'. [[ (x, x') : r; (y, y') : s; \\ c = (Scons\ x\ y, Scons\ x'\ y') ]] ==> P \\ ]] ==> P$$
**by** (*auto simp add: dprod-def*)



**lemma** *dsum-In0I* [intro]:  $(M, M') : r ==> (In0(M), In0(M')) : dsum\ r\ s$   
**by** (auto simp add: dsum-def)

**lemma** *dsum-In1I* [intro]:  $(N, N') : s ==> (In1(N), In1(N')) : dsum\ r\ s$   
**by** (auto simp add: dsum-def)

**lemma** *dsumE* [elim!]:  

$$\begin{aligned} &[[\ w : dsum\ r\ s; \\ &\quad !!x\ x'.\ [[\ (x, x') : r;\ w = (In0(x), In0(x'))\ ]\ ] ==> P; \\ &\quad !!y\ y'.\ [[\ (y, y') : s;\ w = (In1(y), In1(y'))\ ]\ ] ==> P \\ &\quad ]\ ] ==> P \end{aligned}$$
  
**by** (auto simp add: dsum-def)

**lemma** *dprod-mono*:  $[[\ r <= r';\ s <= s'\ ]\ ] ==> dprod\ r\ s <= dprod\ r'\ s'$   
**by** blast

**lemma** *dsum-mono*:  $[[\ r <= r';\ s <= s'\ ]\ ] ==> dsum\ r\ s <= dsum\ r'\ s'$   
**by** blast

**lemma** *dprod-Sigma*:  $(dprod\ (A\ <*>\ B)\ (C\ <*>\ D)) <= (uprod\ A\ C)\ <*>\ (uprod\ B\ D)$   
**by** blast

**lemmas** *dprod-subset-Sigma* = subset-trans [OF dprod-mono dprod-Sigma, standard]

**lemma** *dprod-subset-Sigma2*:  

$$(dprod\ (Sigma\ A\ B)\ (Sigma\ C\ D)) <=$$
  

$$Sigma\ (uprod\ A\ C)\ (Split\ (\%x\ y.\ uprod\ (B\ x)\ (D\ y)))$$
  
**by** auto

**lemma** *dsum-Sigma*:  $(dsum\ (A\ <*>\ B)\ (C\ <*>\ D)) <= (usum\ A\ C)\ <*>\ (usum\ B\ D)$   
**by** blast

**lemmas** *dsum-subset-Sigma* = subset-trans [OF dsum-mono dsum-Sigma, standard]

hides popular names

**hide-type** (open) node item

**hide-const** (**open**) *Push Node Atom Leaf Numb Lim Split Case*

**use** *Tools/Datatype/datatype.ML*

**use** *Tools/inductive-realizer.ML*

**setup** *InductiveRealizer.setup*

**use** *Tools/Datatype/datatype-realizer.ML*

**setup** *Datatype-Realizer.setup*

**end**

## 17 Record: Extensible records with structural subtyping

**theory** *Record*

**imports** *Datatype*

**uses** (*Tools/record.ML*)

**begin**

### 17.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification  $\alpha (beta\text{-update } f \text{ rec}) = \alpha \text{ rec}$  for distinct fields  $\alpha$  and  $\beta$  of some record  $\text{rec}$  with  $n$  fields. There are  $n^2$  such theorems, which prohibits storage of all of them for large  $n$ . The rules can be proved on the fly by case decomposition and simplification in  $O(n)$  time. By creating  $O(n)$  isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in  $O(\log(n)^2)$  time.

The  $O(n)$  cost of case decomposition is not because  $O(n)$  steps are taken, but rather because the resulting rule must contain  $O(n)$  new variables and an  $O(n)$  size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields  $'a$ ,  $'b$ ,  $'c$  and  $'d$  might be introduced as isomorphic to  $'a \times ('b \times ('c \times 'd))$ . If we balance the tuple tree to  $('a \times 'b) \times ('c \times 'd)$  then accessors can be defined by converting to the underlying type then using  $O(\log(n))$  *fst* or *snd* operations. Updaters can be defined similarly, if we introduce a *fst-update* and *snd-update* function. Furthermore, we can prove the access/update theorem in  $O(\log(n))$  steps by using simple rewrites on *fst*, *snd*, *fst-update* and *snd-update*.

The catch is that, although  $O(\log(n))$  steps were taken, the underlying type we converted to is a tuple tree of size  $O(n)$ . Processing this term type wastes

performance. We avoid this for large  $n$  by taking each subtree of size  $K$  and defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these  $O(n/K)$  new types, or, if  $n > K * K$ , we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant  $K$ .

If we prove the access/update theorem on this type with the analagous steps to the tuple tree, we consume  $O(\log(n)^2)$  time as the intermediate terms are  $O(\log(n))$  in size and the types needed have size bounded by  $K$ . To enable this analagous traversal, we define the functions seen below: *iso-tuple-fst*, *iso-tuple-snd*, *iso-tuple-fst-update* and *iso-tuple-snd-update*. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule *isomorphic-tuple.intros* rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

## 17.2 Operators and lemmas for types isomorphic to tuples

**datatype** ( $'a$ ,  $'b$ ,  $'c$ ) *tuple-isomorphism* =  
*Tuple-Isomorphism*  $'a \Rightarrow 'b \times 'c$   $'b \times 'c \Rightarrow 'a$

**primrec**

*repr* :: ( $'a$ ,  $'b$ ,  $'c$ ) *tuple-isomorphism*  $\Rightarrow 'a \Rightarrow 'b \times 'c$  **where**  
*repr* (*Tuple-Isomorphism*  $r$   $a$ ) =  $r$

**primrec**

*abst* :: ( $'a$ ,  $'b$ ,  $'c$ ) *tuple-isomorphism*  $\Rightarrow 'b \times 'c \Rightarrow 'a$  **where**  
*abst* (*Tuple-Isomorphism*  $r$   $a$ ) =  $a$

**definition**

*iso-tuple-fst* :: ( $'a$ ,  $'b$ ,  $'c$ ) *tuple-isomorphism*  $\Rightarrow 'a \Rightarrow 'b$  **where**  
*iso-tuple-fst isom* = *fst*  $\circ$  *repr isom*

**definition**

*iso-tuple-snd* :: ( $'a$ ,  $'b$ ,  $'c$ ) *tuple-isomorphism*  $\Rightarrow 'a \Rightarrow 'c$  **where**  
*iso-tuple-snd isom* = *snd*  $\circ$  *repr isom*

**definition**

*iso-tuple-fst-update* ::  
 $( 'a, 'b, 'c )$  *tuple-isomorphism*  $\Rightarrow ( 'b \Rightarrow 'b ) \Rightarrow ( 'a \Rightarrow 'a )$  **where**

*iso-tuple-fst-update isom*  $f = \text{abst isom} \circ \text{apfst } f \circ \text{repr isom}$

**definition**

*iso-tuple-snd-update* ::  
 (*'a*, *'b*, *'c*) *tuple-isomorphism*  $\Rightarrow$  (*'c*  $\Rightarrow$  *'c*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'a*) **where**  
*iso-tuple-snd-update isom*  $f = \text{abst isom} \circ \text{apsnd } f \circ \text{repr isom}$

**definition**

*iso-tuple-cons* ::  
 (*'a*, *'b*, *'c*) *tuple-isomorphism*  $\Rightarrow$  *'b*  $\Rightarrow$  *'c*  $\Rightarrow$  *'a* **where**  
*iso-tuple-cons isom*  $= \text{curry } (\text{abst isom})$

### 17.3 Logical infrastructure for records

**definition**

*iso-tuple-surjective-proof-assist* :: *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *bool* **where**  
*iso-tuple-surjective-proof-assist*  $x \ y \ f \longleftrightarrow f \ x = y$

**definition**

*iso-tuple-update-accessor-cong-assist* ::  
 ((*'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'a*))  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *bool* **where**  
*iso-tuple-update-accessor-cong-assist*  $\text{upd } \text{acc} \longleftrightarrow$   
 ( $\forall f \ v. \ \text{upd } (\lambda x. f \ (\text{acc } v)) \ v = \text{upd } f \ v$ )  $\wedge$  ( $\forall v. \ \text{upd } \text{id } v = v$ )

**definition**

*iso-tuple-update-accessor-eq-assist* ::  
 ((*'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  (*'a*  $\Rightarrow$  *'a*))  $\Rightarrow$  (*'a*  $\Rightarrow$  *'b*)  $\Rightarrow$  *'a*  $\Rightarrow$  (*'b*  $\Rightarrow$  *'b*)  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\Rightarrow$  *bool*  
**where**  
*iso-tuple-update-accessor-eq-assist*  $\text{upd } \text{acc } v \ f \ v' \ x \longleftrightarrow$   
 $\text{upd } f \ v = v' \wedge \text{acc } v = x \wedge \text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc}$

**lemma** *update-accessor-congruence-foldE*:

**assumes** *uac*: *iso-tuple-update-accessor-cong-assist*  $\text{upd } \text{acc}$   
**and**  $r: r = r'$  **and**  $v: \text{acc } r' = v'$   
**and**  $f: \bigwedge v. v' = v \implies f \ v = f' \ v$   
**shows**  $\text{upd } f \ r = \text{upd } f' \ r'$   
**using** *uac*  $r \ v$  [symmetric]  
**apply** (*subgoal-tac*  $\text{upd } (\lambda x. f \ (\text{acc } r')) \ r' = \text{upd } (\lambda x. f' \ (\text{acc } r')) \ r'$ )  
**apply** (*simp* *add*: *iso-tuple-update-accessor-cong-assist-def*)  
**apply** (*simp* *add*:  $f$ )  
**done**

**lemma** *update-accessor-congruence-unfoldE*:

*iso-tuple-update-accessor-cong-assist*  $\text{upd } \text{acc} \implies$   
 $r = r' \implies \text{acc } r' = v' \implies (\bigwedge v. v = v' \implies f \ v = f' \ v) \implies$   
 $\text{upd } f \ r = \text{upd } f' \ r'$   
**apply** (*erule*(2) *update-accessor-congruence-foldE*)  
**apply** *simp*  
**done**

**lemma** *iso-tuple-update-accessor-cong-assist-id*:

*iso-tuple-update-accessor-cong-assist upd acc  $\implies$  upd id = id*

**by** rule (simp add: iso-tuple-update-accessor-cong-assist-def)

**lemma** *update-accessor-noopE*:

**assumes** *uac*: *iso-tuple-update-accessor-cong-assist upd acc*

**and** *acc*: *f (acc x) = acc x*

**shows** *upd f x = x*

**using** *uac*

**by** (simp add: *acc iso-tuple-update-accessor-cong-assist-id* [OF *uac*, unfolded *id-def*])

*cong*: *update-accessor-congruence-unfoldE* [OF *uac*])

**lemma** *update-accessor-noop-compE*:

**assumes** *uac*: *iso-tuple-update-accessor-cong-assist upd acc*

**and** *acc*: *f (acc x) = acc x*

**shows** *upd (g  $\circ$  f) x = upd g x*

**by** (simp add: *acc cong*: *update-accessor-congruence-unfoldE*[OF *uac*])

**lemma** *update-accessor-cong-assist-idI*:

*iso-tuple-update-accessor-cong-assist id id*

**by** (simp add: iso-tuple-update-accessor-cong-assist-def)

**lemma** *update-accessor-cong-assist-triv*:

*iso-tuple-update-accessor-cong-assist upd acc  $\implies$*

*iso-tuple-update-accessor-cong-assist upd acc*

**by** assumption

**lemma** *update-accessor-accessor-eqE*:

*iso-tuple-update-accessor-eq-assist upd acc v f v' x  $\implies$  acc v = x*

**by** (simp add: iso-tuple-update-accessor-eq-assist-def)

**lemma** *update-accessor-updator-eqE*:

*iso-tuple-update-accessor-eq-assist upd acc v f v' x  $\implies$  upd f v = v'*

**by** (simp add: iso-tuple-update-accessor-eq-assist-def)

**lemma** *iso-tuple-update-accessor-eq-assist-idI*:

*v' = f v  $\implies$  iso-tuple-update-accessor-eq-assist id id v f v' v*

**by** (simp add: iso-tuple-update-accessor-eq-assist-def *update-accessor-cong-assist-idI*)

**lemma** *iso-tuple-update-accessor-eq-assist-triv*:

*iso-tuple-update-accessor-eq-assist upd acc v f v' x  $\implies$*

*iso-tuple-update-accessor-eq-assist upd acc v f v' x*

**by** assumption

**lemma** *iso-tuple-update-accessor-cong-from-eq*:

*iso-tuple-update-accessor-eq-assist upd acc v f v' x  $\implies$*

*iso-tuple-update-accessor-cong-assist upd acc*

**by** (*simp add: iso-tuple-update-accessor-eq-assist-def*)

**lemma** *iso-tuple-surjective-proof-assistI*:  
 $f\ x = y \implies \text{iso-tuple-surjective-proof-assist}\ x\ y\ f$   
**by** (*simp add: iso-tuple-surjective-proof-assist-def*)

**lemma** *iso-tuple-surjective-proof-assist-idE*:  
 $\text{iso-tuple-surjective-proof-assist}\ x\ y\ \text{id} \implies x = y$   
**by** (*simp add: iso-tuple-surjective-proof-assist-def*)

**locale** *isomorphic-tuple* =  
**fixes** *isom* :: ('a, 'b, 'c) *tuple-isomorphism*  
**assumes** *repr-inv*:  $\bigwedge x. \text{abst isom} (\text{repr isom } x) = x$   
**and** *abst-inv*:  $\bigwedge y. \text{repr isom} (\text{abst isom } y) = y$   
**begin**

**lemma** *repr-inj*:  $\text{repr isom } x = \text{repr isom } y \longleftrightarrow x = y$   
**by** (*auto dest: arg-cong [of repr isom x repr isom y abst isom]*  
*simp add: repr-inv*)

**lemma** *abst-inj*:  $\text{abst isom } x = \text{abst isom } y \longleftrightarrow x = y$   
**by** (*auto dest: arg-cong [of abst isom x abst isom y repr isom]*  
*simp add: abst-inv*)

**lemmas** *simps* = *Let-def repr-inv abst-inv repr-inj abst-inj*

**lemma** *iso-tuple-access-update-fst-fst*:  
 $f\ o\ h\ g = j\ o\ f \implies$   
 $(f\ o\ \text{iso-tuple-fst isom})\ o\ (\text{iso-tuple-fst-update isom } o\ h)\ g =$   
 $j\ o\ (f\ o\ \text{iso-tuple-fst isom})$   
**by** (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-fst-def simps*  
*intro!: ext elim!: o-eq-elim*)

**lemma** *iso-tuple-access-update-snd-snd*:  
 $f\ o\ h\ g = j\ o\ f \implies$   
 $(f\ o\ \text{iso-tuple-snd isom})\ o\ (\text{iso-tuple-snd-update isom } o\ h)\ g =$   
 $j\ o\ (f\ o\ \text{iso-tuple-snd isom})$   
**by** (*clarsimp simp: iso-tuple-snd-update-def iso-tuple-snd-def simps*  
*intro!: ext elim!: o-eq-elim*)

**lemma** *iso-tuple-access-update-fst-snd*:  
 $(f\ o\ \text{iso-tuple-fst isom})\ o\ (\text{iso-tuple-snd-update isom } o\ h)\ g =$   
 $\text{id } o\ (f\ o\ \text{iso-tuple-fst isom})$   
**by** (*clarsimp simp: iso-tuple-snd-update-def iso-tuple-fst-def simps*  
*intro!: ext elim!: o-eq-elim*)

**lemma** *iso-tuple-access-update-snd-fst*:  
 $(f\ o\ \text{iso-tuple-snd isom})\ o\ (\text{iso-tuple-fst-update isom } o\ h)\ g =$   
 $\text{id } o\ (f\ o\ \text{iso-tuple-snd isom})$

**by** (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-def\_simps*  
*intro!: ext elim!: o-eq-elim*)

**lemma** *iso-tuple-update-swap-fst-fst:*

$h \circ f \circ j \circ g = j \circ g \circ h \circ f \implies$   
 $(\text{iso-tuple-fst-update isom } o \ h) \circ f \circ (\text{iso-tuple-fst-update isom } o \ j) \circ g =$   
 $(\text{iso-tuple-fst-update isom } o \ j) \circ g \circ (\text{iso-tuple-fst-update isom } o \ h) \circ f$   
**by** (*clarsimp simp: iso-tuple-fst-update-def\_simps apfst-compose intro!: ext*)

**lemma** *iso-tuple-update-swap-snd-snd:*

$h \circ f \circ j \circ g = j \circ g \circ h \circ f \implies$   
 $(\text{iso-tuple-snd-update isom } o \ h) \circ f \circ (\text{iso-tuple-snd-update isom } o \ j) \circ g =$   
 $(\text{iso-tuple-snd-update isom } o \ j) \circ g \circ (\text{iso-tuple-snd-update isom } o \ h) \circ f$   
**by** (*clarsimp simp: iso-tuple-snd-update-def\_simps apsnd-compose intro!: ext*)

**lemma** *iso-tuple-update-swap-fst-snd:*

$(\text{iso-tuple-snd-update isom } o \ h) \circ f \circ (\text{iso-tuple-fst-update isom } o \ j) \circ g =$   
 $(\text{iso-tuple-fst-update isom } o \ j) \circ g \circ (\text{iso-tuple-snd-update isom } o \ h) \circ f$   
**by** (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-update-def*  
*simps intro!: ext*)

**lemma** *iso-tuple-update-swap-snd-fst:*

$(\text{iso-tuple-fst-update isom } o \ h) \circ f \circ (\text{iso-tuple-snd-update isom } o \ j) \circ g =$   
 $(\text{iso-tuple-snd-update isom } o \ j) \circ g \circ (\text{iso-tuple-fst-update isom } o \ h) \circ f$   
**by** (*clarsimp simp: iso-tuple-fst-update-def iso-tuple-snd-update-def\_simps intro!: ext*)

**lemma** *iso-tuple-update-compose-fst-fst:*

$h \circ f \circ j \circ g = k \circ (f \circ g) \implies$   
 $(\text{iso-tuple-fst-update isom } o \ h) \circ f \circ (\text{iso-tuple-fst-update isom } o \ j) \circ g =$   
 $(\text{iso-tuple-fst-update isom } o \ k) \circ (f \circ g)$   
**by** (*clarsimp simp: iso-tuple-fst-update-def\_simps apfst-compose intro!: ext*)

**lemma** *iso-tuple-update-compose-snd-snd:*

$h \circ f \circ j \circ g = k \circ (f \circ g) \implies$   
 $(\text{iso-tuple-snd-update isom } o \ h) \circ f \circ (\text{iso-tuple-snd-update isom } o \ j) \circ g =$   
 $(\text{iso-tuple-snd-update isom } o \ k) \circ (f \circ g)$   
**by** (*clarsimp simp: iso-tuple-snd-update-def\_simps apsnd-compose intro!: ext*)

**lemma** *iso-tuple-surjective-proof-assist-step:*

*iso-tuple-surjective-proof-assist v a (iso-tuple-fst isom o f)  $\implies$*   
*iso-tuple-surjective-proof-assist v b (iso-tuple-snd isom o f)  $\implies$*   
*iso-tuple-surjective-proof-assist v (iso-tuple-cons isom a b) f*  
**by** (*clarsimp simp: iso-tuple-surjective-proof-assist-def\_simps*  
*iso-tuple-fst-def iso-tuple-snd-def iso-tuple-cons-def*)

**lemma** *iso-tuple-fst-update-accessor-cong-assist:*

**assumes** *iso-tuple-update-accessor-cong-assist f g*  
**shows** *iso-tuple-update-accessor-cong-assist*

```

      (iso-tuple-fst-update isom o f) (g o iso-tuple-fst isom)
proof -
  from assms have f id = id
  by (rule iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
  by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def_simps
      iso-tuple-fst-update-def iso-tuple-fst-def)
qed

```

```

lemma iso-tuple-snd-update-accessor-cong-assist:
  assumes iso-tuple-update-accessor-cong-assist f g
  shows iso-tuple-update-accessor-cong-assist
    (iso-tuple-snd-update isom o f) (g o iso-tuple-snd isom)
proof -
  from assms have f id = id
  by (rule iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
  by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def_simps
      iso-tuple-snd-update-def iso-tuple-snd-def)
qed

```

```

lemma iso-tuple-fst-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g a u a' v
  shows iso-tuple-update-accessor-eq-assist
    (iso-tuple-fst-update isom o f) (g o iso-tuple-fst isom)
    (iso-tuple-cons isom a b) u (iso-tuple-cons isom a' b) v
proof -
  from assms have f id = id
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
      intro: iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
      iso-tuple-fst-update-def iso-tuple-fst-def
      iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def_simps)
qed

```

```

lemma iso-tuple-snd-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g b u b' v
  shows iso-tuple-update-accessor-eq-assist
    (iso-tuple-snd-update isom o f) (g o iso-tuple-snd isom)
    (iso-tuple-cons isom a b) u (iso-tuple-cons isom a b') v
proof -
  from assms have f id = id
  by (auto simp add: iso-tuple-update-accessor-eq-assist-def
      intro: iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
  by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
      iso-tuple-snd-update-def iso-tuple-snd-def
      iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def_simps)

```



qed

**lemma** *iso-tuple-cons-conj-eqI*:

$a = c \wedge b = d \wedge P \longleftrightarrow Q \implies$   
 $\text{iso-tuple-cons isom } a \ b = \text{iso-tuple-cons isom } c \ d \wedge P \longleftrightarrow Q$   
**by** (*clarsimp simp: iso-tuple-cons-def\_simps*)

**lemmas** *intros* =

*iso-tuple-access-update-fst-fst*  
*iso-tuple-access-update-snd-snd*  
*iso-tuple-access-update-fst-snd*  
*iso-tuple-access-update-snd-fst*  
*iso-tuple-update-swap-fst-fst*  
*iso-tuple-update-swap-snd-snd*  
*iso-tuple-update-swap-fst-snd*  
*iso-tuple-update-swap-snd-fst*  
*iso-tuple-update-compose-fst-fst*  
*iso-tuple-update-compose-snd-snd*  
*iso-tuple-surjective-proof-assist-step*  
*iso-tuple-fst-update-accessor-eq-assist*  
*iso-tuple-snd-update-accessor-eq-assist*  
*iso-tuple-fst-update-accessor-cong-assist*  
*iso-tuple-snd-update-accessor-cong-assist*  
*iso-tuple-cons-conj-eqI*

end

**lemma** *isomorphic-tuple-intro*:

**fixes** *repr abst*  
**assumes** *repr-inj*:  $\bigwedge x \ y. \text{repr } x = \text{repr } y \longleftrightarrow x = y$   
**and** *abst-inv*:  $\bigwedge z. \text{repr } (\text{abst } z) = z$   
**and** *v*:  $v \equiv \text{Tuple-Isomorphism repr abst}$   
**shows** *isomorphic-tuple v*

**proof**

**fix** *x* **have**  $\text{repr } (\text{abst } (\text{repr } x)) = \text{repr } x$   
**by** (*simp add: abst-inv*)  
**then show**  $\text{Record.abst } v \ (\text{Record.repr } v \ x) = x$   
**by** (*simp add: v repr-inj*)

**next**

**fix** *y*  
**show**  $\text{Record.repr } v \ (\text{Record.abst } v \ y) = y$   
**by** (*simp add: v (fact abst-inv)*)

qed

**definition**

*tuple-iso-tuple*  $\equiv \text{Tuple-Isomorphism id id}$

**lemma** *tuple-iso-tuple*:

*isomorphic-tuple tuple-iso-tuple*

**syntax** (*xsymbols*)

```

-record-type      :: field-types => type          ((3(|-|)))
-record-type-scheme :: field-types => type => type  ((3(|-,/ (2... ::/ -)|)))
-record          :: fields => 'a                  ((3(|-|)))
-record-scheme    :: fields => 'a => 'a            ((3(|-,/ (2... =/ -)|)))
-record-update    :: 'a => field-updates => 'b      (-/(3(|-|)) [900, 0] 900)

```

## 17.5 Record package

```

use Tools/record.ML
setup Record.setup

```

```

hide-const (open) Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd
iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons
iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist
iso-tuple-update-accessor-eq-assist tuple-iso-tuple

```

```
end
```

## 18 Power: Exponentiation

```

theory Power
imports Nat
begin

```

### 18.1 Powers for Arbitrary Monoids

```

class power = one + times
begin

```

```

primrec power :: 'a ⇒ nat ⇒ 'a (infixr ^ 80) where
  power-0: a ^ 0 = 1
| power-Suc: a ^ Suc n = a * a ^ n

```

```

notation (latex output)
power ((-) [1000] 1000)

```

```

notation (HTML output)
power ((-) [1000] 1000)

```

```
end
```

```

context monoid-mult
begin

```

```
subclass power ..
```

```

lemma power-one [simp]:
  1 ^ n = 1

```

```

by (induct n) simp-all

lemma power-one-right [simp]:
   $a ^ 1 = a$ 
by simp

lemma power-commutes:
   $a ^ n * a = a * a ^ n$ 
by (induct n) (simp-all add: mult-assoc)

lemma power-Suc2:
   $a ^ \text{Suc } n = a ^ n * a$ 
by (simp add: power-commutes)

lemma power-add:
   $a ^ (m + n) = a ^ m * a ^ n$ 
by (induct m) (simp-all add: algebra-simps)

lemma power-mult:
   $a ^ (m * n) = (a ^ m) ^ n$ 
by (induct n) (simp-all add: power-add)

end

context comm-monoid-mult
begin

lemma power-mult-distrib:
   $(a * b) ^ n = (a ^ n) * (b ^ n)$ 
by (induct n) (simp-all add: mult-ac)

end

context semiring-1
begin

lemma of-nat-power:
   $\text{of-nat } (m ^ n) = \text{of-nat } m ^ n$ 
by (induct n) (simp-all add: of-nat-mult)

end

context comm-semiring-1
begin

The divides relation

lemma le-imp-power-dvd:
  assumes  $m \leq n$  shows  $a ^ m \text{ dvd } a ^ n$ 
proof

```

```

have  $a ^ n = a ^ (m + (n - m))$ 
  using  $\langle m \leq n \rangle$  by simp
also have  $\dots = a ^ m * a ^ (n - m)$ 
  by (rule power-add)
finally show  $a ^ n = a ^ m * a ^ (n - m)$  .
qed

```

```

lemma power-le-dvd:
 $a ^ n \text{ dvd } b \implies m \leq n \implies a ^ m \text{ dvd } b$ 
  by (rule dvd-trans [OF le-imp-power-dvd])

```

```

lemma dvd-power-same:
 $x \text{ dvd } y \implies x ^ n \text{ dvd } y ^ n$ 
  by (induct n) (auto simp add: mult-dvd-mono)

```

```

lemma dvd-power-le:
 $x \text{ dvd } y \implies m \geq n \implies x ^ n \text{ dvd } y ^ m$ 
  by (rule power-le-dvd [OF dvd-power-same])

```

```

lemma dvd-power [simp]:
  assumes  $n > (0::nat) \vee x = 1$ 
  shows  $x \text{ dvd } (x ^ n)$ 
using assms proof
  assume  $0 < n$ 
  then have  $x ^ n = x ^ \text{Suc } (n - 1)$  by simp
  then show  $x \text{ dvd } (x ^ n)$  by simp
next
  assume  $x = 1$ 
  then show  $x \text{ dvd } (x ^ n)$  by simp
qed

```

end

```

context ring-1
begin

```

```

lemma power-minus:
 $(- a) ^ n = (- 1) ^ n * a ^ n$ 
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc n) then show ?case
    by (simp del: power-Suc add: power-Suc2 mult-assoc)
qed

```

end

```

context linordered-semidom
begin

```

**lemma** *zero-less-power* [*simp*]:

$0 < a \implies 0 < a ^ n$

**by** (*induct* *n*) (*simp-all* *add: mult-pos-pos*)

**lemma** *zero-le-power* [*simp*]:

$0 \leq a \implies 0 \leq a ^ n$

**by** (*induct* *n*) (*simp-all* *add: mult-nonneg-nonneg*)

**lemma** *one-le-power*[*simp*]:

$1 \leq a \implies 1 \leq a ^ n$

**apply** (*induct* *n*)

**apply** *simp-all*

**apply** (*rule* *order-trans* [*OF* - *mult-mono* [*of* 1 - 1]])

**apply** (*simp-all* *add: order-trans* [*OF* *zero-le-one*])

**done**

**lemma** *power-gt1-lemma*:

**assumes** *gt1*:  $1 < a$

**shows**  $1 < a * a ^ n$

**proof** –

**from** *gt1* **have**  $0 \leq a$

**by** (*fact* *order-trans* [*OF* *zero-le-one less-imp-le*])

**have**  $1 * 1 < a * 1$  **using** *gt1* **by** *simp*

**also have**  $\dots \leq a * a ^ n$  **using** *gt1*

**by** (*simp* *only: mult-mono* ( $0 \leq a$ ) *one-le-power* *order-less-imp-le*  
*zero-le-one* *order-refl*)

**finally show** *?thesis* **by** *simp*

**qed**

**lemma** *power-gt1*:

$1 < a \implies 1 < a ^ \text{Suc } n$

**by** (*simp* *add: power-gt1-lemma*)

**lemma** *one-less-power* [*simp*]:

$1 < a \implies 0 < n \implies 1 < a ^ n$

**by** (*cases* *n*) (*simp-all* *add: power-gt1-lemma*)

**lemma** *power-le-imp-le-exp*:

**assumes** *gt1*:  $1 < a$

**shows**  $a ^ m \leq a ^ n \implies m \leq n$

**proof** (*induct* *m* *arbitrary: n*)

**case** 0

**show** *?case* **by** *simp*

**next**

**case** (*Suc* *m*)

**show** *?case*

**proof** (*cases* *n*)

**case** 0

```

with Suc.prems Suc.hyps have  $a * a^m \leq 1$  by simp
with gt1 show ?thesis
  by (force simp only: power-gt1-lemma
      not-less [symmetric])
next
  case (Suc n)
  with Suc.prems Suc.hyps show ?thesis
  by (force dest: mult-left-le-imp-le
      simp add: less-trans [OF zero-less-one gt1])
qed
qed

```

Surely we can strengthen this? It holds for  $0 < a < 1$  too.

```

lemma power-inject-exp [simp]:
   $1 < a \implies a^m = a^n \iff m = n$ 
  by (force simp add: order-antisym power-le-imp-le-exp)

```

Can relax the first premise to  $(0::'a) < a$  in the case of the natural numbers.

```

lemma power-less-imp-less-exp:
   $1 < a \implies a^m < a^n \implies m < n$ 
  by (simp add: order-less-le [of m n] less-le [of a^m a^n]
      power-le-imp-le-exp)

```

```

lemma power-mono:
   $a \leq b \implies 0 \leq a \implies a^n \leq b^n$ 
  by (induct n)
  (auto intro: mult-mono order-trans [of 0 a b])

```

```

lemma power-strict-mono [rule-format]:
   $a < b \implies 0 \leq a \implies 0 < n \longrightarrow a^n < b^n$ 
  by (induct n)
  (auto simp add: mult-strict-mono le-less-trans [of 0 a b])

```

Lemma for *power-strict-decreasing*

```

lemma power-Suc-less:
   $0 < a \implies a < 1 \implies a * a^n < a^n$ 
  by (induct n)
  (auto simp add: mult-strict-left-mono)

```

```

lemma power-strict-decreasing [rule-format]:
   $n < N \implies 0 < a \implies a < 1 \longrightarrow a^N < a^n$ 

```

```

proof (induct N)
  case 0 then show ?case by simp
next
  case (Suc N) then show ?case
  apply (auto simp add: power-Suc-less less-Suc-eq)
  apply (subgoal-tac a * a^N < 1 * a^n)
  apply simp
  apply (rule mult-strict-mono) apply auto

```

done  
qed

Proof resembles that of *power-strict-decreasing*

**lemma** *power-decreasing* [rule-format]:  
 $n \leq N \implies 0 \leq a \implies a \leq 1 \longrightarrow a \wedge N \leq a \wedge n$   
**proof** (*induct N*)  
  **case** 0 **then show** ?case **by** *simp*  
**next**  
  **case** (*Suc N*) **then show** ?case  
  **apply** (*auto simp add: le-Suc-eq*)  
  **apply** (*subgoal-tac a \* a^N ≤ 1 \* a^n, simp*)  
  **apply** (*rule mult-mono*) **apply** *auto*  
  done  
**qed**

**lemma** *power-Suc-less-one*:  
 $0 < a \implies a < 1 \implies a \wedge \text{Suc } n < 1$   
  **using** *power-strict-decreasing* [*of 0 Suc n a*] **by** *simp*

Proof again resembles that of *power-strict-decreasing*

**lemma** *power-increasing* [rule-format]:  
 $n \leq N \implies 1 \leq a \implies a \wedge n \leq a \wedge N$   
**proof** (*induct N*)  
  **case** 0 **then show** ?case **by** *simp*  
**next**  
  **case** (*Suc N*) **then show** ?case  
  **apply** (*auto simp add: le-Suc-eq*)  
  **apply** (*subgoal-tac 1 \* a^n ≤ a \* a^N, simp*)  
  **apply** (*rule mult-mono*) **apply** (*auto simp add: order-trans [OF zero-le-one]*)  
  done  
**qed**

Lemma for *power-strict-increasing*

**lemma** *power-less-power-Suc*:  
 $1 < a \implies a \wedge n < a * a \wedge n$   
  **by** (*induct n*) (*auto simp add: mult-strict-left-mono less-trans [OF zero-less-one]*)

**lemma** *power-strict-increasing* [rule-format]:  
 $n < N \implies 1 < a \longrightarrow a \wedge n < a \wedge N$   
**proof** (*induct N*)  
  **case** 0 **then show** ?case **by** *simp*  
**next**  
  **case** (*Suc N*) **then show** ?case  
  **apply** (*auto simp add: power-less-power-Suc less-Suc-eq*)  
  **apply** (*subgoal-tac 1 \* a^n < a \* a^N, simp*)  
  **apply** (*rule mult-strict-mono*) **apply** (*auto simp add: less-trans [OF zero-less-one]*  
*less-imp-le*)  
  done



qed

**lemma** *power-increasing-iff* [simp]:

$$1 < b \implies b \wedge x \leq b \wedge y \iff x \leq y$$

**by** (blast intro: power-le-imp-le-exp power-increasing less-imp-le)

**lemma** *power-strict-increasing-iff* [simp]:

$$1 < b \implies b \wedge x < b \wedge y \iff x < y$$

**by** (blast intro: power-less-imp-less-exp power-strict-increasing)

**lemma** *power-le-imp-le-base*:

**assumes** *le*:  $a \wedge \text{Suc } n \leq b \wedge \text{Suc } n$

**and** *ynonneg*:  $0 \leq b$

**shows**  $a \leq b$

**proof** (rule ccontr)

**assume**  $\sim a \leq b$

**then have**  $b < a$  **by** (simp only: linorder-not-le)

**then have**  $b \wedge \text{Suc } n < a \wedge \text{Suc } n$

**by** (simp only: prems power-strict-mono)

**from** *le* **and this** **show** False

**by** (simp add: linorder-not-less [symmetric])

qed

**lemma** *power-less-imp-less-base*:

**assumes** *less*:  $a \wedge n < b \wedge n$

**assumes** *nonneg*:  $0 \leq b$

**shows**  $a < b$

**proof** (rule contrapos-pp [OF less])

**assume**  $\sim a < b$

**hence**  $b \leq a$  **by** (simp only: linorder-not-less)

**hence**  $b \wedge n \leq a \wedge n$  **using** *nonneg* **by** (rule power-mono)

**thus**  $\neg a \wedge n < b \wedge n$  **by** (simp only: linorder-not-less)

qed

**lemma** *power-inject-base*:

$$a \wedge \text{Suc } n = b \wedge \text{Suc } n \implies 0 \leq a \implies 0 \leq b \implies a = b$$

**by** (blast intro: power-le-imp-le-base antisym eq-refl sym)

**lemma** *power-eq-imp-eq-base*:

$$a \wedge n = b \wedge n \implies 0 \leq a \implies 0 \leq b \implies 0 < n \implies a = b$$

**by** (cases *n*) (simp-all del: power-Suc, rule power-inject-base)

end

**context** *linordered-idom*

**begin**

**lemma** *power-abs*:

$$\text{abs } (a \wedge n) = \text{abs } a \wedge n$$

```

by (induct n) (auto simp add: abs-mult)

lemma abs-power-minus [simp]:
  abs ((-a) ^ n) = abs (a ^ n)
by (simp add: power-abs)

lemma zero-less-power-abs-iff [simp, no-atp]:
  0 < abs a ^ n  $\longleftrightarrow$  a  $\neq$  0  $\vee$  n = 0
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc n) show ?case by (auto simp add: Suc zero-less-mult-iff)
qed

lemma zero-le-power-abs [simp]:
  0  $\leq$  abs a ^ n
by (rule zero-le-power [OF abs-ge-zero])

end

context ring-1-no-zero-divisors
begin

lemma field-power-not-zero:
  a  $\neq$  0  $\implies$  a ^ n  $\neq$  0
by (induct n) auto

end

context division-ring
begin

  FIXME reorient or rename to nonzero-inverse-power

lemma nonzero-power-inverse:
  a  $\neq$  0  $\implies$  inverse (a ^ n) = (inverse a) ^ n
by (induct n)
  (simp-all add: nonzero-inverse-mult-distrib power-commutes field-power-not-zero)

end

context field
begin

lemma nonzero-power-divide:
  b  $\neq$  0  $\implies$  (a / b) ^ n = a ^ n / b ^ n
by (simp add: divide-inverse power-mult-distrib nonzero-power-inverse)

end

```

**lemma** *power-0-Suc* [simp]:  
 $(0 :: 'a :: \{\text{power}, \text{semiring-0}\}) \wedge \text{Suc } n = 0$   
**by** *simp*

It looks plausible as a simprule, but its effect can be strange.

**lemma** *power-0-left*:  
 $0 \wedge n = (\text{if } n = 0 \text{ then } 1 \text{ else } (0 :: 'a :: \{\text{power}, \text{semiring-0}\}))$   
**by** (*induct n*) *simp-all*

**lemma** *power-eq-0-iff* [simp]:  
 $a \wedge n = 0 \iff$   
 $a = (0 :: 'a :: \{\text{mult-zero}, \text{zero-neq-one}, \text{no-zero-divisors}, \text{power}\}) \wedge n \neq 0$   
**by** (*induct n*)  
 (*auto simp add: no-zero-divisors elim: contrapos-pp*)

**lemma** (*in field*) *power-diff*:  
**assumes** *nz*:  $a \neq 0$   
**shows**  $n \leq m \implies a \wedge (m - n) = a \wedge m / a \wedge n$   
**by** (*induct m n rule: diff-induct*) (*simp-all add: nz field-power-not-zero*)

Perhaps these should be simprules.

**lemma** *power-inverse*:  
**fixes**  $a :: 'a :: \text{division-ring-inverse-zero}$   
**shows**  $\text{inverse } (a \wedge n) = \text{inverse } a \wedge n$   
**apply** (*cases a = 0*)  
**apply** (*simp add: power-0-left*)  
**apply** (*simp add: nonzero-power-inverse*)  
**done**

**lemma** *power-one-over*:  
 $1 / (a :: 'a :: \{\text{field-inverse-zero}, \text{power}\}) \wedge n = (1 / a) \wedge n$   
**by** (*simp add: divide-inverse*) (*rule power-inverse*)

**lemma** *power-divide*:  
 $(a / b) \wedge n = (a :: 'a :: \text{field-inverse-zero}) \wedge n / b \wedge n$   
**apply** (*cases b = 0*)  
**apply** (*simp add: power-0-left*)  
**apply** (*rule nonzero-power-divide*)  
**apply** *assumption*  
**done**

## 18.2 Exponentiation for the Natural Numbers

**lemma** *nat-one-le-power* [simp]:  
 $\text{Suc } 0 \leq i \implies \text{Suc } 0 \leq i \wedge n$   
**by** (*rule one-le-power [of i n, unfolded One-nat-def]*)

**lemma** *nat-zero-less-power-iff* [simp]:  
 $x \wedge n > 0 \iff x > (0 :: \text{nat}) \vee n = 0$

**by** (*induct n*) *auto*

**lemma** *nat-power-eq-Suc-0-iff* [*simp*]:  
 $x \wedge m = \text{Suc } 0 \iff m = 0 \vee x = \text{Suc } 0$   
**by** (*induct m*) *auto*

**lemma** *power-Suc-0* [*simp*]:  
 $\text{Suc } 0 \wedge n = \text{Suc } 0$   
**by** *simp*

Valid for the naturals, but what if  $0 < i < 1$ ? Premises cannot be weakened:  
 consider the case where  $i = (0::'a)$ ,  $m = (1::'a)$  and  $n = (0::'a)$ .

**lemma** *nat-power-less-imp-less*:  
**assumes** *nonneg*:  $0 < (i::\text{nat})$   
**assumes** *less*:  $i \wedge m < i \wedge n$   
**shows**  $m < n$   
**proof** (*cases i = 1*)  
**case** *True* **with** *less power-one* [**where**  $'a = \text{nat}$ ] **show** *?thesis* **by** *simp*  
**next**  
**case** *False* **with** *nonneg* **have**  $1 < i$  **by** *auto*  
**from** *power-strict-increasing-iff* [*OF this*] *less* **show** *?thesis* **..**  
**qed**

**lemma** *power-dvd-imp-le*:  
 $i \wedge m \text{ dvd } i \wedge n \implies (1::\text{nat}) < i \implies m \leq n$   
**apply** (*rule power-le-imp-le-exp, assumption*)  
**apply** (*erule dvd-imp-le, simp*)  
**done**

### 18.3 Code generator tweak

**lemma** *power-power-power* [*code, code-unfold, code-inline del*]:  
 $\text{power} = \text{power}.\text{power } (1::'a::\{\text{power}\}) \text{ (op *)}$   
**unfolding** *power-def power.power-def* **..**

**declare** *power.power.simps* [*code*]

**code-modulename** *SML*  
*Power Arith*

**code-modulename** *OCaml*  
*Power Arith*

**code-modulename** *Haskell*  
*Power Arith*

**end**

## 19 Option: Datatype option

```
theory Option
imports Datatype
begin
```

```
datatype 'a option = None | Some 'a
```

```
lemma not-None-eq [iff]:  $(x \sim = \text{None}) = (\text{EX } y. x = \text{Some } y)$ 
  by (induct x) auto
```

```
lemma not-Some-eq [iff]:  $(\text{ALL } y. x \sim = \text{Some } y) = (x = \text{None})$ 
  by (induct x) auto
```

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

```
lemma inj-Some [simp]: inj-on Some A
by (rule inj-onI) simp
```

```
lemma option-caseE:
  assumes c:  $(\text{case } x \text{ of } \text{None} \Rightarrow P \mid \text{Some } y \Rightarrow Q \ y)$ 
  obtains
     $(\text{None}) \ x = \text{None}$  and  $P$ 
  |  $(\text{Some}) \ y$  where  $x = \text{Some } y$  and  $Q \ y$ 
  using c by (cases x) simp-all
```

```
lemma UNIV-option-conv:  $\text{UNIV} = \text{insert None } (\text{range Some})$ 
by(auto intro: classical)
```

### 19.0.1 Operations

```
primrec the :: 'a option => 'a where
the (Some x) = x
```

```
primrec set :: 'a option => 'a set where
set None = {} |
set (Some x) = {x}
```

```
lemma ospec [dest]:  $(\text{ALL } x:\text{set } A. P \ x) \Rightarrow A = \text{Some } x \Rightarrow P \ x$ 
  by simp
```

```
declaration << fn - =>
  Classical.map-cs (fn cs => cs addSD2 (ospec, thm ospec))
>>
```

```
lemma elem-set [iff]:  $(x : \text{set } xo) = (xo = \text{Some } x)$ 
  by (cases xo) auto
```

**lemma** *set-empty-eq* [*simp*]: (*set* *xo* = {}) = (*xo* = *None*)  
**by** (*cases* *xo*) *auto*

**definition** *map* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a *option*  $\Rightarrow$  'b *option* **where**  
*map* = (%f y. *case* y of *None* => *None* | *Some* x => *Some* (f x))

**lemma** *option-map-None* [*simp*, *code*]: *map* f *None* = *None*  
**by** (*simp* *add*: *map-def*)

**lemma** *option-map-Some* [*simp*, *code*]: *map* f (*Some* x) = *Some* (f x)  
**by** (*simp* *add*: *map-def*)

**lemma** *option-map-is-None* [*iff*]:  
(*map* f *opt* = *None*) = (*opt* = *None*)  
**by** (*simp* *add*: *map-def* *split* *add*: *option.split*)

**lemma** *option-map-eq-Some* [*iff*]:  
(*map* f *xo* = *Some* y) = (EX z. *xo* = *Some* z & f z = y)  
**by** (*simp* *add*: *map-def* *split* *add*: *option.split*)

**lemma** *option-map-comp*:  
*map* f (*map* g *opt*) = *map* (f o g) *opt*  
**by** (*simp* *add*: *map-def* *split* *add*: *option.split*)

**lemma** *option-map-o-sum-case* [*simp*]:  
*map* f o *sum-case* g h = *sum-case* (*map* f o g) (*map* f o h)  
**by** (*rule* *ext*) (*simp* *split*: *sum.split*)

**hide-const** (**open**) *set* *map*

### 19.0.2 Code generator setup

**definition** *is-none* :: 'a *option*  $\Rightarrow$  *bool* **where**  
[*code-post*]: *is-none* x  $\longleftrightarrow$  x = *None*

**lemma** *is-none-code* [*code*]:  
**shows** *is-none* *None*  $\longleftrightarrow$  *True*  
**and** *is-none* (*Some* x)  $\longleftrightarrow$  *False*  
**unfolding** *is-none-def* **by** *simp-all*

**lemma** *is-none-none*:  
*is-none* x  $\longleftrightarrow$  x = *None*  
**by** (*simp* *add*: *is-none-def*)

**lemma** [*code-unfold*]:  
*eq-class.eq* x *None*  $\longleftrightarrow$  *is-none* x  
**by** (*simp* *add*: *eq* *is-none-none*)

```

hide-const (open) is-none

code-type option
  (SML - option)
  (OCaml - option)
  (Haskell Maybe -)
  (Scala !Option[(-)])

code-const None and Some
  (SML NONE and SOME)
  (OCaml None and Some -)
  (Haskell Nothing and Just)
  (Scala None and !Some((-)))

code-instance option :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq option ⇒ 'a option ⇒ bool
  (Haskell infixl 4 ==)

code-reserved SML
  option NONE SOME

code-reserved OCaml
  option None Some

code-reserved Scala
  Option None Some

end

```

## 20 Finite-Set: Finite sets

```

theory Finite-Set
imports Power Option
begin

```

### 20.1 Predicate for finite sets

```

inductive finite :: 'a set => bool
  where
    emptyI [simp, intro!]: finite {}
    | insertI [simp, intro!]: finite A ==> finite (insert a A)

lemma ex-new-if-finite: — does not depend on def of finite at all
  assumes  $\neg \text{finite } (UNIV :: 'a \text{ set})$  and finite A
  shows  $\exists a::'a. a \notin A$ 
proof —

```

from *assms* have  $A \neq \text{UNIV}$  by *blast*  
 thus ?thesis by *blast*  
 qed

**lemma** *finite-induct* [case-names empty insert, induct set: *finite*]:

*finite F ==>*  
 $P \{\} \implies (!x F. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F)) \implies$   
 $P F$

— Discharging  $x \notin F$  entails extra work.

**proof** —

assume  $P \{\}$  and

*insert*:  $!x F. \text{finite } F \implies x \notin F \implies P F \implies P (\text{insert } x F)$

assume *finite F*

thus  $P F$

**proof** *induct*

show  $P \{\}$  by *fact*

fix  $x F$  assume  $F: \text{finite } F$  and  $P: P F$

show  $P (\text{insert } x F)$

**proof** *cases*

assume  $x \in F$

hence  $\text{insert } x F = F$  by (rule *insert-absorb*)

with  $P$  show ?thesis by (simp only:)

next

assume  $x \notin F$

from  $F$  this  $P$  show ?thesis by (rule *insert*)

qed

qed

qed

**lemma** *finite-ne-induct*[case-names singleton insert, consumes 2]:

assumes *fin*: *finite F* shows  $F \neq \{\} \implies$

$\llbracket \bigwedge x. P\{x\};$

$\bigwedge x F. \llbracket \text{finite } F; F \neq \{\}; x \notin F; P F \rrbracket \implies P (\text{insert } x F) \rrbracket$

$\implies P F$

using *fin*

**proof** *induct*

case *empty* thus ?case by *simp*

next

case (*insert x F*)

show ?case

**proof** *cases*

assume  $F = \{\}$

thus ?thesis using  $\langle P \{x\} \rangle$  by *simp*

next

assume  $F \neq \{\}$

thus ?thesis using *insert* by *blast*

qed

qed



```

lemma finite-subset-induct [consumes 2, case-names empty insert]:
  assumes finite F and  $F \subseteq A$ 
    and empty:  $P \{\}$ 
    and insert:  $!!a F. \text{finite } F \implies a \in A \implies a \notin F \implies P F \implies P (\text{insert } a F)$ 
  shows  $P F$ 
proof –
  from  $\langle \text{finite } F \rangle$  and  $\langle F \subseteq A \rangle$ 
  show ?thesis
proof induct
  show  $P \{\}$  by fact
next
  fix  $x F$ 
  assume finite F and  $x \notin F$  and
     $P: F \subseteq A \implies P F$  and  $i: \text{insert } x F \subseteq A$ 
  show  $P (\text{insert } x F)$ 
  proof (rule insert)
    from  $i$  show  $x \in A$  by blast
    from  $i$  have  $F \subseteq A$  by blast
    with  $P$  show  $P F$  .
    show finite F by fact
    show  $x \notin F$  by fact
  qed
qed
qed

```

A finite choice principle. Does not need the SOME choice operator.

```

lemma finite-set-choice:
   $\text{finite } A \implies \text{ALL } x:A. (\text{EX } y. P x y) \implies \text{EX } f. \text{ALL } x:A. P x (f x)$ 
proof (induct set: finite)
  case empty thus ?case by simp
next
  case (insert a A)
  then obtain  $f b$  where  $f: \text{ALL } x:A. P x (f x)$  and  $ab: P a b$  by auto
  show ?case (is EX f. ?P f)
  proof
    show  $?P(\%x. \text{if } x = a \text{ then } b \text{ else } f x)$  using  $f ab$  by auto
  qed
qed

```

Finite sets are the images of initial segments of natural numbers:

```

lemma finite-imp-nat-seg-image-inj-on:
  assumes fin: finite A
  shows  $\exists (n::\text{nat}). f. A = f \, \{i. i < n\} \ \& \ \text{inj-on } f \, \{i. i < n\}$ 
using fin
proof induct
  case empty
  show ?case
  proof show  $\exists f. \{ \} = f \, \{i::\text{nat}. i < 0\} \ \& \ \text{inj-on } f \, \{i. i < 0\}$  by simp

```

```

qed
next
  case (insert a A)
  have notinA:  $a \notin A$  by fact
  from insert.hyps obtain n f
    where  $A = f \text{ ‘ } \{i::\text{nat. } i < n\}$  inj-on f  $\{i. i < n\}$  by blast
  hence insert a A =  $f(n:=a) \text{ ‘ } \{i. i < \text{Suc } n\}$ 
    inj-on ( $f(n:=a)$ )  $\{i. i < \text{Suc } n\}$  using notinA
  by (auto simp add: image-def Ball-def inj-on-def less-Suc-eq)
  thus ?case by blast
qed

lemma nat-seg-image-imp-finite:
  !!f A.  $A = f \text{ ‘ } \{i::\text{nat. } i < n\} \implies \text{finite } A$ 
proof (induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  let ?B =  $f \text{ ‘ } \{i. i < n\}$ 
  have finB: finite ?B by (rule Suc.hyps[OF refl])
  show ?case
  proof cases
    assume  $\exists k < n. f \ n = f \ k$ 
    hence  $A = ?B$  using Suc.prem by (auto simp: less-Suc-eq)
    thus ?thesis using finB by simp
  next
    assume  $\neg(\exists k < n. f \ n = f \ k)$ 
    hence  $A = \text{insert } (f \ n) \ ?B$  using Suc.prem by (auto simp: less-Suc-eq)
    thus ?thesis using finB by simp
  qed
qed

lemma finite-conv-nat-seg-image:
  finite A =  $(\exists (n::\text{nat}) f. A = f \text{ ‘ } \{i::\text{nat. } i < n\})$ 
by (blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

lemma finite-imp-inj-to-nat-seg:
  assumes finite A
  shows EX f n::nat.  $f \ A = \{i. i < n\} \ \& \ \text{inj-on } f \ A$ 
proof -
  from finite-imp-nat-seg-image-inj-on[OF (finite A)]
  obtain f and n::nat where bij: bij-betw f  $\{i. i < n\}$  A
  by (auto simp: bij-betw-def)
  let ?f = the-inv-into  $\{i. i < n\}$  f
  have inj-on ?f A & ?f ‘ A =  $\{i. i < n\}$ 
  by (fold bij-betw-def) (rule bij-betw-the-inv-into[OF bij])
  thus ?thesis by blast
qed

```

**lemma** *finite-Collect-less-nat*[*iff*]:  $\text{finite}\{n::\text{nat}. n < k\}$   
**by**(*fastsimp simp: finite-conv-nat-seg-image*)

Finiteness and set theoretic constructions

**lemma** *finite-UnI*:  $\text{finite } F \implies \text{finite } G \implies \text{finite } (F \text{ Un } G)$   
**by** (*induct set: finite*) *simp-all*

**lemma** *finite-subset*:  $A \subseteq B \implies \text{finite } B \implies \text{finite } A$   
 — Every subset of a finite set is finite.

**proof** —

**assume** *finite B*

**thus** !!*A*.  $A \subseteq B \implies \text{finite } A$

**proof** *induct*

**case** *empty*

**thus** ?*case* **by** *simp*

**next**

**case** (*insert x F A*)

**have** *A*:  $A \subseteq \text{insert } x \text{ } F$  **and** *r*:  $A - \{x\} \subseteq F \implies \text{finite } (A - \{x\})$  **by** *fact+*

**show** *finite A*

**proof** *cases*

**assume** *x*:  $x \in A$

**with** *A* **have**  $A - \{x\} \subseteq F$  **by** (*simp add: subset-insert-iff*)

**with** *r* **have** *finite* ( $A - \{x\}$ ) .

**hence** *finite* (*insert x* ( $A - \{x\}$ )) ..

**also** **have** *insert x* ( $A - \{x\}$ ) = *A* **using** *x* **by** (*rule insert-Diff*)

**finally** **show** ?*thesis* .

**next**

**show**  $A \subseteq F \implies ?\text{thesis}$  **by** *fact*

**assume**  $x \notin A$

**with** *A* **show**  $A \subseteq F$  **by** (*simp add: subset-insert-iff*)

**qed**

**qed**

**qed**

**lemma** *rev-finite-subset*:  $\text{finite } B \implies A \subseteq B \implies \text{finite } A$   
**by** (*rule finite-subset*)

**lemma** *finite-Un* [*iff*]:  $\text{finite } (F \text{ Un } G) = (\text{finite } F \ \& \ \text{finite } G)$   
**by** (*blast intro: finite-subset [of - X Un Y, standard] finite-UnI*)

**lemma** *finite-Collect-disjI*[*simp*]:  
 $\text{finite}\{x. P \ x \mid Q \ x\} = (\text{finite}\{x. P \ x\} \ \& \ \text{finite}\{x. Q \ x\})$   
**by**(*simp add: Collect-disj-eq*)

**lemma** *finite-Int* [*simp, intro*]:  $\text{finite } F \mid \text{finite } G \implies \text{finite } (F \text{ Int } G)$   
 — The converse obviously fails.  
**by** (*blast intro: finite-subset*)

**lemma** *finite-Collect-conjI* [*simp, intro*]:

$finite\{x. P\ x\} \mid finite\{x. Q\ x\} ==> finite\{x. P\ x \ \& \ Q\ x\}$   
 — The converse obviously fails.  
**by**(*simp add: Collect-conj-eq*)

**lemma** *finite-Collect-le-nat*[*iff*]:  $finite\{n::nat. n \leq k\}$   
**by**(*simp add: le-eq-less-or-eq*)

**lemma** *finite-insert* [*simp*]:  $finite\ (insert\ a\ A) = finite\ A$   
**apply** (*subst insert-is-Un*)  
**apply** (*simp only: finite-Un, blast*)  
**done**

**lemma** *finite-Union*[*simp, intro*]:  
 $\llbracket finite\ A; \forall M. M \in A \implies finite\ M \rrbracket \implies finite(\bigcup A)$   
**by** (*induct rule:finite-induct*) *simp-all*

**lemma** *finite-Inter*[*intro*]:  $EX\ A:M. finite(A) \implies finite(Inter\ M)$   
**by** (*blast intro: Inter-lower finite-subset*)

**lemma** *finite-INT*[*intro*]:  $EX\ x:I. finite(A\ x) \implies finite(INT\ x:I. A\ x)$   
**by** (*blast intro: INT-lower finite-subset*)

**lemma** *finite-empty-induct*:  
**assumes** *finite A*  
**and** *P A*  
**and**  $\forall a. A. finite\ A ==> a:A ==> P\ A ==> P\ (A - \{a\})$   
**shows**  $P\ \{\}$   
**proof** —  
**have**  $P\ (A - A)$   
**proof** —  
 $\{$   
**fix**  $c\ b :: 'a\ set$   
**assume**  $c: finite\ c$  **and**  $b: finite\ b$   
**and**  $P1: P\ b$  **and**  $P2: \forall x\ y. finite\ y ==> x \in y ==> P\ y ==> P\ (y - \{x\})$   
**have**  $c \subseteq b ==> P\ (b - c)$   
**using**  $c$   
**proof** *induct*  
**case** *empty*  
**from**  $P1$  **show** *?case* **by** *simp*  
**next**  
**case** (*insert x F*)  
**have**  $P\ (b - F - \{x\})$   
**proof** (*rule P2*)  
**from**  $b$  **show**  $finite\ (b - F)$  **by** (*rule finite-subset*) *blast*  
**from** *insert* **show**  $x \in b - F$  **by** *simp*  
**from** *insert* **show**  $P\ (b - F)$  **by** *simp*  
**qed**  
**also have**  $b - F - \{x\} = b - insert\ x\ F$  **by** (*rule Diff-insert [symmetric]*)

```

      finally show ?case .
    qed
  }
  then show ?thesis by this (simp-all add: assms)
  qed
  then show ?thesis by simp
  qed

lemma finite-Diff [simp]: finite A ==> finite (A - B)
by (rule Diff-subset [THEN finite-subset])

lemma finite-Diff2 [simp]:
  assumes finite B shows finite (A - B) = finite A
proof -
  have finite A <=> finite((A-B) Un (A Int B)) by (simp add: Un-Diff-Int)
  also have ... <=> finite(A-B) using ⟨finite B⟩ by (simp)
  finally show ?thesis ..
  qed

lemma finite-compl[simp]:
  finite(A::'a set) ==> finite(-A) = finite(UNIV::'a set)
by (simp add: Compl-eq-Diff-UNIV)

lemma finite-Collect-not[simp]:
  finite{x::'a. P x} ==> finite{x. ~ P x} = finite(UNIV::'a set)
by (simp add: Collect-neg-eq)

lemma finite-Diff-insert [iff]: finite (A - insert a B) = finite (A - B)
  apply (subst Diff-insert)
  apply (case-tac a : A - B)
  apply (rule finite-insert [symmetric, THEN trans])
  apply (subst insert-Diff, simp-all)
  done

Image and Inverse Image over Finite Sets

lemma finite-imageI[simp]: finite F ==> finite (h ` F)
  — The image of a finite set is finite.
  by (induct set: finite) simp-all

lemma finite-image-set [simp]:
  finite {x. P x} ==> finite { f x | x. P x }
  by (simp add: image-Collect [symmetric])

lemma finite-surj: finite A ==> B <= f ` A ==> finite B
  apply (frule finite-imageI)
  apply (erule finite-subset, assumption)
  done

lemma finite-range-imageI:

```

```

  finite (range g) ==> finite (range (%x. f (g x)))
  apply (drule finite-imageI, simp add: range-composition)
  done

```

**lemma** *finite-imageD*:  $\text{finite } (f^{\ast}A) \implies \text{inj-on } f \ A \implies \text{finite } A$

**proof** –

```

  have aux: !!A. finite (A - { }) = finite A by simp
  fix B :: 'a set
  assume finite B
  thus !!A. f^{\ast}A = B ==> inj-on f A ==> finite A
  apply induct
  apply simp
  apply (subgoal-tac EX y:A. f y = x & F = f^{\ast} (A - {y}))
  apply clarify
  apply (simp (no-asm-use) add: inj-on-def)
  apply (blast dest!: aux [THEN iffD1], atomize)
  apply (erule-tac V = ALL A. ?PP (A) in thin-rl)
  apply (frule subsetD [OF equalityD2 insertI1], clarify)
  apply (rule-tac x = xa in bexI)
  apply (simp-all add: inj-on-image-set-diff)
  done
qed (rule refl)

```

**lemma** *inj-vimage-singleton*:  $\text{inj } f \implies f^{-1}\{a\} \subseteq \{THE\ x. f\ x = a\}$

— The inverse image of a singleton under an injective function is included in a singleton.

```

  apply (auto simp add: inj-on-def)
  apply (blast intro: the-equality [symmetric])
  done

```

**lemma** *finite-vimageI*:  $[[\text{finite } F; \text{inj } h]] \implies \text{finite } (h^{-1} F)$

— The inverse image of a finite set under an injective function is finite.

```

  apply (induct set: finite)
  apply simp-all
  apply (subst vimage-insert)
  apply (simp add: finite-subset [OF inj-vimage-singleton])
  done

```

**lemma** *finite-vimageD*:

```

  assumes fin: finite (h^{-1} F) and surj: surj h
  shows finite F

```

**proof** –

```

  have finite (h^{-1} (h^{-1} F)) using fin by (rule finite-imageI)
  also have h^{-1} (h^{-1} F) = F using surj by (rule surj-image-vimage-eq)
  finally show finite F .

```

**qed**

**lemma** *finite-vimage-iff*:  $\text{bij } h \implies \text{finite } (h^{-1} F) \longleftrightarrow \text{finite } F$

**unfolding** *bij-def* **by** (*auto elim: finite-vimageD finite-vimageI*)

The finite UNION of finite sets

**lemma** *finite-UN-I*:  $\text{finite } A \implies (!a. a:A \implies \text{finite } (B \ a)) \implies \text{finite } (\text{UN } a:A. B \ a)$   
**by** (*induct set: finite*) *simp-all*

Strengthen RHS to  $(\forall x \in A. \text{finite } (B \ x)) \wedge \text{finite } \{x \in A. B \ x \neq \{\}\}$ ?

We’d need to prove  $\text{finite } C \implies \forall A \ B. \text{UNION } A \ B \subseteq C \longrightarrow \text{finite } \{x \in A. B \ x \neq \{\}\}$  by induction.

**lemma** *finite-UN* [*simp*]:  
 $\text{finite } A \implies \text{finite } (\text{UNION } A \ B) = (\text{ALL } x:A. \text{finite } (B \ x))$   
**by** (*blast intro: finite-UN-I finite-subset*)

**lemma** *finite-Collect-bex* [*simp*]:  $\text{finite } A \implies$   
 $\text{finite}\{x. \text{EX } y:A. Q \ x \ y\} = (\text{ALL } y:A. \text{finite}\{x. Q \ x \ y\})$   
**apply** (*subgoal-tac*  $\{x. \text{EX } y:A. Q \ x \ y\} = \text{UNION } A \ (\%y. \{x. Q \ x \ y\})$ )  
**apply** *auto*  
**done**

**lemma** *finite-Collect-bounded-ex* [*simp*]:  $\text{finite}\{y. P \ y\} \implies$   
 $\text{finite}\{x. \text{EX } y. P \ y \ \& \ Q \ x \ y\} = (\text{ALL } y. P \ y \longrightarrow \text{finite}\{x. Q \ x \ y\})$   
**apply** (*subgoal-tac*  $\{x. \text{EX } y. P \ y \ \& \ Q \ x \ y\} = \text{UNION } \{y. P \ y\} \ (\%y. \{x. Q \ x \ y\})$ )  
**apply** *auto*  
**done**

**lemma** *finite-Plus*:  $[\text{finite } A; \text{finite } B] \implies \text{finite } (A \ <+> B)$   
**by** (*simp add: Plus-def*)

**lemma** *finite-PlusD*:

**fixes**  $A :: 'a \text{ set}$  **and**  $B :: 'b \text{ set}$   
**assumes**  $\text{fin}: \text{finite } (A \ <+> B)$   
**shows**  $\text{finite } A \ \text{finite } B$

**proof** –

**have**  $\text{Inl } 'A \subseteq A \ <+> B$  **by** *auto*  
**hence**  $\text{finite } (\text{Inl } 'A :: ('a + 'b) \text{ set})$  **using** *fin* **by** (*rule finite-subset*)  
**thus**  $\text{finite } A$  **by** (*rule finite-imageD*) (*auto intro: inj-onI*)

**next**

**have**  $\text{Inr } 'B \subseteq A \ <+> B$  **by** *auto*  
**hence**  $\text{finite } (\text{Inr } 'B :: ('a + 'b) \text{ set})$  **using** *fin* **by** (*rule finite-subset*)  
**thus**  $\text{finite } B$  **by** (*rule finite-imageD*) (*auto intro: inj-onI*)

**qed**

**lemma** *finite-Plus-iff* [*simp*]:  $\text{finite } (A \ <+> B) \longleftrightarrow \text{finite } A \ \wedge \ \text{finite } B$   
**by** (*auto intro: finite-PlusD finite-Plus*)

**lemma** *finite-Plus-UNIV-iff* [*simp*]:  
 $\text{finite } (\text{UNIV} :: ('a + 'b) \text{ set}) =$

```

  (finite (UNIV :: 'a set) & finite (UNIV :: 'b set))
by(subst UNIV-Plus-UNIV[symmetric])(rule finite-Plus-iff)

```

Sigma of finite sets

```

lemma finite-SigmaI [simp]:
  finite A ==> (!a. a:A ==> finite (B a)) ==> finite (SIGMA a:A. B a)
by (unfold Sigma-def) (blast intro!: finite-UN-I)

```

```

lemma finite-cartesian-product: [| finite A; finite B |] ==>
  finite (A <*> B)
by (rule finite-SigmaI)

```

```

lemma finite-Prod-UNIV:
  finite (UNIV::'a set) ==> finite (UNIV::'b set) ==> finite (UNIV::('a * 'b)
set)
  apply (subgoal-tac (UNIV:: ('a * 'b) set) = Sigma UNIV (%x. UNIV))
  apply (erule ssubst)
  apply (erule finite-SigmaI, auto)
done

```

```

lemma finite-cartesian-productD1:
  [| finite (A <*> B); B ≠ {} |] ==> finite A
  apply (auto simp add: finite-conv-nat-seg-image)
  apply (drule-tac x=n in spec)
  apply (drule-tac x=fst o f in spec)
  apply (auto simp add: o-def)
  prefer 2 apply (force dest!: equalityD2)
  apply (drule equalityD1)
  apply (rename-tac y x)
  apply (subgoal-tac ∃ k. k<n & f k = (x,y))
  prefer 2 apply force
  apply clarify
  apply (rule-tac x=k in image-eqI, auto)
done

```

```

lemma finite-cartesian-productD2:
  [| finite (A <*> B); A ≠ {} |] ==> finite B
  apply (auto simp add: finite-conv-nat-seg-image)
  apply (drule-tac x=n in spec)
  apply (drule-tac x=snd o f in spec)
  apply (auto simp add: o-def)
  prefer 2 apply (force dest!: equalityD2)
  apply (drule equalityD1)
  apply (rename-tac x y)
  apply (subgoal-tac ∃ k. k<n & f k = (x,y))
  prefer 2 apply force
  apply clarify
  apply (rule-tac x=k in image-eqI, auto)
done

```



The powerset of a finite set

**lemma** *finite-Pow-iff* [iff]: *finite* (*Pow* *A*) = *finite* *A*

**proof**

**assume** *finite* (*Pow* *A*)

**with - have** *finite* ((%*x*. {*x*}) ‘ *A*) **by** (*rule* *finite-subset*) *blast*

**thus** *finite* *A* **by** (*rule* *finite-imageD* [*unfolded inj-on-def*]) *simp*

**next**

**assume** *finite* *A*

**thus** *finite* (*Pow* *A*)

**by** *induct* (*simp-all* *add*: *Pow-insert*)

**qed**

**lemma** *finite-Collect-subsets*[*simp,intro*]: *finite* *A*  $\implies$  *finite*{*B*. *B*  $\subseteq$  *A*}

**by**(*simp* *add*: *Pow-def*[*symmetric*])

**lemma** *finite-UnionD*: *finite*( $\bigcup$  *A*)  $\implies$  *finite* *A*

**by**(*blast* *intro*: *finite-subset*[*OF subset-Pow-Union*])

**lemma** *finite-subset-image*:

**assumes** *finite* *B*

**shows** *B*  $\subseteq$  *f* ‘ *A*  $\implies \exists C \subseteq A. \text{finite } C \wedge B = f$  ‘ *C*

**using** *assms* **proof**(*induct*)

**case** *empty* **thus** ?*case* **by** *simp*

**next**

**case** *insert* **thus** ?*case*

**by** (*clarsimp* *simp* *del*: *image-insert* *simp* *add*: *image-insert*[*symmetric*])  
      *blast*

**qed**

## 20.2 Class *finite*

**class** *finite* =

**assumes** *finite-UNIV*: *finite* (*UNIV* :: ‘*a* set)

**begin**

**lemma** *finite* [*simp*]: *finite* (*A* :: ‘*a* set)

**by** (*rule* *subset-UNIV* *finite-UNIV* *finite-subset*) +

**end**

**lemma** *UNIV-unit* [*no-atp*]:

$UNIV = \{()\}$  **by** *auto*

**instance** *unit* :: *finite* **proof**

**qed** (*simp* *add*: *UNIV-unit*)

**lemma** *UNIV-bool* [*no-atp*]:

```

UNIV = {False, True} by auto

instance bool :: finite proof
qed (simp add: UNIV=bool)

instance * :: (finite, finite) finite proof
qed (simp only: UNIV-Times-UNIV [symmetric] finite-cartesian-product finite)

lemma finite-option-UNIV [simp]:
  finite (UNIV :: 'a option set) = finite (UNIV :: 'a set)
  by (auto simp add: UNIV-option-conv elim: finite-imageD intro: inj-Some)

instance option :: (finite) finite proof
qed (simp add: UNIV-option-conv)

lemma inj-graph: inj (%f. {(x, y). y = f x})
  by (rule inj-onI, auto simp add: expand-set-eq expand-fun-eq)

instance fun :: (finite, finite) finite
proof
  show finite (UNIV :: ('a => 'b) set)
  proof (rule finite-imageD)
    let ?graph = %f::'a => 'b. {(x, y). y = f x}
    have range ?graph ⊆ Pow UNIV by simp
    moreover have finite (Pow (UNIV :: ('a * 'b) set))
      by (simp only: finite-Pow-iff finite)
    ultimately show finite (range ?graph)
      by (rule finite-subset)
    show inj ?graph by (rule inj-graph)
  qed
qed

instance + :: (finite, finite) finite proof
qed (simp only: UNIV-Plus-UNIV [symmetric] finite-Plus finite)

```

### 20.3 A basic fold functional for finite sets

The intended behaviour is  $\text{fold } f \ z \ \{x_1, \dots, x_n\} = f \ x_1 \ (\dots (f \ x_n \ z) \dots)$  if  $f$  is “left-commutative”:

```

locale fun-left-comm =
  fixes f :: 'a => 'b => 'b
  assumes fun-left-comm: f x (f y z) = f y (f x z)
begin

```

On a functional level it looks much nicer:

```

lemma fun-comp-comm: f x ∘ f y = f y ∘ f x
by (simp add: fun-left-comm expand-fun-eq)

```

**end**

**inductive** *fold-graph* :: ( $'a \Rightarrow 'b \Rightarrow 'b$ )  $\Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow \text{bool}$   
**for**  $f :: 'a \Rightarrow 'b \Rightarrow 'b$  **and**  $z :: 'b$  **where**  
   *emptyI* [intro]: *fold-graph*  $f\ z\ \{\}$   $z \mid$   
   *insertI* [intro]:  $x \notin A \Longrightarrow \text{fold-graph } f\ z\ A\ y$   
                    $\Longrightarrow \text{fold-graph } f\ z\ (\text{insert } x\ A)\ (f\ x\ y)$

**inductive-cases** *empty-fold-graphE* [elim!]: *fold-graph*  $f\ z\ \{\}$   $x$

**definition** *fold* :: ( $'a \Rightarrow 'b \Rightarrow 'b$ )  $\Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b$  **where**  
 [code del]: *fold*  $f\ z\ A = (\text{THE } y. \text{fold-graph } f\ z\ A\ y)$

A tempting alternative for the definiens is *if finite A then THE y. fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

**lemma** *Diff1-fold-graph*:

*fold-graph*  $f\ z\ (A - \{x\})\ y \Longrightarrow x \in A \Longrightarrow \text{fold-graph } f\ z\ A\ (f\ x\ y)$   
**by** (*erule insert-Diff* [THEN *subst*], *rule fold-graph.intros*, *auto*)

**lemma** *fold-graph-imp-finite*: *fold-graph*  $f\ z\ A\ x \Longrightarrow \text{finite } A$   
**by** (*induct set: fold-graph*) *auto*

**lemma** *finite-imp-fold-graph*: *finite*  $A \Longrightarrow \exists x. \text{fold-graph } f\ z\ A\ x$   
**by** (*induct set: finite*) *auto*

### 20.3.1 From *fold-graph* to *fold*

**context** *fun-left-comm*  
**begin**

**lemma** *fold-graph-insertE-aux*:

*fold-graph*  $f\ z\ A\ y \Longrightarrow a \in A \Longrightarrow \exists y'. y = f\ a\ y' \wedge \text{fold-graph } f\ z\ (A - \{a\})\ y'$

**proof** (*induct set: fold-graph*)

**case** (*insertI*  $x\ A\ y$ ) **show** ?*case*

**proof** (*cases*  $x = a$ )

**assume**  $x = a$  **with** *insertI* **show** ?*case* **by** *auto*

**next**

**assume**  $x \neq a$

**then obtain**  $y'$  **where**  $y = f\ a\ y'$  **and**  $y': \text{fold-graph } f\ z\ (A - \{a\})\ y'$

**using** *insertI* **by** *auto*

**have** 1:  $f\ x\ y = f\ a\ (f\ x\ y')$

**unfolding**  $y$  **by** (*rule fun-left-comm*)

**have** 2: *fold-graph*  $f\ z\ (\text{insert } x\ A - \{a\})\ (f\ x\ y')$

**using**  $y'$  **and**  $\langle x \neq a \rangle$  **and**  $\langle x \notin A \rangle$

**by** (*simp add: insert-Diff-if fold-graph.insertI*)

**from** 1 2 **show** ?*case* **by** *fast*

**qed**

**qed** *simp*

**lemma** *fold-graph-insertE*:

**assumes** *fold-graph*  $f\ z\ (insert\ x\ A)\ v$  **and**  $x \notin A$   
**obtains**  $y$  **where**  $v = f\ x\ y$  **and** *fold-graph*  $f\ z\ A\ y$   
**using** *assms* **by** (*auto dest: fold-graph-insertE-aux [OF - insertI1]*)

**lemma** *fold-graph-determ*:

*fold-graph*  $f\ z\ A\ x \implies fold-graph\ f\ z\ A\ y \implies y = x$   
**proof** (*induct arbitrary: y set: fold-graph*)  
**case** (*insertI*  $x\ A\ y\ v$ )  
**from**  $\langle fold-graph\ f\ z\ (insert\ x\ A)\ v \rangle$  **and**  $\langle x \notin A \rangle$   
**obtain**  $y'$  **where**  $v = f\ x\ y'$  **and** *fold-graph*  $f\ z\ A\ y'$   
**by** (*rule fold-graph-insertE*)  
**from**  $\langle fold-graph\ f\ z\ A\ y' \rangle$  **have**  $y' = y$  **by** (*rule insertI*)  
**with**  $\langle v = f\ x\ y' \rangle$  **show**  $v = f\ x\ y$  **by** *simp*  
**qed** *fast*

**lemma** *fold-equality*:

*fold-graph*  $f\ z\ A\ y \implies fold\ f\ z\ A = y$   
**by** (*unfold fold-def*) (*blast intro: fold-graph-determ*)

**lemma** *fold-graph-fold*: *finite*  $A \implies fold-graph\ f\ z\ A\ (fold\ f\ z\ A)$

**unfolding** *fold-def*  
**apply** (*rule theI'*)  
**apply** (*rule ex-ex1I*)  
**apply** (*erule finite-imp-fold-graph*)  
**apply** (*erule (1) fold-graph-determ*)  
**done**

The base case for *fold*:

**lemma** (*in -*) *fold-empty* [*simp*]: *fold*  $f\ z\ \{\} = z$   
**by** (*unfold fold-def*) *blast*

The various recursion equations for *fold*:

**lemma** *fold-insert* [*simp*]:

*finite*  $A \implies x \notin A \implies fold\ f\ z\ (insert\ x\ A) = f\ x\ (fold\ f\ z\ A)$   
**apply** (*rule fold-equality*)  
**apply** (*erule fold-graph.insertI*)  
**apply** (*erule fold-graph-fold*)  
**done**

**lemma** *fold-fun-comm*:

*finite*  $A \implies f\ x\ (fold\ f\ z\ A) = fold\ f\ (f\ x\ z)\ A$   
**proof** (*induct rule: finite-induct*)  
**case** *empty* **then show** *?case* **by** *simp*  
**next**  
**case** (*insert*  $y\ A$ ) **then show** *?case*  
**by** (*simp add: fun-left-comm[of x]*)

qed

**lemma** *fold-insert2*:

*finite A*  $\implies x \notin A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f \ x \ z) \ A$   
**by** (*simp add: fold-fun-comm*)

**lemma** *fold-rec*:

**assumes** *finite A* **and**  $x \in A$

**shows**  $\text{fold } f \ z \ A = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$

**proof** –

**have**  $A = \text{insert } x \ (A - \{x\})$  **using**  $\langle x \in A \rangle$  **by** *blast*

**then have**  $\text{fold } f \ z \ A = \text{fold } f \ z \ (\text{insert } x \ (A - \{x\}))$  **by** *simp*

**also have**  $\dots = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$

**by** (*rule fold-insert*) (*simp add: finite A*) +

**finally show** *?thesis* .

qed

**lemma** *fold-insert-remove*:

**assumes** *finite A*

**shows**  $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$

**proof** –

**from**  $\langle \text{finite } A \rangle$  **have** *finite (insert x A)* **by** *auto*

**moreover have**  $x \in \text{insert } x \ A$  **by** *auto*

**ultimately have**  $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (\text{insert } x \ A - \{x\}))$

**by** (*rule fold-rec*)

**then show** *?thesis* **by** *simp*

qed

end

A simplified version for idempotent functions:

**locale** *fun-left-comm-idem* = *fun-left-comm* +

**assumes** *fun-left-idem*:  $f \ x \ (f \ x \ z) = f \ x \ z$

**begin**

The nice version:

**lemma** *fun-comp-idem* :  $f \ x \ o \ f \ x = f \ x$

**by** (*simp add: fun-left-idem expand-fun-eq*)

**lemma** *fold-insert-idem*:

**assumes** *fin*: *finite A*

**shows**  $\text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ A)$

**proof** *cases*

**assume**  $x \in A$

**then obtain** *B* **where**  $A = \text{insert } x \ B$  **and**  $x \notin B$  **by** (*rule set-insert*)

**then show** *?thesis* **using** *assms* **by** (*simp add: fun-left-idem*)

**next**

**assume**  $x \notin A$  **then show** *?thesis* **using** *assms* **by** *simp*

qed

**declare** *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

**lemma** *fold-insert-idem2*:

$\text{finite } A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f \ x \ z) \ A$   
**by**(*simp add:fold-fun-comm*)

**end**

### 20.3.2 Expressing set operations via *fold*

**lemma** (**in** *fun-left-comm*) *fun-left-comm-apply*:

*fun-left-comm* ( $\lambda x. f \ (g \ x)$ )

**proof**

**qed** (*simp-all add: fun-left-comm*)

**lemma** (**in** *fun-left-comm-idem*) *fun-left-comm-idem-apply*:

*fun-left-comm-idem* ( $\lambda x. f \ (g \ x)$ )

**by** (*rule fun-left-comm-idem.intro, rule fun-left-comm-apply, unfold-locales*)  
(*simp-all add: fun-left-idem*)

**lemma** *fun-left-comm-idem-insert*:

*fun-left-comm-idem insert*

**proof**

**qed** *auto*

**lemma** *fun-left-comm-idem-remove*:

*fun-left-comm-idem* ( $\lambda x \ A. \ A - \{x\}$ )

**proof**

**qed** *auto*

**lemma** (**in** *semilattice-inf*) *fun-left-comm-idem-inf*:

*fun-left-comm-idem inf*

**proof**

**qed** (*auto simp add: inf-left-commute*)

**lemma** (**in** *semilattice-sup*) *fun-left-comm-idem-sup*:

*fun-left-comm-idem sup*

**proof**

**qed** (*auto simp add: sup-left-commute*)

**lemma** *union-fold-insert*:

**assumes** *finite A*

**shows**  $A \cup B = \text{fold insert } B \ A$

**proof** —

**interpret** *fun-left-comm-idem insert* **by** (*fact fun-left-comm-idem-insert*)  
**from** (*finite A*) **show** ?thesis **by** (*induct A arbitrary: B*) *simp-all*

**qed**

```

lemma minus-fold-remove:
  assumes finite A
  shows  $B - A = \text{fold } (\lambda x A. A - \{x\}) B A$ 
proof –
  interpret fun-left-comm-idem  $\lambda x A. A - \{x\}$  by (fact fun-left-comm-idem-remove)
  from  $\langle \text{finite } A \rangle$  show ?thesis by (induct A arbitrary: B) auto
qed

context complete-lattice
begin

lemma inf-Inf-fold-inf:
  assumes finite A
  shows  $\text{inf } B (\text{Inf } A) = \text{fold inf } B A$ 
proof –
  interpret fun-left-comm-idem inf by (fact fun-left-comm-idem-inf)
  from  $\langle \text{finite } A \rangle$  show ?thesis by (induct A arbitrary: B)
    (simp-all add: Inf-empty Inf-insert inf-commute fold-fun-comm)
qed

lemma sup-Sup-fold-sup:
  assumes finite A
  shows  $\text{sup } B (\text{Sup } A) = \text{fold sup } B A$ 
proof –
  interpret fun-left-comm-idem sup by (fact fun-left-comm-idem-sup)
  from  $\langle \text{finite } A \rangle$  show ?thesis by (induct A arbitrary: B)
    (simp-all add: Sup-empty Sup-insert sup-commute fold-fun-comm)
qed

lemma Inf-fold-inf:
  assumes finite A
  shows  $\text{Inf } A = \text{fold inf top } A$ 
  using assms inf-Inf-fold-inf [of A top] by (simp add: inf-absorb2)

lemma Sup-fold-sup:
  assumes finite A
  shows  $\text{Sup } A = \text{fold sup bot } A$ 
  using assms sup-Sup-fold-sup [of A bot] by (simp add: sup-absorb2)

lemma inf-INFI-fold-inf:
  assumes finite A
  shows  $\text{inf } B (\text{INFI } A f) = \text{fold } (\lambda A. \text{inf } (f A)) B A$  (is ?inf = ?fold)
proof (rule sym)
  interpret fun-left-comm-idem inf by (fact fun-left-comm-idem-inf)
  interpret fun-left-comm-idem  $\lambda A. \text{inf } (f A)$  by (fact fun-left-comm-idem-apply)
  from  $\langle \text{finite } A \rangle$  show  $\text{?fold} = \text{?inf}$ 
  by (induct A arbitrary: B)
    (simp-all add: INFI-def Inf-empty Inf-insert inf-left-commute)
qed

```

```

lemma sup-SUPR-fold-sup:
  assumes finite A
  shows sup B (SUPR A f) = fold (λA. sup (f A)) B A (is ?sup = ?fold)
proof (rule sym)
  interpret fun-left-comm-idem sup by (fact fun-left-comm-idem-sup)
  interpret fun-left-comm-idem λA. sup (f A) by (fact fun-left-comm-idem-apply)
  from ⟨finite A⟩ show ?fold = ?sup
  by (induct A arbitrary: B)
    (simp-all add: SUPR-def Sup-empty Sup-insert sup-left-commute)
qed

```

```

lemma INFI-fold-inf:
  assumes finite A
  shows INFI A f = fold (λA. inf (f A)) top A
  using assms inf-INFI-fold-inf [of A top] by simp

```

```

lemma SUPR-fold-sup:
  assumes finite A
  shows SUPR A f = fold (λA. sup (f A)) bot A
  using assms sup-SUPR-fold-sup [of A bot] by simp

```

**end**

## 20.4 The derived combinator *fold-image*

```

definition fold-image :: ('b ⇒ 'b ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'b ⇒ 'a set ⇒ 'b
where fold-image f g = fold (%x y. f (g x) y)

```

```

lemma fold-image-empty[simp]: fold-image f g z {} = z
by(simp add:fold-image-def)

```

```

context ab-semigroup-mult
begin

```

```

lemma fold-image-insert[simp]:
  assumes finite A and a ∉ A
  shows fold-image times g z (insert a A) = g a * (fold-image times g z A)
proof –
  interpret I: fun-left-comm %x y. (g x) * y
    by unfold-locales (simp add: mult-ac)
  show ?thesis using assms by(simp add:fold-image-def)
qed

```

```

lemma fold-image-reindex:
  assumes fin: finite A
  shows inj-on h A ⇒ fold-image times g z (h `A) = fold-image times (g ∘ h) z A

```



using *fin* by *induct auto*

**lemma** *fold-image-cong*:

```

  finite A ==>
    (!x. x:A ==> g x = h x) ==> fold-image times g z A = fold-image times h z A
apply (subgoal-tac ALL C. C <= A --> (ALL x:C. g x = h x) --> fold-image
times g z C = fold-image times h z C)
apply simp
apply (erule finite-induct, simp)
apply (simp add: subset-insert-iff, clarify)
apply (subgoal-tac finite C)
prefer 2 apply (blast dest: finite-subset [COMP swap-prems-rl])
apply (subgoal-tac C = insert x (C - {x}))
prefer 2 apply blast
apply (erule ssubst)
apply (drule spec)
apply (erule (1) notE impE)
apply (simp add: Ball-def del: insert-Diff-single)
done

```

**end**

**context** *comm-monoid-mult*

**begin**

**lemma** *fold-image-1*:

```

  finite S ==> (∀ x∈S. f x = 1) ==> fold-image op * f 1 S = 1
apply (induct set: finite)
apply simp by auto

```

**lemma** *fold-image-Un-Int*:

```

  finite A ==> finite B ==>
    fold-image times g 1 A * fold-image times g 1 B =
    fold-image times g 1 (A Un B) * fold-image times g 1 (A Int B)
by (induct set: finite)
    (auto simp add: mult-ac insert-absorb Int-insert-left)

```

**lemma** *fold-image-Un-one*:

```

  assumes fS: finite S and fT: finite T
  and I0: ∀ x ∈ S ∩ T. f x = 1
  shows fold-image (op *) f 1 (S ∪ T) = fold-image (op *) f 1 S * fold-image (op
*) f 1 T
proof –
  have fold-image op * f 1 (S ∩ T) = 1
    apply (rule fold-image-1)
    using fS fT I0 by auto
  with fold-image-Un-Int[OF fS fT] show ?thesis by simp

```

qed

**corollary** *fold-Un-disjoint:*

$finite\ A ==> finite\ B ==> A\ Int\ B = \{\} ==>$   
 $fold-image\ times\ g\ 1\ (A\ Un\ B) =$   
 $fold-image\ times\ g\ 1\ A * fold-image\ times\ g\ 1\ B$   
**by** (*simp add: fold-image-Un-Int*)

**lemma** *fold-image-UN-disjoint:*

$\llbracket finite\ I; ALL\ i:I. finite\ (A\ i);$   
 $ALL\ i:I. ALL\ j:I. i \neq j \longrightarrow A\ i\ Int\ A\ j = \{\} \rrbracket$   
 $\implies fold-image\ times\ g\ 1\ (UNION\ I\ A) =$   
 $fold-image\ times\ (\%i. fold-image\ times\ g\ 1\ (A\ i))\ 1\ I$   
**apply** (*induct set: finite, simp, atomize*)  
**apply** (*subgoal-tac ALL i:F. x  $\neq$  i*)  
**prefer** 2 **apply** *blast*  
**apply** (*subgoal-tac A x Int UNION F A = \{\}*)  
**prefer** 2 **apply** *blast*  
**apply** (*simp add: fold-Un-disjoint*)  
**done**

**lemma** *fold-image-Sigma: finite A ==> ALL x:A. finite (B x) ==>*

$fold-image\ times\ (\%x. fold-image\ times\ (g\ x)\ 1\ (B\ x))\ 1\ A =$   
 $fold-image\ times\ (split\ g)\ 1\ (SIGMA\ x:A. B\ x)$   
**apply** (*subst Sigma-def*)  
**apply** (*subst fold-image-UN-disjoint, assumption, simp*)  
**apply** *blast*  
**apply** (*erule fold-image-cong*)  
**apply** (*subst fold-image-UN-disjoint, simp, simp*)  
**apply** *blast*  
**apply** *simp*  
**done**

**lemma** *fold-image-distrib: finite A  $\implies$*

$fold-image\ times\ (\%x. g\ x * h\ x)\ 1\ A =$   
 $fold-image\ times\ g\ 1\ A * fold-image\ times\ h\ 1\ A$   
**by** (*erule finite-induct*) (*simp-all add: mult-ac*)

**lemma** *fold-image-related:*

**assumes** *Re: R e e*  
**and** *Rop:  $\forall x1\ y1\ x2\ y2. R\ x1\ x2 \wedge R\ y1\ y2 \longrightarrow R\ (x1 * y1)\ (x2 * y2)$*   
**and** *fS: finite S and Rfg:  $\forall x \in S. R\ (h\ x)\ (g\ x)$*   
**shows**  $R\ (fold-image\ (op\ *)\ h\ e\ S)\ (fold-image\ (op\ *)\ g\ e\ S)$   
**using** *fS by (rule finite-subset-induct) (insert assms, auto)*

**lemma** *fold-image-eq-general:*

**assumes** *fS: finite S*  
**and** *h:  $\forall y \in S'. \exists! x. x \in S \wedge h(x) = y$*   
**and** *f12:  $\forall x \in S. h\ x \in S' \wedge f2(h\ x) = f1\ x$*

```

shows fold-image (op *) f1 e S = fold-image (op *) f2 e S'
proof -
  from h f12 have hS: h ` S = S' by auto
  {fix x y assume H: x ∈ S y ∈ S h x = h y
   from f12 h H have x = y by auto }
  hence hinj: inj-on h S unfolding inj-on-def Ex1-def by blast
  from f12 have th:  $\bigwedge x. x \in S \implies (f2 \circ h) x = f1 x$  by auto
  from hS have fold-image (op *) f2 e S' = fold-image (op *) f2 e (h ` S) by
simp
  also have ... = fold-image (op *) (f2 o h) e S
    using fold-image-reindex[OF fS hinj, of f2 e] .
  also have ... = fold-image (op *) f1 e S using th fold-image-cong[OF fS, of f2
o h f1 e]
    by blast
  finally show ?thesis ..
qed

```

lemma fold-image-eq-general-inverses:

```

assumes fS: finite S
and kh:  $\bigwedge y. y \in T \implies k y \in S \wedge h (k y) = y$ 
and hk:  $\bigwedge x. x \in S \implies h x \in T \wedge k (h x) = x \wedge g (h x) = f x$ 
shows fold-image (op *) f e S = fold-image (op *) g e T

```

```

apply (rule fold-image-eq-general[OF fS, of T h g f e])
apply (rule ballI)
apply (frule kh)
apply (rule ex1I[])
apply blast
apply clarsimp
apply (drule hk) apply simp
apply (rule sym)
apply (erule conjunct1[OF conjunct2[OF hk]])
apply (rule ballI)
apply (drule hk)
apply blast
done

```

end

## 20.5 A fold functional for non-empty sets

Does not require start value.

inductive

```

fold1Set :: ('a => 'a => 'a) => 'a set => 'a => bool
for f :: 'a => 'a => 'a

```

where

```

fold1Set-insertI [intro]:
 $\llbracket \text{fold-graph } f \text{ a } A \text{ x; } a \notin A \rrbracket \implies \text{fold1Set } f (\text{insert } a \text{ } A) \text{ x}$ 

```

**definition** *fold1* :: ('a => 'a => 'a) => 'a set => 'a **where**  
*fold1* f A == THE x. fold1Set f A x

**lemma** *fold1Set-nonempty*:  
*fold1Set* f A x  $\implies$  A  $\neq$  {}  
**by**(erule *fold1Set.cases*, *simp-all*)

**inductive-cases** *empty-fold1SetE* [elim!]: *fold1Set* f {} x

**inductive-cases** *insert-fold1SetE* [elim!]: *fold1Set* f (insert a X) x

**lemma** *fold1Set-sing* [iff]: (*fold1Set* f {a} b) = (a = b)  
**by** (blast elim: *fold-graph.cases*)

**lemma** *fold1-singleton* [simp]: *fold1* f {a} = a  
**by** (unfold *fold1-def*) blast

**lemma** *finite-nonempty-imp-fold1Set*:  
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \exists x. \text{fold1Set } f A x$   
**apply** (induct A rule: *finite-induct*)  
**apply** (auto dest: *finite-imp-fold-graph* [of - f])  
**done**

First, some lemmas about *fold-graph*.

**context** *ab-semigroup-mult*  
**begin**

**lemma** *fun-left-comm*: *fun-left-comm*(op \*)  
**by** *unfold-locales* (*simp add: mult-ac*)

**lemma** *fold-graph-insert-swap*:  
**assumes** *fold*: *fold-graph times* (b::'a) A y **and** b  $\notin$  A  
**shows** *fold-graph times* z (insert b A) (z \* y)  
**proof** —  
**interpret** *fun-left-comm* op \*::'a  $\Rightarrow$  'a  $\Rightarrow$  'a **by** (rule *fun-left-comm*)  
**from** *assms* **show** ?thesis  
**proof** (induct rule: *fold-graph.induct*)  
**case** *emptyI* **show** ?case **by** (subst *mult-commute* [of z b], fast)  
**next**  
**case** (*insertI* x A y)  
**have** *fold-graph times* z (insert x (insert b A)) (x \* (z \* y))  
**using** *insertI* **by** force — how does *id* get unfolded?  
**thus** ?case **by** (*simp add: insert-commute mult-ac*)  
**qed**  
**qed**

**lemma** *fold-graph-permute-diff*:  
**assumes** *fold*: *fold-graph times* b A x

```

shows !!a.  $\llbracket a \in A; b \notin A \rrbracket \implies \text{fold-graph times } a \text{ (insert } b \text{ (} A - \{a\} \text{)) } x$ 
using fold
proof (induct rule: fold-graph.induct)
  case emptyI thus ?case by simp
next
  case (insertI x A y)
  have  $a = x \vee a \in A$  using insertI by simp
  thus ?case
  proof
    assume  $a = x$ 
    with insertI show ?thesis
    by (simp add: id-def [symmetric], blast intro: fold-graph-insert-swap)
  next
    assume  $a \in A$ 
    hence fold-graph times a (insert x (insert b (A - {a}))) (x * y)
    using insertI by force
    moreover
    have insert x (insert b (A - {a})) = insert b (insert x A - {a})
    using ainA insertI by blast
    ultimately show ?thesis by simp
  qed
qed

```

**lemma** fold1-eq-fold:

**assumes** finite A  $a \notin A$  **shows** fold1 times (insert a A) = fold times a A

**proof** –

```

  interpret fun-left-comm op *: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a by (rule fun-left-comm)
  from assms show ?thesis
  apply (simp add: fold1-def fold-def)
  apply (rule the-equality)
  apply (best intro: fold-graph-determ theI dest: finite-imp-fold-graph [of - times])
  apply (rule sym, clarify)
  apply (case-tac Aa=A)
  apply (best intro: fold-graph-determ)
  apply (subgoal-tac fold-graph times a A x)
  apply (best intro: fold-graph-determ)
  apply (subgoal-tac insert aa (Aa - {a}) = A)
  prefer 2 apply (blast elim: equalityE)
  apply (auto dest: fold-graph-permute-diff [where a=a])
  done
qed

```

**lemma** nonempty-iff:  $(A \neq \{\}) = (\exists x B. A = \text{insert } x B \ \& \ x \notin B)$

**apply** safe

**apply** simp

**apply** (drule-tac  $x=x$  **in** spec)

**apply** (drule-tac  $x=A-\{x\}$  **in** spec, auto)

**done**

```

lemma fold1-insert:
  assumes nonempty:  $A \neq \{\}$  and  $A$ : finite  $A$   $x \notin A$ 
  shows fold1 times (insert  $x$   $A$ ) =  $x * \text{fold1 times } A$ 
proof –
  interpret fun-left-comm  $op$   $:: 'a \Rightarrow 'a \Rightarrow 'a$  by (rule fun-left-comm)
  from nonempty obtain  $a$   $A'$  where  $A = \text{insert } a$   $A'$  &  $a \sim: A'$ 
    by (auto simp add: nonempty-iff)
  with  $A$  show ?thesis
    by (simp add: insert-commute [of x] fold1-eq-fold eq-commute)
qed

end

context ab-semigroup-idem-mult
begin

lemma fun-left-comm-idem: fun-left-comm-idem( $op$   $*$ )
apply unfold-locales
  apply (rule mult-left-commute)
apply (rule mult-left-idem)
done

lemma fold1-insert-idem [simp]:
  assumes nonempty:  $A \neq \{\}$  and  $A$ : finite  $A$ 
  shows fold1 times (insert  $x$   $A$ ) =  $x * \text{fold1 times } A$ 
proof –
  interpret fun-left-comm-idem  $op$   $:: 'a \Rightarrow 'a \Rightarrow 'a$ 
    by (rule fun-left-comm-idem)
  from nonempty obtain  $a$   $A'$  where  $A': A = \text{insert } a$   $A'$  &  $a \sim: A'$ 
    by (auto simp add: nonempty-iff)
  show ?thesis
  proof cases
    assume  $a = x$ 
    thus ?thesis
    proof cases
      assume  $A' = \{\}$ 
      with prems show ?thesis by simp
    next
      assume  $A' \neq \{\}$ 
      with prems show ?thesis
        by (simp add: fold1-insert mult-assoc [symmetric])
    qed
  next
    assume  $a \neq x$ 
    with prems show ?thesis
      by (simp add: insert-commute fold1-eq-fold)
    qed
  qed

```

```

lemma hom-fold1-commute:
assumes hom:  $\forall x y. h (x * y) = h x * h y$ 
and N: finite N  $N \neq \{\}$  shows  $h (fold1\ times\ N) = fold1\ times\ (h\ 'N)$ 
using N proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case (insert n N)
  then have  $h (fold1\ times\ (insert\ n\ N)) = h (n * fold1\ times\ N)$  by simp
  also have  $\dots = h\ n * h (fold1\ times\ N)$  by (rule hom)
  also have  $h (fold1\ times\ N) = fold1\ times\ (h\ 'N)$  by (rule insert)
  also have  $times\ (h\ n) \dots = fold1\ times\ (insert\ (h\ n)\ (h\ 'N))$ 
    using insert by (simp)
  also have  $insert\ (h\ n)\ (h\ 'N) = h\ 'insert\ n\ N$  by simp
  finally show ?case .
qed

lemma fold1-eq-fold-idem:
assumes finite A
shows  $fold1\ times\ (insert\ a\ A) = fold\ times\ a\ A$ 
proof (cases  $a \in A$ )
  case False
  with assms show ?thesis by (simp add: fold1-eq-fold)
next
  interpret fun-left-comm-idem times by (fact fun-left-comm-idem)
  case True then obtain b B
    where A:  $A = insert\ a\ B$  and  $a \notin B$  by (rule set-insert)
  with assms have finite B by auto
  then have  $fold\ times\ a\ (insert\ a\ B) = fold\ times\ (a * a)\ B$ 
    using  $\langle a \notin B \rangle$  by (rule fold-insert2)
  then show ?thesis
    using  $\langle a \notin B \rangle \langle finite\ B \rangle$  by (simp add: fold1-eq-fold A)
qed

end

```

Now the recursion rules for definitions:

```

lemma fold1-singleton-def:  $g = fold1\ f \implies g\ \{a\} = a$ 
by simp

lemma (in ab-semigroup-mult) fold1-insert-def:
   $\llbracket g = fold1\ times; finite\ A; x \notin A; A \neq \{\} \rrbracket \implies g\ (insert\ x\ A) = x * g\ A$ 
by (simp add: fold1-insert)

lemma (in ab-semigroup-idem-mult) fold1-insert-idem-def:
   $\llbracket g = fold1\ times; finite\ A; A \neq \{\} \rrbracket \implies g\ (insert\ x\ A) = x * g\ A$ 
by simp

```

### 20.5.1 Determinacy for *fold1Set*

**declare**

*empty-fold-graphE* [rule del] *fold-graph.intros* [rule del]  
*empty-fold1SetE* [rule del] *insert-fold1SetE* [rule del]  
 — No more proofs involve these relations.

### 20.5.2 Lemmas about *fold1*

**context** *ab-semigroup-mult*

**begin**

**lemma** *fold1-Un*:

**assumes** *A*: *finite A*  $A \neq \{\}$

**shows** *finite B*  $\implies B \neq \{\} \implies A \text{ Int } B = \{\} \implies$

$\text{fold1 times } (A \text{ Un } B) = \text{fold1 times } A * \text{fold1 times } B$

**using** *A* **by** (*induct rule: finite-ne-induct*)

(*simp-all add: fold1-insert mult-assoc*)

**lemma** *fold1-in*:

**assumes** *A*: *finite (A)*  $A \neq \{\}$  **and** *elem*:  $\bigwedge x y. x * y \in \{x, y\}$

**shows** *fold1 times A*  $\in A$

**using** *A*

**proof** (*induct rule: finite-ne-induct*)

**case** *singleton* **thus** ?*case* **by** *simp*

**next**

**case** *insert* **thus** ?*case* **using** *elem* **by** (*force simp add: fold1-insert*)

**qed**

**end**

**lemma** (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:

**assumes** *A*: *finite A*  $A \neq \{\}$

**shows** *finite B*  $\implies B \neq \{\} \implies$

$\text{fold1 times } (A \text{ Un } B) = \text{fold1 times } A * \text{fold1 times } B$

**using** *A*

**proof** (*induct rule: finite-ne-induct*)

**case** *singleton* **thus** ?*case* **by** *simp*

**next**

**case** *insert* **thus** ?*case* **by** (*simp add: mult-assoc*)

**qed**

## 20.6 Locales as mini-packages for fold operations

### 20.6.1 The natural case

**locale** *folding* =

**fixes** *f* :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'b

**fixes** *F* :: 'a *set*  $\Rightarrow$  'b  $\Rightarrow$  'b

**assumes** *commute-comp*:  $f y \circ f x = f x \circ f y$



**assumes** *eq-fold*:  $\text{finite } A \implies F\ A\ s = \text{fold } f\ s\ A$   
**begin**

**lemma** *empty* [*simp*]:

$F\ \{\} = \text{id}$

**by** (*simp add: eq-fold expand-fun-eq*)

**lemma** *insert* [*simp*]:

**assumes** *finite*  $A$  **and**  $x \notin A$

**shows**  $F\ (\text{insert } x\ A) = F\ A \circ f\ x$

**proof** –

**interpret** *fun-left-comm*  $f$  **proof**

**qed** (*insert commute-comp, simp add: expand-fun-eq*)

**from** *fold-insert2 assms*

**have**  $\bigwedge s. \text{fold } f\ s\ (\text{insert } x\ A) = \text{fold } f\ (f\ x\ s)\ A$ .

**with**  $\langle \text{finite } A \rangle$  **show** *?thesis* **by** (*simp add: eq-fold expand-fun-eq*)

**qed**

**lemma** *remove*:

**assumes** *finite*  $A$  **and**  $x \in A$

**shows**  $F\ A = F\ (A - \{x\}) \circ f\ x$

**proof** –

**from**  $\langle x \in A \rangle$  **obtain**  $B$  **where**  $A = \text{insert } x\ B$  **and**  $x \notin B$

**by** (*auto dest: mk-disjoint-insert*)

**moreover from**  $\langle \text{finite } A \rangle$  **this have** *finite*  $B$  **by** *simp*

**ultimately show** *?thesis* **by** *simp*

**qed**

**lemma** *insert-remove*:

**assumes** *finite*  $A$

**shows**  $F\ (\text{insert } x\ A) = F\ (A - \{x\}) \circ f\ x$

**using** *assms* **by** (*cases*  $x \in A$ ) (*simp-all add: remove insert-absorb*)

**lemma** *commute-left-comp*:

$f\ y \circ (f\ x \circ g) = f\ x \circ (f\ y \circ g)$

**by** (*simp add: o-assoc commute-comp*)

**lemma** *commute-comp'*:

**assumes** *finite*  $A$

**shows**  $f\ x \circ F\ A = F\ A \circ f\ x$

**using** *assms* **by** (*induct*  $A$ )

(*simp, simp del: o-apply add: o-assoc, simp del: o-apply add: o-assoc [symmetric]*  
*commute-comp*)

**lemma** *commute-left-comp'*:

**assumes** *finite*  $A$

**shows**  $f\ x \circ (F\ A \circ g) = F\ A \circ (f\ x \circ g)$

**using** *assms* **by** (*simp add: o-assoc commute-comp'*)

**lemma** *commute-comp''*:

**assumes** *finite A* **and** *finite B*

**shows**  $F B \circ F A = F A \circ F B$

**using** *assms* **by** (*induct A*)

(*simp-all add: o-assoc, simp add: o-assoc [symmetric] commute-comp'*)

**lemma** *commute-left-comp''*:

**assumes** *finite A* **and** *finite B*

**shows**  $F B \circ (F A \circ g) = F A \circ (F B \circ g)$

**using** *assms* **by** (*simp add: o-assoc commute-comp''*)

**lemmas** *commute-comps = o-assoc [symmetric] commute-comp commute-left-comp  
commute-comp' commute-left-comp' commute-comp'' commute-left-comp''*

**lemma** *union-inter*:

**assumes** *finite A* **and** *finite B*

**shows**  $F (A \cup B) \circ F (A \cap B) = F A \circ F B$

**using** *assms* **by** (*induct A*)

(*simp-all del: o-apply add: insert-absorb Int-insert-left commute-comps,  
simp add: o-assoc*)

**lemma** *union*:

**assumes** *finite A* **and** *finite B*

**and**  $A \cap B = \{\}$

**shows**  $F (A \cup B) = F A \circ F B$

**proof** –

**from** *union-inter*  $\langle \text{finite } A \rangle \langle \text{finite } B \rangle$  **have**  $F (A \cup B) \circ F (A \cap B) = F A \circ F B$

$B$  .

**with**  $\langle A \cap B = \{\} \rangle$  **show** *?thesis* **by** *simp*

**qed**

**end**

## 20.6.2 The natural case with idempotency

**locale** *folding-idem = folding +*

**assumes** *idem-comp*:  $f x \circ f x = f x$

**begin**

**lemma** *idem-left-comp*:

$f x \circ (f x \circ g) = f x \circ g$

**by** (*simp add: o-assoc idem-comp*)

**lemma** *in-comp-idem*:

**assumes** *finite A* **and**  $x \in A$

**shows**  $F A \circ f x = F A$

**using** *assms* **by** (*induct A*)

(*auto simp add: commute-comps idem-comp, simp add: commute-left-comp' [symmetric] commute-comp'*)

```

lemma subset-comp-idem:
  assumes finite A and  $B \subseteq A$ 
  shows  $F A \circ F B = F A$ 
proof –
  from assms have finite B by (blast dest: finite-subset)
  then show ?thesis using  $\langle B \subseteq A \rangle$  by (induct B)
    (simp-all add: o-assoc in-comp-idem <finite A>)
qed

declare insert [simp del]

lemma insert-idem [simp]:
  assumes finite A
  shows  $F (\text{insert } x A) = F A \circ f x$ 
  using assms by (cases x ∈ A) (simp-all add: insert in-comp-idem insert-absorb)

lemma union-idem:
  assumes finite A and finite B
  shows  $F (A \cup B) = F A \circ F B$ 
proof –
  from assms have finite (A ∪ B) and  $A \cap B \subseteq A \cup B$  by auto
  then have  $F (A \cup B) \circ F (A \cap B) = F (A \cup B)$  by (rule subset-comp-idem)
  with assms show ?thesis by (simp add: union-inter)
qed

end

```

### 20.6.3 The image case with fixed function

```

no-notation times (infixl * 70)
no-notation Groups.one (1)

locale folding-image-simple = comm-monoid +
  fixes  $g :: 'b \Rightarrow 'a$ 
  fixes  $F :: 'b \text{ set} \Rightarrow 'a$ 
  assumes eq-fold-g:  $\text{finite } A \implies F A = \text{fold-image } f g 1 A$ 
begin

lemma empty [simp]:
   $F \{\} = 1$ 
  by (simp add: eq-fold-g)

lemma insert [simp]:
  assumes finite A and  $x \notin A$ 
  shows  $F (\text{insert } x A) = g x * F A$ 
proof –
  interpret fun-left-comm  $\%x y. (g x) * y$  proof
  qed (simp add: ac-simps)

```

**with** *assms* **have** *fold-image* (*op* \*) *g* 1 (*insert* *x* *A*) = *g* *x* \* *fold-image* (*op* \*)  
*g* 1 *A*  
**by** (*simp add: fold-image-def*)  
**with**  $\langle \text{finite } A \rangle$  **show** ?thesis **by** (*simp add: eq-fold-g*)  
**qed**

**lemma** *remove*:  
**assumes** *finite* *A* **and**  $x \in A$   
**shows**  $F\ A = g\ x * F\ (A - \{x\})$   
**proof** –  
**from**  $\langle x \in A \rangle$  **obtain** *B* **where** *A*:  $A = \text{insert } x\ B$  **and**  $x \notin B$   
**by** (*auto dest: mk-disjoint-insert*)  
**moreover from**  $\langle \text{finite } A \rangle$  **this have** *finite* *B* **by** *simp*  
**ultimately show** ?thesis **by** *simp*  
**qed**

**lemma** *insert-remove*:  
**assumes** *finite* *A*  
**shows**  $F\ (\text{insert } x\ A) = g\ x * F\ (A - \{x\})$   
**using** *assms* **by** (*cases*  $x \in A$ ) (*simp-all add: remove insert-absorb*)

**lemma** *neutral*:  
**assumes** *finite* *A* **and**  $\forall x \in A. g\ x = 1$   
**shows**  $F\ A = 1$   
**using** *assms* **by** (*induct* *A*) *simp-all*

**lemma** *union-inter*:  
**assumes** *finite* *A* **and** *finite* *B*  
**shows**  $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$   
**using** *assms* **proof** (*induct* *A*)  
**case** *empty* **then show** ?case **by** *simp*  
**next**  
**case** (*insert* *x* *A*) **then show** ?case  
**by** (*auto simp add: insert-absorb Int-insert-left commute [of - g x] assoc left-commute*)  
**qed**

**corollary** *union-inter-neutral*:  
**assumes** *finite* *A* **and** *finite* *B*  
**and** *I0*:  $\forall x \in A \cap B. g\ x = 1$   
**shows**  $F\ (A \cup B) = F\ A * F\ B$   
**using** *assms* **by** (*simp add: union-inter [symmetric] neutral*)

**corollary** *union-disjoint*:  
**assumes** *finite* *A* **and** *finite* *B*  
**assumes**  $A \cap B = \{\}$   
**shows**  $F\ (A \cup B) = F\ A * F\ B$   
**using** *assms* **by** (*simp add: union-inter-neutral*)

end

#### 20.6.4 The image case with flexible function

```

locale folding-image = comm-monoid +
  fixes  $F :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'a$ 
  assumes eq-fold:  $\bigwedge g. \text{finite } A \Longrightarrow F\ g\ A = \text{fold-image } f\ g\ 1\ A$ 

sublocale folding-image < folding-image-simple op * 1 g F g proof
qed (fact eq-fold)

context folding-image
begin

lemma reindex:
  assumes finite A and inj-on h A
  shows  $F\ g\ (h\ ` A) = F\ (g \circ h)\ A$ 
  using assms by (induct A) auto

lemma cong:
  assumes finite A and  $\bigwedge x. x \in A \Longrightarrow g\ x = h\ x$ 
  shows  $F\ g\ A = F\ h\ A$ 
proof –
  from assms have  $ALL\ C. C \leq A \longleftrightarrow (ALL\ x:C. g\ x = h\ x) \longleftrightarrow F\ g\ C = F\ h\ C$ 
  apply – apply (erule finite-induct) apply simp
  apply (simp add: subset-insert-iff, clarify)
  apply (subgoal-tac finite C)
  prefer 2 apply (blast dest: finite-subset [COMP swap-prems-rl])
  apply (subgoal-tac C = insert x (C - {x}))
  prefer 2 apply blast
  apply (erule ssubst)
  apply (erule spec)
  apply (erule (1) notE impE)
  apply (simp add: Ball-def del: insert-Diff-single)
  done
  with assms show ?thesis by simp
qed

lemma UNION-disjoint:
  assumes finite I and  $\forall i \in I. \text{finite } (A\ i)$ 
  and  $\forall i \in I. \forall j \in I. i \neq j \longrightarrow A\ i \cap A\ j = \{\}$ 
  shows  $F\ g\ (\text{UNION } I\ A) = F\ (F\ g \circ A)\ I$ 
apply (insert assms)
apply (induct set: finite, simp, atomize)
apply (subgoal-tac  $\forall i \in Fa. x \neq i$ )
  prefer 2 apply blast
apply (subgoal-tac  $A\ x\ Int\ \text{UNION } Fa\ A = \{\}$ )
  prefer 2 apply blast

```

**apply** (*simp add: union-disjoint*)  
**done**

**lemma** *distrib*:  
 assumes *finite A*  
 shows  $F (\lambda x. g x * h x) A = F g A * F h A$   
 using *assms* **by** (*rule finite-induct*) (*simp-all add: assoc commute left-commute*)

**lemma** *related*:  
 assumes *Re: R 1 1*  
 and *Rop:  $\forall x1 y1 x2 y2. R x1 x2 \wedge R y1 y2 \longrightarrow R (x1 * y1) (x2 * y2)$*   
 and *fS: finite S* and *Rfg:  $\forall x \in S. R (h x) (g x)$*   
 shows  $R (F h S) (F g S)$   
 using *fS* **by** (*rule finite-subset-induct*) (*insert assms, auto*)

**lemma** *eq-general*:  
 assumes *fS: finite S*  
 and *h:  $\forall y \in S'. \exists !x. x \in S \wedge h x = y$*   
 and *f12:  $\forall x \in S. h x \in S' \wedge f2 (h x) = f1 x$*   
 shows  $F f1 S = F f2 S'$   
**proof**–  
 from *h f12* **have** *hS:  $h ' S = S'$*  **by** *blast*  
 {**fix** *x y* **assume** *H:  $x \in S \wedge y \in S \wedge h x = h y$*   
   **from** *f12 h H* **have**  $x = y$  **by** *auto* }  
 hence *hinj: inj-on h S* **unfolding** *inj-on-def Ex1-def* **by** *blast*  
 from *f12* **have** *th:  $\bigwedge x. x \in S \implies (f2 \circ h) x = f1 x$*  **by** *auto*  
 from *hS* **have**  $F f2 S' = F f2 (h ' S)$  **by** *simp*  
 also **have**  $\dots = F (f2 \circ h) S$  **using** *reindex [OF fS hinj, of f2]* .  
 also **have**  $\dots = F f1 S$  **using** *th cong [OF fS, of f2 o h f1]*  
   **by** *blast*  
 finally **show** *?thesis ..*  
**qed**

**lemma** *eq-general-inverses*:  
 assumes *fS: finite S*  
 and *kh:  $\bigwedge y. y \in T \implies k y \in S \wedge h (k y) = y$*   
 and *hk:  $\bigwedge x. x \in S \implies h x \in T \wedge k (h x) = x \wedge g (h x) = j x$*   
 shows  $F j S = F g T$

**apply** (*rule eq-general [OF fS, of T h g j]*)  
**apply** (*rule ballI*)  
**apply** (*frule kh*)  
**apply** (*rule ex1I[]*)  
**apply** *blast*  
**apply** *clarsimp*  
**apply** (*drule hk*) **apply** *simp*  
**apply** (*rule sym*)  
**apply** (*erule conjunct1 [OF conjunct2 [OF hk]]*)  
**apply** (*rule ballI*)

```

    apply (drule hk)
    apply blast
  done

```

```

end

```

### 20.6.5 The image case with fixed function and idempotency

```

locale folding-image-simple-idem = folding-image-simple +
  assumes idem:  $x * x = x$ 

```

```

sublocale folding-image-simple-idem < semilattice proof
qed (fact idem)

```

```

context folding-image-simple-idem
begin

```

```

lemma in-idem:
  assumes finite A and  $x \in A$ 
  shows  $g\ x * F\ A = F\ A$ 
  using assms by (induct A) (auto simp add: left-commute)

```

```

lemma subset-idem:
  assumes finite A and  $B \subseteq A$ 
  shows  $F\ B * F\ A = F\ A$ 
proof –
  from assms have finite B by (blast dest: finite-subset)
  then show ?thesis using  $\langle B \subseteq A \rangle$  by (induct B)
    (auto simp add: assoc in-idem finite A)
qed

```

```

declare insert [simp del]

```

```

lemma insert-idem [simp]:
  assumes finite A
  shows  $F\ (\text{insert } x\ A) = g\ x * F\ A$ 
  using assms by (cases x ∈ A) (simp-all add: insert in-idem insert-absorb)

```

```

lemma union-idem:
  assumes finite A and finite B
  shows  $F\ (A \cup B) = F\ A * F\ B$ 
proof –
  from assms have finite (A ∪ B) and  $A \cap B \subseteq A \cup B$  by auto
  then have  $F\ (A \cap B) * F\ (A \cup B) = F\ (A \cup B)$  by (rule subset-idem)
  with assms show ?thesis by (simp add: union-inter [of A B, symmetric] com-
mute)
qed

```

```

end

```

### 20.6.6 The image case with flexible function and idempotency

**locale** *folding-image-idem* = *folding-image* +  
**assumes** *idem*:  $x * x = x$

**sublocale** *folding-image-idem* < *folding-image-simple-idem* *op* \* 1 *g* *F* *g* **proof**  
**qed** (*fact idem*)

### 20.6.7 The neutral-less case

**locale** *folding-one* = *abel-semigroup* +  
**fixes** *F* ::  $'a \text{ set} \Rightarrow 'a$   
**assumes** *eq-fold*:  $\text{finite } A \Longrightarrow F A = \text{fold1 } f A$   
**begin**

**lemma** *singleton* [*simp*]:  
 $F \{x\} = x$   
**by** (*simp add: eq-fold*)

**lemma** *eq-fold'*:  
**assumes** *finite A* **and**  $x \notin A$   
**shows**  $F (\text{insert } x A) = \text{fold } (op *) x A$   
**proof** –  
**interpret** *ab-semigroup-mult* *op* \* **proof** **qed** (*simp-all add: ac-simps*)  
**with** *assms* **show** ?thesis **by** (*simp add: eq-fold fold1-eq-fold*)  
**qed**

**lemma** *insert* [*simp*]:  
**assumes** *finite A* **and**  $x \notin A$  **and**  $A \neq \{\}$   
**shows**  $F (\text{insert } x A) = x * F A$   
**proof** –  
**from**  $\langle A \neq \{\} \rangle$  **obtain** *b* **where**  $b \in A$  **by** *blast*  
**then obtain** *B* **where** \*:  $A = \text{insert } b B$   $b \notin B$  **by** (*blast dest: mk-disjoint-insert*)  
**with**  $\langle \text{finite } A \rangle$  **have** *finite B* **by** *simp*  
**interpret** *fold*:  $\text{folding } op * \lambda a b. \text{fold } (op *) b a$  **proof**  
**qed** (*simp-all add: expand-fun-eq ac-simps*)  
**thm** *fold commute-comp'* [*of B b, simplified expand-fun-eq, simplified*]  
**from**  $\langle \text{finite } B \rangle$  *fold commute-comp'* [*of B x*]  
**have**  $op * x \circ (\lambda b. \text{fold } op * b B) = (\lambda b. \text{fold } op * b B) \circ op * x$  **by** *simp*  
**then have** *A*:  $x * \text{fold } op * b B = \text{fold } op * (b * x) B$  **by** (*simp add: expand-fun-eq commute*)  
**from**  $\langle \text{finite } B \rangle$  \* *fold.insert* [*of B b*]  
**have**  $(\lambda x. \text{fold } op * x (\text{insert } b B)) = (\lambda x. \text{fold } op * x B) \circ op * b$  **by** *simp*  
**then have** *B*:  $\text{fold } op * x (\text{insert } b B) = \text{fold } op * (b * x) B$  **by** (*simp add: expand-fun-eq*)  
**from** *A B assms* \* **show** ?thesis **by** (*simp add: eq-fold' del: fold.insert*)  
**qed**

**lemma** *remove*:  
**assumes** *finite A* **and**  $x \in A$



shows  $F A = (\text{if } A - \{x\} = \{\} \text{ then } x \text{ else } x * F (A - \{x\}))$   
**proof** –  
 from *assms* obtain  $B$  where  $A = \text{insert } x B$  and  $x \notin B$  by (*blast dest: mk-disjoint-insert*)  
 with *assms* show ?thesis by *simp*  
**qed**

**lemma** *insert-remove*:  
 assumes *finite A*  
 shows  $F (\text{insert } x A) = (\text{if } A - \{x\} = \{\} \text{ then } x \text{ else } x * F (A - \{x\}))$   
 using *assms* by (*cases x ∈ A*) (*simp-all add: insert-absorb remove*)

**lemma** *union-disjoint*:  
 assumes *finite A A ≠ {}* and *finite B B ≠ {}* and  $A \cap B = \{\}$   
 shows  $F (A \cup B) = F A * F B$   
 using *assms* by (*induct A rule: finite-ne-induct*) (*simp-all add: ac-simps*)

**lemma** *union-inter*:  
 assumes *finite A* and *finite B* and  $A \cap B \neq \{\}$   
 shows  $F (A \cup B) * F (A \cap B) = F A * F B$   
**proof** –  
 from *assms* have  $A \neq \{\}$  and  $B \neq \{\}$  by *auto*  
 from  $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle \langle A \cap B \neq \{\} \rangle$  show ?thesis **proof** (*induct A rule: finite-ne-induct*)  
 case (*singleton x*) then show ?case by (*simp add: insert-absorb ac-simps*)  
 next  
 case (*insert x A*) show ?case **proof** (*cases x ∈ B*)  
 case *True* then have  $B \neq \{\}$  by *auto*  
 with *insert True*  $\langle \text{finite } B \rangle$  show ?thesis by (*cases A ∩ B = {}*)  
 (*simp-all add: insert-absorb ac-simps union-disjoint*)  
 next  
 case *False* with *insert* have  $F (A \cup B) * F (A \cap B) = F A * F B$  by *simp*  
 moreover from *False*  $\langle \text{finite } B \rangle$  *insert* have *finite*  $(A \cup B)$   $x \notin A \cup B$   $A \cup B \neq \{\}$   
 by *auto*  
 ultimately show ?thesis using *False*  $\langle \text{finite } A \rangle \langle x \notin A \rangle \langle A \neq \{\} \rangle$  by (*simp add: assoc*)  
**qed**  
**qed**  
**qed**

**lemma** *closed*:  
 assumes *finite A A ≠ {}* and *elem:  $\bigwedge x y. x * y \in \{x, y\}$*   
 shows  $F A \in A$   
 using  $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle$  **proof** (*induct rule: finite-ne-induct*)  
 case *singleton* then show ?case by *simp*  
 next  
 case *insert* with *elem* show ?case by *force*  
**qed**

end

### 20.6.8 The neutral-less case with idempotency

locale *folding-one-idem* = *folding-one* +  
 assumes *idem*:  $x * x = x$

sublocale *folding-one-idem* < *semilattice* **proof**  
**qed** (*fact idem*)

context *folding-one-idem*  
 begin

lemma *in-idem*:  
 assumes *finite* *A* and  $x \in A$   
 shows  $x * F A = F A$   
**proof** –  
 from *assms* have  $A \neq \{\}$  by *auto*  
 with  $\langle \text{finite } A \rangle$  show ?thesis using  $\langle x \in A \rangle$  by (induct *A* rule: *finite-ne-induct*)  
 (auto simp add: *ac-simps*)  
**qed**

lemma *subset-idem*:  
 assumes *finite* *A*  $B \neq \{\}$  and  $B \subseteq A$   
 shows  $F B * F A = F A$   
**proof** –  
 from *assms* have *finite* *B* by (blast dest: *finite-subset*)  
 then show ?thesis using  $\langle B \neq \{\} \rangle \langle B \subseteq A \rangle$  by (induct *B* rule: *finite-ne-induct*)  
 (simp-all add: *assoc in-idem*  $\langle \text{finite } A \rangle$ )  
**qed**

lemma *eq-fold-idem'*:  
 assumes *finite* *A*  
 shows  $F (\text{insert } a A) = \text{fold } (op *) a A$   
**proof** –  
 interpret *ab-semigroup-idem-mult* *op* \* **proof** **qed** (*simp-all* add: *ac-simps*)  
 with *assms* show ?thesis by (simp add: *eq-fold fold1-eq-fold-idem*)  
**qed**

lemma *insert-idem* [*simp*]:  
 assumes *finite* *A* and  $A \neq \{\}$   
 shows  $F (\text{insert } x A) = x * F A$   
**proof** (*cases*  $x \in A$ )  
 case *False* from  $\langle \text{finite } A \rangle \langle x \notin A \rangle \langle A \neq \{\} \rangle$  show ?thesis by (rule *insert*)  
 next  
 case *True*  
 from  $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle$  show ?thesis by (simp add: *in-idem insert-absorb True*)  
**qed**

**lemma** *union-idem*:

assumes *finite A*  $A \neq \{\}$  and *finite B*  $B \neq \{\}$   
 shows  $F (A \cup B) = F A * F B$   
**proof** (*cases A ∩ B = {}*)  
 case *True* **with** *assms* **show** *?thesis* **by** (*simp add: union-disjoint*)  
**next**  
 case *False*  
 from *assms* **have** *finite (A ∪ B)* and  $A \cap B \subseteq A \cup B$  **by** *auto*  
 with *False* **have**  $F (A \cap B) * F (A \cup B) = F (A \cup B)$  **by** (*auto intro: subset-idem*)  
 with *assms False* **show** *?thesis* **by** (*simp add: union-inter [of A B, symmetric] commute*)  
**qed**

**lemma** *hom-commute*:

assumes *hom*:  $\bigwedge x y. h (x * y) = h x * h y$   
 and *N*: *finite N*  $N \neq \{\}$  **shows**  $h (F N) = F (h ' N)$   
**using N proof** (*induct rule: finite-ne-induct*)  
 case *singleton* **thus** *?case* **by** *simp*  
**next**  
 case (*insert n N*)  
 then **have**  $h (F (insert n N)) = h (n * F N)$  **by** *simp*  
 also **have**  $\dots = h n * h (F N)$  **by** (*rule hom*)  
 also **have**  $h (F N) = F (h ' N)$  **by** (*rule insert*)  
 also **have**  $h n * \dots = F (insert (h n) (h ' N))$   
 using *insert* **by** (*simp*)  
 also **have**  $insert (h n) (h ' N) = h ' insert n N$  **by** *simp*  
 finally **show** *?case* .  
**qed**

**end**

**notation** *times* (*infixl* \* 70)

**notation** *Groups.one* (1)

## 20.7 Finite cardinality

This definition, although traditional, is ugly to work with:  $card A == LEAST n. EX f. A = \{f i \mid i. i < n\}$ . But now that we have *fold-image* things are easy:

**definition** *card* :: 'a set  $\Rightarrow$  nat **where**

$card A = (if\ finite\ A\ then\ fold-image\ (op\ +)\ (\lambda x. 1)\ 0\ A\ else\ 0)$

**interpretation** *card!*: *folding-image-simple op + 0 λx. 1 card* **proof**

**qed** (*simp add: card-def*)

**lemma** *card-infinite* [*simp*]:

$\neg finite\ A \Longrightarrow card\ A = 0$

```

by (simp add: card-def)

lemma card-empty:
  card {} = 0
by (fact card.empty)

lemma card-insert-disjoint:
  finite A ==> x ∉ A ==> card (insert x A) = Suc (card A)
by simp

lemma card-insert-if:
  finite A ==> card (insert x A) = (if x ∈ A then card A else Suc (card A))
by auto (simp add: card.insert-remove card.remove)

lemma card-ge-0-finite:
  card A > 0 ==> finite A
by (rule ccontr) simp

lemma card-0-eq [simp, no-atp]:
  finite A ==> card A = 0 ⟷ A = {}
by (auto dest: mk-disjoint-insert)

lemma finite-UNIV-card-ge-0:
  finite (UNIV :: 'a set) ==> card (UNIV :: 'a set) > 0
by (rule ccontr) simp

lemma card-eq-0-iff:
  card A = 0 ⟷ A = {} ∨ ¬ finite A
by auto

lemma card-gt-0-iff:
  0 < card A ⟷ A ≠ {} ∧ finite A
by (simp add: neq0-conv [symmetric] card-eq-0-iff)

lemma card-Suc-Diff1: finite A ==> x: A ==> Suc (card (A - {x})) = card A
apply(rule-tac t = A in insert-Diff [THEN subst], assumption)
apply(simp del:insert-Diff-single)
done

lemma card-Diff-singleton:
  finite A ==> x: A ==> card (A - {x}) = card A - 1
by (simp add: card-Suc-Diff1 [symmetric])

lemma card-Diff-singleton-if:
  finite A ==> card (A - {x}) = (if x : A then card A - 1 else card A)
by (simp add: card-Diff-singleton)

lemma card-Diff-insert[simp]:
assumes finite A and a:A and a ~: B

```

**shows**  $\text{card}(A - \text{insert } a \ B) = \text{card}(A - B) - 1$

**proof** –

**have**  $A - \text{insert } a \ B = (A - B) - \{a\}$  **using** *assms* **by** *blast*

**then show** *?thesis* **using** *assms* **by** (*simp add: card-Diff-singleton*)

**qed**

**lemma** *card-insert: finite A ==> card (insert x A) = Suc (card (A - {x}))*

**by** (*simp add: card-insert-if card-Suc-Diff1 del: card-Diff-insert*)

**lemma** *card-insert-le: finite A ==> card A <= card (insert x A)*

**by** (*simp add: card-insert-if*)

**lemma** *card-mono:*

**assumes** *finite B* **and**  $A \subseteq B$

**shows**  $\text{card } A \leq \text{card } B$

**proof** –

**from** *assms* **have** *finite A* **by** (*auto intro: finite-subset*)

**then show** *?thesis* **using** *assms* **proof** (*induct A arbitrary: B*)

**case empty** **then show** *?case* **by** *simp*

**next**

**case** (*insert x A*)

**then have**  $x \in B$  **by** *simp*

**from** *insert* **have**  $A \subseteq B - \{x\}$  **and** *finite (B - {x})* **by** *auto*

**with** *insert.hyps* **have**  $\text{card } A \leq \text{card } (B - \{x\})$  **by** *auto*

**with** (*finite A*)  $x \notin A$  (*finite B*)  $x \in B$  **show** *?case* **by** *simp (simp only:*

*card.remove*)

**qed**

**qed**

**lemma** *card-seteq: finite B ==> (!A. A <= B ==> card B <= card A ==> A = B)*

**apply** (*induct set: finite, simp, clarify*)

**apply** (*subgoal-tac finite A & A - {x} <= F*)

**prefer** 2 **apply** (*blast intro: finite-subset, atomize*)

**apply** (*drule-tac x = A - {x} in spec*)

**apply** (*simp add: card-Diff-singleton-if split add: split-if-asm*)

**apply** (*case-tac card A, auto*)

**done**

**lemma** *psubset-card-mono: finite B ==> A < B ==> card A < card B*

**apply** (*simp add: psubset-eq linorder-not-le [symmetric]*)

**apply** (*blast dest: card-seteq*)

**done**

**lemma** *card-Un-Int: finite A ==> finite B*

**==>**  $\text{card } A + \text{card } B = \text{card } (A \text{ Un } B) + \text{card } (A \text{ Int } B)$

**by** (*fact card.union-inter [symmetric]*)

**lemma** *card-Un-disjoint: finite A ==> finite B*

$\Rightarrow A \text{ Int } B = \{\} \Rightarrow \text{card } (A \text{ Un } B) = \text{card } A + \text{card } B$   
**by** (fact card.union-disjoint)

**lemma** card-Diff-subset:  
 assumes finite  $B$  and  $B \subseteq A$   
 shows  $\text{card } (A - B) = \text{card } A - \text{card } B$   
**proof** (cases finite  $A$ )  
 case False with assms show ?thesis by simp  
 next  
 case True with assms show ?thesis by (induct  $B$  arbitrary:  $A$ ) simp-all  
**qed**

**lemma** card-Diff-subset-Int:  
 assumes  $AB$ : finite  $(A \cap B)$  shows  $\text{card } (A - B) = \text{card } A - \text{card } (A \cap B)$   
**proof** –  
 have  $A - B = A - A \cap B$  by auto  
 thus ?thesis  
 by (simp add: card-Diff-subset  $AB$ )  
**qed**

**lemma** card-Diff1-less: finite  $A \Rightarrow x: A \Rightarrow \text{card } (A - \{x\}) < \text{card } A$   
**apply** (rule Suc-less-SucD)  
**apply** (simp add: card-Suc-Diff1 del:card-Diff-insert)  
**done**

**lemma** card-Diff2-less:  
 finite  $A \Rightarrow x: A \Rightarrow y: A \Rightarrow \text{card } (A - \{x\} - \{y\}) < \text{card } A$   
**apply** (case-tac  $x = y$ )  
**apply** (simp add: card-Diff1-less del:card-Diff-insert)  
**apply** (rule less-trans)  
**prefer** 2 **apply** (auto intro!: card-Diff1-less simp del:card-Diff-insert)  
**done**

**lemma** card-Diff1-le: finite  $A \Rightarrow \text{card } (A - \{x\}) \leq \text{card } A$   
**apply** (case-tac  $x : A$ )  
**apply** (simp-all add: card-Diff1-less less-imp-le)  
**done**

**lemma** card-psubset: finite  $B \Rightarrow A \subseteq B \Rightarrow \text{card } A < \text{card } B \Rightarrow A < B$   
**by** (erule psubsetI, blast)

**lemma** insert-partition:  
 $\llbracket x \notin F; \forall c1 \in \text{insert } x F. \forall c2 \in \text{insert } x F. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \rrbracket$   
 $\Rightarrow x \cap \bigcup F = \{\}$   
**by** auto

**lemma** finite-psubset-induct[consumes 1, case-names psubset]:  
 assumes fin: finite  $A$   
 and major:  $\bigwedge A. \text{finite } A \Rightarrow (\bigwedge B. B \subset A \Rightarrow P B) \Rightarrow P A$

```

shows  $P A$ 
using fin
proof (induct  $A$  taking: card rule: measure-induct-rule)
  case (less A)
  have fin: finite A by fact
  have ih:  $\bigwedge B. \llbracket \text{card } B < \text{card } A; \text{finite } B \rrbracket \implies P B$  by fact
  { fix  $B$ 
    assume asm:  $B \subset A$ 
    from asm have  $\text{card } B < \text{card } A$  using psubset-card-mono fin by blast
    moreover
    from asm have  $B \subseteq A$  by auto
    then have finite B using fin finite-subset by blast
    ultimately
    have  $P B$  using ih by simp
  }
  with fin show  $P A$  using major by blast
qed

```

main cardinality theorem

```

lemma card-partition [rule-format]:
  finite C ==>
    finite ( $\bigcup C$ ) -->
      ( $\forall c \in C. \text{card } c = k$ ) -->
        ( $\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \implies c1 \cap c2 = \{\}$ ) -->
           $k * \text{card}(C) = \text{card}(\bigcup C)$ 
  apply (erule finite-induct, simp)
  apply (simp add: card-Un-disjoint insert-partition
    finite-subset [of -  $\bigcup$  (insert  $x F$ )])
done

```

```

lemma card-eq-UNIV-imp-eq-UNIV:
  assumes fin: finite (UNIV :: 'a set)
  and card:  $\text{card } A = \text{card } (\text{UNIV} :: 'a \text{ set})$ 
  shows  $A = (\text{UNIV} :: 'a \text{ set})$ 
proof
  show  $A \subseteq \text{UNIV}$  by simp
  show  $\text{UNIV} \subseteq A$ 
  proof
    fix  $x$ 
    show  $x \in A$ 
    proof (rule ccontr)
      assume  $x \notin A$ 
      then have  $A \subset \text{UNIV}$  by auto
      with fin have  $\text{card } A < \text{card } (\text{UNIV} :: 'a \text{ set})$  by (fact psubset-card-mono)
      with card show False by simp
    qed
  qed
qed

```

The form of a finite set of given cardinality

```

lemma card-eq-SucD:
assumes card A = Suc k
shows  $\exists b \in B. A = \text{insert } b \ B \ \& \ b \notin B \ \& \ \text{card } B = k \ \& \ (k=0 \longrightarrow B=\{\})$ 
proof –
  have fin: finite A using assms by (auto intro: ccontr)
  moreover have card A  $\neq$  0 using assms by auto
  ultimately obtain b where b: b  $\in$  A by auto
  show ?thesis
  proof (intro exI conjI)
    show A = insert b (A - {b}) using b by blast
    show b  $\notin$  A - {b} by blast
    show card (A - {b}) = k and k = 0  $\longrightarrow$  A - {b} = {}
      using assms b fin by(fastsimp dest:mk-disjoint-insert)+
  qed
qed

```

```

lemma card-Suc-eq:
  (card A = Suc k) =
    ( $\exists b \in B. A = \text{insert } b \ B \ \& \ b \notin B \ \& \ \text{card } B = k \ \& \ (k=0 \longrightarrow B=\{\})$ )
apply(rule iffI)
  apply(erule card-eq-SucD)
apply(auto)
apply(subst card-insert)
apply(auto intro: ccontr)
done

```

```

lemma finite-fun-UNIVD2:
  assumes fin: finite (UNIV :: ('a  $\Rightarrow$  'b) set)
  shows finite (UNIV :: 'b set)
proof –
  from fin have finite (range ( $\lambda f :: 'a \Rightarrow 'b. f$  arbitrary))
    by(rule finite-imageI)
  moreover have UNIV = range ( $\lambda f :: 'a \Rightarrow 'b. f$  arbitrary)
    by(rule UNIV-eq-I) auto
  ultimately show finite (UNIV :: 'b set) by simp
qed

```

```

lemma card-UNIV-unit: card (UNIV :: unit set) = 1
  unfolding UNIV-unit by simp

```

### 20.7.1 Cardinality of image

```

lemma card-image-le: finite A  $\implies$  card (f ` A)  $\leq$  card A
apply (induct set: finite)
apply simp
apply (simp add: le-SucI card-insert-if)
done

```

```

lemma card-image:

```



```

  assumes inj-on  $f$   $A$ 
  shows  $\text{card } (f \text{ `` } A) = \text{card } A$ 
proof (cases finite  $A$ )
  case True then show ?thesis using assms by (induct  $A$ ) simp-all
next
  case False then have  $\neg \text{finite } (f \text{ `` } A)$  using assms by (auto dest: finite-imageD)
  with False show ?thesis by simp
qed

```

**lemma** *bij-betw-same-card*:  $\text{bij-betw } f \text{ } A \text{ } B \implies \text{card } A = \text{card } B$   
 by(*auto* *simp: card-image bij-betw-def*)

**lemma** *endo-inj-surj*:  $\text{finite } A \implies f \text{ `` } A \subseteq A \implies \text{inj-on } f \text{ } A \implies f \text{ `` } A = A$   
 by (*simp* *add: card-seteq card-image*)

**lemma** *eq-card-imp-inj-on*:  
 $[[ \text{finite } A; \text{card}(f \text{ `` } A) = \text{card } A ]] \implies \text{inj-on } f \text{ } A$   
 apply (*induct* *rule:finite-induct*)  
 apply *simp*  
 apply (*frule* *card-image-le*[*where*  $f = f$ ])  
 apply (*simp* *add:card-insert-if split:if-splits*)  
 done

**lemma** *inj-on-iff-eq-card*:  
 $\text{finite } A \implies \text{inj-on } f \text{ } A = (\text{card}(f \text{ `` } A) = \text{card } A)$   
 by(*blast* *intro: card-image eq-card-imp-inj-on*)

**lemma** *card-inj-on-le*:  
 $[[ \text{inj-on } f \text{ } A; f \text{ `` } A \subseteq B; \text{finite } B ]] \implies \text{card } A \leq \text{card } B$   
 apply (*subgoal-tac* *finite*  $A$ )  
 apply (*force* *intro: card-mono* *simp* *add: card-image* [*symmetric*])  
 apply (*blast* *intro: finite-imageD* *dest: finite-subset*)  
 done

**lemma** *card-bij-eq*:  
 $[[ \text{inj-on } f \text{ } A; f \text{ `` } A \subseteq B; \text{inj-on } g \text{ } B; g \text{ `` } B \subseteq A; \text{finite } A; \text{finite } B ]] \implies \text{card } A = \text{card } B$   
 by (*auto* *intro: le-antisym card-inj-on-le*)

### 20.7.2 Cardinality of sums

**lemma** *card-Plus*:  
 assumes *finite*  $A$  and *finite*  $B$   
 shows  $\text{card } (A <+> B) = \text{card } A + \text{card } B$   
 proof –  
 have  $\text{Inl `` } A \cap \text{Inr `` } B = \{\}$  by *fast*  
 with *assms* show ?thesis  
 unfolding *Plus-def*

by (simp add: card-Un-disjoint card-image)  
qed

**lemma** card-Plus-conv-if:  
card ( $A <+> B$ ) = (if finite  $A \wedge$  finite  $B$  then card  $A +$  card  $B$  else 0)  
by (auto simp add: card-Plus)

### 20.7.3 Cardinality of the Powerset

**lemma** card-Pow: finite  $A \implies$  card (Pow  $A$ ) = Suc (Suc 0)  $^$  card  $A$   
apply (induct set: finite)  
apply (simp-all add: Pow-insert)  
apply (subst card-Un-disjoint, blast)  
apply (blast intro: finite-imageI, blast)  
apply (subgoal-tac inj-on (insert  $x$ ) (Pow  $F$ ))  
apply (simp add: card-image Pow-insert)  
apply (unfold inj-on-def)  
apply (blast elim!: equalityE)  
done

Relates to equivalence classes. Based on a theorem of F. Kamm $\tilde{A}_{\frac{1}{4}}$ ller.

**lemma** dvd-partition:  
finite (Union  $C$ )  $\implies$   
ALL  $c : C$ .  $k$  dvd card  $c \implies$   
(ALL  $c1 : C$ . ALL  $c2 : C$ .  $c1 \neq c2 \longrightarrow c1 \text{ Int } c2 = \{\}$ )  $\implies$   
 $k$  dvd card (Union  $C$ )  
apply (frule finite-UnionD)  
apply (rotate-tac -1)  
apply (induct set: finite, simp-all, clarify)  
apply (subst card-Un-disjoint)  
apply (auto simp add: disjoint-eq-subset-Compl)  
done

### 20.7.4 Relating injectivity and surjectivity

**lemma** finite-surj-inj: finite( $A$ )  $\implies A \leq f^*A \implies$  inj-on  $f$   $A$   
apply (rule eq-card-imp-inj-on, assumption)  
apply (frule finite-imageI)  
apply (drule (1) card-seteq)  
apply (erule card-image-le)  
apply simp  
done

**lemma** finite-UNIV-surj-inj: fixes  $f :: 'a \Rightarrow 'a$   
shows finite (UNIV ::  $'a$  set)  $\implies$  surj  $f \implies$  inj  $f$   
by (blast intro: finite-surj-inj subset-UNIV dest: surj-range)

**lemma** finite-UNIV-inj-surj: fixes  $f :: 'a \Rightarrow 'a$   
shows finite (UNIV ::  $'a$  set)  $\implies$  inj  $f \implies$  surj  $f$   
by (fastsimp simp: surj-def dest!: endo-inj-surj)

```

corollary infinite-UNIV-nat[iff]:  $\sim \text{finite}(\text{UNIV}::\text{nat set})$ 
proof
  assume finite(UNIV::nat set)
  with finite-UNIV-inj-surj[of Suc]
  show False by simp (blast dest: Suc-neq-Zero surjD)
qed

```

```

lemma infinite-UNIV-char-0[no-atp]:
   $\neg \text{finite}(\text{UNIV}::'a::\text{semiring-char-0 set})$ 
proof
  assume finite (UNIV::'a set)
  with subset-UNIV have finite (range of-nat::'a set)
    by (rule finite-subset)
  moreover have inj (of-nat::nat  $\Rightarrow$  'a)
    by (simp add: inj-on-def)
  ultimately have finite (UNIV::nat set)
    by (rule finite-imageD)
  then show False
    by simp
qed

end

```

## 21 Relation: Relations

```

theory Relation
imports Datatype Finite-Set
begin

```

### 21.1 Definitions

```

definition
  converse :: ('a * 'b set  $\Rightarrow$  'b * 'a set
    ((- ^ -1) [1000] 999) where
    r ^ -1 ==  $\{(y, x). (x, y) : r\}$ 

```

```

notation (xsymbols)
  converse ((--1) [1000] 999)

```

```

definition
  rel-comp :: ['a * 'b set, 'b * 'c set]  $\Rightarrow$  'a * 'c set
    (infixr 0 75) where
    r O s ==  $\{(x, z). \exists y. (x, y) : r \ \& \ (y, z) : s\}$ 

```

```

definition
  Image :: ['a * 'b set, 'a set]  $\Rightarrow$  'b set

```

(infixl “*90*”) **where**  
 $r \text{ “} s == \{y. EX x:s. (x,y):r\}$

**definition**

$Id :: ('a * 'a) \text{ set} \text{ where}$  — the identity relation  
 $Id == \{p. EX x. p = (x,x)\}$

**definition**

$Id-on :: 'a \text{ set} ==> ('a * 'a) \text{ set} \text{ where}$  — diagonal: identity over a set  
 $Id-on A == \bigcup_{x \in A}. \{(x,x)\}$

**definition**

$Domain :: ('a * 'b) \text{ set} ==> 'a \text{ set} \text{ where}$   
 $Domain r == \{x. EX y. (x,y):r\}$

**definition**

$Range :: ('a * 'b) \text{ set} ==> 'b \text{ set} \text{ where}$   
 $Range r == Domain(r^{-1})$

**definition**

$Field :: ('a * 'a) \text{ set} ==> 'a \text{ set} \text{ where}$   
 $Field r == Domain r \cup Range r$

**definition**

$refl-on :: ['a \text{ set}, ('a * 'a) \text{ set}] ==> bool \text{ where}$  — reflexivity over a set  
 $refl-on A r == r \subseteq A \times A \ \& \ (ALL x: A. (x,x) : r)$

**abbreviation**

$refl :: ('a * 'a) \text{ set} ==> bool \text{ where}$  — reflexivity over a type  
 $refl == refl-on UNIV$

**definition**

$sym :: ('a * 'a) \text{ set} ==> bool \text{ where}$  — symmetry predicate  
 $sym r == ALL x y. (x,y):r \longrightarrow (y,x):r$

**definition**

$antisym :: ('a * 'a) \text{ set} ==> bool \text{ where}$  — antisymmetry predicate  
 $antisym r == ALL x y. (x,y):r \longrightarrow (y,x):r \longrightarrow x=y$

**definition**

$trans :: ('a * 'a) \text{ set} ==> bool \text{ where}$  — transitivity predicate  
 $trans r == (ALL x y z. (x,y):r \longrightarrow (y,z):r \longrightarrow (x,z):r)$

**definition**

$irrefl :: ('a * 'a) \text{ set} ==> bool \text{ where}$   
 $irrefl r \equiv \forall x. (x,x) \notin r$

**definition**

$total-on :: 'a \text{ set} ==> ('a * 'a) \text{ set} ==> bool \text{ where}$

*total-on*  $A$   $r \equiv \forall x \in A. \forall y \in A. x \neq y \longrightarrow (x, y) \in r \vee (y, x) \in r$

**abbreviation** *total*  $\equiv$  *total-on UNIV*

**definition**

*single-valued*  $:: ('a * 'b) \text{ set} \Rightarrow \text{bool}$  **where**  
*single-valued*  $r == \text{ALL } x \ y. (x, y) : r \longrightarrow (\text{ALL } z. (x, z) : r \longrightarrow y = z)$

**definition**

*inv-image*  $:: ('b * 'b) \text{ set} \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a * 'a) \text{ set}$  **where**  
*inv-image*  $r \ f == \{(x, y). (f \ x, f \ y) : r\}$

## 21.2 The identity relation

**lemma** *IdI* [*intro*]:  $(a, a) : \text{Id}$

**by** (*simp add: Id-def*)

**lemma** *IdE* [*elim!*]:  $p : \text{Id} \Longrightarrow (!x. p = (x, x) \Longrightarrow P) \Longrightarrow P$

**by** (*unfold Id-def*) (*iprover elim: CollectE*)

**lemma** *pair-in-Id-conv* [*iff*]:  $((a, b) : \text{Id}) = (a = b)$

**by** (*unfold Id-def*) *blast*

**lemma** *refl-Id*: *refl Id*

**by** (*simp add: refl-on-def*)

**lemma** *antisym-Id*: *antisym Id*

— A strange result, since *Id* is also symmetric.

**by** (*simp add: antisym-def*)

**lemma** *sym-Id*: *sym Id*

**by** (*simp add: sym-def*)

**lemma** *trans-Id*: *trans Id*

**by** (*simp add: trans-def*)

## 21.3 Diagonal: identity over a set

**lemma** *Id-on-empty* [*simp*]:  $\text{Id-on } \{\} = \{\}$

**by** (*simp add: Id-on-def*)

**lemma** *Id-on-eqI*:  $a = b \Longrightarrow a : A \Longrightarrow (a, b) : \text{Id-on } A$

**by** (*simp add: Id-on-def*)

**lemma** *Id-onI* [*intro!, no-atp*]:  $a : A \Longrightarrow (a, a) : \text{Id-on } A$

**by** (*rule Id-on-eqI*) (*rule refl*)

**lemma** *Id-onE* [*elim!*]:

$c : \text{Id-on } A \Longrightarrow (!x. x : A \Longrightarrow c = (x, x) \Longrightarrow P) \Longrightarrow P$

— The general elimination rule.

**by** (*unfold Id-on-def*) (*iprover elim!*: *UN-E singletonE*)

**lemma** *Id-on-iff*:  $((x, y) : \text{Id-on } A) = (x = y \ \& \ x : A)$   
**by** *blast*

**lemma** *Id-on-subset-Times*:  $\text{Id-on } A \subseteq A \times A$   
**by** *blast*

## 21.4 Composition of two relations

**lemma** *rel-compI* [*intro*]:  
 $(a, b) : r \implies (b, c) : s \implies (a, c) : r \ O \ s$   
**by** (*unfold rel-comp-def*) *blast*

**lemma** *rel-compE* [*elim!*]:  $xz : r \ O \ s \implies$   
 $(!!x \ y \ z. \ xz = (x, z) \implies (x, y) : r \implies (y, z) : s \implies P) \implies P$   
**by** (*unfold rel-comp-def*) (*iprover elim!*: *CollectE splitE exE conjE*)

**lemma** *rel-compEpair*:  
 $(a, c) : r \ O \ s \implies (!y. (a, y) : r \implies (y, c) : s \implies P) \implies P$   
**by** (*iprover elim*: *rel-compE Pair-inject ssubst*)

**lemma** *R-O-Id* [*simp*]:  $R \ O \ \text{Id} = R$   
**by** *fast*

**lemma** *Id-O-R* [*simp*]:  $\text{Id} \ O \ R = R$   
**by** *fast*

**lemma** *rel-comp-empty1* [*simp*]:  $\{\} \ O \ R = \{\}$   
**by** *blast*

**lemma** *rel-comp-empty2* [*simp*]:  $R \ O \ \{\} = \{\}$   
**by** *blast*

**lemma** *O-assoc*:  $(R \ O \ S) \ O \ T = R \ O \ (S \ O \ T)$   
**by** *blast*

**lemma** *trans-O-subset*:  $\text{trans } r \implies r \ O \ r \subseteq r$   
**by** (*unfold trans-def*) *blast*

**lemma** *rel-comp-mono*:  $r' \subseteq r \implies s' \subseteq s \implies (r' \ O \ s') \subseteq (r \ O \ s)$   
**by** *blast*

**lemma** *rel-comp-subset-Sigma*:  
 $r \subseteq A \times B \implies s \subseteq B \times C \implies (r \ O \ s) \subseteq A \times C$   
**by** *blast*

**lemma** *rel-comp-distrib* [*simp*]:  $R \ O \ (S \cup T) = (R \ O \ S) \cup (R \ O \ T)$   
**by** *auto*

**lemma** *rel-comp-distrib2[simp]*:  $(S \cup T) \ O \ R = (S \ O \ R) \cup (T \ O \ R)$   
**by** *auto*

**lemma** *rel-comp-UNION-distrib*:  $s \ O \ \text{UNION } I \ r = \text{UNION } I \ (\%i. s \ O \ r \ i)$   
**by** *auto*

**lemma** *rel-comp-UNION-distrib2*:  $\text{UNION } I \ r \ O \ s = \text{UNION } I \ (\%i. r \ i \ O \ s)$   
**by** *auto*

## 21.5 Reflexivity

**lemma** *refl-onI*:  $r \subseteq A \times A \implies (!x. x : A \implies (x, x) : r) \implies \text{refl-on } A \ r$   
**by** (*unfold refl-on-def*) (*iprover intro!: ballI*)

**lemma** *refl-onD*:  $\text{refl-on } A \ r \implies a : A \implies (a, a) : r$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-onD1*:  $\text{refl-on } A \ r \implies (x, y) : r \implies x : A$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-onD2*:  $\text{refl-on } A \ r \implies (x, y) : r \implies y : A$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-Int*:  $\text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cap B) \ (r \cap s)$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-Un*:  $\text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cup B) \ (r \cup s)$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-INTER*:  
 $\text{ALL } x:S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\text{INTER } S \ A) \ (\text{INTER } S \ r)$   
**by** (*unfold refl-on-def*) *fast*

**lemma** *refl-on-UNION*:  
 $\text{ALL } x:S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\text{UNION } S \ A) \ (\text{UNION } S \ r)$   
**by** (*unfold refl-on-def*) *blast*

**lemma** *refl-on-empty[simp]*:  $\text{refl-on } \{\} \ \{\}$   
**by** (*simp add:refl-on-def*)

**lemma** *refl-on-Id-on*:  $\text{refl-on } A \ (\text{Id-on } A)$   
**by** (*rule refl-onI [OF Id-on-subset-Times Id-onI]*)

## 21.6 Antisymmetry

**lemma** *antisymI*:  
 $(!x \ y. (x, y) : r \implies (y, x) : r \implies x=y) \implies \text{antisym } r$   
**by** (*unfold antisym-def*) *iprover*

**lemma** *antisymD*:  $\text{antisym } r \implies (a, b) : r \implies (b, a) : r \implies a = b$   
**by** (*unfold antisym-def*) *iprover*

**lemma** *antisym-subset*:  $r \subseteq s \implies \text{antisym } s \implies \text{antisym } r$   
**by** (*unfold antisym-def*) *blast*

**lemma** *antisym-empty* [*simp*]:  $\text{antisym } \{\}$   
**by** (*unfold antisym-def*) *blast*

**lemma** *antisym-Id-on* [*simp*]:  $\text{antisym } (\text{Id-on } A)$   
**by** (*unfold antisym-def*) *blast*

## 21.7 Symmetry

**lemma** *symI*:  $(\forall a\ b. (a, b) : r \implies (b, a) : r) \implies \text{sym } r$   
**by** (*unfold sym-def*) *iprover*

**lemma** *symD*:  $\text{sym } r \implies (a, b) : r \implies (b, a) : r$   
**by** (*unfold sym-def*, *blast*)

**lemma** *sym-Int*:  $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cap s)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-Un*:  $\text{sym } r \implies \text{sym } s \implies \text{sym } (r \cup s)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-INTER*:  $\text{ALL } x:S. \text{sym } (r\ x) \implies \text{sym } (\text{INTER } S\ r)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-UNION*:  $\text{ALL } x:S. \text{sym } (r\ x) \implies \text{sym } (\text{UNION } S\ r)$   
**by** (*fast intro: symI dest: symD*)

**lemma** *sym-Id-on* [*simp*]:  $\text{sym } (\text{Id-on } A)$   
**by** (*rule symI*) *clarify*

## 21.8 Transitivity

**lemma** *transI*:  
 $(\forall x\ y\ z. (x, y) : r \implies (y, z) : r \implies (x, z) : r) \implies \text{trans } r$   
**by** (*unfold trans-def*) *iprover*

**lemma** *transD*:  $\text{trans } r \implies (a, b) : r \implies (b, c) : r \implies (a, c) : r$   
**by** (*unfold trans-def*) *iprover*

**lemma** *trans-Int*:  $\text{trans } r \implies \text{trans } s \implies \text{trans } (r \cap s)$   
**by** (*fast intro: transI elim: transD*)

**lemma** *trans-INTER*:  $\text{ALL } x:S. \text{trans } (r\ x) \implies \text{trans } (\text{INTER } S\ r)$   
**by** (*fast intro: transI elim: transD*)



**lemma** *trans-Id-on* [*simp*]: *trans* (*Id-on* *A*)  
**by** (*fast intro: transI elim: transD*)

**lemma** *trans-diff-Id*: *trans* *r*  $\implies$  *antisym* *r*  $\implies$  *trans* (*r-Id*)  
**unfolding** *antisym-def trans-def* **by** *blast*

## 21.9 Irreflexivity

**lemma** *irrefl-diff-Id* [*simp*]: *irrefl*(*r-Id*)  
**by**(*simp add:irrefl-def*)

## 21.10 Totality

**lemma** *total-on-empty* [*simp*]: *total-on* {} *r*  
**by**(*simp add:total-on-def*)

**lemma** *total-on-diff-Id* [*simp*]: *total-on* *A* (*r-Id*) = *total-on* *A* *r*  
**by**(*simp add: total-on-def*)

## 21.11 Converse

**lemma** *converse-iff* [*iff*]:  $((a,b): r^{-1}) = ((b,a) : r)$   
**by** (*simp add: converse-def*)

**lemma** *converseI* [*sym*]:  $(a, b) : r \implies (b, a) : r^{-1}$   
**by** (*simp add: converse-def*)

**lemma** *converseD* [*sym*]:  $(a,b) : r^{-1} \implies (b, a) : r$   
**by** (*simp add: converse-def*)

**lemma** *converseE* [*elim!*]:  
 $yx : r^{-1} \implies (!x y. yx = (y, x) \implies (x, y) : r \implies P) \implies P$   
— More general than *converseD*, as it “splits” the member of the relation.  
**by** (*unfold converse-def*) (*iprover elim!: CollectE splitE bexE*)

**lemma** *converse-converse* [*simp*]:  $(r^{-1})^{-1} = r$   
**by** (*unfold converse-def*) *blast*

**lemma** *converse-rel-comp*:  $(r \circ s)^{-1} = s^{-1} \circ r^{-1}$   
**by** *blast*

**lemma** *converse-Int*:  $(r \cap s)^{-1} = r^{-1} \cap s^{-1}$   
**by** *blast*

**lemma** *converse-Un*:  $(r \cup s)^{-1} = r^{-1} \cup s^{-1}$   
**by** *blast*

**lemma** *converse-INTER*:  $(\text{INTER } S \ r)^{-1} = (\text{INT } x:S. (r \ x)^{-1})$   
**by** *fast*

**lemma** *converse-UNION*:  $(\text{UNION } S \ r)^{\wedge-1} = (\text{UN } x:S. (r \ x)^{\wedge-1})$   
**by** *blast*

**lemma** *converse-Id* [*simp*]:  $\text{Id}^{\wedge-1} = \text{Id}$   
**by** *blast*

**lemma** *converse-Id-on* [*simp*]:  $(\text{Id-on } A)^{\wedge-1} = \text{Id-on } A$   
**by** *blast*

**lemma** *refl-on-converse* [*simp*]:  $\text{refl-on } A \ (\text{converse } r) = \text{refl-on } A \ r$   
**by** (*unfold refl-on-def*) *auto*

**lemma** *sym-converse* [*simp*]:  $\text{sym} \ (\text{converse } r) = \text{sym } r$   
**by** (*unfold sym-def*) *blast*

**lemma** *antisym-converse* [*simp*]:  $\text{antisym} \ (\text{converse } r) = \text{antisym } r$   
**by** (*unfold antisym-def*) *blast*

**lemma** *trans-converse* [*simp*]:  $\text{trans} \ (\text{converse } r) = \text{trans } r$   
**by** (*unfold trans-def*) *blast*

**lemma** *sym-conv-converse-eq*:  $\text{sym } r = (r^{\wedge-1} = r)$   
**by** (*unfold sym-def*) *fast*

**lemma** *sym-Un-converse*:  $\text{sym} \ (r \cup r^{\wedge-1})$   
**by** (*unfold sym-def*) *blast*

**lemma** *sym-Int-converse*:  $\text{sym} \ (r \cap r^{\wedge-1})$   
**by** (*unfold sym-def*) *blast*

**lemma** *total-on-converse*[*simp*]:  $\text{total-on } A \ (r^{\wedge-1}) = \text{total-on } A \ r$   
**by** (*auto simp: total-on-def*)

## 21.12 Domain

**declare** *Domain-def* [*no-atp*]

**lemma** *Domain-iff*:  $(a : \text{Domain } r) = (\text{EX } y. (a, y) : r)$   
**by** (*unfold Domain-def*) *blast*

**lemma** *DomainI* [*intro*]:  $(a, b) : r ==> a : \text{Domain } r$   
**by** (*iprover intro!: iffD2 [OF Domain-iff]*)

**lemma** *DomainE* [*elim!*]:  
 $a : \text{Domain } r ==> (!y. (a, y) : r ==> P) ==> P$   
**by** (*iprover dest!: iffD1 [OF Domain-iff]*)

**lemma** *Domain-empty* [*simp*]:  $\text{Domain } \{\} = \{\}$   
**by** *blast*

**lemma** *Domain-empty-iff*:  $\text{Domain } r = \{\} \longleftrightarrow r = \{\}$   
**by** *auto*

**lemma** *Domain-insert*:  $\text{Domain } (\text{insert } (a, b) \ r) = \text{insert } a \ (\text{Domain } r)$   
**by** *blast*

**lemma** *Domain-Id* [simp]:  $\text{Domain } \text{Id} = \text{UNIV}$   
**by** *blast*

**lemma** *Domain-Id-on* [simp]:  $\text{Domain } (\text{Id-on } A) = A$   
**by** *blast*

**lemma** *Domain-Un-eq*:  $\text{Domain}(A \cup B) = \text{Domain}(A) \cup \text{Domain}(B)$   
**by** *blast*

**lemma** *Domain-Int-subset*:  $\text{Domain}(A \cap B) \subseteq \text{Domain}(A) \cap \text{Domain}(B)$   
**by** *blast*

**lemma** *Domain-Diff-subset*:  $\text{Domain}(A) - \text{Domain}(B) \subseteq \text{Domain}(A - B)$   
**by** *blast*

**lemma** *Domain-Union*:  $\text{Domain } (\text{Union } S) = (\bigcup A \in S. \text{Domain } A)$   
**by** *blast*

**lemma** *Domain-converse*[simp]:  $\text{Domain}(r^{-1}) = \text{Range } r$   
**by** (*auto simp: Range-def*)

**lemma** *Domain-mono*:  $r \subseteq s \implies \text{Domain } r \subseteq \text{Domain } s$   
**by** *blast*

**lemma** *fst-eq-Domain*:  $\text{fst } R = \text{Domain } R$   
**by** (*auto intro!: image-eqI*)

**lemma** *Domain-dprod* [simp]:  $\text{Domain } (\text{dprod } r \ s) = \text{uprod } (\text{Domain } r) \ (\text{Domain } s)$   
**by** *auto*

**lemma** *Domain-dsum* [simp]:  $\text{Domain } (\text{dsum } r \ s) = \text{usum } (\text{Domain } r) \ (\text{Domain } s)$   
**by** *auto*

### 21.13 Range

**lemma** *Range-iff*:  $(a : \text{Range } r) = (\exists y. (y, a) : r)$   
**by** (*simp add: Domain-def Range-def*)

**lemma** *RangeI* [intro]:  $(a, b) : r \implies b : \text{Range } r$   
**by** (*unfold Range-def (iprover intro!: converseI DomainI)*)

**lemma** *RangeE* [elim!]:  $b : \text{Range } r \implies (!x. (x, b) : r \implies P) \implies P$   
**by** (*unfold Range-def*) (*iprover elim! : DomainE dest! : converseD*)

**lemma** *Range-empty* [simp]:  $\text{Range } \{\} = \{\}$   
**by** *blast*

**lemma** *Range-empty-iff*:  $\text{Range } r = \{\} \longleftrightarrow r = \{\}$   
**by** *auto*

**lemma** *Range-insert*:  $\text{Range } (\text{insert } (a, b) r) = \text{insert } b (\text{Range } r)$   
**by** *blast*

**lemma** *Range-Id* [simp]:  $\text{Range } \text{Id} = \text{UNIV}$   
**by** *blast*

**lemma** *Range-Id-on* [simp]:  $\text{Range } (\text{Id-on } A) = A$   
**by** *auto*

**lemma** *Range-Un-eq*:  $\text{Range}(A \cup B) = \text{Range}(A) \cup \text{Range}(B)$   
**by** *blast*

**lemma** *Range-Int-subset*:  $\text{Range}(A \cap B) \subseteq \text{Range}(A) \cap \text{Range}(B)$   
**by** *blast*

**lemma** *Range-Diff-subset*:  $\text{Range}(A) - \text{Range}(B) \subseteq \text{Range}(A - B)$   
**by** *blast*

**lemma** *Range-Union*:  $\text{Range } (\text{Union } S) = (\bigcup A \in S. \text{Range } A)$   
**by** *blast*

**lemma** *Range-converse*[simp]:  $\text{Range}(r^{-1}) = \text{Domain } r$   
**by** *blast*

**lemma** *snd-eq-Range*:  $\text{snd} \circ R = \text{Range } R$   
**by** (*auto intro! : image-eqI*)

## 21.14 Field

**lemma** *mono-Field*:  $r \subseteq s \implies \text{Field } r \subseteq \text{Field } s$   
**by** (*auto simp : Field-def Domain-def Range-def*)

**lemma** *Field-empty*[simp]:  $\text{Field } \{\} = \{\}$   
**by** (*auto simp : Field-def*)

**lemma** *Field-insert*[simp]:  $\text{Field } (\text{insert } (a, b) r) = \{a, b\} \cup \text{Field } r$   
**by** (*auto simp : Field-def*)

**lemma** *Field-Un*[simp]:  $\text{Field } (r \cup s) = \text{Field } r \cup \text{Field } s$

**by** (*auto simp:Field-def*)

**lemma** *Field-Union* [*simp*]:  $\text{Field } (\bigcup R) = \bigcup (\text{Field } \cdot R)$   
**by** (*auto simp:Field-def*)

**lemma** *Field-converse* [*simp*]:  $\text{Field}(r^{-1}) = \text{Field } r$   
**by** (*auto simp:Field-def*)

### 21.15 Image of a set under a relation

**declare** *Image-def* [*no-atp*]

**lemma** *Image-iff*:  $(b : r \cdot A) = (EX x:A. (x, b) : r)$   
**by** (*simp add: Image-def*)

**lemma** *Image-singleton*:  $r \cdot \{a\} = \{b. (a, b) : r\}$   
**by** (*simp add: Image-def*)

**lemma** *Image-singleton-iff* [*iff*]:  $(b : r \cdot \{a\}) = ((a, b) : r)$   
**by** (*rule Image-iff [THEN trans]*) *simp*

**lemma** *ImageI* [*intro,no-atp*]:  $(a, b) : r ==> a : A ==> b : r \cdot A$   
**by** (*unfold Image-def*) *blast*

**lemma** *ImageE* [*elim!*]:  
 $b : r \cdot A ==> (!x. (x, b) : r ==> x : A ==> P) ==> P$   
**by** (*unfold Image-def*) (*iprover elim!: CollectE bexE*)

**lemma** *rev-ImageI*:  $a : A ==> (a, b) : r ==> b : r \cdot A$   
 — This version’s more effective when we already have the required  $a$   
**by** *blast*

**lemma** *Image-empty* [*simp*]:  $R \cdot \{\} = \{\}$   
**by** *blast*

**lemma** *Image-Id* [*simp*]:  $\text{Id } \cdot A = A$   
**by** *blast*

**lemma** *Image-Id-on* [*simp*]:  $\text{Id-on } A \cdot B = A \cap B$   
**by** *blast*

**lemma** *Image-Int-subset*:  $R \cdot (A \cap B) \subseteq R \cdot A \cap R \cdot B$   
**by** *blast*

**lemma** *Image-Int-eq*:  
 $\text{single-valued } (\text{converse } R) ==> R \cdot (A \cap B) = R \cdot A \cap R \cdot B$   
**by** (*simp add: single-valued-def, blast*)

**lemma** *Image-Un*:  $R \cdot (A \cup B) = R \cdot A \cup R \cdot B$

by *blast*

**lemma** *Un-Image*:  $(R \cup S) \text{ “ } A = R \text{ “ } A \cup S \text{ “ } A$   
by *blast*

**lemma** *Image-subset*:  $r \subseteq A \times B \implies r \text{ “ } C \subseteq B$   
by (*iprover intro!*: *subsetI elim!*: *ImageE dest!*: *subsetD SigmaD2*)

**lemma** *Image-eq-UN*:  $r \text{ “ } B = (\bigcup y \in B. r \text{ “ } \{y\})$   
— NOT suitable for rewriting  
by *blast*

**lemma** *Image-mono*:  $r' \subseteq r \implies A' \subseteq A \implies (r' \text{ “ } A') \subseteq (r \text{ “ } A)$   
by *blast*

**lemma** *Image-UN*:  $(r \text{ “ } (\text{UNION } A \ B)) = (\bigcup x \in A. r \text{ “ } (B \ x))$   
by *blast*

**lemma** *Image-INT-subset*:  $(r \text{ “ } \text{INTER } A \ B) \subseteq (\bigcap x \in A. r \text{ “ } (B \ x))$   
by *blast*

Converse inclusion requires some assumptions

**lemma** *Image-INT-eq*:  
   $[[\text{single-valued } (r^{-1}); A \neq \{\}]] \implies r \text{ “ } \text{INTER } A \ B = (\bigcap x \in A. r \text{ “ } B \ x)$   
**apply** (*rule equalityI*)  
  **apply** (*rule Image-INT-subset*)  
**apply** (*simp add: single-valued-def, blast*)  
**done**

**lemma** *Image-subset-eq*:  $(r \text{ “ } A \subseteq B) = (A \subseteq - ((r^{-1}) \text{ “ } (-B)))$   
by *blast*

## 21.16 Single valued relations

**lemma** *single-valuedI*:  
   $\text{ALL } x \ y. (x, y) : r \dashrightarrow (\text{ALL } z. (x, z) : r \dashrightarrow y = z) \implies \text{single-valued } r$   
by (*unfold single-valued-def*)

**lemma** *single-valuedD*:  
   $\text{single-valued } r \implies (x, y) : r \implies (x, z) : r \implies y = z$   
by (*simp add: single-valued-def*)

**lemma** *single-valued-rel-comp*:  
   $\text{single-valued } r \implies \text{single-valued } s \implies \text{single-valued } (r \ O \ s)$   
by (*unfold single-valued-def*) *blast*

**lemma** *single-valued-subset*:  
   $r \subseteq s \implies \text{single-valued } s \implies \text{single-valued } r$   
by (*unfold single-valued-def*) *blast*

**lemma** *single-valued-Id* [simp]: *single-valued Id*  
**by** (*unfold single-valued-def*) *blast*

**lemma** *single-valued-Id-on* [simp]: *single-valued (Id-on A)*  
**by** (*unfold single-valued-def*) *blast*

### 21.17 Graphs given by *Collect*

**lemma** *Domain-Collect-split* [simp]:  $\text{Domain}\{(x,y). P\ x\ y\} = \{x. \text{EX } y. P\ x\ y\}$   
**by** *auto*

**lemma** *Range-Collect-split* [simp]:  $\text{Range}\{(x,y). P\ x\ y\} = \{y. \text{EX } x. P\ x\ y\}$   
**by** *auto*

**lemma** *Image-Collect-split* [simp]:  $\{(x,y). P\ x\ y\} \text{ “ } A = \{y. \text{EX } x:A. P\ x\ y\}$   
**by** *auto*

### 21.18 Inverse image

**lemma** *sym-inv-image*:  $\text{sym } r \implies \text{sym } (\text{inv-image } r\ f)$   
**by** (*unfold sym-def inv-image-def*) *blast*

**lemma** *trans-inv-image*:  $\text{trans } r \implies \text{trans } (\text{inv-image } r\ f)$   
**apply** (*unfold trans-def inv-image-def*)  
**apply** (*simp (no-asm)*)  
**apply** *blast*  
**done**

**lemma** *in-inv-image*[simp]:  $((x,y) : \text{inv-image } r\ f) = ((f\ x, f\ y) : r)$   
**by** (*auto simp:inv-image-def*)

**lemma** *converse-inv-image*[simp]:  $(\text{inv-image } R\ f)^{-1} = \text{inv-image } (R^{-1})\ f$   
**unfolding** *inv-image-def converse-def* **by** *auto*

### 21.19 Finiteness

**lemma** *finite-converse* [iff]:  $\text{finite } (r^{-1}) = \text{finite } r$   
**apply** (*subgoal-tac*  $r^{-1} = (\%(x,y). (y,x))\ r$ )  
**apply** *simp*  
**apply** (*rule iffI*)  
**apply** (*erule finite-imageD [unfolded inj-on-def]*)  
**apply** (*simp split add: split-split*)  
**apply** (*erule finite-imageI*)  
**apply** (*simp add: converse-def image-def, auto*)  
**apply** (*rule bexI*)  
**prefer** 2 **apply** *assumption*  
**apply** *simp*  
**done**

```

lemma finite-Domain: finite  $r \implies$  finite (Domain  $r$ )
  by (induct set: finite) (auto simp add: Domain-insert)

lemma finite-Range: finite  $r \implies$  finite (Range  $r$ )
  by (induct set: finite) (auto simp add: Range-insert)

lemma finite-Field: finite  $r \implies$  finite (Field  $r$ )
  — A finite relation has a finite field (= domain  $\cup$  range.
  apply (induct set: finite)
  apply (auto simp add: Field-def Domain-insert Range-insert)
  done

```

## 21.20 Miscellaneous

Version of *lfp-induct* for binary relations

```

lemmas lfp-induct2 =
  lfp-induct-set [of ( $a, b$ ), split-format (complete)]

```

Version of *subsetI* for binary relations

```

lemma subrelI:  $(\bigwedge x y. (x, y) \in r \implies (x, y) \in s) \implies r \subseteq s$ 
by auto

```

**end**

## 22 Predicate: Predicates as relations and enumerations

```

theory Predicate
imports Inductive Relation
begin

```

```

notation
  inf (infixl  $\sqcap$  70) and
  sup (infixl  $\sqcup$  65) and
  Inf ( $\bigsqcap$  - [900] 900) and
  Sup ( $\bigsqcup$  - [900] 900) and
  top ( $\top$ ) and
  bot ( $\perp$ )

```

### 22.1 Predicates as (complete) lattices

Handy introduction and elimination rules for  $\leq$  on unary and binary predicates

```

lemma predicate1I:
  assumes  $PQ: \bigwedge x. P x \implies Q x$ 
  shows  $P \leq Q$ 

```



```

apply (rule le-funI)
apply (rule le-boolI)
apply (rule PQ)
apply assumption
done

```

```

lemma predicate1D [Pure.dest?, dest?]:
   $P \leq Q \implies P\ x \implies Q\ x$ 
  apply (erule le-funE)
  apply (erule le-boolE)
  apply assumption+
  done

```

```

lemma rev-predicate1D:
   $P\ x \implies P \leq Q \implies Q\ x$ 
  by (rule predicate1D)

```

```

lemma predicate2I [Pure.intro!, intro!]:
  assumes PQ:  $\bigwedge x\ y. P\ x\ y \implies Q\ x\ y$ 
  shows  $P \leq Q$ 
  apply (rule le-funI)+
  apply (rule le-boolI)
  apply (rule PQ)
  apply assumption
  done

```

```

lemma predicate2D [Pure.dest, dest]:
   $P \leq Q \implies P\ x\ y \implies Q\ x\ y$ 
  apply (erule le-funE)+
  apply (erule le-boolE)
  apply assumption+
  done

```

```

lemma rev-predicate2D:
   $P\ x\ y \implies P \leq Q \implies Q\ x\ y$ 
  by (rule predicate2D)

```

### 22.1.1 Equality

```

lemma pred-equals-eq:  $((\lambda x. x \in R) = (\lambda x. x \in S)) = (R = S)$ 
  by (simp add: mem-def)

```

```

lemma pred-equals-eq2 [pred-set-conv]:  $((\lambda x\ y. (x, y) \in R) = (\lambda x\ y. (x, y) \in S))$ 
   $= (R = S)$ 
  by (simp add: expand-fun-eq mem-def)

```

### 22.1.2 Order relation

```

lemma pred-subset-eq:  $((\lambda x. x \in R) \leq (\lambda x. x \in S)) = (R \leq S)$ 
  by (simp add: mem-def)

```

**lemma** *pred-subset-eq2* [*pred-set-conv*]:  $((\lambda x y. (x, y) \in R) \leq (\lambda x y. (x, y) \in S)) = (R \leq S))$   
**by** *fast*

### 22.1.3 Top and bottom elements

**lemma** *top1I* [*intro!*]:  $\text{top } x$   
**by** (*simp add: top-fun-eq top-bool-eq*)

**lemma** *top2I* [*intro!*]:  $\text{top } x y$   
**by** (*simp add: top-fun-eq top-bool-eq*)

**lemma** *bot1E* [*elim!*]:  $\text{bot } x \implies P$   
**by** (*simp add: bot-fun-eq bot-bool-eq*)

**lemma** *bot2E* [*elim!*]:  $\text{bot } x y \implies P$   
**by** (*simp add: bot-fun-eq bot-bool-eq*)

**lemma** *bot-empty-eq*:  $\text{bot} = (\lambda x. x \in \{\})$   
**by** (*auto simp add: expand-fun-eq*)

**lemma** *bot-empty-eq2*:  $\text{bot} = (\lambda x y. (x, y) \in \{\})$   
**by** (*auto simp add: expand-fun-eq*)

### 22.1.4 Binary union

**lemma** *sup1I1* [*elim?*]:  $A x \implies \text{sup } A B x$   
**by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup2I1* [*elim?*]:  $A x y \implies \text{sup } A B x y$   
**by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup1I2* [*elim?*]:  $B x \implies \text{sup } A B x$   
**by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup2I2* [*elim?*]:  $B x y \implies \text{sup } A B x y$   
**by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup1E* [*elim!*]:  $\text{sup } A B x \implies (A x \implies P) \implies (B x \implies P) \implies P$   
**by** (*simp add: sup-fun-eq sup-bool-eq iprover*)

**lemma** *sup2E* [*elim!*]:  $\text{sup } A B x y \implies (A x y \implies P) \implies (B x y \implies P) \implies P$   
**by** (*simp add: sup-fun-eq sup-bool-eq iprover*)

Classical introduction rule: no commitment to  $A$  vs  $B$ .

**lemma** *sup1CI* [*intro!*]:  $(\sim B x \implies A x) \implies \text{sup } A B x$

**by** (*auto simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup2CI* [*intro!*]:  $(\sim B\ x\ y \implies A\ x\ y) \implies \sup A\ B\ x\ y$   
**by** (*auto simp add: sup-fun-eq sup-bool-eq*)

**lemma** *sup-Un-eq*:  $\sup (\lambda x. x \in R) (\lambda x. x \in S) = (\lambda x. x \in R \cup S)$   
**by** (*simp add: sup-fun-eq sup-bool-eq mem-def*)

**lemma** *sup-Un-eq2* [*pred-set-conv*]:  $\sup (\lambda x\ y. (x, y) \in R) (\lambda x\ y. (x, y) \in S) =$   
 $(\lambda x\ y. (x, y) \in R \cup S)$   
**by** (*simp add: sup-fun-eq sup-bool-eq mem-def*)

### 22.1.5 Binary intersection

**lemma** *inf1I* [*intro!*]:  $A\ x \implies B\ x \implies \inf A\ B\ x$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf2I* [*intro!*]:  $A\ x\ y \implies B\ x\ y \implies \inf A\ B\ x\ y$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf1E* [*elim!*]:  $\inf A\ B\ x \implies (A\ x \implies B\ x \implies P) \implies P$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf2E* [*elim!*]:  $\inf A\ B\ x\ y \implies (A\ x\ y \implies B\ x\ y \implies P) \implies P$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf1D1*:  $\inf A\ B\ x \implies A\ x$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf2D1*:  $\inf A\ B\ x\ y \implies A\ x\ y$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf1D2*:  $\inf A\ B\ x \implies B\ x$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf2D2*:  $\inf A\ B\ x\ y \implies B\ x\ y$   
**by** (*simp add: inf-fun-eq inf-bool-eq*)

**lemma** *inf-Int-eq*:  $\inf (\lambda x. x \in R) (\lambda x. x \in S) = (\lambda x. x \in R \cap S)$   
**by** (*simp add: inf-fun-eq inf-bool-eq mem-def*)

**lemma** *inf-Int-eq2* [*pred-set-conv*]:  $\inf (\lambda x\ y. (x, y) \in R) (\lambda x\ y. (x, y) \in S) =$   
 $(\lambda x\ y. (x, y) \in R \cap S)$   
**by** (*simp add: inf-fun-eq inf-bool-eq mem-def*)

### 22.1.6 Unions of families

**lemma** *SUP1-iff*:  $(\sup x:A. B\ x)\ b = (\exists x:A. B\ x\ b)$   
**by** (*simp add: SUPR-def Sup-fun-def Sup-bool-def*) *blast*

**lemma** *SUP2-iff*:  $(\text{SUP } x:A. B \ x) \ b \ c = (\text{EX } x:A. B \ x \ b \ c)$   
**by** (*simp add: SUPR-def Sup-fun-def Sup-bool-def*) *blast*

**lemma** *SUP1-I* [*intro*]:  $a : A \implies B \ a \ b \implies (\text{SUP } x:A. B \ x) \ b$   
**by** (*auto simp add: SUP1-iff*)

**lemma** *SUP2-I* [*intro*]:  $a : A \implies B \ a \ b \ c \implies (\text{SUP } x:A. B \ x) \ b \ c$   
**by** (*auto simp add: SUP2-iff*)

**lemma** *SUP1-E* [*elim*!]:  $(\text{SUP } x:A. B \ x) \ b \implies (!x. x : A \implies B \ x \ b \implies R) \implies R$   
**by** (*auto simp add: SUP1-iff*)

**lemma** *SUP2-E* [*elim*!]:  $(\text{SUP } x:A. B \ x) \ b \ c \implies (!x. x : A \implies B \ x \ b \ c \implies R) \implies R$   
**by** (*auto simp add: SUP2-iff*)

**lemma** *SUP-UN-eq*:  $(\text{SUP } i. (\lambda x. x \in r \ i)) = (\lambda x. x \in (\text{UN } i. r \ i))$   
**by** (*simp add: SUP1-iff expand-fun-eq*)

**lemma** *SUP-UN-eq2*:  $(\text{SUP } i. (\lambda x \ y. (x, y) \in r \ i)) = (\lambda x \ y. (x, y) \in (\text{UN } i. r \ i))$   
**by** (*simp add: SUP2-iff expand-fun-eq*)

### 22.1.7 Intersections of families

**lemma** *INF1-iff*:  $(\text{INF } x:A. B \ x) \ b = (\text{ALL } x:A. B \ x \ b)$   
**by** (*simp add: INF1-def Inf-fun-def Inf-bool-def*) *blast*

**lemma** *INF2-iff*:  $(\text{INF } x:A. B \ x) \ b \ c = (\text{ALL } x:A. B \ x \ b \ c)$   
**by** (*simp add: INF1-def Inf-fun-def Inf-bool-def*) *blast*

**lemma** *INF1-I* [*intro*!]:  $(!x. x : A \implies B \ x \ b) \implies (\text{INF } x:A. B \ x) \ b$   
**by** (*auto simp add: INF1-iff*)

**lemma** *INF2-I* [*intro*!]:  $(!x. x : A \implies B \ x \ b \ c) \implies (\text{INF } x:A. B \ x) \ b \ c$   
**by** (*auto simp add: INF2-iff*)

**lemma** *INF1-D* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \implies a : A \implies B \ a \ b$   
**by** (*auto simp add: INF1-iff*)

**lemma** *INF2-D* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \ c \implies a : A \implies B \ a \ b \ c$   
**by** (*auto simp add: INF2-iff*)

**lemma** *INF1-E* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \implies (B \ a \ b \implies R) \implies (a \sim: A \implies R) \implies R$   
**by** (*auto simp add: INF1-iff*)

**lemma** *INF2-E* [*elim*]:  $(\text{INF } x:A. B \ x) \ b \ c \implies (B \ a \ b \ c \implies R) \implies (a \sim: A \implies R) \implies R$

**by** (*auto simp add: INF2-iff*)

**lemma** *INF-INT-eq*:  $(\text{INF } i. (\lambda x. x \in r \ i)) = (\lambda x. x \in (\text{INT } i. r \ i))$   
**by** (*simp add: INF1-iff expand-fun-eq*)

**lemma** *INF-INT-eq2*:  $(\text{INF } i. (\lambda x y. (x, y) \in r \ i)) = (\lambda x y. (x, y) \in (\text{INT } i. r \ i))$   
**by** (*simp add: INF2-iff expand-fun-eq*)

## 22.2 Predicates as relations

### 22.2.1 Composition

**inductive**

*pred-comp* ::  $['a \Rightarrow 'b \Rightarrow \text{bool}, 'b \Rightarrow 'c \Rightarrow \text{bool}] \Rightarrow 'a \Rightarrow 'c \Rightarrow \text{bool}$   
*(infixr OO 75)*

**for**  $r :: 'a \Rightarrow 'b \Rightarrow \text{bool}$  **and**  $s :: 'b \Rightarrow 'c \Rightarrow \text{bool}$

**where**

*pred-compI* [intro]:  $r \ a \ b \implies s \ b \ c \implies (r \ OO \ s) \ a \ c$

**inductive-cases** *pred-compE* [elim!]:  $(r \ OO \ s) \ a \ c$

**lemma** *pred-comp-rel-comp-eq* [pred-set-conv]:

$((\lambda x y. (x, y) \in r) \ OO \ (\lambda x y. (x, y) \in s)) = (\lambda x y. (x, y) \in r \ O \ s)$

**by** (*auto simp add: expand-fun-eq elim: pred-compE*)

### 22.2.2 Converse

**inductive**

*conversep* ::  $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow 'a \Rightarrow \text{bool}$   
*(( $\hat{\ } - - 1$ ) [1000] 1000)*

**for**  $r :: 'a \Rightarrow 'b \Rightarrow \text{bool}$

**where**

*conversepI*:  $r \ a \ b \implies r^{\hat{\ } - - 1} \ b \ a$

**notation** (*xsymbols*)

*conversep* ( $(\hat{\ }^{-1} - 1)$  [1000] 1000)

**lemma** *conversepD*:

**assumes**  $ab: r^{\hat{\ } - - 1} \ a \ b$

**shows**  $r \ b \ a$  **using**  $ab$

**by** *cases simp*

**lemma** *conversep-iff* [iff]:  $r^{\hat{\ } - - 1} \ a \ b = r \ b \ a$

**by** (*iprover intro: conversepI dest: conversepD*)

**lemma** *conversep-converse-eq* [pred-set-conv]:

$(\lambda x y. (x, y) \in r)^{\hat{\ } - - 1} = (\lambda x y. (x, y) \in r^{\hat{\ } - 1})$

**by** (*auto simp add: expand-fun-eq*)

**lemma** *conversep-conversep* [simp]:  $(r^{\hat{\ } - - 1})^{\hat{\ } - - 1} = r$

**by** (*iprover intro: order-antisym conversepI dest: conversepD*)

**lemma** *converse-pred-comp*:  $(r \text{ OO } s)^{\hat{---}1} = s^{\hat{---}1} \text{ OO } r^{\hat{---}1}$   
**by** (*iprover intro: order-antisym conversepI pred-compI*  
*elim: pred-compE dest: conversepD*)

**lemma** *converse-meet*:  $(\inf r s)^{\hat{---}1} = \inf r^{\hat{---}1} s^{\hat{---}1}$   
**by** (*simp add: inf-fun-eq inf-bool-eq*  
*(iprover intro: conversepI ext dest: conversepD)*)

**lemma** *converse-join*:  $(\sup r s)^{\hat{---}1} = \sup r^{\hat{---}1} s^{\hat{---}1}$   
**by** (*simp add: sup-fun-eq sup-bool-eq*  
*(iprover intro: conversepI ext dest: conversepD)*)

**lemma** *conversep-noteq [simp]*:  $(op \sim)^{\hat{---}1} = op \sim$   
**by** (*auto simp add: expand-fun-eq*)

**lemma** *conversep-eq [simp]*:  $(op =)^{\hat{---}1} = op =$   
**by** (*auto simp add: expand-fun-eq*)

### 22.2.3 Domain

**inductive**

*DomainP* ::  $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool}$   
**for** *r* ::  $'a \Rightarrow 'b \Rightarrow \text{bool}$

**where**

*DomainPI* [*intro*]:  $r a b \Rightarrow \text{DomainP } r a$

**inductive-cases** *DomainPE* [*elim!*]:  $\text{DomainP } r a$

**lemma** *DomainP-Domain-eq [pred-set-conv]*:  $\text{DomainP } (\lambda x y. (x, y) \in r) = (\lambda x. x \in \text{Domain } r)$   
**by** (*blast intro!: Orderings.order-antisym predicate1I*)

### 22.2.4 Range

**inductive**

*RangeP* ::  $('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'b \Rightarrow \text{bool}$   
**for** *r* ::  $'a \Rightarrow 'b \Rightarrow \text{bool}$

**where**

*RangePI* [*intro*]:  $r a b \Rightarrow \text{RangeP } r b$

**inductive-cases** *RangePE* [*elim!*]:  $\text{RangeP } r b$

**lemma** *RangeP-Range-eq [pred-set-conv]*:  $\text{RangeP } (\lambda x y. (x, y) \in r) = (\lambda x. x \in \text{Range } r)$   
**by** (*blast intro!: Orderings.order-antisym predicate1I*)

### 22.2.5 Inverse image

**definition**

$inv\text{-}imagep :: ('b \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$  **where**  
 $inv\text{-}imagep\ r\ f == \%x\ y.\ r\ (f\ x)\ (f\ y)$

**lemma**  $[pred\text{-}set\text{-}conv]$ :  $inv\text{-}imagep\ (\lambda x\ y.\ (x, y) \in r)\ f = (\lambda x\ y.\ (x, y) \in inv\text{-}image\ r\ f)$

**by**  $(simp\ add:\ inv\text{-}image\text{-}def\ inv\text{-}imagep\text{-}def)$

**lemma**  $in\text{-}inv\text{-}imagep\ [simp]$ :  $inv\text{-}imagep\ r\ f\ x\ y = r\ (f\ x)\ (f\ y)$

**by**  $(simp\ add:\ inv\text{-}imagep\text{-}def)$

### 22.2.6 Powerset

**definition**  $Powp :: ('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$  **where**

$Powp\ A == \lambda B.\ \forall x \in B.\ A\ x$

**lemma**  $Powp\text{-}Pow\text{-}eq\ [pred\text{-}set\text{-}conv]$ :  $Powp\ (\lambda x.\ x \in A) = (\lambda x.\ x \in Pow\ A)$

**by**  $(auto\ simp\ add:\ Powp\text{-}def\ expand\text{-}fun\text{-}eq)$

**lemmas**  $Powp\text{-}mono\ [mono] = Pow\text{-}mono\ [to\text{-}pred\ pred\text{-}subset\text{-}eq]$

### 22.2.7 Properties of relations

**abbreviation**  $antisymP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$  **where**

$antisymP\ r == antisym\ \{(x, y).\ r\ x\ y\}$

**abbreviation**  $transP :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$  **where**

$transP\ r == trans\ \{(x, y).\ r\ x\ y\}$

**abbreviation**  $single\text{-}valuedP :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$  **where**

$single\text{-}valuedP\ r == single\text{-}valued\ \{(x, y).\ r\ x\ y\}$

## 22.3 Predicates as enumerations

### 22.3.1 The type of predicate enumerations (a monad)

**datatype**  $'a\ pred = Pred\ 'a \Rightarrow bool$

**primrec**  $eval :: 'a\ pred \Rightarrow 'a \Rightarrow bool$  **where**

$eval\text{-}pred:\ eval\ (Pred\ f) = f$

**lemma**  $Pred\text{-}eval\ [simp]$ :

$Pred\ (eval\ x) = x$

**by**  $(cases\ x)\ simp$

**lemma**  $eval\text{-}inject:$   $eval\ x = eval\ y \longleftrightarrow x = y$

**by**  $(cases\ x)\ auto$

**definition**  $single :: 'a \Rightarrow 'a\ pred$  **where**

$single\ x = Pred\ ((op\ =)\ x)$

**definition**  $bind :: 'a\ pred \Rightarrow ('a \Rightarrow 'b\ pred) \Rightarrow 'b\ pred$  (**infixl**  $\gg=$  70) **where**  
 $P\ \gg= f = Pred\ (\lambda x. (\exists y. eval\ P\ y \wedge eval\ (f\ y)\ x))$

**instantiation**  $pred :: (type) \{complete-lattice, boolean-algebra\}$   
**begin**

**definition**

$P \leq Q \longleftrightarrow eval\ P \leq eval\ Q$

**definition**

$P < Q \longleftrightarrow eval\ P < eval\ Q$

**definition**

$\perp = Pred\ \perp$

**definition**

$\top = Pred\ \top$

**definition**

$P \sqcap Q = Pred\ (eval\ P \sqcap eval\ Q)$

**definition**

$P \sqcup Q = Pred\ (eval\ P \sqcup eval\ Q)$

**definition**

[code del]:  $\bigcap A = Pred\ (INFI\ A\ eval)$

**definition**

[code del]:  $\bigcup A = Pred\ (SUPR\ A\ eval)$

**definition**

$- P = Pred\ (-\ eval\ P)$

**definition**

$P - Q = Pred\ (eval\ P - eval\ Q)$

**instance proof**

**qed** (*auto simp add: less-eq-pred-def less-pred-def*  
*inf-pred-def sup-pred-def bot-pred-def top-pred-def*  
*Inf-pred-def Sup-pred-def uminus-pred-def minus-pred-def fun-Compl-def bool-Compl-def,*  
*auto simp add: le-fun-def less-fun-def le-bool-def less-bool-def*  
*eval-inject mem-def*)

**end**

**lemma** *bind-bind*:

$(P\ \gg= Q)\ \gg= R = P\ \gg= (\lambda x. Q\ x\ \gg= R)$



**by** (*auto simp add: bind-def expand-fun-eq*)

**lemma** *bind-single*:

$P \gg= \text{single} = P$

**by** (*simp add: bind-def single-def*)

**lemma** *single-bind*:

$\text{single } x \gg= P = P \ x$

**by** (*simp add: bind-def single-def*)

**lemma** *bottom-bind*:

$\perp \gg= P = \perp$

**by** (*auto simp add: bot-pred-def bind-def expand-fun-eq*)

**lemma** *sup-bind*:

$(P \sqcup Q) \gg= R = P \gg= R \sqcup Q \gg= R$

**by** (*auto simp add: bind-def sup-pred-def expand-fun-eq*)

**lemma** *Sup-bind*:  $(\bigsqcup A \gg= f) = \bigsqcup ((\lambda x. x \gg= f) \text{ ` } A)$

**by** (*auto simp add: bind-def Sup-pred-def SUP1-iff expand-fun-eq*)

**lemma** *pred-iffI*:

**assumes**  $\bigwedge x. \text{eval } A \ x \Longrightarrow \text{eval } B \ x$

**and**  $\bigwedge x. \text{eval } B \ x \Longrightarrow \text{eval } A \ x$

**shows**  $A = B$

**proof** –

**from** *assms* **have**  $\bigwedge x. \text{eval } A \ x \longleftrightarrow \text{eval } B \ x$  **by** *blast*

**then show** *?thesis* **by** (*cases A, cases B*) (*simp add: expand-fun-eq*)

**qed**

**lemma** *singleI*:  $\text{eval } (\text{single } x) \ x$

**unfolding** *single-def* **by** *simp*

**lemma** *singleI-unit*:  $\text{eval } (\text{single } ()) \ x$

**by** *simp* (*rule singleI*)

**lemma** *singleE*:  $\text{eval } (\text{single } x) \ y \Longrightarrow (y = x \Longrightarrow P) \Longrightarrow P$

**unfolding** *single-def* **by** *simp*

**lemma** *singleE'*:  $\text{eval } (\text{single } x) \ y \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$

**by** (*erule singleE*) *simp*

**lemma** *bindI*:  $\text{eval } P \ x \Longrightarrow \text{eval } (Q \ x) \ y \Longrightarrow \text{eval } (P \gg= Q) \ y$

**unfolding** *bind-def* **by** *auto*

**lemma** *bindE*:  $\text{eval } (R \gg= Q) \ y \Longrightarrow (\bigwedge x. \text{eval } R \ x \Longrightarrow \text{eval } (Q \ x) \ y \Longrightarrow P) \Longrightarrow P$

**unfolding** *bind-def* **by** *auto*

**lemma** *botE*:  $eval \perp x \implies P$   
**unfolding** *bot-pred-def* **by** *auto*

**lemma** *supI1*:  $eval A x \implies eval (A \sqcup B) x$   
**unfolding** *sup-pred-def* **by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *supI2*:  $eval B x \implies eval (A \sqcup B) x$   
**unfolding** *sup-pred-def* **by** (*simp add: sup-fun-eq sup-bool-eq*)

**lemma** *supE*:  $eval (A \sqcup B) x \implies (eval A x \implies P) \implies (eval B x \implies P) \implies P$   
**unfolding** *sup-pred-def* **by** *auto*

**lemma** *single-not-bot* [*simp*]:  
 $single\ x \neq \perp$   
**by** (*auto simp add: single-def bot-pred-def expand-fun-eq*)

**lemma** *not-bot*:  
**assumes**  $A \neq \perp$   
**obtains**  $x$  **where**  $eval\ A\ x$   
**using** *assms* **by** (*cases A*)  
(*auto simp add: bot-pred-def, auto simp add: mem-def*)

### 22.3.2 Emptiness check and definite choice

**definition** *is-empty* ::  $'a\ pred \Rightarrow bool$  **where**  
 $is\_empty\ A \longleftrightarrow A = \perp$

**lemma** *is-empty-bot*:  
 $is\_empty\ \perp$   
**by** (*simp add: is-empty-def*)

**lemma** *not-is-empty-single*:  
 $\neg is\_empty\ (single\ x)$   
**by** (*auto simp add: is-empty-def single-def bot-pred-def expand-fun-eq*)

**lemma** *is-empty-sup*:  
 $is\_empty\ (A \sqcup B) \longleftrightarrow is\_empty\ A \wedge is\_empty\ B$   
**by** (*auto simp add: is-empty-def*)

**definition** *singleton* ::  $(unit \Rightarrow 'a) \Rightarrow 'a\ pred \Rightarrow 'a$  **where**  
 $singleton\ dfault\ A = (if\ \exists!x. eval\ A\ x\ then\ THE\ x. eval\ A\ x\ else\ dfault\ ())$

**lemma** *singleton-eqI*:  
 $\exists!x. eval\ A\ x \implies eval\ A\ x \implies singleton\ dfault\ A = x$   
**by** (*auto simp add: singleton-def*)

**lemma** *eval-singletonI*:  
 $\exists!x. eval\ A\ x \implies eval\ A\ (singleton\ dfault\ A)$

**proof** –

```

assume assm:  $\exists!x. \text{eval } A \ x$ 
then obtain x where  $\text{eval } A \ x \ ..$ 
moreover with assm have  $\text{singleton dfault } A = x$  by (rule singleton-eqI)
ultimately show ?thesis by simp
qed

```

```

lemma single-singleton:
   $\exists!x. \text{eval } A \ x \implies \text{single } (\text{singleton dfault } A) = A$ 
proof –
  assume assm:  $\exists!x. \text{eval } A \ x$ 
  then have  $\text{eval } A \ (\text{singleton dfault } A)$ 
    by (rule eval-singletonI)
  moreover from assm have  $\bigwedge x. \text{eval } A \ x \implies \text{singleton dfault } A = x$ 
    by (rule singleton-eqI)
  ultimately have  $\text{eval } (\text{single } (\text{singleton dfault } A)) = \text{eval } A$ 
    by (simp (no-asm-use) add: single-def expand-fun-eq) blast
  then show ?thesis by (simp add: eval-inject)
qed

```

```

lemma singleton-undefinedI:
   $\neg (\exists!x. \text{eval } A \ x) \implies \text{singleton dfault } A = \text{dfault } ()$ 
  by (simp add: singleton-def)

```

```

lemma singleton-bot:
   $\text{singleton dfault } \perp = \text{dfault } ()$ 
  by (auto simp add: bot-pred-def intro: singleton-undefinedI)

```

```

lemma singleton-single:
   $\text{singleton dfault } (\text{single } x) = x$ 
  by (auto simp add: intro: singleton-eqI singleI elim: singleE)

```

```

lemma singleton-sup-single-single:
   $\text{singleton dfault } (\text{single } x \sqcup \text{single } y) = (\text{if } x = y \text{ then } x \text{ else dfault } ())$ 
proof (cases x = y)
  case True then show ?thesis by (simp add: singleton-single)
next
  case False
  have  $\text{eval } (\text{single } x \sqcup \text{single } y) \ x$ 
    and  $\text{eval } (\text{single } x \sqcup \text{single } y) \ y$ 
    by (auto intro: supI1 supI2 singleI)
  with False have  $\neg (\exists!z. \text{eval } (\text{single } x \sqcup \text{single } y) \ z)$ 
    by blast
  then have  $\text{singleton dfault } (\text{single } x \sqcup \text{single } y) = \text{dfault } ()$ 
    by (rule singleton-undefinedI)
  with False show ?thesis by simp
qed

```

```

lemma singleton-sup-aux:
   $\text{singleton dfault } (A \sqcup B) = (\text{if } A = \perp \text{ then singleton dfault } B$ 

```

```

    else if  $B = \perp$  then singleton dfault A
    else singleton dfault
      (single (singleton dfault A)  $\sqcup$  single (singleton dfault B)))
proof (cases ( $\exists!x. \text{eval } A \ x$ )  $\wedge$  ( $\exists!y. \text{eval } B \ y$ ))
  case True then show ?thesis by (simp add: single-singleton)
next
case False
from False have A-or-B:
  singleton dfault A = dfault ()  $\vee$  singleton dfault B = dfault ()
  by (auto intro!: singleton-undefinedI)
then have rhs: singleton dfault
  (single (singleton dfault A)  $\sqcup$  single (singleton dfault B)) = dfault ()
  by (auto simp add: singleton-sup-single-single singleton-single)
from False have not-unique:
   $\neg (\exists!x. \text{eval } A \ x) \vee \neg (\exists!y. \text{eval } B \ y)$  by simp
show ?thesis proof (cases  $A \neq \perp \wedge B \neq \perp$ )
  case True
  then obtain a b where a: eval A a and b: eval B b
  by (blast elim: not-bot)
  with True not-unique have  $\neg (\exists!x. \text{eval } (A \sqcup B) \ x)$ 
  by (auto simp add: sup-pred-def bot-pred-def)
  then have singleton dfault (A  $\sqcup$  B) = dfault () by (rule singleton-undefinedI)
  with True rhs show ?thesis by simp
next
case False then show ?thesis by auto
qed
qed

lemma singleton-sup:
  singleton dfault (A  $\sqcup$  B) = (if A =  $\perp$  then singleton dfault B
    else if B =  $\perp$  then singleton dfault A
    else if singleton dfault A = singleton dfault B then singleton dfault A else dfault
  ())
using singleton-sup-aux [of dfault A B] by (simp only: singleton-sup-single-single)

```

### 22.3.3 Derived operations

**definition** if-pred :: bool  $\Rightarrow$  unit pred **where**  
 if-pred-eq: if-pred b = (if b then single () else  $\perp$ )

**definition** holds :: unit pred  $\Rightarrow$  bool **where**  
 holds-eq: holds P = eval P ()

**definition** not-pred :: unit pred  $\Rightarrow$  unit pred **where**  
 not-pred-eq: not-pred P = (if eval P () then  $\perp$  else single ())

**lemma** if-predI:  $P \Rightarrow \text{eval } (\text{if-pred } P) \ ()$   
**unfolding** if-pred-eq **by** (auto intro: singleI)

**lemma** *if-predE*:  $\text{eval } (\text{if-pred } b) \ x \implies (b \implies x = () \implies P) \implies P$   
**unfolding** *if-pred-eq* **by** (*cases b*) (*auto elim: botE*)

**lemma** *not-predI*:  $\neg P \implies \text{eval } (\text{not-pred } (\text{Pred } (\lambda u. P))) \ ()$   
**unfolding** *not-pred-eq eval-pred* **by** (*auto intro: singleI*)

**lemma** *not-predI'*:  $\neg \text{eval } P \ () \implies \text{eval } (\text{not-pred } P) \ ()$   
**unfolding** *not-pred-eq* **by** (*auto intro: singleI*)

**lemma** *not-predE*:  $\text{eval } (\text{not-pred } (\text{Pred } (\lambda u. P))) \ x \implies (\neg P \implies \text{thesis}) \implies \text{thesis}$   
**unfolding** *not-pred-eq*  
**by** (*auto split: split-if-asm elim: botE*)

**lemma** *not-predE'*:  $\text{eval } (\text{not-pred } P) \ x \implies (\neg \text{eval } P \ x \implies \text{thesis}) \implies \text{thesis}$   
**unfolding** *not-pred-eq*  
**by** (*auto split: split-if-asm elim: botE*)  
**lemma** *f*  $() = \text{False} \vee f \ () = \text{True}$   
**by** *simp*

**lemma** *closure-of-bool-cases*:  
**assumes**  $(f :: \text{unit} \Rightarrow \text{bool}) = (\%u. \text{False}) \implies P \ f$   
**assumes**  $f = (\%u. \text{True}) \implies P \ f$   
**shows**  $P \ f$   
**proof** –  
  **have**  $f = (\%u. \text{False}) \vee f = (\%u. \text{True})$   
  **apply** (*cases f*  $()$ )  
  **apply** (*rule disjI2*)  
  **apply** (*rule ext*)  
  **apply** (*simp add: unit-eq*)  
  **apply** (*rule disjI1*)  
  **apply** (*rule ext*)  
  **apply** (*simp add: unit-eq*)  
  **done**  
**from this prems show ?thesis by blast**  
**qed**

**lemma** *unit-pred-cases*:  
**assumes**  $P \perp$   
**assumes**  $P \ (\text{single } ())$   
**shows**  $P \ Q$   
**using** *assms*  
**unfolding** *bot-pred-def Collect-def empty-def single-def*  
**apply** (*cases Q*)  
**apply** *simp*  
**apply** (*rule-tac f=fun in closure-of-bool-cases*)  
**apply** *auto*  
**apply** (*subgoal-tac*  $(\%x. () = x) = (\%x. \text{True})$ )  
**apply** *auto*

done

**lemma** *holds-if-pred*:

*holds (if-pred b) = b*

**unfolding** *if-pred-eq holds-eq*

**by** (*cases b*) (*auto intro: singleI elim: botE*)

**lemma** *if-pred-holds*:

*if-pred (holds P) = P*

**unfolding** *if-pred-eq holds-eq*

**by** (*rule unit-pred-cases*) (*auto intro: singleI elim: botE*)

**lemma** *is-empty-holds*:

*is-empty P  $\longleftrightarrow$   $\neg$  holds P*

**unfolding** *is-empty-def holds-eq*

**by** (*rule unit-pred-cases*) (*auto elim: botE intro: singleI*)

### 22.3.4 Implementation

**datatype** *'a seq* = *Empty* | *Insert 'a 'a pred* | *Join 'a pred 'a seq*

**primrec** *pred-of-seq* :: *'a seq  $\Rightarrow$  'a pred* **where**

*pred-of-seq Empty =  $\perp$*

| *pred-of-seq (Insert x P) = single x  $\sqcup$  P*

| *pred-of-seq (Join P xq) = P  $\sqcup$  pred-of-seq xq*

**definition** *Seq* :: (*unit  $\Rightarrow$  'a seq*)  $\Rightarrow$  *'a pred* **where**

*Seq f = pred-of-seq (f ())*

**code-datatype** *Seq*

**primrec** *member* :: *'a seq  $\Rightarrow$  'a  $\Rightarrow$  bool* **where**

*member Empty x  $\longleftrightarrow$  False*

| *member (Insert y P) x  $\longleftrightarrow$  x = y  $\vee$  eval P x*

| *member (Join P xq) x  $\longleftrightarrow$  eval P x  $\vee$  member xq x*

**lemma** *eval-member*:

*member xq = eval (pred-of-seq xq)*

**proof** (*induct xq*)

**case** *Empty* **show** *?case*

**by** (*auto simp add: expand-fun-eq elim: botE*)

**next**

**case** *Insert* **show** *?case*

**by** (*auto simp add: expand-fun-eq elim: supE singleE intro: supI1 supI2 singleI*)

**next**

**case** *Join* **then show** *?case*

**by** (*auto simp add: expand-fun-eq elim: supE intro: supI1 supI2*)

**qed**

**lemma** *eval-code* [code]:  $\text{eval } (\text{Seq } f) = \text{member } (f \ ())$   
**unfolding** *Seq-def* **by** (*rule sym*, *rule eval-member*)

**lemma** *single-code* [code]:  
 $\text{single } x = \text{Seq } (\lambda u. \text{Insert } x \perp)$   
**unfolding** *Seq-def* **by** *simp*

**primrec** *apply* ::  $('a \Rightarrow 'b \text{ Predicate.pred}) \Rightarrow 'a \text{ seq} \Rightarrow 'b \text{ seq}$  **where**  
 $\text{apply } f \text{ Empty} = \text{Empty}$   
 $|\text{ apply } f (\text{Insert } x P) = \text{Join } (f x) (\text{Join } (P \gg= f) \text{ Empty})$   
 $|\text{ apply } f (\text{Join } P xq) = \text{Join } (P \gg= f) (\text{apply } f xq)$

**lemma** *apply-bind*:  
 $\text{pred-of-seq } (\text{apply } f xq) = \text{pred-of-seq } xq \gg= f$   
**proof** (*induct xq*)  
**case** *Empty* **show** ?case  
**by** (*simp add: bottom-bind*)  
**next**  
**case** *Insert* **show** ?case  
**by** (*simp add: single-bind sup-bind*)  
**next**  
**case** *Join* **then show** ?case  
**by** (*simp add: sup-bind*)  
**qed**

**lemma** *bind-code* [code]:  
 $\text{Seq } g \gg= f = \text{Seq } (\lambda u. \text{apply } f (g \ ()))$   
**unfolding** *Seq-def* **by** (*rule sym*, *rule apply-bind*)

**lemma** *bot-set-code* [code]:  
 $\perp = \text{Seq } (\lambda u. \text{Empty})$   
**unfolding** *Seq-def* **by** *simp*

**primrec** *adjunct* ::  $'a \text{ pred} \Rightarrow 'a \text{ seq} \Rightarrow 'a \text{ seq}$  **where**  
 $\text{adjunct } P \text{ Empty} = \text{Join } P \text{ Empty}$   
 $|\text{ adjunct } P (\text{Insert } x Q) = \text{Insert } x (Q \sqcup P)$   
 $|\text{ adjunct } P (\text{Join } Q xq) = \text{Join } Q (\text{adjunct } P xq)$

**lemma** *adjunct-sup*:  
 $\text{pred-of-seq } (\text{adjunct } P xq) = P \sqcup \text{pred-of-seq } xq$   
**by** (*induct xq*) (*simp-all add: sup-assoc sup-commute sup-left-commute*)

**lemma** *sup-code* [code]:  
 $\text{Seq } f \sqcup \text{Seq } g = \text{Seq } (\lambda u. \text{case } f \ ())$   
 $\text{of Empty} \Rightarrow g \ ()$   
 $|\text{ Insert } x P \Rightarrow \text{Insert } x (P \sqcup \text{Seq } g)$   
 $|\text{ Join } P xq \Rightarrow \text{adjunct } (\text{Seq } g) (\text{Join } P xq)$   
**proof** (*cases f* ())  
**case** *Empty*

```

thus ?thesis
  unfolding Seq-def by (simp add: sup-commute [of  $\perp$ ])
next
  case Insert
  thus ?thesis
    unfolding Seq-def by (simp add: sup-assoc)
next
  case Join
  thus ?thesis
    unfolding Seq-def
    by (simp add: adjunct-sup sup-assoc sup-commute sup-left-commute)
qed

```

```

primrec contained :: 'a seq  $\Rightarrow$  'a pred  $\Rightarrow$  bool where
  contained Empty Q  $\longleftrightarrow$  True
| contained (Insert x P) Q  $\longleftrightarrow$  eval Q x  $\wedge$  P  $\leq$  Q
| contained (Join P xq) Q  $\longleftrightarrow$  P  $\leq$  Q  $\wedge$  contained xq Q

```

```

lemma single-less-eq-eval:
  single x  $\leq$  P  $\longleftrightarrow$  eval P x
by (auto simp add: single-def less-eq-pred-def mem-def)

```

```

lemma contained-less-eq:
  contained xq Q  $\longleftrightarrow$  pred-of-seq xq  $\leq$  Q
by (induct xq) (simp-all add: single-less-eq-eval)

```

```

lemma less-eq-pred-code [code]:
  Seq f  $\leq$  Q = (case f ()
    of Empty  $\Rightarrow$  True
    | Insert x P  $\Rightarrow$  eval Q x  $\wedge$  P  $\leq$  Q
    | Join P xq  $\Rightarrow$  P  $\leq$  Q  $\wedge$  contained xq Q)
by (cases f ())
  (simp-all add: Seq-def single-less-eq-eval contained-less-eq)

```

```

lemma eq-pred-code [code]:
  fixes P Q :: 'a pred
  shows eq-class.eq P Q  $\longleftrightarrow$  P  $\leq$  Q  $\wedge$  Q  $\leq$  P
  unfolding eq by auto

```

```

lemma [code]:
  pred-case f P = f (eval P)
by (cases P) simp

```

```

lemma [code]:
  pred-rec f P = f (eval P)
by (cases P) simp

```

```

inductive eq :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool where eq x x

```



**lemma** *eq-is-eq*:  $eq\ x\ y \equiv (x = y)$   
**by** (*rule eq-reflection*) (*auto intro: eq.intros elim: eq.cases*)

**definition** *map* ::  $('a \Rightarrow 'b) \Rightarrow 'a\ pred \Rightarrow 'b\ pred$  **where**  
*map* *f* *P* = *P*  $\gg$  (*single o f*)

**primrec** *null* ::  $'a\ seq \Rightarrow bool$  **where**  
*null* *Empty*  $\longleftrightarrow True$   
| *null* (*Insert* *x* *P*)  $\longleftrightarrow False$   
| *null* (*Join* *P* *xq*)  $\longleftrightarrow is\_empty\ P \wedge null\ xq$

**lemma** *null-is-empty*:  
*null* *xq*  $\longleftrightarrow is\_empty\ (pred\_of\_seq\ xq)$   
**by** (*induct xq*) (*simp-all add: is-empty-bot not-is-empty-single is-empty-sup*)

**lemma** *is-empty-code* [*code*]:  
*is-empty* (*Seq* *f*)  $\longleftrightarrow null\ (f\ ())$   
**by** (*simp add: null-is-empty Seq-def*)

**primrec** *the-only* ::  $(unit \Rightarrow 'a) \Rightarrow 'a\ seq \Rightarrow 'a$  **where**  
[*code del*]: *the-only* *dfault* *Empty* = *dfault* ()  
| *the-only* *dfault* (*Insert* *x* *P*) = (*if is-empty* *P* *then* *x* *else* *let* *y* = *singleton* *dfault* *P* *in* *if* *x* = *y* *then* *x* *else* *dfault* ())  
| *the-only* *dfault* (*Join* *P* *xq*) = (*if is-empty* *P* *then* *the-only* *dfault* *xq* *else* *if* *null* *xq* *then* *singleton* *dfault* *P* *else* *let* *x* = *singleton* *dfault* *P*; *y* = *the-only* *dfault* *xq* *in* *if* *x* = *y* *then* *x* *else* *dfault* ())

**lemma** *the-only-singleton*:  
*the-only* *dfault* *xq* = *singleton* *dfault* (*pred-of-seq* *xq*)  
**by** (*induct xq*)  
(*auto simp add: singleton-bot singleton-single is-empty-def*  
*null-is-empty Let-def singleton-sup*)

**lemma** *singleton-code* [*code*]:  
*singleton* *dfault* (*Seq* *f*) = (*case* *f* ())  
*of* *Empty*  $\Rightarrow$  *dfault* ()  
| *Insert* *x* *P*  $\Rightarrow$  *if is-empty* *P* *then* *x*  
*else* *let* *y* = *singleton* *dfault* *P* *in*  
*if* *x* = *y* *then* *x* *else* *dfault* ()  
| *Join* *P* *xq*  $\Rightarrow$  *if is-empty* *P* *then* *the-only* *dfault* *xq*  
*else* *if* *null* *xq* *then* *singleton* *dfault* *P*  
*else* *let* *x* = *singleton* *dfault* *P*; *y* = *the-only* *dfault* *xq* *in*  
*if* *x* = *y* *then* *x* *else* *dfault* ()  
**by** (*cases f* ())  
(*auto simp add: Seq-def the-only-singleton is-empty-def*  
*null-is-empty singleton-bot singleton-single singleton-sup Let-def*)

**definition** *not-unique* ::  $'a\ pred \Rightarrow 'a$

**where**

[code del]: *not-unique*  $A = (THE\ x.\ eval\ A\ x)$

**definition** *the* ::  $'a\ pred \Rightarrow 'a$

**where**

[code del]: *the*  $A = (THE\ x.\ eval\ A\ x)$

**lemma** *the-eq*[code]: *the*  $A = singleton\ (\lambda x.\ not\_unique\ A)\ A$

**by** (*auto simp add: the-def singleton-def not-unique-def*)

**code-abort** *not-unique*

**code-reflect** *Predicate*

**datatypes** *pred* = *Seq* **and** *seq* = *Empty* | *Insert* | *Join*

**functions** *map*

**ML**  $\langle\langle$

*signature* *PREDICATE* =

*sig*

*datatype*  $'a\ pred = Seq\ of\ (unit \rightarrow 'a\ seq)$

*and*  $'a\ seq = Empty\ |\ Insert\ of\ 'a\ *\ 'a\ pred\ |\ Join\ of\ 'a\ pred\ *\ 'a\ seq$

*val* *yield*:  $'a\ pred \rightarrow ('a\ *\ 'a\ pred)\ option$

*val* *yieldn*:  $int \rightarrow 'a\ pred \rightarrow 'a\ list\ *\ 'a\ pred$

*val* *map*:  $('a \rightarrow 'b) \rightarrow 'a\ pred \rightarrow 'b\ pred$

*end*;

*structure* *Predicate* : *PREDICATE* =

*struct*

*datatype* *pred* = *datatype* *Predicate.pred*

*datatype* *seq* = *datatype* *Predicate.seq*

*fun* *map* *f* = *Predicate.map* *f*;

*fun* *yield* (*Seq* *f*) = *next* (*f* ())

*and* *next* *Empty* = *NONE*

| *next* (*Insert* (*x*, *P*)) = *SOME* (*x*, *P*)

| *next* (*Join* (*P*, *xq*)) = (case *yield* *P*

of *NONE* => *next* *xq*

| *SOME* (*x*, *Q*) => *SOME* (*x*, *Seq* (*fn* - => *Join* (*Q*, *xq*))));

*fun* *anamorph* *f* *k* *x* = (if *k* = 0 then ([], *x*)

else case *f* *x*

of *NONE* => ([], *x*)

| *SOME* (*v*, *y*) => let

*val* (*vs*, *z*) = *anamorph* *f* (*k* - 1) *y*

in (*v* :: *vs*, *z*) *end*);

*fun* *yieldn* *P* = *anamorph* *yield* *P*;

end;  
 >>

#### no-notation

*inf* (**infixl**  $\sqcap$  70) **and**  
*sup* (**infixl**  $\sqcup$  65) **and**  
*Inf* ( $\sqcap$  - [900] 900) **and**  
*Sup* ( $\sqcup$  - [900] 900) **and**  
*top* ( $\top$ ) **and**  
*bot* ( $\perp$ ) **and**  
*bind* (**infixl**  $\gg=$  70)

**hide-type** (**open**) *pred seq*

**hide-const** (**open**) *Pred eval single bind is-empty singleton if-pred not-pred holds*  
*Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map*  
*not-unique the*

**end**

## 23 Transitive-Closure: Reflexive and Transitive closure of a relation

**theory** *Transitive-Closure*

**imports** *Predicate*

**uses**  $\sim\sim$ /src/Provers/trancl.ML

**begin**

*rtrancl* is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

#### inductive-set

*rtrancl* :: ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set (( $\hat{*}$ ) [1000] 999)

**for** *r* :: ('a  $\times$  'a) set

**where**

*rtrancl-refl* [*intro!*, *Pure.intro!*, *simp*]: (*a*, *a*) :  $r^{\hat{*}}$   
 | *rtrancl-into-rtrancl* [*Pure.intro*]: (*a*, *b*) :  $r^{\hat{*}} \Rightarrow (b, c) : r \Rightarrow (a, c) : r^{\hat{*}}$

#### inductive-set

*trancl* :: ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\times$  'a) set (( $\hat{+}$ ) [1000] 999)

**for** *r* :: ('a  $\times$  'a) set

**where**

*r-into-trancl* [*intro*, *Pure.intro*]: (*a*, *b*) :  $r \Rightarrow (a, b) : r^{\hat{+}}$   
 | *trancl-into-trancl* [*Pure.intro*]: (*a*, *b*) :  $r^{\hat{+}} \Rightarrow (b, c) : r \Rightarrow (a, c) : r^{\hat{+}}$

**declare** *rtrancl-def* [*nitpick-def del*]

*rtranclp-def* [nitpick-def del]  
*trancl-def* [nitpick-def del]  
*tranclp-def* [nitpick-def del]

**notation**

*rtranclp*  $((-^{**}) [1000] 1000)$  **and**  
*tranclp*  $((-^{++}) [1000] 1000)$

**abbreviation**

*reflclp*  $:: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool} \quad ((-^{\hat{=}}) [1000] 1000)$   
**where**  
 $r^{\hat{=}} == \sup r \text{ op} =$

**abbreviation**

*reflcl*  $:: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \quad ((-^{\hat{=}}) [1000] 999)$  **where**  
 $r^{\hat{=}} == r \cup \text{Id}$

**notation** (*xsymbols*)

*rtranclp*  $((-^{**}) [1000] 1000)$  **and**  
*tranclp*  $((-^{++}) [1000] 1000)$  **and**  
*reflclp*  $((-^{\hat{=}}) [1000] 1000)$  **and**  
*rtrancl*  $((-^{*}) [1000] 999)$  **and**  
*trancl*  $((-^{+}) [1000] 999)$  **and**  
*reflcl*  $((-^{=}) [1000] 999)$

**notation** (*HTML output*)

*rtranclp*  $((-^{**}) [1000] 1000)$  **and**  
*tranclp*  $((-^{++}) [1000] 1000)$  **and**  
*reflclp*  $((-^{\hat{=}}) [1000] 1000)$  **and**  
*rtrancl*  $((-^{*}) [1000] 999)$  **and**  
*trancl*  $((-^{+}) [1000] 999)$  **and**  
*reflcl*  $((-^{=}) [1000] 999)$

**23.1 Reflexive closure**

**lemma** *refl-reflcl[simp]*:  $\text{refl}(r^{\hat{=}})$   
**by**(*simp add:refl-on-def*)

**lemma** *antisym-reflcl[simp]*:  $\text{antisym}(r^{\hat{=}}) = \text{antisym } r$   
**by**(*simp add:antisym-def*)

**lemma** *trans-reflclI[simp]*:  $\text{trans } r \Longrightarrow \text{trans}(r^{\hat{=}})$   
**unfolding** *trans-def* **by** *blast*

**23.2 Reflexive-transitive closure**

**lemma** *reflcl-set-eq [pred-set-conv]*:  $(\sup (\lambda x y. (x, y) \in r) \text{ op } =) = (\lambda x y. (x, y) \in r \cup \text{Id})$   
**by** (*auto simp add: expand-fun-eq*)

**lemma** *r-into-rtrancl* [intro]:  $!!p. p \in r \implies p \in r^*$   
 — *rtrancl* of *r* contains *r*  
**apply** (*simp only: split-tupled-all*)  
**apply** (*erule rtrancl-refl [THEN rtrancl-into-rtrancl]*)  
**done**

**lemma** *r-into-rtranclp* [intro]:  $r\ x\ y \implies r^{**}\ x\ y$   
 — *rtrancl* of *r* contains *r*  
**by** (*erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl]*)

**lemma** *rtranclp-mono*:  $r \leq s \implies r^{**} \leq s^{**}$   
 — monotonicity of *rtrancl*  
**apply** (*rule predicate2I*)  
**apply** (*erule rtranclp.induct*)  
**apply** (*rule-tac [2] rtranclp.rtrancl-into-rtrancl, blast+*)  
**done**

**lemmas** *rtrancl-mono* = *rtranclp-mono* [to-set]

**theorem** *rtranclp-induct* [consumes 1, case-names *base step*, induct set: *rtranclp*]:  
**assumes** *a*:  $r^{**}\ a\ b$   
**and cases**:  $P\ a\ !!y\ z. [\ [r^{**}\ a\ y; r\ y\ z; P\ y\ ] \implies P\ z$   
**shows**  $P\ b$  **using** *a*  
**by** (*induct x $\equiv$ a b*) (*rule cases*)+

**lemmas** *rtrancl-induct* [induct set: *rtrancl*] = *rtranclp-induct* [to-set]

**lemmas** *rtranclp-induct2* =  
*rtranclp-induct*[of - (*ax,ay*) (*bx,by*), *split-rule*,  
 consumes 1, case-names *refl step*]

**lemmas** *rtrancl-induct2* =  
*rtrancl-induct*[of (*ax,ay*) (*bx,by*), *split-format* (*complete*),  
 consumes 1, case-names *refl step*]

**lemma** *refl-rtrancl*: *refl* ( $r^*$ )  
**by** (*unfold refl-on-def*) *fast*

Transitivity of transitive closure.

**lemma** *trans-rtrancl*: *trans* ( $r^*$ )  
**proof** (*rule transI*)  
**fix** *x y z*  
**assume**  $(x, y) \in r^*$   
**assume**  $(y, z) \in r^*$   
**then show**  $(x, z) \in r^*$   
**proof** *induct*  
**case** *base*  
**show**  $(x, y) \in r^*$  **by** *fact*  
**next**

```

    case (step u v)
    from  $\langle (x, u) \in r^* \rangle$  and  $\langle (u, v) \in r \rangle$ 
    show  $\langle (x, v) \in r^* \rangle$  ..
qed
qed

```

lemmas *rtrancl-trans* = *trans-rtrancl* [THEN *transD*, *standard*]

```

lemma rtranclp-trans:
  assumes  $xy: r^{**} x y$ 
  and  $yz: r^{**} y z$ 
  shows  $r^{**} x z$  using yz xy
  by induct iprover+

```

```

lemma rtranclE [cases set: rtrancl]:
  assumes major:  $(a::'a, b) : r^*$ 
  obtains
    (base)  $a = b$ 
  | (step)  $y$  where  $(a, y) : r^*$  and  $(y, b) : r$ 
  — elimination of rtrancl – by induction on a special formula
  apply (subgoal-tac  $(a::'a) = b \mid (EX y. (a, y) : r^* \ \& \ (y, b) : r)$ )
  apply (rule-tac [2] major [THEN rtrancl-induct])
  prefer 2 apply blast
  prefer 2 apply blast
  apply (erule asm-rl exE disjE conjE base step)+
  done

```

```

lemma rtrancl-Int-subset: [ $Id \subseteq s; (r^* \cap s) \subseteq r \subseteq s$ ] ==>  $r^* \subseteq s$ 
  apply (rule subsetI)
  apply (rule-tac p=x in PairE, clarify)
  apply (erule rtrancl-induct, auto)
  done

```

```

lemma converse-rtranclp-into-rtranclp:
   $r a b \implies r^{**} b c \implies r^{**} a c$ 
  by (rule rtranclp-trans) iprover+

```

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [to-set]

More  $r^*$  equations and inclusions.

```

lemma rtranclp-idemp [simp]:  $(r^{**})^{**} = r^{**}$ 
  apply (auto intro!: order-antisym)
  apply (erule rtranclp-induct)
  apply (rule rtranclp.rtrancl-refl)
  apply (blast intro: rtranclp-trans)
  done

```

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [to-set]

```

lemma rtrancl-idemp-self-comp [simp]:  $R^{\wedge*} \circ R^{\wedge*} = R^{\wedge*}$ 
  apply (rule set-ext)
  apply (simp only: split-tupled-all)
  apply (blast intro: rtrancl-trans)
  done

```

```

lemma rtrancl-subset-rtrancl:  $r \subseteq s^{\wedge*} \implies r^{\wedge*} \subseteq s^{\wedge*}$ 
  apply (drule rtrancl-mono)
  apply simp
  done

```

```

lemma rtranclp-subset:  $R \leq S \implies S \leq R^{\wedge**} \implies S^{\wedge**} = R^{\wedge**}$ 
  apply (drule rtranclp-mono)
  apply (drule rtranclp-mono)
  apply simp
  done

```

```

lemmas rtrancl-subset = rtranclp-subset [to-set]

```

```

lemma rtranclp-sup-rtranclp:  $(\sup (R^{\wedge**}) (S^{\wedge**}))^{\wedge**} = (\sup R S)^{\wedge**}$ 
  by (blast intro!: rtranclp-subset intro: rtranclp-mono [THEN predicate2D])

```

```

lemmas rtrancl-Un-rtrancl = rtranclp-sup-rtranclp [to-set]

```

```

lemma rtranclp-reflcl [simp]:  $(R^{\wedge==})^{\wedge**} = R^{\wedge**}$ 
  by (blast intro!: rtranclp-subset)

```

```

lemmas rtrancl-reflcl [simp] = rtranclp-reflcl [to-set]

```

```

lemma rtrancl-r-diff-Id:  $(r - Id)^{\wedge*} = r^{\wedge*}$ 
  apply (rule sym)
  apply (rule rtrancl-subset, blast, clarify)
  apply (rename-tac a b)
  apply (case-tac a = b)
  apply blast
  apply (blast intro!: r-into-rtrancl)
  done

```

```

lemma rtranclp-r-diff-Id:  $(\inf r \text{ op } \sim)^{\wedge**} = r^{\wedge**}$ 
  apply (rule sym)
  apply (rule rtranclp-subset)
  apply blast+
  done

```

```

theorem rtranclp-converseD:
  assumes  $r: (r^{\wedge--1})^{\wedge**} x y$ 
  shows  $r^{\wedge**} y x$ 
proof –
  from  $r$  show ?thesis

```

by *induct* (*iprover* *intro*: *rtranclp-trans* *dest*!: *conversepD*) +  
qed

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

**theorem** *rtranclp-converseI*:

assumes  $r^{**} y x$

shows  $(r^{-1})^{**} x y$

using *assms*

by *induct* (*iprover* *intro*: *rtranclp-trans* *conversepI*) +

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

**lemma** *rtrancl-converse*:  $(r^{-1})^* = (r^*)^{-1}$

by (*fast* *dest*!: *rtrancl-converseD* *intro*!: *rtrancl-converseI*)

**lemma** *sym-rtrancl*:  $\text{sym } r \implies \text{sym } (r^*)$

by (*simp* *only*: *sym-conv-converse-eq* *rtrancl-converse* [*symmetric*])

**theorem** *converse-rtranclp-induct* [*consumes 1, case-names base step*]:

assumes *major*:  $r^{**} a b$

and *cases*:  $P b \text{ !! } y z. [r y z; r^{**} z b; P z] \implies P y$

shows  $P a$

using *rtranclp-converseI* [*OF major*]

by *induct* (*iprover* *intro*: *cases* *dest*!: *conversepD* *rtranclp-converseD*) +

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [*to-set*]

lemmas *converse-rtranclp-induct2* =

*converse-rtranclp-induct* [*of* - (*ax,ay*) (*bx,by*), *split-rule*,  
*consumes 1, case-names refl step*]

lemmas *converse-rtrancl-induct2* =

*converse-rtrancl-induct* [*of* (*ax,ay*) (*bx,by*), *split-format* (*complete*),  
*consumes 1, case-names refl step*]

**lemma** *converse-rtranclpE* [*consumes 1, case-names base step*]:

assumes *major*:  $r^{**} x z$

and *cases*:  $x = z \implies P$

!!*y*.  $[r x y; r^{**} y z] \implies P$

shows  $P$

apply (*subgoal-tac*  $x = z \mid (EX y. r x y \ \& \ r^{**} y z)$ )

apply (*rule-tac* [2] *major* [*THEN* *converse-rtranclp-induct*])

prefer 2 apply *iprover*

prefer 2 apply *iprover*

apply (*erule* *asm-rl* *exE* *disjE* *conjE* *cases*) +

done

lemmas *converse-rtranclE* = *converse-rtranclpE* [*to-set*]



**lemmas** *converse-rtranclpE2* = *converse-rtranclpE* [*of* - (*xa,xb*) (*za,zb*), *split-rule*]

**lemmas** *converse-rtranclE2* = *converse-rtranclE* [*of* (*xa,xb*) (*za,zb*), *split-rule*]

**lemma** *r-comp-rtrancl-eq*:  $r \circ r^* = r^* \circ r$

**by** (*blast elim: rtranclE converse-rtranclE*  
*intro: rtrancl-into-rtrancl converse-rtrancl-into-rtrancl*)

**lemma** *rtrancl-unfold*:  $r^* = Id \cup r^* \circ r$

**by** (*auto intro: rtrancl-into-rtrancl elim: rtranclE*)

**lemma** *rtrancl-Un-separatorE*:

(*a,b*) :  $(P \cup Q)^* \implies \forall x y. (a,x) : P^* \longrightarrow (x,y) : Q \longrightarrow x=y \implies (a,b) : P^*$   
**apply** (*induct rule:rtrancl.induct*)  
**apply** *blast*  
**apply** (*blast intro:rtrancl-trans*)  
**done**

**lemma** *rtrancl-Un-separator-converseE*:

(*a,b*) :  $(P \cup Q)^* \implies \forall x y. (x,b) : P^* \longrightarrow (y,x) : Q \longrightarrow y=x \implies (a,b) : P^*$   
**apply** (*induct rule:converse-rtrancl-induct*)  
**apply** *blast*  
**apply** (*blast intro:rtrancl-trans*)  
**done**

**lemma** *Image-closed-trancl*:

**assumes**  $r \text{ “ } X \subseteq X \text{ shows } r^* \text{ “ } X = X$   
**proof** –  
**from** *assms* **have** \*\*:  $\{y. \exists x \in X. (x, y) \in r\} \subseteq X$  **by** *auto*  
**have**  $\bigwedge x y. (y, x) \in r^* \implies y \in X \implies x \in X$   
**proof** –  
**fix** *x y*  
**assume** \*:  $y \in X$   
**assume**  $(y, x) \in r^*$   
**then show**  $x \in X$   
**proof** *induct*  
**case** *base* **show** ?*case* **by** (*fact* \*)  
**next**  
**case** *step* **with** \*\* **show** ?*case* **by** *auto*  
**qed**  
**qed**  
**then show** ?*thesis* **by** *auto*  
**qed**

### 23.3 Transitive closure

**lemma** *trancl-mono*:  $!!p. p \in r^+ \implies r \subseteq s \implies p \in s^+$

**apply** (*simp add: split-tupled-all*)

```

apply (erule trancl.induct)
apply (iprover dest: subsetD)+
done

```

```

lemma r-into-trancl': !!p. p : r ==> p : r^+
by (simp only: split-tupled-all) (erule r-into-trancl)

```

Conversions between *trancl* and *rtrancl*.

```

lemma tranclp-into-rtranclp: r^++ a b ==> r^** a b
by (erule tranclp.induct) iprover+

```

```

lemmas trancl-into-rtrancl = tranclp-into-rtranclp [to-set]

```

```

lemma rtranclp-into-tranclp1: assumes r: r^** a b
shows !!c. r b c ==> r^++ a c using r
by induct iprover+

```

```

lemmas rtrancl-into-trancl1 = rtranclp-into-tranclp1 [to-set]

```

```

lemma rtranclp-into-tranclp2: [| r a b; r^** b c |] ==> r^++ a c
— intro rule from r and rtrancl
apply (erule rtranclp.cases)
apply iprover
apply (rule rtranclp-trans [THEN rtranclp-into-tranclp1])
apply (simp | rule r-into-rtranclp)+
done

```

```

lemmas rtrancl-into-trancl2 = rtranclp-into-tranclp2 [to-set]

```

Nice induction rule for *trancl*

```

lemma tranclp-induct [consumes 1, case-names base step, induct pred: tranclp]:
assumes a: r^++ a b
and cases: !!y. r a y ==> P y
!!y z. r^++ a y ==> r y z ==> P y ==> P z
shows P b using a
by (induct x≡a b) (iprover intro: cases)+

```

```

lemmas trancl-induct [induct set: trancl] = tranclp-induct [to-set]

```

```

lemmas tranclp-induct2 =
tranclp-induct [of - (ax,ay) (bx,by), split-rule,
consumes 1, case-names base step]

```

```

lemmas trancl-induct2 =
trancl-induct [of (ax,ay) (bx,by), split-format (complete),
consumes 1, case-names base step]

```

```

lemma tranclp-trans-induct:
assumes major: r^++ x y

```

**and cases:**  $!!x\ y. r\ x\ y \implies P\ x\ y$   
 $!!x\ y\ z. [| r^++\ x\ y; P\ x\ y; r^++\ y\ z; P\ y\ z |] \implies P\ x\ z$   
**shows**  $P\ x\ y$   
 — Another induction rule for *tranc1*, incorporating transitivity  
**by** (*iprover intro: major [THEN tranc1p-induct] cases*)

**lemmas** *tranc1-trans-induct* = *tranc1p-trans-induct* [*to-set*]

**lemma** *tranc1E* [*cases set: tranc1*]:  
**assumes**  $(a, b) : r^+$   
**obtains**  
 (*base*)  $(a, b) : r$   
 | (*step*)  $c$  **where**  $(a, c) : r^+$  **and**  $(c, b) : r$   
**using** *assms* **by** *cases simp-all*

**lemma** *tranc1-Int-subset*:  $[| r \subseteq s; (r^+ \cap s) \ O\ r \subseteq s |] \implies r^+ \subseteq s$   
**apply** (*rule subsetI*)  
**apply** (*rule-tac p = x in PairE*)  
**apply** *clarify*  
**apply** (*erule tranc1-induct*)  
**apply** *auto*  
**done**

**lemma** *tranc1-unfold*:  $r^+ = r \ Un\ r^+ \ O\ r$   
**by** (*auto intro: tranc1-into-tranc1 elim: tranc1E*)

Transitivity of  $r^+$

**lemma** *trans-tranc1* [*simp*]: *trans*  $(r^+)$   
**proof** (*rule transI*)  
**fix**  $x\ y\ z$   
**assume**  $(x, y) \in r^+$   
**assume**  $(y, z) \in r^+$   
**then show**  $(x, z) \in r^+$   
**proof** *induct*  
**case** (*base u*)  
**from**  $\langle (x, y) \in r^+ \rangle$  **and**  $\langle (y, u) \in r \rangle$   
**show**  $(x, u) \in r^+ \ ..$   
**next**  
**case** (*step u v*)  
**from**  $\langle (x, u) \in r^+ \rangle$  **and**  $\langle (u, v) \in r \rangle$   
**show**  $(x, v) \in r^+ \ ..$   
**qed**  
**qed**

**lemmas** *tranc1-trans* = *trans-tranc1* [*THEN transD, standard*]

**lemma** *tranc1p-trans*:  
**assumes**  $xy: r^++\ x\ y$   
**and**  $yz: r^++\ y\ z$

```

shows  $r^{++} x z$  using  $yz xy$ 
by induct iprover+

lemma trancl-id [simp]: trans r  $\implies r^+ = r$ 
  apply auto
  apply (erule trancl-induct)
  apply assumption
  apply (unfold trans-def)
  apply blast
done

lemma rtranclp-tranclp-tranclp:
  assumes  $r^{**} x y$ 
  shows  $!!z. r^{++} y z \implies r^{++} x z$  using assms
  by induct (iprover intro: tranclp-trans)+

lemmas rtrancl-trancl-trancl = rtranclp-tranclp-tranclp [to-set]

lemma tranclp-into-tranclp2: r a b  $\implies r^{++} b c \implies r^{++} a c$ 
  by (erule tranclp-trans [OF tranclp.r-into-trancl])

lemmas trancl-into-trancl2 = tranclp-into-tranclp2 [to-set]

lemma trancl-insert:
   $(\text{insert } (y, x) r)^+ = r^+ \cup \{(a, b). (a, y) \in r^* \wedge (x, b) \in r^*\}$ 
  — primitive recursion for trancl over finite relations
  apply (rule equalityI)
  apply (rule subsetI)
  apply (simp only: split-tupled-all)
  apply (erule trancl-induct, blast)
  apply (blast intro: rtrancl-into-trancl1 trancl-into-rtrancl trancl-trans)
  apply (rule subsetI)
  apply (blast intro: trancl-mono rtrancl-mono
    [THEN [2] rev-subsetD] rtrancl-trancl-trancl rtrancl-into-trancl2)
done

lemma tranclp-converseI:  $(r^{++})^{--1} x y \implies (r^{--1})^{++} x y$ 
  apply (drule conversepD)
  apply (erule tranclp-induct)
  apply (iprover intro: conversepI tranclp-trans) +
done

lemmas trancl-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD:  $(r^{--1})^{++} x y \implies (r^{++})^{--1} x y$ 
  apply (rule conversepI)
  apply (erule tranclp-induct)
  apply (iprover dest: conversepD intro: tranclp-trans) +
done

```

**lemmas** *trancl-converseD* = *tranclp-converseD* [*to-set*]

**lemma** *tranclp-converse*:  $(r^{--1})^{++} = (r^{++})^{--1}$   
**by** (*fastsimp simp add: expand-fun-eq*  
*intro!: tranclp-converseI dest!: tranclp-converseD*)

**lemmas** *trancl-converse* = *tranclp-converse* [*to-set*]

**lemma** *sym-trancl*:  $\text{sym } r \implies \text{sym } (r^+)$   
**by** (*simp only: sym-conv-converse-eq trancl-converse [symmetric]*)

**lemma** *converse-trancl-induct* [*consumes 1, case-names base step*]:  
**assumes** *major*:  $r^{++} a b$   
**and cases**:  $\forall y. r y b \implies P(y)$   
 $\forall y z. [r y z; r^{++} z b; P(z)] \implies P(y)$   
**shows**  $P a$   
**apply** (*rule tranclp-induct [OF tranclp-converseI, OF conversepI, OF major]*)  
**apply** (*rule cases*)  
**apply** (*erule conversepD*)  
**apply** (*blast intro: assms dest!: tranclp-converseD*)  
**done**

**lemmas** *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

**lemma** *tranclpD*:  $R^{++} x y \implies \exists z. R x z \wedge R^{**} z y$   
**apply** (*erule converse-tranclp-induct*)  
**apply** *auto*  
**apply** (*blast intro: rtranclp-trans*)  
**done**

**lemmas** *tranclD* = *tranclpD* [*to-set*]

**lemma** *converse-tranclpE*:  
**assumes** *major*: *tranclp*  $r x z$   
**assumes** *base*:  $r x z \implies P$   
**assumes** *step*:  $\bigwedge y. [r x y; \text{tranclp } r y z] \implies P$   
**shows**  $P$   
**proof** –  
**from** *tranclpD* [*OF major*]  
**obtain**  $y$  **where**  $r x y$  **and** *rtranclp*  $r y z$  **by** *iprover*  
**from** *this*(2) **show**  $P$   
**proof** (*cases rule: rtranclp.cases*)  
**case** *rtrancl-refl*  
**with**  $\langle r x y \rangle$  *base* **show**  $P$  **by** *iprover*  
**next**  
**case** *rtrancl-into-rtrancl*  
**from** *this* **have** *tranclp*  $r y z$   
**by** (*iprover intro: rtranclp-into-tranclp1*)

```

    with ⟨r x y⟩ step show P by iprover
  qed
qed

lemmas converse-tranclE = converse-tranclpE [to-set]

lemma tranclD2:
  (x, y) ∈ R+ ⇒ ∃ z. (x, z) ∈ R* ∧ (z, y) ∈ R
  by (blast elim: tranclE intro: trancl-into-rtrancl)

lemma irrefl-tranclI: r+-1 ∩ r* = {} ==> (x, x) ∉ r+
  by (blast elim: tranclE dest: trancl-into-rtrancl)

lemma irrefl-trancl-rD: !!X. ALL x. (x, x) ∉ r+ ==> (x, y) ∈ r ==> x ≠ y
  by (blast dest: r-into-trancl)

lemma trancl-subset-Sigma-aux:
  (a, b) ∈ r* ==> r ⊆ A × A ==> a = b ∨ a ∈ A
  by (induct rule: rtrancl-induct) auto

lemma trancl-subset-Sigma: r ⊆ A × A ==> r+ ⊆ A × A
  apply (rule subsetI)
  apply (simp only: split-tupled-all)
  apply (erule tranclE)
  apply (blast dest!: trancl-into-rtrancl trancl-subset-Sigma-aux)+
  done

lemma reflcl-tranclp [simp]: (r++)+ = r**
  apply (safe intro!: order-antisym)
  apply (erule tranclp-into-rtranclp)
  apply (blast elim: rtranclp.cases dest: rtranclp-into-tranclp1)
  done

lemmas reflcl-trancl [simp] = reflcl-tranclp [to-set]

lemma trancl-reflcl [simp]: (r=)+ = r*
  apply safe
  apply (drule trancl-into-rtrancl, simp)
  apply (erule rtranclE, safe)
  apply (rule r-into-trancl, simp)
  apply (rule rtrancl-into-trancl1)
  apply (erule rtrancl-reflcl [THEN equalityD2, THEN subsetD], fast)
  done

lemma trancl-empty [simp]: {}+ = {}
  by (auto elim: trancl-induct)

lemma rtrancl-empty [simp]: {}* = Id
  by (rule subst [OF reflcl-trancl]) simp

```

**lemma** *rtranclpD*:  $R^* a b \implies a = b \vee a \neq b \wedge R^{++} a b$   
**by** (*force simp add: reflcl-tranclp [symmetric] simp del: reflcl-tranclp*)

**lemmas** *rtranclD* = *rtranclpD* [*to-set*]

**lemma** *rtrancl-eq-or-trancl*:  
 $(x, y) \in R^* = (x = y \vee x \neq y \wedge (x, y) \in R^+)$   
**by** (*fast elim: trancl-into-rtrancl dest: rtranclD*)

**lemma** *trancl-unfold-right*:  $r^+ = r^* \circ r$   
**by** (*auto dest: tranclD2 intro: rtrancl-into-trancl1*)

**lemma** *trancl-unfold-left*:  $r^+ = r \circ r^*$   
**by** (*auto dest: tranclD intro: rtrancl-into-trancl2*)

Simplifying nested closures

**lemma** *rtrancl-trancl-absorb[simp]*:  $(R^*)^+ = R^*$   
**by** (*simp add: trans-rtrancl*)

**lemma** *trancl-rtrancl-absorb[simp]*:  $(R^+)^* = R^*$   
**by** (*subst reflcl-trancl[symmetric] simp*)

**lemma** *rtrancl-reflcl-absorb[simp]*:  $(R^*)^+ = R^*$   
**by** *auto*

*Domain and Range*

**lemma** *Domain-rtrancl [simp]*:  $\text{Domain } (R^*) = \text{UNIV}$   
**by** *blast*

**lemma** *Range-rtrancl [simp]*:  $\text{Range } (R^*) = \text{UNIV}$   
**by** *blast*

**lemma** *rtrancl-Un-subset*:  $(R^* \cup S^*) \subseteq (R \cup S)^*$   
**by** (*rule rtrancl-Un-rtrancl [THEN subst] fast*)

**lemma** *in-rtrancl-UnI*:  $x \in R^* \vee x \in S^* \implies x \in (R \cup S)^*$   
**by** (*blast intro: subsetD [OF rtrancl-Un-subset]*)

**lemma** *trancl-domain [simp]*:  $\text{Domain } (r^+) = \text{Domain } r$   
**by** (*unfold Domain-def (blast dest: tranclD)*)

**lemma** *trancl-range [simp]*:  $\text{Range } (r^+) = \text{Range } r$   
**unfolding** *Range-def* **by** (*simp add: trancl-converse [symmetric]*)

**lemma** *Not-Domain-rtrancl*:  
 $x \sim: \text{Domain } R \implies ((x, y) : R^*) = (x = y)$   
**apply** *auto*  
**apply** (*erule rev-mp*)

```

apply (erule rtrancl-induct)
apply auto
done

```

```

lemma trancl-subset-Field2:  $r^+ \leq \text{Field } r \times \text{Field } r$ 
apply clarify
apply (erule trancl-induct)
apply (auto simp add: Field-def)
done

```

```

lemma finite-trancl:  $\text{finite } (r^+) = \text{finite } r$ 
apply auto
prefer 2
apply (rule trancl-subset-Field2 [THEN finite-subset])
apply (rule finite-SigmaI)
prefer 3
apply (blast intro: r-into-trancl' finite-subset)
apply (auto simp add: finite-Field)
done

```

More about converse *rtrancl* and *trancl*, should be merged with main body.

```

lemma single-valued-confluent:
  [| single-valued  $r$ ;  $(x,y) \in r^*$ ;  $(x,z) \in r^*$  |]
   $\implies (y,z) \in r^* \vee (z,y) \in r^*$ 
apply (erule rtrancl-induct)
apply simp
apply (erule disjE)
apply (blast elim: converse-rtranclE dest: single-valuedD)
apply (blast intro: rtrancl-trans)
done

```

```

lemma r-r-into-trancl:  $(a, b) \in R \implies (b, c) \in R \implies (a, c) \in R^+$ 
by (fast intro: trancl-trans)

```

```

lemma trancl-into-trancl [rule-format]:
   $(a, b) \in r^+ \implies (b, c) \in r \longrightarrow (a, c) \in r^+$ 
apply (erule trancl-induct)
apply (fast intro: r-r-into-trancl)
apply (fast intro: r-r-into-trancl trancl-trans)
done

```

```

lemma tranclp-rtranclp-tranclp:
   $r^{++} a b \implies r^{**} b c \implies r^{++} a c$ 
apply (drule tranclpD)
apply (elim exE conjE)
apply (drule rtranclp-trans, assumption)
apply (drule rtranclp-into-tranclp2, assumption, assumption)
done

```



**lemmas** *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

**lemmas** *transitive-closure-trans* [*trans*] =  
*r-r-into-trancl trancl-trans rtrancl-trans*  
*trancl.trancl-into-trancl trancl-into-trancl2*  
*rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl*  
*rtrancl-trancl-trancl trancl-rtrancl-trancl*

**lemmas** *transitive-closurep-trans'* [*trans*] =  
*tranclp-trans rtranclp-trans*  
*tranclp.trancl-into-trancl tranclp-into-tranclp2*  
*rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp*  
*rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp*

**declare** *trancl-into-rtrancl* [*elim*]

### 23.4 The power operation on relations

$R \hat{\ }^n = R \circ \dots \circ R$ , the  $n$ -fold composition of  $R$

**overloading**

*relpow* == *compow* ::  $\text{nat} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$   
**begin**

**primrec** *relpow* ::  $\text{nat} \Rightarrow ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set}$  **where**  
*relpow* 0  $R = \text{Id}$   
| *relpow* (*Suc*  $n$ )  $R = (R \hat{\ }^n) \circ R$

**end**

**lemma** *rel-pow-1* [*simp*]:  
**fixes**  $R :: ('a \times 'a) \text{ set}$   
**shows**  $R \hat{\ }^1 = R$   
**by** *simp*

**lemma** *rel-pow-0-I*:  
 $(x, x) \in R \hat{\ }^0$   
**by** *simp*

**lemma** *rel-pow-Suc-I*:  
 $(x, y) \in R \hat{\ }^n \implies (y, z) \in R \implies (x, z) \in R \hat{\ }^{Suc\ n}$   
**by** *auto*

**lemma** *rel-pow-Suc-I2*:  
 $(x, y) \in R \implies (y, z) \in R \hat{\ }^n \implies (x, z) \in R \hat{\ }^{Suc\ n}$   
**by** (*induct*  $n$  *arbitrary*:  $z$ ) (*simp*, *fastsimp*)

**lemma** *rel-pow-0-E*:  
 $(x, y) \in R \hat{\ }^0 \implies (x = y \implies P) \implies P$   
**by** *simp*

**lemma** *rel-pow-Suc-E*:

$(x, z) \in R^{\wedge\wedge} \text{Suc } n \implies (\bigwedge y. (x, y) \in R^{\wedge\wedge} n \implies (y, z) \in R \implies P) \implies P$   
**by** *auto*

**lemma** *rel-pow-E*:

$(x, z) \in R^{\wedge\wedge} n \implies (n = 0 \implies x = z \implies P)$   
 $\implies (\bigwedge y m. n = \text{Suc } m \implies (x, y) \in R^{\wedge\wedge} m \implies (y, z) \in R \implies P)$   
 $\implies P$   
**by** (*cases n*) *auto*

**lemma** *rel-pow-Suc-D2*:

$(x, z) \in R^{\wedge\wedge} \text{Suc } n \implies (\exists y. (x, y) \in R \wedge (y, z) \in R^{\wedge\wedge} n)$   
**apply** (*induct n arbitrary: x z*)  
**apply** (*blast intro: rel-pow-0-I elim: rel-pow-0-E rel-pow-Suc-E*)  
**apply** (*blast intro: rel-pow-Suc-I elim: rel-pow-0-E rel-pow-Suc-E*)  
**done**

**lemma** *rel-pow-Suc-E2*:

$(x, z) \in R^{\wedge\wedge} \text{Suc } n \implies (\bigwedge y. (x, y) \in R \implies (y, z) \in R^{\wedge\wedge} n \implies P) \implies P$   
**by** (*blast dest: rel-pow-Suc-D2*)

**lemma** *rel-pow-Suc-D2'*:

$\forall x y z. (x, y) \in R^{\wedge\wedge} n \wedge (y, z) \in R \longrightarrow (\exists w. (x, w) \in R \wedge (w, z) \in R^{\wedge\wedge} n)$   
**by** (*induct n*) (*simp-all, blast*)

**lemma** *rel-pow-E2*:

$(x, z) \in R^{\wedge\wedge} n \implies (n = 0 \implies x = z \implies P)$   
 $\implies (\bigwedge y m. n = \text{Suc } m \implies (x, y) \in R \implies (y, z) \in R^{\wedge\wedge} m \implies P)$   
 $\implies P$   
**apply** (*cases n, simp*)  
**apply** (*cut-tac n=nat and R=R in rel-pow-Suc-D2', simp, blast*)  
**done**

**lemma** *rel-pow-add*:  $R^{\wedge\wedge} (m+n) = R^{\wedge\wedge} m \circ R^{\wedge\wedge} n$

**by**(*induct n*) *auto*

**lemma** *rel-pow-commute*:  $R \circ R^{\wedge\wedge} n = R^{\wedge\wedge} n \circ R$

**by** (*induct n*) (*simp, simp add: O-assoc [symmetric]*)

**lemma** *rtrancl-imp-UN-rel-pow*:

**assumes**  $p \in R^{\wedge*}$   
**shows**  $p \in (\bigcup n. R^{\wedge\wedge} n)$   
**proof** (*cases p*)  
**case** (*Pair x y*)  
**with** *assms have*  $(x, y) \in R^{\wedge*}$  **by** *simp*  
**then have**  $(x, y) \in (\bigcup n. R^{\wedge\wedge} n)$  **proof** *induct*  
**case** *base* **show** ?*case* **by** (*blast intro: rel-pow-0-I*)  
**next**

```

    case step then show ?case by (blast intro: rel-pow-Suc-I)
  qed
  with Pair show ?thesis by simp
qed

```

```

lemma rel-pow-imp-rtrancl:
  assumes  $p \in R^{\wedge n}$ 
  shows  $p \in R^*$ 
proof (cases p)
  case (Pair x y)
  with assms have  $(x, y) \in R^{\wedge n}$  by simp
  then have  $(x, y) \in R^*$  proof (induct n arbitrary: x y)
    case 0 then show ?case by simp
  next
    case Suc then show ?case
      by (blast elim: rel-pow-Suc-E intro: rtrancl-into-rtrancl)
  qed
  with Pair show ?thesis by simp
qed

```

```

lemma rtrancl-is-UN-rel-pow:
   $R^* = (\bigcup n. R^{\wedge n})$ 
  by (blast intro: rtrancl-imp-UN-rel-pow rel-pow-imp-rtrancl)

```

```

lemma rtrancl-power:
   $p \in R^* \longleftrightarrow (\exists n. p \in R^{\wedge n})$ 
  by (simp add: rtrancl-is-UN-rel-pow)

```

```

lemma trancl-power:
   $p \in R^+ \longleftrightarrow (\exists n > 0. p \in R^{\wedge n})$ 
  apply (cases p)
  apply simp
  apply (rule iffI)
  apply (drule tranclD2)
  apply (clarsimp simp: rtrancl-is-UN-rel-pow)
  apply (rule-tac x=Suc n in exI)
  apply (clarsimp simp: rel-comp-def)
  apply fastsimp
  apply clarsimp
  apply (case-tac n, simp)
  apply clarsimp
  apply (drule rel-pow-imp-rtrancl)
  apply (drule rtrancl-into-trancl1) apply auto
done

```

```

lemma rtrancl-imp-rel-pow:
   $p \in R^* \implies \exists n. p \in R^{\wedge n}$ 
  by (auto dest: rtrancl-imp-UN-rel-pow)

```

```

lemma single-valued-rel-pow:
  fixes  $R :: ('a * 'a) \text{ set}$ 
  shows  $\text{single-valued } R \implies \text{single-valued } (R \wedge^n)$ 
  apply (induct n arbitrary: R)
  apply simp-all
  apply (rule single-valuedI)
  apply (fast dest: single-valuedD elim: rel-pow-Suc-E)
  done

```

### 23.5 Setup of transitivity reasoner

ML  $\ll$

```

structure Trancl-Tac = Trancl-Tac
(
  val r-into-trancl = @{thm trancl.r-into-trancl};
  val trancl-trans = @{thm trancl-trans};
  val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
  val r-into-rtrancl = @{thm r-into-rtrancl};
  val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
  val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
  val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
  val rtrancl-trans = @{thm rtrancl-trans};

  fun decomp (@{const Trueprop} $ t) =
    let fun dec (Const (op :, -) $ (Const (Pair, -) $ a $ b) $ rel) =
        let fun decr (Const (Transitive-Closure.rtrancl, -) $ r) = (r, r*)
            | decr (Const (Transitive-Closure.trancl, -) $ r) = (r, r+)
            | decr r = (r, r);
        val (rel, r) = decr (Envir.beta-eta-contract rel);
        in SOME (a, b, rel, r) end
      | dec - = NONE
    in dec t end
  | decomp - = NONE;
);

```

```

structure Tranclp-Tac = Trancl-Tac
(
  val r-into-trancl = @{thm tranclp.r-into-trancl};
  val trancl-trans = @{thm tranclp-trans};
  val rtrancl-refl = @{thm rtranclp.rtrancl-refl};
  val r-into-rtrancl = @{thm r-into-rtranclp};
  val trancl-into-rtrancl = @{thm tranclp-into-rtranclp};
  val rtrancl-trancl-trancl = @{thm rtranclp-tranclp-tranclp};
  val trancl-rtrancl-trancl = @{thm tranclp-rtranclp-tranclp};
  val rtrancl-trans = @{thm rtranclp-trans};

```

```

  fun decomp (@{const Trueprop} $ t) =
    let fun dec (rel $ a $ b) =

```

```

    let fun decr (Const (Transitive-Closure.rtranclp, -) $ r) = (r,r*)
      | decr (Const (Transitive-Closure.tranclp, -) $ r) = (r,r+)
      | decr r = (r,r);
    val (rel,r) = decr rel;
    in SOME (a, b, rel, r) end
  | dec - = NONE
in dec t end
| decomp - = NONE;
);
>>

declaration << fn - =>
  Simplifier.map-ss (fn ss => ss
    addSolver (mk-solver' Trancl (Trancl-Tac.trancl-tac o Simplifier.the-context))
    addSolver (mk-solver' Rtrancl (Trancl-Tac.rtrancl-tac o Simplifier.the-context))
    addSolver (mk-solver' Tranclp (Tranclp-Tac.trancl-tac o Simplifier.the-context))
    addSolver (mk-solver' Rtranclp (Tranclp-Tac.rtrancl-tac o Simplifier.the-context)))
  >>

```

Optional methods.

```

method-setup trancl =
  << Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.trancl-tac) >>
  << simple transitivity reasoner >>
method-setup rtrancl =
  << Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.rtrancl-tac) >>
  << simple transitivity reasoner >>
method-setup tranclp =
  << Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.trancl-tac) >>
  << simple transitivity reasoner (predicate version) >>
method-setup rtranclp =
  << Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.rtrancl-tac) >>
  << simple transitivity reasoner (predicate version) >>

end

```

## 24 Wellfounded: Well-founded Recursion

```

theory Wellfounded
imports Transitive-Closure
uses (Tools/Function/size.ML)
begin

```

### 24.1 Basic Definitions

```

definition wf :: ('a * 'a) set => bool where
  wf(r) == (!P. (!x. (!y. (y,x):r --> P(y)) --> P(x)) --> (!x. P(x)))

```

```

definition wfP :: ('a => 'a => bool) => bool where

```

$wfP\ r == wf\ \{(x, y). r\ x\ y\}$

**lemma** *wfP-wf-eq* [*pred-set-conv*]:  $wfP\ (\lambda x\ y. (x, y) \in r) = wf\ r$   
**by** (*simp add: wfP-def*)

**lemma** *wfUNIVI*:

$(!!P\ x. (ALL\ y. (y, x) : r \dashrightarrow P(y)) \dashrightarrow P(x)) ==> P(x) ==>$   
 $wf(r)$   
**unfolding** *wf-def* **by** *blast*

**lemmas** *wfPUNIVI* = *wfUNIVI* [*to-pred*]

Restriction to domain  $A$  and range  $B$ . If  $r$  is well-founded over their intersection, then  $wf\ r$

**lemma** *wfI*:

$[| r \subseteq A <*> B;$   
 $!!x\ P. [| \forall y. (y, x) : r \dashrightarrow P(y) \dashrightarrow P(x); x : A; x : B |] ==> P\ x |]$   
 $==> wf\ r$   
**unfolding** *wf-def* **by** *blast*

**lemma** *wf-induct*:

$[| wf(r);$   
 $!!x. [| ALL\ y. (y, x) : r \dashrightarrow P(y) |] ==> P(x)$   
 $|] ==> P(a)$   
**unfolding** *wf-def* **by** *blast*

**lemmas** *wfP-induct* = *wf-induct* [*to-pred*]

**lemmas** *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*,  
*induct set: wf*]

**lemmas** *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set: wfP*]

**lemma** *wf-not-sym*:  $wf\ r ==> (a, x) : r ==> (x, a) \sim: r$   
**by** (*induct a arbitrary: x set: wf*) *blast*

**lemma** *wf-asym*:

**assumes**  $wf\ r\ (a, x) \in r$   
**obtains**  $(x, a) \notin r$   
**by** (*drule wf-not-sym[OF assms]*)

**lemma** *wf-not-refl* [*simp*]:  $wf\ r ==> (a, a) \sim: r$   
**by** (*blast elim: wf-asym*)

**lemma** *wf-irrefl*: **assumes**  $wf\ r$  **obtains**  $(a, a) \notin r$   
**by** (*drule wf-not-refl[OF assms]*)

**lemma** *wf-wellorderI*:

**assumes**  $wf: wf\ \{(x::'a::ord, y). x < y\}$

```

assumes lin: OFCLASS('a::ord, linorder-class)
shows OFCLASS('a::ord, wellorder-class)
using lin by (rule wellorder-class.intro)
      (blast intro: class.wellorder-axioms.intro wf-induct-rule [OF wf])

```

```

lemma (in wellorder) wf:
  wf {(x, y). x < y}
unfolding wf-def by (blast intro: less-induct)

```

## 24.2 Basic Results

Point-free characterization of well-foundedness

```

lemma wfE-pf:
  assumes wf: wf R
  assumes a:  $A \subseteq R \text{ “ } A$ 
  shows  $A = \{\}$ 
proof –
  { fix x
    from wf have  $x \notin A$ 
    proof induct
      fix x assume  $\bigwedge y. (y, x) \in R \implies y \notin A$ 
      then have  $x \notin R \text{ “ } A$  by blast
      with a show  $x \notin A$  by blast
    qed
  } thus ?thesis by auto
qed

```

```

lemma wfI-pf:
  assumes a:  $\bigwedge A. A \subseteq R \text{ “ } A \implies A = \{\}$ 
  shows wf R
proof (rule wfUNIVI)
  fix P :: 'a  $\Rightarrow$  bool and x
  let ?A = {x.  $\neg P$  x}
  assume  $\forall x. (\forall y. (y, x) \in R \longrightarrow P$  y)  $\longrightarrow P$  x
  then have  $?A \subseteq R \text{ “ } ?A$  by blast
  with a show  $P$  x by blast
qed

```

Minimal-element characterization of well-foundedness

```

lemma wfE-min:
  assumes wf: wf R and Q:  $x \in Q$ 
  obtains z where  $z \in Q \bigwedge y. (y, z) \in R \implies y \notin Q$ 
  using Q wfE-pf[OF wf, of Q] by blast

lemma wfI-min:
  assumes a:  $\bigwedge x Q. x \in Q \implies \exists z \in Q. \forall y. (y, z) \in R \longrightarrow y \notin Q$ 
  shows wf R
proof (rule wfI-pf)
  fix A assume b:  $A \subseteq R \text{ “ } A$ 

```

```

{ fix x assume x ∈ A
  from a[OF this] b have False by blast
}
thus A = {} by blast
qed

```

```

lemma wf-eq-minimal: wf r = (∀ Q x. x ∈ Q --> (∃ z ∈ Q. ∀ y. (y, z) ∈ r -->
y ∉ Q))
apply auto
apply (erule wfE-min, assumption, blast)
apply (rule wfI-min, auto)
done

```

```

lemmas wfP-eq-minimal = wf-eq-minimal [to-pred]

```

Well-foundedness of transitive closure

```

lemma wf-trancl:
  assumes wf r
  shows wf (r+)
proof -
  {
    fix P and x
    assume induct-step: !!x. (!!y. (y, x) : r+ ==> P y) ==> P x
    have P x
    proof (rule induct-step)
      fix y assume (y, x) : r+
      with ⟨wf r⟩ show P y
    proof (induct x arbitrary: y)
      case (less x)
      note hyp = ⟨∧ x' y'. (x', x) : r ==> (y', x') : r+ ==> P y'⟩
      from ⟨(y, x) : r+⟩ show P y
    proof cases
      case base
      show P y
    proof (rule induct-step)
      fix y' assume (y', y) : r+
      with ⟨(y, x) : r⟩ show P y' by (rule hyp [of y y'])
    qed
  } next
  case step
  then obtain x' where (x', x) : r and (y, x') : r+ by simp
  then show P y by (rule hyp [of x' y])
qed
qed
qed
} then show ?thesis unfolding wf-def by blast
qed

```

```

lemmas wfP-trancl = wf-trancl [to-pred]

```



```

lemma wf-converse-trancl: wf (r-1) ==> wf ((r+)-1)
  apply (subst trancl-converse [symmetric])
  apply (erule wf-trancl)
  done

```

Well-foundedness of subsets

```

lemma wf-subset: [| wf(r); p<=r |] ==> wf(p)
  apply (simp (no-asm-use) add: wf-eq-minimal)
  apply fast
  done

```

```

lemmas wfP-subset = wf-subset [to-pred]

```

Well-foundedness of the empty relation

```

lemma wf-empty [iff]: wf {}
  by (simp add: wf-def)

```

```

lemma wfP-empty [iff]:
  wfP (λx y. False)
proof -
  have wfP bot by (fact wf-empty [to-pred bot-empty-eq2])
  then show ?thesis by (simp add: bot-fun-eq bot-bool-eq)
qed

```

```

lemma wf-Int1: wf r ==> wf (r Int r')
  apply (erule wf-subset)
  apply (rule Int-lower1)
  done

```

```

lemma wf-Int2: wf r ==> wf (r' Int r)
  apply (erule wf-subset)
  apply (rule Int-lower2)
  done

```

Exponentiation

```

lemma wf-exp:
  assumes wf (R^ n)
  shows wf R
proof (rule wfI-pf)
  fix A assume A ⊆ R “ A
  then have A ⊆ (R^ n) “ A by (induct n) force+
  with ⟨wf (R^ n)⟩
  show A = {} by (rule wfE-pf)
qed

```

Well-foundedness of insert

```

lemma wf-insert [iff]: wf(insert (y,x) r) = (wf(r) & (x,y) ~: r*)

```

```

apply (rule iffI)
  apply (blast elim: wf-trancl [THEN wf-irrefl]
    intro: rtrancl-into-trancl1 wf-subset
      rtrancl-mono [THEN [2] rev-subsetD])
apply (simp add: wf-eq-minimal, safe)
apply (rule allE, assumption, erule impE, blast)
apply (erule bexE)
apply (rename-tac a, case-tac a = x)
  prefer 2
apply blast
apply (case-tac y:Q)
  prefer 2 apply blast
apply (rule-tac x = {z. z:Q & (z,y) : r^*} in allE)
  apply assumption
apply (erule-tac V = ALL Q. (EX x. x : Q) --> ?P Q in thin-rl)
  — essential for speed

```

Blast with new substOccur fails

```

apply (fast intro: converse-rtrancl-into-rtrancl)
done

```

Well-foundedness of image

```

lemma wf-prod-fun-image: [| wf r; inj f |] ==> wf (prod-fun f f ' r)
apply (simp only: wf-eq-minimal, clarify)
apply (case-tac EX p. f p : Q)
apply (erule-tac x = {p. f p : Q} in allE)
apply (fast dest: inj-onD, blast)
done

```

### 24.3 Well-Foundedness Results for Unions

```

lemma wf-union-compatible:
  assumes wf R wf S
  assumes R O S ⊆ R
  shows wf (R ∪ S)
proof (rule wfI-min)
  fix x :: 'a and Q
  let ?Q' = {x ∈ Q. ∀ y. (y, x) ∈ R ⟶ y ∉ Q}
  assume x ∈ Q
  obtain a where a ∈ ?Q'
  by (rule wfE-min [OF ⟨wf R⟩ ⟨x ∈ Q⟩]) blast
  with ⟨wf S⟩
  obtain z where z ∈ ?Q' and zmin: ∧ y. (y, z) ∈ S ⟹ y ∉ ?Q' by (erule
wfE-min)
  {
    fix y assume (y, z) ∈ S
    then have y ∉ ?Q' by (rule zmin)

    have y ∉ Q
  }

```

```

proof
  assume  $y \in Q$ 
  with  $\langle y \notin ?Q' \rangle$ 
  obtain  $w$  where  $(w, y) \in R$  and  $w \in Q$  by auto
  from  $\langle (w, y) \in R \rangle \langle (y, z) \in S \rangle$  have  $(w, z) \in R \circ S$  by (rule rel-compI)
  with  $\langle R \circ S \subseteq R \rangle$  have  $(w, z) \in R$  ..
  with  $\langle z \in ?Q' \rangle$  have  $w \notin Q$  by blast
  with  $\langle w \in Q \rangle$  show False by contradiction
qed
}
with  $\langle z \in ?Q' \rangle$  show  $\exists z \in Q. \forall y. (y, z) \in R \cup S \longrightarrow y \notin Q$  by blast
qed

```

Well-foundedness of indexed union with disjoint domains and ranges

```

lemma wf-UN: [|  $\text{ALL } i:I. \text{wf}(r\ i);$   

    $\text{ALL } i:I. \text{ALL } j:I. r\ i \sim = r\ j \longrightarrow \text{Domain}(r\ i) \text{ Int } \text{Range}(r\ j) = \{\}$   

  |]  $\implies \text{wf}(\text{UN } i:I. r\ i)$ 
apply (simp only: wf-eq-minimal, clarify)
apply (rename-tac A a, case-tac EX i:I. EX a:A. EX b:A. (b,a) : r i)
prefer 2
apply force
apply clarify
apply (drule bspec, assumption)
apply (erule-tac x={a. a:A & (EX b:A. (b,a) : r i) } in allE)
apply (blast elim!: allE)
done

```

```

lemma wfP-SUP:
   $\forall i. \text{wfP}(r\ i) \implies \forall i\ j. r\ i \neq r\ j \longrightarrow \inf(\text{DomainP}(r\ i))(\text{RangeP}(r\ j)) = \text{bot}$ 
 $\implies \text{wfP}(\text{SUPR } \text{UNIV } r)$ 
  by (rule wf-UN [where I=UNIV and r= $\lambda i. \{(x, y). r\ i\ x\ y\}, \text{to-pred SUP-UN-eq2}]$ )
  (simp-all add: Collect-def)

```

```

lemma wf-Union:
  [|  $\text{ALL } r:R. \text{wf } r;$   

    $\text{ALL } r:R. \text{ALL } s:R. r \sim = s \longrightarrow \text{Domain } r \text{ Int } \text{Range } s = \{\}$   

  |]  $\implies \text{wf}(\text{Union } R)$ 
apply (simp add: Union-def)
apply (blast intro: wf-UN)
done

```

```

lemma wf-Un:
  [|  $\text{wf } r; \text{wf } s; \text{Domain } r \text{ Int } \text{Range } s = \{\}$  |]  $\implies \text{wf}(r \text{ Un } s)$ 
  using wf-union-compatible[of s r]
  by (auto simp: Un-ac)

```

```

lemma wf-union-merge:
   $\text{wf}(R \cup S) = \text{wf}(R \circ R \cup S \circ R \cup S)$  (is  $\text{wf } ?A = \text{wf } ?B$ )

```

```

proof
  assume  $wf\ ?A$ 
  with  $wf\text{-}trancl$  have  $wfT: wf\ (?A\ ^+)$  .
  moreover have  $?B \subseteq ?A\ ^+$ 
    by ( $subst\ trancl\text{-}unfold$ ,  $subst\ trancl\text{-}unfold$ ) blast
  ultimately show  $wf\ ?B$  by ( $rule\ wf\text{-}subset$ )
next
  assume  $wf\ ?B$ 

  show  $wf\ ?A$ 
  proof ( $rule\ wfI\text{-}min$ )
    fix  $Q :: 'a\ set$  and  $x$ 
    assume  $x \in Q$ 

    with  $\langle wf\ ?B \rangle$ 
    obtain  $z$  where  $z \in Q$  and  $\bigwedge y. (y, z) \in ?B \implies y \notin Q$ 
      by ( $erule\ wfE\text{-}min$ )
    then have  $A1: \bigwedge y. (y, z) \in R\ O\ R \implies y \notin Q$ 
      and  $A2: \bigwedge y. (y, z) \in S\ O\ R \implies y \notin Q$ 
      and  $A3: \bigwedge y. (y, z) \in S \implies y \notin Q$ 
      by auto

    show  $\exists z \in Q. \forall y. (y, z) \in ?A \longrightarrow y \notin Q$ 
    proof ( $cases\ \forall y. (y, z) \in R \longrightarrow y \notin Q$ )
      case True
        with  $\langle z \in Q \rangle\ A3$  show  $?thesis$  by blast
      next
        case False
          then obtain  $z'$  where  $z' \in Q\ (z', z) \in R$  by blast

          have  $\forall y. (y, z') \in ?A \longrightarrow y \notin Q$ 
          proof ( $intro\ allI\ impI$ )
            fix  $y$  assume  $(y, z') \in ?A$ 
            then show  $y \notin Q$ 
            proof
              assume  $(y, z') \in R$ 
              then have  $(y, z) \in R\ O\ R$  using  $\langle (z', z) \in R \rangle$  ..
              with  $A1$  show  $y \notin Q$  .
            next
              assume  $(y, z') \in S$ 
              then have  $(y, z) \in S\ O\ R$  using  $\langle (z', z) \in R \rangle$  ..
              with  $A2$  show  $y \notin Q$  .
            qed
          qed
          with  $\langle z' \in Q \rangle$  show  $?thesis$  ..
        qed
      qed
    qed
  qed

```

**lemma** *wf-comp-self*:  $wf\ R = wf\ (R\ O\ R)$  — special case  
**by** (*rule wf-union-merge* [**where**  $S = \{\}$ , *simplified*])

## 24.4 Acyclic relations

**definition** *acyclic* ::  $('a * 'a)\ set \Rightarrow bool$  **where**  
 $acyclic\ r == !x. (x, x) \sim: r^+ +$

**abbreviation** *acyclicP* ::  $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$  **where**  
 $acyclicP\ r == acyclic\ \{(x, y). r\ x\ y\}$

**lemma** *acyclicI*:  $ALL\ x. (x, x) \sim: r^+ + \Rightarrow acyclic\ r$   
**by** (*simp add: acyclic-def*)

**lemma** *wf-acyclic*:  $wf\ r \Rightarrow acyclic\ r$   
**apply** (*simp add: acyclic-def*)  
**apply** (*blast elim: wf-trancl [THEN wf-irrefl]*)  
**done**

**lemmas** *wfP-acyclicP* = *wf-acyclic* [*to-pred*]

**lemma** *acyclic-insert* [*iff*]:  
 $acyclic(insert\ (y, x)\ r) = (acyclic\ r \ \&\ (x, y) \sim: r^+ \cdot)$   
**apply** (*simp add: acyclic-def trancl-insert*)  
**apply** (*blast intro: rtrancl-trans*)  
**done**

**lemma** *acyclic-converse* [*iff*]:  $acyclic(r^+ - 1) = acyclic\ r$   
**by** (*simp add: acyclic-def trancl-converse*)

**lemmas** *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

**lemma** *acyclic-impl-antisym-rtrancl*:  $acyclic\ r \Rightarrow antisym(r^+ \cdot)$   
**apply** (*simp add: acyclic-def antisym-def*)  
**apply** (*blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl*)  
**done**

**lemma** *acyclic-subset*:  $[| acyclic\ s; r \leq s |] \Rightarrow acyclic\ r$   
**apply** (*simp add: acyclic-def*)  
**apply** (*blast intro: trancl-mono*)  
**done**

Wellfoundedness of finite acyclic relations

**lemma** *finite-acyclic-wf* [*rule-format*]:  $finite\ r \Rightarrow acyclic\ r \dashrightarrow wf\ r$   
**apply** (*erule finite-induct, blast*)  
**apply** (*simp (no-asm-simp) only: split-tupled-all*)  
**apply** *simp*

done

**lemma** *finite-acyclic-wf-converse*:  $[|finite\ r;\ acyclic\ r|] \implies wf\ (r^{\wedge}-1)$   
**apply** (*erule* *finite-converse* [*THEN* *iffD2*, *THEN* *finite-acyclic-wf*])  
**apply** (*erule* *acyclic-converse* [*THEN* *iffD2*])  
done

**lemma** *wf-iff-acyclic-if-finite*:  $finite\ r \implies wf\ r = acyclic\ r$   
**by** (*blast* *intro*: *finite-acyclic-wf* *wf-acyclic*)

## 24.5 nat is well-founded

**lemma** *less-nat-rel*:  $op\ < = (\lambda m\ n.\ n = Suc\ m)^{\wedge}++$   
**proof** (*rule* *ext*, *rule* *ext*, *rule* *iffI*)  
  **fix** *n m* :: *nat*  
  **assume**  $m < n$   
  **then show**  $(\lambda m\ n.\ n = Suc\ m)^{\wedge}++\ m\ n$   
  **proof** (*induct* *n*)  
    **case** 0 **then show** ?*case* **by** *auto*  
  **next**  
    **case** (*Suc* *n*) **then show** ?*case*  
      **by** (*auto* *simp* *add*: *less-Suc-eq-le* *le-less* *intro*: *tranclp.trancl-into-trancl*)  
  **qed**  
**next**  
  **fix** *n m* :: *nat*  
  **assume**  $(\lambda m\ n.\ n = Suc\ m)^{\wedge}++\ m\ n$   
  **then show**  $m < n$   
  **by** (*induct* *n*)  
    (*simp-all* *add*: *less-Suc-eq-le* *reflexive* *le-less*)  
**qed**

### definition

*pred-nat* :: (*nat* \* *nat*) *set* **where**  
*pred-nat* =  $\{(m, n).\ n = Suc\ m\}$

### definition

*less-than* :: (*nat* \* *nat*) *set* **where**  
*less-than* = *pred-nat*<sup>+</sup>

**lemma** *less-eq*:  $(m, n) \in pred\text{-}nat^{\wedge}+ \iff m < n$   
**unfolding** *less-nat-rel* *pred-nat-def* *trancl-def* **by** *simp*

**lemma** *pred-nat-trancl-eq-le*:  
 $(m, n) \in pred\text{-}nat^{\wedge}* \iff m \leq n$   
**unfolding** *less-eq* *rtrancl-eq-or-trancl* **by** *auto*

**lemma** *wf-pred-nat*: *wf* *pred-nat*  
**apply** (*unfold* *wf-def* *pred-nat-def*, *clarify*)  
**apply** (*induct-tac* *x*, *blast*+) )

**done**

**lemma** *wf-less-than* [iff]: *wf less-than*  
**by** (*simp add: less-than-def wf-pred-nat [THEN wf-trancl]*)

**lemma** *trans-less-than* [iff]: *trans less-than*  
**by** (*simp add: less-than-def*)

**lemma** *less-than-iff* [iff]:  $((x,y): \text{less-than}) = (x < y)$   
**by** (*simp add: less-than-def less-eq*)

**lemma** *wf-less*: *wf*  $\{(x, y::\text{nat}). x < y\}$   
**using** *wf-less-than* **by** (*simp add: less-than-def less-eq [symmetric]*)

## 24.6 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [?].

**inductive-set**  
*acc* ::  $('a * 'a) \text{ set} \Rightarrow 'a \text{ set}$   
**for** *r* ::  $('a * 'a) \text{ set}$   
**where**  
*accI*:  $(!!y. (y, x) : r \Rightarrow y : \text{acc } r) \Rightarrow x : \text{acc } r$

**abbreviation**  
*termip* ::  $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool}$  **where**  
*termip* *r* == *accp*  $(r^{-1-1})$

**abbreviation**  
*termi* ::  $('a * 'a) \text{ set} \Rightarrow 'a \text{ set}$  **where**  
*termi* *r* == *acc*  $(r^{-1})$

**lemmas** *accpI* = *accp.accI*

Induction rules

**theorem** *accp-induct*:  
**assumes** *major*: *accp r a*  
**assumes** *hyp*:  $!!x. \text{accp } r \ x \Rightarrow \forall y. r \ y \ x \longrightarrow P \ y \Rightarrow P \ x$   
**shows** *P a*  
**apply** (*rule major [THEN accp.induct]*)  
**apply** (*rule hyp*)  
**apply** (*rule accp.accI*)  
**apply** *fast*  
**apply** *fast*  
**done**

**theorems** *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set: accp*]

**theorem** *accp-downward*: *accp r b*  $\Rightarrow r \ a \ b \Rightarrow \text{accp } r \ a$   
**apply** (*erule accp.cases*)

```

apply fast
done

lemma not-accp-down:
  assumes na:  $\neg \text{accp } R \ x$ 
  obtains z where  $R \ z \ x$  and  $\neg \text{accp } R \ z$ 
proof –
  assume a:  $\bigwedge z. \llbracket R \ z \ x; \neg \text{accp } R \ z \rrbracket \implies \text{thesis}$ 

  show thesis
  proof (cases  $\forall z. R \ z \ x \longrightarrow \text{accp } R \ z$ )
    case True
    hence  $\bigwedge z. R \ z \ x \implies \text{accp } R \ z$  by auto
    hence  $\text{accp } R \ x$ 
    by (rule accp.accI)
    with na show thesis ..
  next
    case False then obtain z where  $R \ z \ x$  and  $\neg \text{accp } R \ z$ 
    by auto
    with a show thesis .
  qed
qed

lemma accp-downwards-aux:  $r^{**} \ b \ a \implies \text{accp } r \ a \dashrightarrow \text{accp } r \ b$ 
  apply (erule rtranclp-induct)
  apply blast
  apply (blast dest: accp-downward)
  done

theorem accp-downwards:  $\text{accp } r \ a \implies r^{**} \ b \ a \implies \text{accp } r \ b$ 
  apply (blast dest: accp-downwards-aux)
  done

theorem accp-wfPI:  $\forall x. \text{accp } r \ x \implies \text{wfP } r$ 
  apply (rule wfPUNIVI)
  apply (induct-tac P x rule: accp-induct)
  apply blast
  apply blast
  done

theorem accp-wfPD:  $\text{wfP } r \implies \text{accp } r \ x$ 
  apply (erule wfP-induct-rule)
  apply (rule accp.accI)
  apply blast
  done

theorem wfP-accp-iff:  $\text{wfP } r = (\forall x. \text{accp } r \ x)$ 
  apply (blast intro: accp-wfPI dest: accp-wfPD)
  done

```



Smaller relations have bigger accessible parts:

```

lemma accp-subset:
  assumes sub:  $R1 \leq R2$ 
  shows  $\text{accp } R2 \leq \text{accp } R1$ 
proof (rule predicate1I)
  fix x assume  $\text{accp } R2 \ x$ 
  then show  $\text{accp } R1 \ x$ 
  proof (induct x)
    fix x
    assume ih:  $\bigwedge y. R2 \ y \ x \implies \text{accp } R1 \ y$ 
    with sub show  $\text{accp } R1 \ x$ 
    by (blast intro: accp.accI)
  qed
qed

```

This is a generalized induction theorem that works on subsets of the accessible part.

```

lemma accp-subset-induct:
  assumes subset:  $D \leq \text{accp } R$ 
  and dcl:  $\bigwedge x \ z. \llbracket D \ x; R \ z \ x \rrbracket \implies D \ z$ 
  and D x
  and istep:  $\bigwedge x. \llbracket D \ x; (\bigwedge z. R \ z \ x \implies P \ z) \rrbracket \implies P \ x$ 
  shows  $P \ x$ 
proof –
  from subset and  $\langle D \ x \rangle$ 
  have  $\text{accp } R \ x \ ..$ 
  then show  $P \ x$  using  $\langle D \ x \rangle$ 
  proof (induct x)
    fix x
    assume D x
    and  $\bigwedge y. R \ y \ x \implies D \ y \implies P \ y$ 
    with dcl and istep show  $P \ x$  by blast
  qed
qed

```

Set versions of the above theorems

```

lemmas acc-induct = accp-induct [to-set]

```

```

lemmas acc-induct-rule = acc-induct [rule-format, induct set: acc]

```

```

lemmas acc-downward = accp-downward [to-set]

```

```

lemmas not-acc-down = not-accp-down [to-set]

```

```

lemmas acc-downwards-aux = accp-downwards-aux [to-set]

```

```

lemmas acc-downwards = accp-downwards [to-set]

```

**lemmas**  $acc\text{-}wfI = accp\text{-}wfPI$  [*to-set*]

**lemmas**  $acc\text{-}wfD = accp\text{-}wfPD$  [*to-set*]

**lemmas**  $wf\text{-}acc\text{-}iff = wfP\text{-}accp\text{-}iff$  [*to-set*]

**lemmas**  $acc\text{-}subset = accp\text{-}subset$  [*to-set pred-subset-eq*]

**lemmas**  $acc\text{-}subset\text{-}induct = accp\text{-}subset\text{-}induct$  [*to-set pred-subset-eq*]

## 24.7 Tools for building wellfounded relations

Inverse Image

**lemma**  $wf\text{-}inv\text{-}image$  [*simp,intro!*]:  $wf(r) ==> wf(inv\text{-}image\ r\ (f::'a=>'b))$   
**apply** (*simp (no-asm-use) add: inv-image-def wf-eq-minimal*)  
**apply** *clarify*  
**apply** (*subgoal-tac EX (w::'b) . w : {w. EX (x::'a) . x: Q & (f x = w) }*)  
**prefer** 2 **apply** (*blast del: allE*)  
**apply** (*erule allE*)  
**apply** (*erule (1) notE impE*)  
**apply** *blast*  
**done**

Measure functions into *nat*

**definition**  $measure :: ('a ==> nat) ==> ('a * 'a) set$   
**where**  $measure == inv\text{-}image\ less\text{-}than$

**lemma**  $in\text{-}measure$  [*simp*]:  $((x,y) : measure\ f) = (f\ x < f\ y)$   
**by** (*simp add:measure-def*)

**lemma**  $wf\text{-}measure$  [*iff*]:  $wf\ (measure\ f)$   
**apply** (*unfold measure-def*)  
**apply** (*rule wf-less-than [THEN wf-inv-image]*)  
**done**

Lexicographic combinations

**definition**  
 $lex\text{-}prod :: [( 'a * 'a) set, ('b * 'b) set] ==> (('a * 'b) * ('a * 'b)) set$   
 $(\mathbf{infixr} < * lex * > 80)$

**where**

$ra < * lex * > rb == \{((a,b),(a',b')). (a,a') : ra \mid a=a' \ \& \ (b,b') : rb\}$

**lemma**  $wf\text{-}lex\text{-}prod$  [*intro!*]:  $[| wf(ra); wf(rb) |] ==> wf(ra < * lex * > rb)$   
**apply** (*unfold wf-def lex-prod-def*)  
**apply** (*rule allI, rule impI*)  
**apply** (*simp (no-asm-use) only: split-paired-All*)  
**apply** (*drule spec, erule mp*)  
**apply** (*rule allI, rule impI*)

**apply** (*drule spec, erule mp, blast*)  
**done**

**lemma** *in-lex-prod*[*simp*]:  
 $((a,b),(a',b')): r <_{*lex*} s = ((a,a'): r \vee (a = a' \wedge (b, b') : s))$   
**by** (*auto simp:lex-prod-def*)

*op*  $<_{*lex*}$  preserves transitivity

**lemma** *trans-lex-prod* [*intro!*]:  
 $[trans\ R1; trans\ R2] \implies trans\ (R1 <_{*lex*} R2)$   
**by** (*unfold trans-def lex-prod-def, blast*)

lexicographic combinations with measure functions

**definition**

*mlex-prod* ::  $('a \Rightarrow nat) \Rightarrow ('a \times 'a) set \Rightarrow ('a \times 'a) set$  (**infixr**  $<_{*mlex*}$  80)  
**where**  
 $f <_{*mlex*} R = inv-image\ (less-than\ <_{*lex*}\ R)\ (\%x. (f\ x, x))$

**lemma** *wf-mlex*:  $wf\ R \implies wf\ (f <_{*mlex*} R)$   
**unfolding** *mlex-prod-def*  
**by** *auto*

**lemma** *mlex-less*:  $f\ x < f\ y \implies (x, y) \in f <_{*mlex*} R$   
**unfolding** *mlex-prod-def* **by** *simp*

**lemma** *mlex-leq*:  $f\ x \leq f\ y \implies (x, y) \in R \implies (x, y) \in f <_{*mlex*} R$   
**unfolding** *mlex-prod-def* **by** *auto*

proper subset relation on finite sets

**definition** *finite-psubset* ::  $('a\ set * 'a\ set)\ set$   
**where** *finite-psubset* ==  $\{(A,B). A < B \ \& \ finite\ B\}$

**lemma** *wf-finite-psubset*[*simp*]:  $wf(finite-psubset)$   
**apply** (*unfold finite-psubset-def*)  
**apply** (*rule wf-measure [THEN wf-subset]*)  
**apply** (*simp add: measure-def inv-image-def less-than-def less-eq*)  
**apply** (*fast elim!: psubset-card-mono*)  
**done**

**lemma** *trans-finite-psubset*:  $trans\ finite-psubset$   
**by** (*simp add: finite-psubset-def less-le trans-def, blast*)

**lemma** *in-finite-psubset*[*simp*]:  $(A, B) \in finite-psubset = (A < B \ \& \ finite\ B)$   
**unfolding** *finite-psubset-def* **by** *auto*

max- and min-extension of order to finite sets

**inductive-set** *max-ext* ::  $('a \times 'a) set \Rightarrow ('a\ set \times 'a\ set) set$   
**for** *R* ::  $('a \times 'a) set$

where

$max-extI[intro]: finite\ X \implies finite\ Y \implies Y \neq \{\} \implies (\bigwedge x. x \in X \implies \exists y \in Y. (x, y) \in R) \implies (X, Y) \in max-ext\ R$

**lemma** *max-ext-wf*:

**assumes** *wf*:  $wf\ r$

**shows**  $wf\ (max-ext\ r)$

**proof** (*rule acc-wfI, intro allI*)

**fix** *M* **show**  $M \in acc\ (max-ext\ r)\ (is - \in ?W)$

**proof** *cases*

**assume** *finite M*

**thus** *?thesis*

**proof** (*induct M*)

**show**  $\{\} \in ?W$

**by** (*rule accI*) (*auto elim: max-ext.cases*)

**next**

**fix** *M a* **assume**  $M \in ?W\ finite\ M$

**with** *wf* **show**  $insert\ a\ M \in ?W$

**proof** (*induct arbitrary: M*)

**fix** *M a*

**assume**  $M \in ?W$  **and** [*intro*]: *finite M*

**assume** *hyp*:  $\bigwedge b\ M. (b, a) \in r \implies M \in ?W \implies finite\ M \implies insert\ b\ M$

$\in ?W$

{

**fix** *N M* :: '*a* set

**assume** *finite N finite M*

**then**

**have**  $\llbracket M \in ?W ; (\bigwedge y. y \in N \implies (y, a) \in r) \rrbracket \implies N \cup M \in ?W$

**by** (*induct N arbitrary: M*) (*auto simp: hyp*)

}

**note** *add-less = this*

**show**  $insert\ a\ M \in ?W$

**proof** (*rule accI*)

**fix** *N* **assume** *Nless*:  $(N, insert\ a\ M) \in max-ext\ r$

**hence** *asm1*:  $\bigwedge x. x \in N \implies (x, a) \in r \vee (\exists y \in M. (x, y) \in r)$

**by** (*auto elim!: max-ext.cases*)

**let**  $?N1 = \{ n \in N. (n, a) \in r \}$

**let**  $?N2 = \{ n \in N. (n, a) \notin r \}$

**have**  $N: ?N1 \cup ?N2 = N$  **by** (*rule set-ext*) *auto*

**from** *Nless* **have** *finite N* **by** (*auto elim: max-ext.cases*)

**then** **have** *finites*: *finite ?N1 finite ?N2* **by** *auto*

**have**  $?N2 \in ?W$

**proof** *cases*

**assume** [*simp*]:  $M = \{\}$

**have** *Mw*:  $\{\} \in ?W$  **by** (*rule accI*) (*auto elim: max-ext.cases*)

```

      from asm1 have ?N2 = {} by auto
      with Mw show ?N2 ∈ ?W by (simp only:)
    next
      assume M ≠ {}
      have N2: (?N2, M) ∈ max-ext r
        by (rule max-extI[OF - - ⟨M ≠ {}⟩]) (insert asm1, auto intro: finites)

      with ⟨M ∈ ?W⟩ show ?N2 ∈ ?W by (rule acc-downward)
    qed
  with finites have ?N1 ∪ ?N2 ∈ ?W
    by (rule add-less) simp
  then show N ∈ ?W by (simp only: N)
qed
qed
qed
next
  assume [simp]: ¬ finite M
  show ?thesis
    by (rule accI) (auto elim: max-ext.cases)
qed
qed

```

**lemma** *max-ext-additive*:  
 $(A, B) \in \text{max-ext } R \implies (C, D) \in \text{max-ext } R \implies$   
 $(A \cup C, B \cup D) \in \text{max-ext } R$   
 by (*force elim*!: *max-ext.cases*)

**definition**

*min-ext* :: ('*a* × '*a*) set ⇒ ('*a* set × '*a* set) set  
**where**  
 [*code del*]: *min-ext* *r* = {(*X*, *Y*) | *X* *Y*. *X* ≠ {} ∧ (∀ *y* ∈ *Y*. (∃ *x* ∈ *X*. (*x*, *y*) ∈ *r*))}

**lemma** *min-ext-wf*:

```

  assumes wf r
  shows wf (min-ext r)
proof (rule wfI-min)
  fix Q :: 'a set set
  fix x
  assume nonempty: x ∈ Q
  show ∃ m ∈ Q. (∀ n. (n, m) ∈ min-ext r ⟶ n ∉ Q)
proof cases
  assume Q = {{}} thus ?thesis by (simp add: min-ext-def)
next
  assume Q ≠ {{}}
  with nonempty
  obtain e x where x ∈ Q e ∈ x by force
  then have eU: e ∈ ⋃ Q by auto

```

```

with  $\langle wf\ r \rangle$ 
obtain  $z$  where  $z: z \in \bigcup Q \wedge y. (y, z) \in r \implies y \notin \bigcup Q$ 
  by  $(erule\ wfE\text{-}min)$ 
from  $z$  obtain  $m$  where  $m \in Q\ z \in m$  by  $auto$ 
from  $\langle m \in Q \rangle$ 
show  $?thesis$ 
proof  $(rule, intro\ beqI\ allI\ impI)$ 
  fix  $n$ 
  assume  $smaller: (n, m) \in min\text{-}ext\ r$ 
  with  $\langle z \in m \rangle$  obtain  $y$  where  $y: y \in n\ (y, z) \in r$  by  $(auto\ simp: min\text{-}ext\text{-}def)$ 
  then show  $n \notin Q$  using  $z(2)$  by  $auto$ 
qed
qed
qed

```

## 24.8 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

**lemma** *sequence-trans*:  $[[\ ALL\ i. (f\ (Suc\ i), f\ i) : r^* ] ] \implies (f\ (i+k), f\ i) : r^*$   
**by**  $(induct\ k)\ (auto\ intro: rtrancl\text{-}trans)$

**lemma** *wf-weak-decr-stable*:

**assumes**  $as: \ALL\ i. (f\ (Suc\ i), f\ i) : r^* \wedge wf\ (r^+)$

**shows**  $EX\ i. \ALL\ k. f\ (i+k) = f\ i$

**proof** –

**have**  $lem: !!x. [[\ ALL\ i. (f\ (Suc\ i), f\ i) : r^*; wf\ (r^+) ] ]$

$\implies \ALL\ m. f\ m = x \dashrightarrow (EX\ i. \ALL\ k. f\ (m+i+k) = f\ (m+i))$

**apply**  $(erule\ wf\text{-}induct, clarify)$

**apply**  $(case\text{-}tac\ EX\ j. (f\ (m+j), f\ m) : r^+)$

**apply**  $clarify$

**apply**  $(subgoal\text{-}tac\ EX\ i. \ALL\ k. f\ ((m+j) + i + k) = f\ ((m+j) + i))$

**apply**  $clarify$

**apply**  $(rule\text{-}tac\ x = j + i\ \mathbf{in}\ exI)$

**apply**  $(simp\ add: add\text{-}ac, blast)$

**apply**  $(rule\text{-}tac\ x = 0\ \mathbf{in}\ exI, clarsimp)$

**apply**  $(drule\text{-}tac\ i = m\ \mathbf{and}\ k = k\ \mathbf{in}\ sequence\text{-}trans)$

**apply**  $(blast\ elim: rtranclE\ dest: rtrancl\text{-}into\text{-}trancl1)$

**done**

**from**  $lem[OF\ as, THEN\ spec, of\ 0, simplified]$

**show**  $?thesis$  **by**  $auto$

**qed**

**lemma** *weak-decr-stable*:

$\ALL\ i. f\ (Suc\ i) \leq ((f\ i)::nat) \implies EX\ i. \ALL\ k. f\ (i+k) = f\ i$

**apply**  $(rule\text{-}tac\ r = pred\text{-}nat\ \mathbf{in}\ wf\text{-}weak\text{-}decr\text{-}stable)$

**apply**  $(simp\ add: pred\text{-}nat\text{-}trancl\text{-}eq\text{-}le)$

```

apply (intro wf-trancl wf-pred-nat)
done

```

## 24.9 size of a datatype value

```

use Tools/Function/size.ML

```

```

setup Size.setup

```

```

lemma size-bool [code]:
  size (b::bool) = 0 by (cases b) auto

```

```

lemma nat-size [simp, code]: size (n::nat) = n
  by (induct n) simp-all

```

```

declare prod.size [no-atp]

```

```

lemma [code]:
  size (P :: 'a Predicate.pred) = 0 by (cases P) simp

```

```

lemma [code]:
  pred-size f P = 0 by (cases P) simp

```

```

end

```

## 25 FunDef: Function Definitions and Termination Proofs

```

theory FunDef
imports Wellfounded
uses
  Tools/prop-logic.ML
  Tools/sat-solver.ML
  (Tools/Function/function-lib.ML)
  (Tools/Function/function-common.ML)
  (Tools/Function/context-tree.ML)
  (Tools/Function/function-core.ML)
  (Tools/Function/sum-tree.ML)
  (Tools/Function/mutual.ML)
  (Tools/Function/pattern-split.ML)
  (Tools/Function/function.ML)
  (Tools/Function/relation.ML)
  (Tools/Function/measure-functions.ML)
  (Tools/Function/lexicographic-order.ML)
  (Tools/Function/pat-completeness.ML)
  (Tools/Function/fun.ML)
  (Tools/Function/induction-schema.ML)

```

```

(Tools/Function/termination.ML)
(Tools/Function/scnp-solve.ML)
(Tools/Function/scnp-reconstruct.ML)
begin

```

### 25.1 Definitions with default value.

#### definition

```

THE-default :: 'a ⇒ ('a ⇒ bool) ⇒ 'a where
THE-default d P = (if (∃!x. P x) then (THE x. P x) else d)

```

**lemma** *THE-defaultI'*:  $\exists!x. P x \implies P (THE\text{-}default\ d\ P)$   
**by** (*simp add: theI' THE-default-def*)

**lemma** *THE-default1-equality*:

```

[[∃!x. P x; P a]] ⇒ THE-default d P = a
by (simp add: the1-equality THE-default-def)

```

**lemma** *THE-default-none*:

```

¬(∃!x. P x) ⇒ THE-default d P = d
by (simp add: THE-default-def)

```

**lemma** *fundef-ex1-existence*:

```

assumes f-def: f == (λx::'a. THE-default (d x) (λy. G x y))
assumes ex1: ∃!y. G x y
shows G x (f x)
apply (simp only: f-def)
apply (rule THE-defaultI')
apply (rule ex1)
done

```

**lemma** *fundef-ex1-uniqueness*:

```

assumes f-def: f == (λx::'a. THE-default (d x) (λy. G x y))
assumes ex1: ∃!y. G x y
assumes elm: G x (h x)
shows h x = f x
apply (simp only: f-def)
apply (rule THE-default1-equality [symmetric])
apply (rule ex1)
apply (rule elm)
done

```

**lemma** *fundef-ex1-iff*:

```

assumes f-def: f == (λx::'a. THE-default (d x) (λy. G x y))
assumes ex1: ∃!y. G x y
shows (G x y) = (f x = y)
apply (auto simp: ex1 f-def THE-default1-equality)
apply (rule THE-defaultI')

```



```

apply (rule ex1)
done

lemma fundef-default-value:
  assumes f-def:  $f == (\lambda x::'a. \text{THE-default } (d \ x) (\lambda y. G \ x \ y))$ 
  assumes graph:  $\bigwedge x \ y. G \ x \ y \implies D \ x$ 
  assumes  $\neg D \ x$ 
  shows  $f \ x = d \ x$ 
proof -
  have  $\neg(\exists y. G \ x \ y)$ 
  proof
    assume  $\exists y. G \ x \ y$ 
    hence  $D \ x$  using graph ..
    with  $\neg D \ x$  show False ..
  qed
  hence  $\neg(\exists! y. G \ x \ y)$  by blast

thus ?thesis
  unfolding f-def
  by (rule THE-default-none)
qed

definition in-rel-def[simp]:
  in-rel  $R \ x \ y == (x, y) \in R$ 

lemma wf-in-rel:
   $wf \ R \implies wfP \ (in-rel \ R)$ 
  by (simp add: wfP-def)

use Tools/Function/function-lib.ML
use Tools/Function/function-common.ML
use Tools/Function/context-tree.ML
use Tools/Function/function-core.ML
use Tools/Function/sum-tree.ML
use Tools/Function/mutual.ML
use Tools/Function/pattern-split.ML
use Tools/Function/relation.ML
use Tools/Function/function.ML
use Tools/Function/pat-completeness.ML
use Tools/Function/fun.ML
use Tools/Function/induction-schema.ML

setup ⟨⟨
  Function.setup
  #> Pat-Completeness.setup
  #> Function-Fun.setup
  #> Induction-Schema.setup
  ⟩⟩

```

## 25.2 Measure Functions

**inductive** *is-measure* :: ('a  $\Rightarrow$  nat)  $\Rightarrow$  bool  
**where** *is-measure-trivial*: *is-measure* f

**use** *Tools/Function/measure-functions.ML*  
**setup** *MeasureFunctions.setup*

**lemma** *measure-size*[*measure-function*]: *is-measure* size  
**by** (rule *is-measure-trivial*)

**lemma** *measure-fst*[*measure-function*]: *is-measure* f  $\Longrightarrow$  *is-measure* ( $\lambda p. f (fst p)$ )  
**by** (rule *is-measure-trivial*)

**lemma** *measure-snd*[*measure-function*]: *is-measure* f  $\Longrightarrow$  *is-measure* ( $\lambda p. f (snd p)$ )  
**by** (rule *is-measure-trivial*)

**use** *Tools/Function/lexicographic-order.ML*  
**setup** *Lexicographic-Order.setup*

## 25.3 Congruence Rules

**lemma** *let-cong* [*fundef-cong*]:  
 $M = N \Longrightarrow (\bigwedge x. x = N \Longrightarrow f x = g x) \Longrightarrow Let\ M\ f = Let\ N\ g$   
**unfolding** *Let-def* **by** *blast*

**lemmas** [*fundef-cong*] =  
*if-cong image-cong INT-cong UN-cong*  
*bex-cong ball-cong imp-cong*

**lemma** *split-cong* [*fundef-cong*]:  
 $(\bigwedge x\ y. (x, y) = q \Longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow p = q$   
 $\Longrightarrow split\ f\ p = split\ g\ q$   
**by** (*auto simp: split-def*)

**lemma** *comp-cong* [*fundef-cong*]:  
 $f\ (g\ x) = f'\ (g'\ x') \Longrightarrow (f\ o\ g)\ x = (f'\ o\ g')\ x'$   
**unfolding** *o-apply* .

## 25.4 Simp rules for termination proofs

**lemma** *termination-basic-simps*[*termination-simp*]:  
 $x < (y::nat) \Longrightarrow x < y + z$   
 $x < z \Longrightarrow x < y + z$   
 $x \leq y \Longrightarrow x \leq y + (z::nat)$   
 $x \leq z \Longrightarrow x \leq y + (z::nat)$   
 $x < y \Longrightarrow x \leq (y::nat)$   
**by** *arith+*

**declare** *le-imp-less-Suc*[*termination-simp*]

**lemma** *prod-size-simp*[*termination-simp*]:  
 $\text{prod-size } f \ g \ p = f \ (\text{fst } p) + g \ (\text{snd } p) + \text{Suc } 0$   
**by** (*induct p*) *auto*

## 25.5 Decomposition

**lemma** *less-by-empty*:  
 $A = \{\} \implies A \subseteq B$   
**and** *union-comp-emptyL*:  
 $\llbracket A \ O \ C = \{\}; B \ O \ C = \{\} \rrbracket \implies (A \cup B) \ O \ C = \{\}$   
**and** *union-comp-emptyR*:  
 $\llbracket A \ O \ B = \{\}; A \ O \ C = \{\} \rrbracket \implies A \ O \ (B \cup C) = \{\}$   
**and** *wf-no-loop*:  
 $R \ O \ R = \{\} \implies \text{wf } R$   
**by** (*auto simp add: wf-comp-self*[*of R*])

## 25.6 Reduction Pairs

**definition**  
 $\text{reduction-pair } P = (\text{wf } (\text{fst } P) \wedge \text{fst } P \ O \ \text{snd } P \subseteq \text{fst } P)$

**lemma** *reduction-pairI*[*intro*]:  $\text{wf } R \implies R \ O \ S \subseteq R \implies \text{reduction-pair } (R, S)$   
**unfolding** *reduction-pair-def* **by** *auto*

**lemma** *reduction-pair-lemma*:  
**assumes** *rp*: *reduction-pair P*  
**assumes**  $R \subseteq \text{fst } P$   
**assumes**  $S \subseteq \text{snd } P$   
**assumes** *wf S*  
**shows**  $\text{wf } (R \cup S)$   
**proof** –  
**from** *rp*  $\langle S \subseteq \text{snd } P \rangle$  **have**  $\text{wf } (\text{fst } P) \ \text{fst } P \ O \ S \subseteq \text{fst } P$   
**unfolding** *reduction-pair-def* **by** *auto*  
**with**  $\langle \text{wf } S \rangle$  **have**  $\text{wf } (\text{fst } P \cup S)$   
**by** (*auto intro: wf-union-compatible*)  
**moreover from**  $\langle R \subseteq \text{fst } P \rangle$  **have**  $R \cup S \subseteq \text{fst } P \cup S$  **by** *auto*  
**ultimately show** *?thesis* **by** (*rule wf-subset*)  
**qed**

**definition**  
 $\text{rp-inv-image} = (\lambda(R, S) \ f. (\text{inv-image } R \ f, \text{inv-image } S \ f))$

**lemma** *rp-inv-image-rp*:  
 $\text{reduction-pair } P \implies \text{reduction-pair } (\text{rp-inv-image } P \ f)$   
**unfolding** *reduction-pair-def rp-inv-image-def split-def*  
**by** *force*

## 25.7 Concrete orders for SCNP termination proofs

**definition** *pair-less* = *less-than*  $\langle *lex* \rangle$  *less-than*

**definition** [code del]: *pair-leq* = *pair-less*  $\hat{=}$

**definition** *max-strict* = *max-ext* *pair-less*

**definition** [code del]: *max-weak* = *max-ext* *pair-leq*  $\cup \{(\{\}, \{\})\}$

**definition** [code del]: *min-strict* = *min-ext* *pair-less*

**definition** [code del]: *min-weak* = *min-ext* *pair-leq*  $\cup \{(\{\}, \{\})\}$

**lemma** *wf-pair-less*[simp]: *wf* *pair-less*

by (*auto* simp: *pair-less-def*)

Introduction rules for *pair-less*/*pair-leq*

**lemma** *pair-leqI1*:  $a < b \implies ((a, s), (b, t)) \in \text{pair-leq}$

and *pair-leqI2*:  $a \leq b \implies s \leq t \implies ((a, s), (b, t)) \in \text{pair-leq}$

and *pair-lessI1*:  $a < b \implies ((a, s), (b, t)) \in \text{pair-less}$

and *pair-lessI2*:  $a \leq b \implies s < t \implies ((a, s), (b, t)) \in \text{pair-less}$

**unfolding** *pair-leq-def* *pair-less-def* by *auto*

Introduction rules for *max*

**lemma** *smax-emptyI*:

*finite*  $Y \implies Y \neq \{\} \implies (\{\}, Y) \in \text{max-strict}$

and *smax-insertI*:

$\llbracket y \in Y; (x, y) \in \text{pair-less}; (X, Y) \in \text{max-strict} \rrbracket \implies (\text{insert } x \ X, Y) \in \text{max-strict}$

and *wmax-emptyI*:

*finite*  $X \implies (\{\}, X) \in \text{max-weak}$

and *wmax-insertI*:

$\llbracket y \in YS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{max-weak} \rrbracket \implies (\text{insert } x \ XS, YS) \in \text{max-weak}$

**unfolding** *max-strict-def* *max-weak-def* by (*auto* elim!: *max-ext.cases*)

Introduction rules for *min*

**lemma** *smin-emptyI*:

$X \neq \{\} \implies (X, \{\}) \in \text{min-strict}$

and *smin-insertI*:

$\llbracket x \in XS; (x, y) \in \text{pair-less}; (XS, YS) \in \text{min-strict} \rrbracket \implies (XS, \text{insert } y \ YS) \in \text{min-strict}$

and *wmin-emptyI*:

$(X, \{\}) \in \text{min-weak}$

and *wmin-insertI*:

$\llbracket x \in XS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{min-weak} \rrbracket \implies (XS, \text{insert } y \ YS) \in \text{min-weak}$

by (*auto* simp: *min-strict-def* *min-weak-def* *min-ext-def*)

Reduction Pairs

**lemma** *max-ext-compat*:

assumes  $R \ O \ S \subseteq R$

shows  $\text{max-ext } R \ O \ (\text{max-ext } S \cup \{(\{\}, \{\})\}) \subseteq \text{max-ext } R$

using *assms*

```

apply auto
apply (elim max-ext.cases)
apply rule
apply auto[3]
apply (drule-tac x=xa in meta-spec)
apply simp
apply (erule bexE)
apply (drule-tac x=xb in meta-spec)
by auto

lemma max-rpair-set: reduction-pair (max-strict, max-weak)
  unfolding max-strict-def max-weak-def
apply (intro reduction-pairI max-ext-wf)
apply simp
apply (rule max-ext-compat)
by (auto simp: pair-less-def pair-leq-def)

lemma min-ext-compat:
  assumes  $R \ O \ S \subseteq R$ 
  shows  $\text{min-ext } R \ O \ (\text{min-ext } S \cup \{\{\}, \{\}\}) \subseteq \text{min-ext } R$ 
using assms
apply (auto simp: min-ext-def)
apply (drule-tac x=ya in bspec, assumption)
apply (erule bexE)
apply (drule-tac x=xc in bspec)
apply assumption
by auto

lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
apply (intro reduction-pairI min-ext-wf)
apply simp
apply (rule min-ext-compat)
by (auto simp: pair-less-def pair-leq-def)

```

## 25.8 Tool setup

```

use Tools/Function/termination.ML
use Tools/Function/scnp-solve.ML
use Tools/Function/scnp-reconstruct.ML

setup  $\ll \text{ScnpReconstruct.setup} \gg$ 

ML-val — setup inactive
 $\ll$ 
  Context.theory-map (Function.Common.set-termination-prover
    (ScnpReconstruct.decomp-scnp-tac [ScnpSolve.MAX, ScnpSolve.MIN, Scnp-
      Solve.MS]))
 $\gg$ 

```

end

## 26 Extraction: Program extraction for HOL

```
theory Extraction
imports Option
uses Tools/rewrite-hol-proof.ML
begin
```

### 26.1 Setup

```
setup <<
  Extraction.add-types
    [(bool, ([], NONE))] #>
  Extraction.set-preprocessor (fn thy =>
    Proofterm.rewrite-proof-notypes
      ([], RewriteHOLProof.elim-cong :: ProofRewriteRules.rprocs true) o
    Proofterm.rewrite-proof thy
      (RewriteHOLProof.rews,
        ProofRewriteRules.rprocs true @ [ProofRewriteRules.expand-of-class thy]) o
    ProofRewriteRules.elim-vars (curry Const @ {const-name default}))
  >>
```

```
lemmas [extraction-expand] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
  induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
  induct-atomize induct-atomize' induct-rulify induct-rulify'
  induct-rulify-fallback induct-trueI
  True-implies-equals TrueE
```

```
lemmas [extraction-expand-def] =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-true-def induct-false-def
```

```
datatype sumbool = Left | Right
```

### 26.2 Type of extracted program

```
extract-type
  typeof (Trueprop P)  $\equiv$  typeof P

  typeof P  $\equiv$  Type (TYPE(Null))  $\implies$  typeof Q  $\equiv$  Type (TYPE('Q))  $\implies$ 
    typeof (P  $\longrightarrow$  Q)  $\equiv$  Type (TYPE('Q))

  typeof Q  $\equiv$  Type (TYPE(Null))  $\implies$  typeof (P  $\longrightarrow$  Q)  $\equiv$  Type (TYPE(Null))
```

$$\begin{aligned}
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\
& \quad \text{typeof } (P \longrightarrow Q) \equiv \text{Type } (\text{TYPE}('P \Rightarrow 'Q)) \\
& (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& \quad \text{typeof } (\forall x. P x) \equiv \text{Type } (\text{TYPE}(\text{Null})) \\
& (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}('P))) \implies \\
& \quad \text{typeof } (\forall x::'a. P x) \equiv \text{Type } (\text{TYPE}('a \Rightarrow 'P)) \\
& (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& \quad \text{typeof } (\exists x::'a. P x) \equiv \text{Type } (\text{TYPE}('a)) \\
& (\lambda x. \text{typeof } (P x)) \equiv (\lambda x. \text{Type } (\text{TYPE}('P))) \implies \\
& \quad \text{typeof } (\exists x::'a. P x) \equiv \text{Type } (\text{TYPE}('a \times 'P)) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \quad \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}(\text{sumbool})) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\
& \quad \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('Q \text{ option})) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \quad \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('P \text{ option})) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\
& \quad \text{typeof } (P \vee Q) \equiv \text{Type } (\text{TYPE}('P + 'Q)) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\
& \quad \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('Q)) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \quad \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P)) \\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\
& \quad \text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P \times 'Q)) \\
& \text{typeof } (P = Q) \equiv \text{typeof } ((P \longrightarrow Q) \wedge (Q \longrightarrow P)) \\
& \text{typeof } (x \in P) \equiv \text{typeof } P
\end{aligned}$$

### 26.3 Realizability

#### realizability

$$\begin{aligned}
& (\text{realizes } t \text{ (Trueprop } P)) \equiv (\text{Trueprop } (\text{realizes } t \text{ } P)) \\
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& \quad (\text{realizes } t \text{ } (P \longrightarrow Q)) \equiv (\text{realizes } \text{Null } P \longrightarrow \text{realizes } t \text{ } Q)
\end{aligned}$$

$$\begin{aligned}
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}('P))) \implies \\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \longrightarrow Q)) \equiv (\forall x::'P. \text{realizes } x \ P \longrightarrow \text{realizes } \text{Null } Q) \\
\\
& (\text{realizes } t \ (P \longrightarrow Q)) \equiv (\forall x. \text{realizes } x \ P \longrightarrow \text{realizes } (t \ x) \ Q) \\
\\
& (\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (\forall x. P \ x)) \equiv (\forall x. \text{realizes } \text{Null } (P \ x)) \\
\\
& (\text{realizes } t \ (\forall x. P \ x)) \equiv (\forall x. \text{realizes } (t \ x) \ (P \ x)) \\
\\
& (\lambda x. \text{typeof } (P \ x)) \equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (\exists x. P \ x)) \equiv (\text{realizes } \text{Null } (P \ t)) \\
\\
& (\text{realizes } t \ (\exists x. P \ x)) \equiv (\text{realizes } (\text{snd } t) \ (P \ (\text{fst } t))) \\
\\
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of Left } \Rightarrow \text{realizes } \text{Null } P \mid \text{Right } \Rightarrow \text{realizes } \text{Null } Q) \\
\\
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of None } \Rightarrow \text{realizes } \text{Null } P \mid \text{Some } q \Rightarrow \text{realizes } q \ Q) \\
\\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of None } \Rightarrow \text{realizes } \text{Null } Q \mid \text{Some } p \Rightarrow \text{realizes } p \ P) \\
\\
& (\text{realizes } t \ (P \vee Q)) \equiv \\
& (\text{case } t \text{ of Inl } p \Rightarrow \text{realizes } p \ P \mid \text{Inr } q \Rightarrow \text{realizes } q \ Q) \\
\\
& (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } \text{Null } P \wedge \text{realizes } t \ Q) \\
\\
& (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } t \ P \wedge \text{realizes } \text{Null } Q) \\
\\
& (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } (\text{fst } t) \ P \wedge \text{realizes } (\text{snd } t) \ Q) \\
\\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \text{realizes } t \ (\neg P) \equiv \neg \text{realizes } \text{Null } P \\
\\
& \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \\
& \text{realizes } t \ (\neg P) \equiv (\forall x::'P. \neg \text{realizes } x \ P) \\
\\
& \text{typeof } (P::\text{bool}) \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
& \text{realizes } t \ (P = Q) \equiv \text{realizes } \text{Null } P = \text{realizes } \text{Null } Q
\end{aligned}$$



$$(\text{realizes } t \ (P = Q)) \equiv (\text{realizes } t \ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$$

## 26.4 Computational content of basic inference rules

**theorem** *disjE-realizer*:

assumes  $r$ :  $\text{case } x \text{ of } \text{Inl } p \Rightarrow P \ p \mid \text{Inr } q \Rightarrow Q \ q$   
 and  $r1$ :  $\bigwedge p. P \ p \Longrightarrow R \ (f \ p)$  and  $r2$ :  $\bigwedge q. Q \ q \Longrightarrow R \ (g \ q)$   
 shows  $R \ (\text{case } x \text{ of } \text{Inl } p \Rightarrow f \ p \mid \text{Inr } q \Rightarrow g \ q)$   
**proof** (*cases x*)  
 case *Inl*  
 with  $r$  show ?thesis by simp (rule  $r1$ )  
 next  
 case *Inr*  
 with  $r$  show ?thesis by simp (rule  $r2$ )  
 qed

**theorem** *disjE-realizer2*:

assumes  $r$ :  $\text{case } x \text{ of } \text{None} \Rightarrow P \mid \text{Some } q \Rightarrow Q \ q$   
 and  $r1$ :  $P \Longrightarrow R \ f$  and  $r2$ :  $\bigwedge q. Q \ q \Longrightarrow R \ (g \ q)$   
 shows  $R \ (\text{case } x \text{ of } \text{None} \Rightarrow f \mid \text{Some } q \Rightarrow g \ q)$   
**proof** (*cases x*)  
 case *None*  
 with  $r$  show ?thesis by simp (rule  $r1$ )  
 next  
 case *Some*  
 with  $r$  show ?thesis by simp (rule  $r2$ )  
 qed

**theorem** *disjE-realizer3*:

assumes  $r$ :  $\text{case } x \text{ of } \text{Left} \Rightarrow P \mid \text{Right} \Rightarrow Q$   
 and  $r1$ :  $P \Longrightarrow R \ f$  and  $r2$ :  $Q \Longrightarrow R \ g$   
 shows  $R \ (\text{case } x \text{ of } \text{Left} \Rightarrow f \mid \text{Right} \Rightarrow g)$   
**proof** (*cases x*)  
 case *Left*  
 with  $r$  show ?thesis by simp (rule  $r1$ )  
 next  
 case *Right*  
 with  $r$  show ?thesis by simp (rule  $r2$ )  
 qed

**theorem** *conjI-realizer*:

$P \ p \Longrightarrow Q \ q \Longrightarrow P \ (fst \ (p, q)) \wedge Q \ (snd \ (p, q))$   
 by simp

**theorem** *exI-realizer*:

$P \ y \ x \Longrightarrow P \ (snd \ (x, y)) \ (fst \ (x, y))$  by simp

**theorem** *exE-realizer*:  $P \ (snd \ p) \ (fst \ p) \Longrightarrow$

$$(\bigwedge x y. P y x \implies Q (f x y)) \implies Q (\text{let } (x, y) = p \text{ in } f x y) \\ \mathbf{by} \text{ (cases } p \text{) (simp add: Let-def)}$$

$$\mathbf{theorem} \text{ } exE\text{-realizer'}: P \text{ (snd } p \text{) (fst } p \text{)} \implies \\ (\bigwedge x y. P y x \implies Q) \implies Q \mathbf{by} \text{ (cases } p \text{) simp}$$

**realizers**

$$\text{impI } (P, Q): \lambda pq. pq \\ \Lambda (c: -) (d: -) P Q pq (h: -). \text{allI } \cdot \cdot \cdot c \cdot (\Lambda x. \text{impI } \cdot \cdot \cdot \cdot (h \cdot x))$$

$$\text{impI } (P): \text{Null} \\ \Lambda (c: -) P Q (h: -). \text{allI } \cdot \cdot \cdot c \cdot (\Lambda x. \text{impI } \cdot \cdot \cdot \cdot (h \cdot x))$$

$$\text{impI } (Q): \lambda q. q \Lambda (c: -) P Q q. \text{impI } \cdot \cdot \cdot$$

$$\text{impI}: \text{Null impI}$$

$$\text{mp } (P, Q): \lambda pq. pq \\ \Lambda (c: -) (d: -) P Q pq (h: -) p. \text{mp } \cdot \cdot \cdot \cdot (\text{spec } \cdot \cdot \cdot p \cdot c \cdot h)$$

$$\text{mp } (P): \text{Null} \\ \Lambda (c: -) P Q (h: -) p. \text{mp } \cdot \cdot \cdot \cdot (\text{spec } \cdot \cdot \cdot p \cdot c \cdot h)$$

$$\text{mp } (Q): \lambda q. q \Lambda (c: -) P Q q. \text{mp } \cdot \cdot \cdot$$

$$\text{mp}: \text{Null mp}$$

$$\text{allI } (P): \lambda p. p \Lambda (c: -) P (d: -) p. \text{allI } \cdot \cdot \cdot d$$

$$\text{allI}: \text{Null allI}$$

$$\text{spec } (P): \lambda x p. p x \Lambda (c: -) P x (d: -) p. \text{spec } \cdot \cdot \cdot x \cdot d$$

$$\text{spec}: \text{Null spec}$$

$$\text{exI } (P): \lambda x p. (x, p) \Lambda (c: -) P x (d: -) p. \text{exI-realizer} \cdot P \cdot p \cdot x \cdot c \cdot d$$

$$\text{exI}: \lambda x. x \Lambda P x (c: -) (h: -). h$$

$$\text{exE } (P, Q): \lambda p pq. \text{let } (x, y) = p \text{ in } pq x y \\ \Lambda (c: -) (d: -) P Q (e: -) p (h: -) pq. \text{exE-realizer} \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h$$

$$\text{exE } (P): \text{Null} \\ \Lambda (c: -) P Q (d: -) p. \text{exE-realizer'} \cdot \cdot \cdot \cdot \cdot c \cdot d$$

$$\text{exE } (Q): \lambda x pq. pq x \\ \Lambda (c: -) P Q (d: -) x (h1: -) pq (h2: -). h2 \cdot x \cdot h1$$

$$\text{exE}: \text{Null}$$

$$\Lambda P Q (c: -) x (h1: -) (h2: -). h2 \cdot x \cdot h1$$

$$\text{conjI} (P, Q): \text{Pair}$$

$$\Lambda (c: -) (d: -) P Q p (h: -) q. \text{conjI-realizer} \cdot P \cdot p \cdot Q \cdot q \cdot c \cdot d \cdot h$$

$$\text{conjI} (P): \lambda p. p$$

$$\Lambda (c: -) P Q p. \text{conjI} \cdot - \cdot - \cdot -$$

$$\text{conjI} (Q): \lambda q. q$$

$$\Lambda (c: -) P Q (h: -) q. \text{conjI} \cdot - \cdot - \cdot - \cdot h$$

$$\text{conjI}: \text{Null conjI}$$

$$\text{conjunct1} (P, Q): \text{fst}$$

$$\Lambda (c: -) (d: -) P Q pq. \text{conjunct1} \cdot - \cdot - \cdot -$$

$$\text{conjunct1} (P): \lambda p. p$$

$$\Lambda (c: -) P Q p. \text{conjunct1} \cdot - \cdot - \cdot -$$

$$\text{conjunct1} (Q): \text{Null}$$

$$\Lambda (c: -) P Q q. \text{conjunct1} \cdot - \cdot - \cdot -$$

$$\text{conjunct1}: \text{Null conjunct1}$$

$$\text{conjunct2} (P, Q): \text{snd}$$

$$\Lambda (c: -) (d: -) P Q pq. \text{conjunct2} \cdot - \cdot - \cdot -$$

$$\text{conjunct2} (P): \text{Null}$$

$$\Lambda (c: -) P Q p. \text{conjunct2} \cdot - \cdot - \cdot -$$

$$\text{conjunct2} (Q): \lambda p. p$$

$$\Lambda (c: -) P Q p. \text{conjunct2} \cdot - \cdot - \cdot -$$

$$\text{conjunct2}: \text{Null conjunct2}$$

$$\text{disjI1} (P, Q): \text{Inl}$$

$$\Lambda (c: -) (d: -) P Q p. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{sum.cases-1} \cdot P \cdot - \cdot p \cdot \text{arity-type-bool} \cdot c \cdot d)$$

$$\text{disjI1} (P): \text{Some}$$

$$\Lambda (c: -) P Q p. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{option.cases-2} \cdot - \cdot P \cdot p \cdot \text{arity-type-bool} \cdot c)$$

$$\text{disjI1} (Q): \text{None}$$

$$\Lambda (c: -) P Q. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{option.cases-1} \cdot - \cdot - \cdot \text{arity-type-bool} \cdot c)$$

$$\text{disjI1}: \text{Left}$$

$$\Lambda P Q. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{sumbool.cases-1} \cdot - \cdot - \cdot \text{arity-type-bool})$$

$$\text{disjI2} (P, Q): \text{Inr}$$

$\Lambda (d: -) (c: -) Q P q. \text{iffD2} \cdot \dots \cdot (\text{sum.cases-2} \cdot \dots \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c \cdot d)$

$\text{disjI2} (P): \text{None}$

$\Lambda (c: -) Q P. \text{iffD2} \cdot \dots \cdot (\text{option.cases-1} \cdot \dots \cdot \text{arity-type-bool} \cdot c)$

$\text{disjI2} (Q): \text{Some}$

$\Lambda (c: -) Q P q. \text{iffD2} \cdot \dots \cdot (\text{option.cases-2} \cdot \dots \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c)$

$\text{disjI2}: \text{Right}$

$\Lambda Q P. \text{iffD2} \cdot \dots \cdot (\text{sumbool.cases-2} \cdot \dots \cdot \text{arity-type-bool})$

$\text{disjE} (P, Q, R): \lambda pq \text{ pr } qr.$

$(\text{case } pq \text{ of } \text{Inl } p \Rightarrow \text{pr } p \mid \text{Inr } q \Rightarrow \text{qr } q)$

$\Lambda (c: -) (d: -) (e: -) P Q R pq (h1: -) \text{pr} (h2: -) \text{qr}.$

$\text{disjE-realizer} \cdot \dots \cdot pq \cdot R \cdot \text{pr} \cdot \text{qr} \cdot c \cdot d \cdot e \cdot h1 \cdot h2$

$\text{disjE} (Q, R): \lambda pq \text{ pr } qr.$

$(\text{case } pq \text{ of } \text{None} \Rightarrow \text{pr} \mid \text{Some } q \Rightarrow \text{qr } q)$

$\Lambda (c: -) (d: -) P Q R pq (h1: -) \text{pr} (h2: -) \text{qr}.$

$\text{disjE-realizer2} \cdot \dots \cdot pq \cdot R \cdot \text{pr} \cdot \text{qr} \cdot c \cdot d \cdot h1 \cdot h2$

$\text{disjE} (P, R): \lambda pq \text{ pr } qr.$

$(\text{case } pq \text{ of } \text{None} \Rightarrow \text{qr} \mid \text{Some } p \Rightarrow \text{pr } p)$

$\Lambda (c: -) (d: -) P Q R pq (h1: -) \text{pr} (h2: -) \text{qr} (h3: -).$

$\text{disjE-realizer2} \cdot \dots \cdot pq \cdot R \cdot \text{qr} \cdot \text{pr} \cdot c \cdot d \cdot h1 \cdot h3 \cdot h2$

$\text{disjE} (R): \lambda pq \text{ pr } qr.$

$(\text{case } pq \text{ of } \text{Left} \Rightarrow \text{pr} \mid \text{Right} \Rightarrow \text{qr})$

$\Lambda (c: -) P Q R pq (h1: -) \text{pr} (h2: -) \text{qr}.$

$\text{disjE-realizer3} \cdot \dots \cdot pq \cdot R \cdot \text{pr} \cdot \text{qr} \cdot c \cdot h1 \cdot h2$

$\text{disjE} (P, Q): \text{Null}$

$\Lambda (c: -) (d: -) P Q R pq. \text{disjE-realizer} \cdot \dots \cdot pq \cdot (\lambda x. R) \cdot \dots \cdot c \cdot d \cdot \text{arity-type-bool}$

$\text{disjE} (Q): \text{Null}$

$\Lambda (c: -) P Q R pq. \text{disjE-realizer2} \cdot \dots \cdot pq \cdot (\lambda x. R) \cdot \dots \cdot c \cdot \text{arity-type-bool}$

$\text{disjE} (P): \text{Null}$

$\Lambda (c: -) P Q R pq (h1: -) (h2: -) (h3: -).$

$\text{disjE-realizer2} \cdot \dots \cdot pq \cdot (\lambda x. R) \cdot \dots \cdot c \cdot \text{arity-type-bool} \cdot h1 \cdot h3 \cdot h2$

$\text{disjE}: \text{Null}$

$\Lambda P Q R pq. \text{disjE-realizer3} \cdot \dots \cdot pq \cdot (\lambda x. R) \cdot \dots \cdot \text{arity-type-bool}$

$\text{FalseE} (P): \text{default}$

$\Lambda (c: -) P. \text{FalseE} \cdot -$

*FalseE*: *Null FalseE*

*notI* (*P*): *Null*

$\Lambda (c: -) P (h: -). \text{allI} \cdot - \cdot c \cdot (\Lambda x. \text{notI} \cdot - \cdot (h \cdot x))$

*notI*: *Null notI*

*notE* (*P*, *R*):  $\lambda p. \text{default}$

$\Lambda (c: -) (d: -) P R (h: -) p. \text{notE} \cdot - \cdot - \cdot (\text{spec} \cdot - \cdot p \cdot c \cdot h)$

*notE* (*P*): *Null*

$\Lambda (c: -) P R (h: -) p. \text{notE} \cdot - \cdot - \cdot (\text{spec} \cdot - \cdot p \cdot c \cdot h)$

*notE* (*R*): *default*

$\Lambda (c: -) P R. \text{notE} \cdot - \cdot -$

*notE*: *Null notE*

*subst* (*P*):  $\lambda s \ t \ ps. ps$

$\Lambda (c: -) s \ t \ P (d: -) (h: -) ps. \text{subst} \cdot s \cdot t \cdot P \ ps \cdot d \cdot h$

*subst*: *Null subst*

*iffD1* (*P*, *Q*): *fst*

$\Lambda (d: -) (c: -) Q P pq (h: -) p. \\ mp \cdot - \cdot - \cdot (\text{spec} \cdot - \cdot p \cdot d \cdot (\text{conjunct1} \cdot - \cdot - \cdot h))$

*iffD1* (*P*):  $\lambda p. p$

$\Lambda (c: -) Q P p (h: -). mp \cdot - \cdot - \cdot (\text{conjunct1} \cdot - \cdot - \cdot h)$

*iffD1* (*Q*): *Null*

$\Lambda (c: -) Q P q1 (h: -) q2. \\ mp \cdot - \cdot - \cdot (\text{spec} \cdot - \cdot q2 \cdot c \cdot (\text{conjunct1} \cdot - \cdot - \cdot h))$

*iffD1*: *Null iffD1*

*iffD2* (*P*, *Q*): *snd*

$\Lambda (c: -) (d: -) P Q pq (h: -) q. \\ mp \cdot - \cdot - \cdot (\text{spec} \cdot - \cdot q \cdot d \cdot (\text{conjunct2} \cdot - \cdot - \cdot h))$

*iffD2* (*P*):  $\lambda p. p$

$\Lambda (c: -) P Q p (h: -). mp \cdot - \cdot - \cdot (\text{conjunct2} \cdot - \cdot - \cdot h)$

*iffD2* (*Q*): *Null*

$\Lambda (c: -) P Q q1 (h: -) q2. \\ mp \cdot - \cdot - \cdot (\text{spec} \cdot - \cdot q2 \cdot c \cdot (\text{conjunct2} \cdot - \cdot - \cdot h))$

*iffD2*: *Null iffD2*

```

iffI (P, Q): Pair
  Λ (c: -) (d: -) P Q pq (h1 : -) qp (h2 : -). conjI-realizer ·
    (λpq. ∀ x. P x → Q (pq x)) · pq ·
    (λqp. ∀ x. Q x → P (qp x)) · qp ·
    (arity-type-fun · c · d) ·
    (arity-type-fun · d · c) ·
    (allI · - · c · (Λ x. impI · - · - · (h1 · x))) ·
    (allI · - · d · (Λ x. impI · - · - · (h2 · x)))

iffI (P): λp. p
  Λ (c: -) P Q (h1 : -) p (h2 : -). conjI · - · - ·
    (allI · - · c · (Λ x. impI · - · - · (h1 · x))) ·
    (impI · - · - · h2)

iffI (Q): λq. q
  Λ (c: -) P Q q (h1 : -) (h2 : -). conjI · - · - ·
    (impI · - · - · h1) ·
    (allI · - · c · (Λ x. impI · - · - · (h2 · x)))

iffI: Null iffI

end

```

## 27 Plain: Plain HOL

```

theory Plain
imports Datatype Record FunDef Extraction
begin

```

Plain bootstrap of fundamental HOL tools and packages; does not include *Hilbert-Choice*.

```
ML << path-add ~~/src/HOL/Library >>
```

```
end
```

## 28 Big-Operators: Big operators and finite (non-empty) sets

```

theory Big-Operators
imports Plain
begin

```

### 28.1 Generic monoid operation over a set

```
no-notation times (infixl * 70)
```

**no-notation** *Groups.one* (1)

**locale** *comm-monoid-big* = *comm-monoid* +  
**fixes**  $F :: ('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'a$   
**assumes**  $F\text{-eq}: F\ g\ A = (\text{if finite } A \text{ then fold-image } (op\ *)\ g\ 1\ A \text{ else } 1)$

**sublocale** *comm-monoid-big* < *folding-image* **proof**  
**qed** (*simp add: F-eq*)

**context** *comm-monoid-big*  
**begin**

**lemma** *infinite* [*simp*]:  
 $\neg \text{finite } A \Longrightarrow F\ g\ A = 1$   
**by** (*simp add: F-eq*)

**end**

for ad-hoc proofs for *fold-image*

**lemma** (**in** *comm-monoid-add*) *comm-monoid-mult*:  
 $\text{class.comm-monoid-mult } (op\ +)\ 0$   
**proof qed** (*auto intro: add-assoc add-commute*)

**notation** *times* (**infixl** \* 70)  
**notation** *Groups.one* (1)

## 28.2 Generalized summation over a set

**definition** (**in** *comm-monoid-add*) *setsum* ::  $('b \Rightarrow 'a) \Rightarrow 'b \text{ set} \Rightarrow 'a$  **where**  
 $\text{setsum } f\ A = (\text{if finite } A \text{ then fold-image } (op\ +)\ f\ 0\ A \text{ else } 0)$

**sublocale** *comm-monoid-add* < *setsum!*: *comm-monoid-big*  $op\ +\ 0$  *setsum* **proof**  
**qed** (*fact setsum-def*)

**abbreviation**  
 $\text{Setsum } (\sum - [1000]\ 999)$  **where**  
 $\sum A == \text{setsum } (\%x. x)\ A$

Now: lot's of fancy syntax. First, *setsum*  $(\lambda x. e)\ A$  is written  $\sum_{x \in A}. e$ .

**syntax**  
 $\text{-setsum} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-add} \quad ((\mathfrak{S}SUM\ -::-. )\ [0, 51, 10]\ 10)$   
**syntax** (*xsymbols*)  
 $\text{-setsum} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-add} \quad ((\mathfrak{S}\sum\ -\in-. )\ [0, 51, 10]\ 10)$   
**syntax** (*HTML output*)  
 $\text{-setsum} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-add} \quad ((\mathfrak{S}\sum\ -\in-. )\ [0, 51, 10]\ 10)$

**translations** — Beware of argument permutation!

$SUM\ i:A. b == CONST\ setsum\ (\%i. b)\ A$

$\sum\ i \in A. b == CONST\ setsum\ (\%i. b)\ A$

Instead of  $\sum\ x \in \{x. P\}. e$  we introduce the shorter  $\sum\ x|P. e$ .

**syntax**

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ ((3SUM - | / - / -)\ [0,0,10]\ 10)$

**syntax** (*xsymbols*)

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ ((3\sum - | (-). / -)\ [0,0,10]\ 10)$

**syntax** (*HTML output*)

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a\ ((3\sum - | (-). / -)\ [0,0,10]\ 10)$

**translations**

$SUM\ x|P. t \Rightarrow CONST\ setsum\ (\%x. t)\ \{x. P\}$

$\sum\ x|P. t \Rightarrow CONST\ setsum\ (\%x. t)\ \{x. P\}$

**print-translation**  $\ll$

*let*

*fun setsum-tr' [Abs (x, Tx, t), Const (@{const-syntax Collect}, -) \$ Abs (y, Ty, P)] =*

*if x <> y then raise Match*

*else*

*let*

*val x' = Syntax.mark-bound x;*

*val t' = subst-bound (x', t);*

*val P' = subst-bound (x', P);*

*in Syntax.const @ {syntax-const -qsetsum} \$ Syntax.mark-bound x \$ P' \$*

*t' end*

*| setsum-tr' - = raise Match;*

*in [(@ {const-syntax setsum}, setsum-tr')] end*

$\gg$

**lemma** *setsum-empty:*

*setsum f {} = 0*

**by** (*fact setsum.empty*)

**lemma** *setsum-insert:*

*finite F ==> a \notin F ==> setsum f (insert a F) = f a + setsum f F*

**by** (*fact setsum.insert*)

**lemma** *setsum-infinite:*

*\sim finite A ==> setsum f A = 0*

**by** (*fact setsum.infinite*)

**lemma** (*in comm-monoid-add*) *setsum-reindex:*

**assumes** *inj-on f B* **shows** *setsum h (f ' B) = setsum (h \circ f) B*

**proof** —

**interpret** *comm-monoid-mult op + 0* **by** (*fact comm-monoid-mult*)

**from** *assms* **show** *?thesis* **by** (*auto simp add: setsum-def fold-image-reindex*)



*dest!:(finite-imageD)*  
**qed**

**lemma** (in *comm-monoid-add*) *setsum-reindex-id*:  
*inj-on f B ==> setsum f B = setsum id (f ‘ B)*  
**by** (*simp add: setsum-reindex*)

**lemma** (in *comm-monoid-add*) *setsum-reindex-nonzero*:  
**assumes** *fS: finite S*  
**and** *nz:  $\bigwedge x y. x \in S \implies y \in S \implies x \neq y \implies f x = f y \implies h (f x) = 0$*   
**shows** *setsum h (f ‘ S) = setsum (h o f) S*  
**using** *nz*  
**proof**(*induct rule: finite-induct[OF fS]*)  
**case** 1 **thus** ?*case* **by** *simp*  
**next**  
**case** (2 *x F*)  
**{assume** *fxF: f x ∈ f ‘ F* **hence**  $\exists y \in F. f y = f x$  **by** *auto*  
**then obtain** *y* **where** *y: y ∈ F f x = f y* **by** *auto*  
**from** 2.*hyps y* **have** *xy: x ≠ y* **by** *auto*  
  
**from** 2.*prems*[*of x y*] 2.*hyps xy y* **have** *h0: h (f x) = 0* **by** *simp*  
**have** *setsum h (f ‘ insert x F) = setsum h (f ‘ F)* **using** *fxF* **by** *auto*  
**also have**  $\dots = \text{setsum } (h \circ f) (\text{insert } x F)$   
**unfolding** *setsum.insert[OF ⟨finite F⟩ ⟨x ∉ F⟩]*  
**using** *h0*  
**apply** *simp*  
**apply** (*rule 2.hyps(3)*)  
**apply** (*rule-tac y=y in 2.prems*)  
**apply** *simp-all*  
**done**  
**finally have** ?*case* **..}**  
**moreover**  
**{assume** *fxF: f x ∉ f ‘ F*  
**have** *setsum h (f ‘ insert x F) = h (f x) + setsum h (f ‘ F)*  
**using** *fxF 2.hyps* **by** *simp*  
**also have**  $\dots = \text{setsum } (h \circ f) (\text{insert } x F)$   
**unfolding** *setsum.insert[OF ⟨finite F⟩ ⟨x ∉ F⟩]*  
**apply** *simp*  
**apply** (*rule cong [OF refl [of op + (h (f x))]]*)  
**apply** (*rule 2.hyps(3)*)  
**apply** (*rule-tac y=y in 2.prems*)  
**apply** *simp-all*  
**done**  
**finally have** ?*case* **..}**  
**ultimately show** ?*case* **by** *blast*  
**qed**

**lemma** (in *comm-monoid-add*) *setsum-cong*:  
 $A = B \implies (!x. x:B \implies f x = g x) \implies \text{setsum } f A = \text{setsum } g B$

```

by (cases finite A) (auto intro: setsum.cong)

lemma (in comm-monoid-add) strong-setsum-cong [cong]:
  A = B ==> (!!x. x:B =simp=> f x = g x)
  ==> setsum (%x. f x) A = setsum (%x. g x) B
by (rule setsum-cong) (simp-all add: simp-implies-def)

lemma (in comm-monoid-add) setsum-cong2: [!x. x ∈ A ==> f x = g x] ==>
  setsum f A = setsum g A
by (auto intro: setsum-cong)

lemma (in comm-monoid-add) setsum-reindex-cong:
  [|inj-on f A; B = f ' A; !!a. a:A ==> g a = h (f a)|]
  ==> setsum h B = setsum g A
by (simp add: setsum-reindex cong: setsum-cong)

lemma (in comm-monoid-add) setsum-0[simp]: setsum (%i. 0) A = 0
by (cases finite A) (erule finite-induct, auto)

lemma (in comm-monoid-add) setsum-0': ALL a:A. f a = 0 ==> setsum f A =
0
by (simp add: setsum-cong)

lemma (in comm-monoid-add) setsum-Un-Int: finite A ==> finite B ==>
  setsum g (A Un B) + setsum g (A Int B) = setsum g A + setsum g B
— The reversed orientation looks more natural, but LOOPS as a simprule!
by (fact setsum.union-inter)

lemma (in comm-monoid-add) setsum-Un-disjoint: finite A ==> finite B
==> A Int B = {} ==> setsum g (A Un B) = setsum g A + setsum g B
by (fact setsum.union-disjoint)

lemma setsum-mono-zero-left:
  assumes fT: finite T and ST: S ⊆ T
  and z: ∀ i ∈ T - S. f i = 0
  shows setsum f S = setsum f T
proof—
  have eq: T = S ∪ (T - S) using ST by blast
  have d: S ∩ (T - S) = {} using ST by blast
  from fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)
  show ?thesis
  by (simp add: setsum-Un-disjoint[OF f d, unfolded eq[symmetric]] setsum-0'[OF
z])
qed

lemma setsum-mono-zero-right:
  finite T ==> S ⊆ T ==> ∀ i ∈ T - S. f i = 0 ==> setsum f T = setsum f S
by (blast intro!: setsum-mono-zero-left[symmetric])

```

**lemma** *setsum-mono-zero-cong-left*:  
 assumes  $fT$ : *finite*  $T$  and  $ST$ :  $S \subseteq T$   
 and  $z$ :  $\forall i \in T - S. g\ i = 0$   
 and  $fg$ :  $\bigwedge x. x \in S \implies f\ x = g\ x$   
 shows  $\text{setsum } f\ S = \text{setsum } g\ T$   
**proof**–  
 have  $eq$ :  $T = S \cup (T - S)$  **using**  $ST$  **by** *blast*  
 have  $d$ :  $S \cap (T - S) = \{\}$  **using**  $ST$  **by** *blast*  
 from  $fT\ ST$  have  $f$ : *finite*  $S$  *finite*  $(T - S)$  **by** (*auto intro: finite-subset*)  
 show *?thesis*  
   **using**  $fg$  **by** (*simp add: setsum-Un-disjoint[OF f d, unfolded eq[symmetric]]*)  
*setsum-0*[ $OF\ z$ ])  
**qed**

**lemma** *setsum-mono-zero-cong-right*:  
 assumes  $fT$ : *finite*  $T$  and  $ST$ :  $S \subseteq T$   
 and  $z$ :  $\forall i \in T - S. f\ i = 0$   
 and  $fg$ :  $\bigwedge x. x \in S \implies f\ x = g\ x$   
 shows  $\text{setsum } f\ T = \text{setsum } g\ S$   
**using** *setsum-mono-zero-cong-left*[ $OF\ fT\ ST\ z$ ]  $fg$ [*symmetric*] **by** *auto*

**lemma** *setsum-delta*:  
 assumes  $fS$ : *finite*  $S$   
 shows  $\text{setsum } (\lambda k. \text{if } k=a \text{ then } b\ k \text{ else } 0)\ S = (\text{if } a \in S \text{ then } b\ a \text{ else } 0)$   
**proof**–  
 let  $?f = (\lambda k. \text{if } k=a \text{ then } b\ k \text{ else } 0)$   
 {**assume**  $a$ :  $a \notin S$   
   **hence**  $\forall k \in S. ?f\ k = 0$  **by** *simp*  
   **hence** *?thesis* **using**  $a$  **by** *simp*}

**moreover**  
 {**assume**  $a$ :  $a \in S$   
   let  $?A = S - \{a\}$   
   let  $?B = \{a\}$   
   have  $eq$ :  $S = ?A \cup ?B$  **using**  $a$  **by** *blast*  
   have  $dj$ :  $?A \cap ?B = \{\}$  **by** *simp*  
   from  $fS$  have  $fAB$ : *finite*  $?A$  *finite*  $?B$  **by** *auto*  
   have  $\text{setsum } ?f\ S = \text{setsum } ?f\ ?A + \text{setsum } ?f\ ?B$   
     **using** *setsum-Un-disjoint*[ $OF\ fAB\ dj$ , *of*  $?f$ , *unfolded eq[symmetric]*]  
     **by** *simp*  
   **then have** *?thesis* **using**  $a$  **by** *simp*}

**ultimately show** *?thesis* **by** *blast*  
**qed**

**lemma** *setsum-delta'*:  
 assumes  $fS$ : *finite*  $S$  **shows**  
 $\text{setsum } (\lambda k. \text{if } a = k \text{ then } b\ k \text{ else } 0)\ S =$   
    $(\text{if } a \in S \text{ then } b\ a \text{ else } 0)$   
**using** *setsum-delta*[ $OF\ fS$ , *of*  $a\ b$ , *symmetric*]  
**by** (*auto intro: setsum-cong*)

```

lemma setsum-restrict-set:
  assumes fA: finite A
  shows setsum f (A ∩ B) = setsum (λx. if x ∈ B then f x else 0) A
proof –
  from fA have fab: finite (A ∩ B) by auto
  have aba: A ∩ B ⊆ A by blast
  let ?g = λx. if x ∈ A ∩ B then f x else 0
  from setsum-mono-zero-left[OF fA aba, of ?g]
  show ?thesis by simp
qed

```

```

lemma setsum-cases:
  assumes fA: finite A
  shows setsum (λx. if P x then f x else g x) A =
    setsum f (A ∩ {x. P x}) + setsum g (A ∩ -{x. P x})
proof –
  have a: A = A ∩ {x. P x} ∪ A ∩ -{x. P x}
    (A ∩ {x. P x}) ∩ (A ∩ -{x. P x}) = {}
    by blast+
  from fA
  have f: finite (A ∩ {x. P x}) finite (A ∩ -{x. P x}) by auto
  let ?g = λx. if P x then f x else g x
  from setsum-Un-disjoint[OF f a(2), of ?g] a(1)
  show ?thesis by simp
qed

```

```

lemma (in comm-monoid-add) setsum-UN-disjoint:
  assumes finite I and ALL i:I. finite (A i)
  and ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {}
  shows setsum f (UNION I A) = (∑ i∈I. setsum f (A i))
proof –
  interpret comm-monoid-mult op + 0 by (fact comm-monoid-mult)
  from assms show ?thesis by (simp add: setsum-def fold-image-UN-disjoint cong: setsum-cong)
qed

```

No need to assume that  $C$  is finite. If infinite, the rhs is directly 0, and  $\bigcup C$  is also infinite, hence the lhs is also 0.

```

lemma setsum-Union-disjoint:
  [| (ALL A:C. finite A);
    (ALL A:C. ALL B:C. A ≠ B --> A Int B = {}) |]
  ==> setsum f (Union C) = setsum (setsum f) C
apply (cases finite C)
prefer 2 apply (force dest: finite-UnionD simp add: setsum-def)
apply (frule setsum-UN-disjoint [of C id f])
apply (unfold Union-def id-def, assumption+)
done

```

```

lemma (in comm-monoid-add) setsum-Sigma:
  assumes finite A and ALL x:A. finite (B x)
  shows  $(\sum x \in A. (\sum y \in B x. f x y)) = (\sum (x,y) \in (\text{SIGMA } x:A. B x). f x y)$ 
proof –
  interpret comm-monoid-mult op + 0 by (fact comm-monoid-mult)
  from assms show ?thesis by (simp add: setsum-def fold-image-Sigma split-def
  cong: setsum-cong)
qed

```

Here we can eliminate the finiteness assumptions, by cases.

```

lemma setsum-cartesian-product:
   $(\sum x \in A. (\sum y \in B. f x y)) = (\sum (x,y) \in A <*> B. f x y)$ 
apply (cases finite A)
apply (cases finite B)
apply (simp add: setsum-Sigma)
apply (cases A={}, simp)
apply (simp)
apply (auto simp add: setsum-def
  dest: finite-cartesian-productD1 finite-cartesian-productD2)
done

```

```

lemma (in comm-monoid-add) setsum-addf: setsum (%x. f x + g x) A = (setsum
f A + setsum g A)
by (cases finite A) (simp-all add: setsum.distrib)

```

### 28.2.1 Properties in more restricted classes of structures

```

lemma setsum-SucD: setsum f A = Suc n ==> EX a:A. 0 < f a
apply (case-tac finite A)
prefer 2 apply (simp add: setsum-def)
apply (erule rev-mp)
apply (erule finite-induct, auto)
done

```

```

lemma setsum-eq-0-iff [simp]:
  finite F ==> (setsum f F = 0) = (ALL a:F. f a = (0::nat))
by (induct set: finite) auto

```

```

lemma setsum-eq-Suc0-iff: finite A ==>
  (setsum f A = Suc 0) = (EX a:A. f a = Suc 0 & (ALL b:A. a ≠ b → f b = 0))
apply (erule finite-induct)
apply (auto simp add: add-is-1)
done

```

```

lemmas setsum-eq-1-iff = setsum-eq-Suc0-iff [simplified One-nat-def [symmetric]]

```

```

lemma setsum-Un-nat: finite A ==> finite B ==>

```

( $\text{setsum } f \ (A \text{ Un } B) :: \text{nat}$ ) =  $\text{setsum } f \ A + \text{setsum } f \ B - \text{setsum } f \ (A \text{ Int } B)$   
 — For the natural numbers, we have subtraction.

**by** (*subst setsum-Un-Int [symmetric], auto simp add: algebra-simps*)

**lemma** *setsum-Un: finite A ==> finite B ==>*

( $\text{setsum } f \ (A \text{ Un } B) :: 'a :: \text{ab-group-add}$ ) =  
 $\text{setsum } f \ A + \text{setsum } f \ B - \text{setsum } f \ (A \text{ Int } B)$

**by** (*subst setsum-Un-Int [symmetric], auto simp add: algebra-simps*)

**lemma** (*in comm-monoid-add*) *setsum-eq-general-reverses:*

**assumes** *fS: finite S and fT: finite T*

**and** *kh:  $\bigwedge y. y \in T \implies k \ y \in S \wedge h \ (k \ y) = y$*

**and** *hk:  $\bigwedge x. x \in S \implies h \ x \in T \wedge k \ (h \ x) = x \wedge g \ (h \ x) = f \ x$*

**shows**  $\text{setsum } f \ S = \text{setsum } g \ T$

**proof** —

**interpret** *comm-monoid-mult op + 0 by (fact comm-monoid-mult)*

**show** *?thesis*

**apply** (*simp add: setsum-def fS fT*)

**apply** (*rule fold-image-eq-general-inverses*)

**apply** (*rule fS*)

**apply** (*erule kh*)

**apply** (*erule hk*)

**done**

**qed**

**lemma** (*in comm-monoid-add*) *setsum-Un-zero:*

**assumes** *fS: finite S and fT: finite T*

**and** *I0:  $\forall x \in S \cap T. f \ x = 0$*

**shows**  $\text{setsum } f \ (S \cup T) = \text{setsum } f \ S + \text{setsum } f \ T$

**proof** —

**interpret** *comm-monoid-mult op + 0 by (fact comm-monoid-mult)*

**show** *?thesis*

**using** *fS fT*

**apply** (*simp add: setsum-def*)

**apply** (*rule fold-image-Un-one*)

**using** *I0 by auto*

**qed**

**lemma** *setsum-UNION-zero:*

**assumes** *fS: finite S and fSS:  $\forall T \in S. \text{finite } T$*

**and** *f0:  $\bigwedge T1 \ T2 \ x. T1 \in S \implies T2 \in S \implies T1 \neq T2 \implies x \in T1 \implies x \in T2 \implies f \ x = 0$*

**shows**  $\text{setsum } f \ (\bigcup S) = \text{setsum } (\lambda T. \text{setsum } f \ T) \ S$

**using** *fSS f0*

**proof** (*induct rule: finite-induct[OF fS]*)

**case** 1 **thus** *?case by simp*

**next**

**case** (2 *T F*)

**then have** *fTF: finite T  $\forall T \in F. \text{finite } T$  finite F and TF:  $T \notin F$*

**and**  $H$ :  $\text{setsum } f (\bigcup F) = \text{setsum } (\text{setsum } f) F$  **by** *auto*  
**from**  $fTF$  **have**  $fUF$ :  $\text{finite } (\bigcup F)$  **by** *auto*  
**from**  $2.\text{prems } TF fTF$   
**show**  $?case$   
**by** (*auto simp add: H[symmetric] intro: setsum-Un-zero[OF fTF(1) fUF, of f]*)  
**qed**

**lemma** *setsum-diff1-nat*:  $(\text{setsum } f (A - \{a\}) :: \text{nat}) =$   
 $(\text{if } a:A \text{ then } \text{setsum } f A - f a \text{ else } \text{setsum } f A)$   
**apply** (*case-tac finite A*)  
**prefer** 2 **apply** (*simp add: setsum-def*)  
**apply** (*erule finite-induct*)  
**apply** (*auto simp add: insert-Diff-if*)  
**apply** (*drule-tac a = a in mk-disjoint-insert, auto*)  
**done**

**lemma** *setsum-diff1*:  $\text{finite } A \implies$   
 $(\text{setsum } f (A - \{a\}) :: ('a::\text{ab-group-add})) =$   
 $(\text{if } a:A \text{ then } \text{setsum } f A - f a \text{ else } \text{setsum } f A)$   
**by** (*erule finite-induct (auto simp add: insert-Diff-if)*)

**lemma** *setsum-diff1 '[rule-format]*:  
 $\text{finite } A \implies a \in A \longrightarrow (\sum x \in A. f x) = f a + (\sum x \in (A - \{a\}). f x)$   
**apply** (*erule finite-induct[where F=A and P=% A. (a \in A \longrightarrow (\sum x \in A. f x) = f a + (\sum x \in (A - \{a\}). f x))]*)  
**apply** (*auto simp add: insert-Diff-if add-ac*)  
**done**

**lemma** *setsum-diff1-ring*: **assumes**  $\text{finite } A \ a \in A$   
**shows**  $\text{setsum } f (A - \{a\}) = \text{setsum } f A - (f a :: 'a::\text{ring})$   
**unfolding** *setsum-diff1 '[OF assms]* **by** *auto*

**lemma** *setsum-diff-nat*:  
**assumes**  $\text{finite } B \text{ and } B \subseteq A$   
**shows**  $(\text{setsum } f (A - B) :: \text{nat}) = (\text{setsum } f A) - (\text{setsum } f B)$   
**using** *assms*  
**proof** *induct*  
**show**  $\text{setsum } f (A - \{\}) = (\text{setsum } f A) - (\text{setsum } f \{\})$  **by** *simp*  
**next**  
**fix**  $F x$  **assume**  $\text{finF}$ :  $\text{finite } F$  **and**  $x\text{notinF}$ :  $x \notin F$   
**and**  $x\text{FinA}$ :  $\text{insert } x F \subseteq A$   
**and**  $IH$ :  $F \subseteq A \implies \text{setsum } f (A - F) = \text{setsum } f A - \text{setsum } f F$   
**from**  $x\text{notinF } x\text{FinA}$  **have**  $x\text{inAF}$ :  $x \in (A - F)$  **by** *simp*  
**from**  $x\text{inAF}$  **have**  $A$ :  $\text{setsum } f ((A - F) - \{x\}) = \text{setsum } f (A - F) - f x$   
**by** (*simp add: setsum-diff1-nat*)  
**from**  $x\text{FinA}$  **have**  $F \subseteq A$  **by** *simp*

```

with  $IH$  have  $\text{setsum } f (A - F) = \text{setsum } f A - \text{setsum } f F$  by simp
with  $A$  have  $B$ :  $\text{setsum } f ((A - F) - \{x\}) = \text{setsum } f A - \text{setsum } f F - f x$ 
  by simp
from  $xnotin F$  have  $A - \text{insert } x F = (A - F) - \{x\}$  by auto
with  $B$  have  $C$ :  $\text{setsum } f (A - \text{insert } x F) = \text{setsum } f A - \text{setsum } f F - f x$ 
  by simp
from  $fin F$   $xnotin F$  have  $\text{setsum } f (\text{insert } x F) = \text{setsum } f F + f x$  by simp
with  $C$  have  $\text{setsum } f (A - \text{insert } x F) = \text{setsum } f A - \text{setsum } f (\text{insert } x F)$ 
  by simp
thus  $\text{setsum } f (A - \text{insert } x F) = \text{setsum } f A - \text{setsum } f (\text{insert } x F)$  by simp
qed

```

**lemma** *setsum-diff*:

```

assumes  $le$ :  $\text{finite } A \ B \subseteq A$ 
shows  $\text{setsum } f (A - B) = \text{setsum } f A - ((\text{setsum } f B)::('a::\text{ab-group-add}))$ 
proof -
  from  $le$  have  $\text{finite} B$ :  $\text{finite } B$  using finite-subset by auto
  show ?thesis using  $\text{finite} B$   $le$ 
  proof induct
    case empty
    thus ?case by auto
  next
    case  $(\text{insert } x F)$ 
    thus ?case using  $le$   $\text{finite} B$ 
    by (simp add: Diff-insert[where  $a=x$  and  $B=F$ ] setsum-diff1 insert-absorb)
  qed
qed

```

**lemma** *setsum-mono*:

```

assumes  $le$ :  $\bigwedge i. i \in K \implies f(i::'a) \leq ((g\ i)::('b::\{\text{comm-monoid-add}, \text{ordered-ab-semigroup-add}\}))$ 
shows  $(\sum i \in K. f\ i) \leq (\sum i \in K. g\ i)$ 
proof (cases finite K)
  case True
  thus ?thesis using  $le$ 
  proof induct
    case empty
    thus ?case by simp
  next
    case insert
    thus ?case using add-mono by fastsimp
  qed
next
  case False
  thus ?thesis
  by (simp add: setsum-def)
qed

```

**lemma** *setsum-strict-mono*:

```

fixes  $f :: 'a \Rightarrow 'b::\{\text{ordered-cancel-ab-semigroup-add}, \text{comm-monoid-add}\}$ 

```



```

assumes finite  $A$   $A \neq \{\}$ 
and  $\forall x. x:A \implies f\ x < g\ x$ 
shows  $\text{setsum } f\ A < \text{setsum } g\ A$ 
using prems
proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case insert thus ?case by (auto simp: add-strict-mono)
qed

lemma setsum-negf:
   $\text{setsum } (\%x. - (f\ x)::'a::\text{ab-group-add})\ A = - \text{setsum } f\ A$ 
proof (cases finite A)
  case True thus ?thesis by (induct set: finite) auto
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-subtractf:
   $\text{setsum } (\%x. ((f\ x)::'a::\text{ab-group-add}) - g\ x)\ A =$ 
   $\text{setsum } f\ A - \text{setsum } g\ A$ 
proof (cases finite A)
  case True thus ?thesis by (simp add: diff-minus setsum-addf setsum-negf)
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-nonneg:
  assumes nn:  $\forall x \in A. (0::'a::\{\text{ordered-ab-semigroup-add}, \text{comm-monoid-add}\}) \leq f\ x$ 
shows  $0 \leq \text{setsum } f\ A$ 
proof (cases finite A)
  case True thus ?thesis using nn
  proof induct
    case empty then show ?case by simp
  next
    case (insert  $x\ F$ )
    then have  $0 + 0 \leq f\ x + \text{setsum } f\ F$  by (blast intro: add-mono)
    with insert show ?case by simp
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-nonpos:
  assumes np:  $\forall x \in A. f\ x \leq (0::'a::\{\text{ordered-ab-semigroup-add}, \text{comm-monoid-add}\})$ 
shows  $\text{setsum } f\ A \leq 0$ 
proof (cases finite A)
  case True thus ?thesis using np

```

```

proof induct
  case empty then show ?case by simp
next
  case (insert x F)
  then have  $f\ x + \text{setsum } f\ F \leq 0 + 0$  by (blast intro: add-mono)
  with insert show ?case by simp
qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

```

lemma setsum-nonneg-leq-bound:
  fixes  $f :: 'a \Rightarrow 'b::\{\text{ordered-ab-group-add}\}$ 
  assumes  $\text{finite } s \wedge i. i \in s \implies f\ i \geq 0$  ( $\sum i \in s. f\ i = B$   $i \in s$ )
  shows  $f\ i \leq B$ 
proof –
  have  $0 \leq (\sum i \in s - \{i\}. f\ i)$  and  $0 \leq f\ i$ 
    using assms by (auto intro!: setsum-nonneg)
  moreover
  have  $(\sum i \in s - \{i\}. f\ i) + f\ i = B$ 
    using assms by (simp add: setsum-diff1)
  ultimately show ?thesis by auto
qed

```

```

lemma setsum-nonneg-0:
  fixes  $f :: 'a \Rightarrow 'b::\{\text{ordered-ab-group-add}\}$ 
  assumes  $\text{finite } s$  and  $\text{pos: } \bigwedge i. i \in s \implies f\ i \geq 0$ 
  and  $(\sum i \in s. f\ i) = 0$  and  $i: i \in s$ 
  shows  $f\ i = 0$ 
  using setsum-nonneg-leq-bound [OF assms] pos [OF i] by auto

```

```

lemma setsum-mono2:
  fixes  $f :: 'a \Rightarrow 'b :: \text{ordered-comm-monoid-add}$ 
  assumes fin:  $\text{finite } B$  and sub:  $A \subseteq B$  and nn:  $\bigwedge b. b \in B - A \implies 0 \leq f\ b$ 
  shows  $\text{setsum } f\ A \leq \text{setsum } f\ B$ 
proof –
  have  $\text{setsum } f\ A \leq \text{setsum } f\ A + \text{setsum } f\ (B - A)$ 
    by (simp add: add-increasing2 [OF setsum-nonneg] nn Ball-def)
  also have  $\dots = \text{setsum } f\ (A \cup (B - A))$  using fin finite-subset [OF sub fin]
    by (simp add: setsum-Un-disjoint del: Un-Diff-cancel)
  also have  $A \cup (B - A) = B$  using sub by blast
  finally show ?thesis .
qed

```

```

lemma setsum-mono3:  $\text{finite } B \implies A \leq B \implies$ 
   $\text{ALL } x: B - A.$ 
   $0 \leq ((f\ x)::'a::\{\text{comm-monoid-add, ordered-ab-semigroup-add}\}) \implies$ 
   $\text{setsum } f\ A \leq \text{setsum } f\ B$ 
  apply (subgoal-tac  $\text{setsum } f\ B = \text{setsum } f\ A + \text{setsum } f\ (B - A)$ )

```

```

apply (erule ssubst)
apply (subgoal-tac setsum f A + 0 <= setsum f A + setsum f (B - A))
apply simp
apply (rule add-left-mono)
apply (erule setsum-nonneg)
apply (subst setsum-Un-disjoint [THEN sym])
apply (erule finite-subset, assumption)
apply (rule finite-subset)
prefer 2
apply assumption
apply (auto simp add: sup-absorb2)
done

```

```

lemma setsum-right-distrib:
  fixes f :: 'a => ('b::semiring-0)
  shows  $r * \text{setsum } f A = \text{setsum } (\%n. r * f n) A$ 
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (simp add: right-distrib)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

```

lemma setsum-left-distrib:
   $\text{setsum } f A * (r::'a::semiring-0) = (\sum n \in A. f n * r)$ 
proof (cases finite A)
  case True
  then show ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (simp add: left-distrib)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

```

```

lemma setsum-divide-distrib:
   $\text{setsum } f A / (r::'a::field) = (\sum n \in A. f n / r)$ 
proof (cases finite A)
  case True
  then show ?thesis
  proof induct
    case empty thus ?case by simp

```

```

next
  case (insert x A) thus ?case by (simp add: add-divide-distrib)
qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-abs[iff]:
  fixes f :: 'a => ('b::ordered-ab-group-add-abs)
  shows abs (setsum f A) ≤ setsum (%i. abs(f i)) A
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A)
    thus ?case by (auto intro: abs-triangle-ineq order-trans)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-abs-ge-zero[iff]:
  fixes f :: 'a => ('b::ordered-ab-group-add-abs)
  shows 0 ≤ setsum (%i. abs(f i)) A
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by auto
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma abs-setsum-abs[simp]:
  fixes f :: 'a => ('b::ordered-ab-group-add-abs)
  shows abs (∑ a∈A. abs(f a)) = (∑ a∈A. abs(f a))
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert a A)
    hence |∑ a∈insert a A. |f a|| = ||f a| + (∑ a∈A. |f a|)| by simp

```

```

    also have ... = ||f a| + | $\sum_{a \in A} f a$ || using insert by simp
    also have ... = |f a| + | $\sum_{a \in A} f a$ |
      by (simp del: abs-of-nonneg)
    also have ... = ( $\sum_{a \in \text{insert } a \ A} |f a|$ ) using insert by simp
    finally show ?case .
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-Plus:
  fixes A :: 'a set and B :: 'b set
  assumes fin: finite A finite B
  shows setsum f (A <+> B) = setsum (f o Inl) A + setsum (f o Inr) B
proof -
  have A <+> B = Inl ' A  $\cup$  Inr ' B by auto
  moreover from fin have finite (Inl ' A :: ('a + 'b) set) finite (Inr ' B :: ('a + 'b) set)
  by (auto intro: finite-imageI)
  moreover have Inl ' A  $\cap$  Inr ' B = ({}) :: ('a + 'b) set by auto
  moreover have inj-on (Inl :: 'a  $\Rightarrow$  'a + 'b) A inj-on (Inr :: 'b  $\Rightarrow$  'a + 'b) B
  by (auto intro: inj-onI)
  ultimately show ?thesis using fin by (simp add: setsum-Un-disjoint setsum-reindex)
qed

```

Commuting outer and inner summation

```

lemma setsum-commute:
  ( $\sum_{i \in A} \sum_{j \in B} f i j$ ) = ( $\sum_{j \in B} \sum_{i \in A} f i j$ )
proof (simp add: setsum-cartesian-product)
  have ( $\sum_{(x,y) \in A <*> B} f x y$ ) =
    ( $\sum_{(y,x) \in (\% (i,j). (j, i)) ' (A \times B)} f x y$ )
    (is ?s = -)
  apply (simp add: setsum-reindex [where f =  $\% (i,j). (j, i)$ ] swap-inj-on)
  apply (simp add: split-def)
  done
  also have ... = ( $\sum_{(y,x) \in B \times A} f x y$ )
    (is - = ?t)
  apply (simp add: swap-product)
  done
  finally show ?s = ?t .
qed

```

```

lemma setsum-product:
  fixes f :: 'a => ('b::semiring-0)
  shows setsum f A * setsum g B = ( $\sum_{i \in A} \sum_{j \in B} f i * g j$ )
  by (simp add: setsum-right-distrib setsum-left-distrib) (rule setsum-commute)

```

```

lemma setsum-mult-setsum-if-inj:
  fixes f :: 'a => ('b::semiring-0)

```

**shows** *inj-on* ( $\%(a,b). f\ a * g\ b$ ) ( $A \times B$ )  $==>$   
 $setsum\ f\ A * setsum\ g\ B = setsum\ id\ \{f\ a * g\ b \mid a\ b. a:A \ \&\ b:B\}$   
**by** (*auto simp: setsum-product setsum-cartesian-product*  
*intro!: setsum-reindex-cong[symmetric]*)

**lemma** *setsum-constant* [*simp*]:  $(\sum x \in A. y) = of\_nat(card\ A) * y$   
**apply** (*cases finite A*)  
**apply** (*erule finite-induct*)  
**apply** (*auto simp add: algebra-simps*)  
**done**

**lemma** *setsum-bounded*:  
**assumes** *le*:  $\bigwedge i. i \in A \implies f\ i \leq (K :: 'a :: \{semiring-1, ordered-ab-semigroup-add\})$   
**shows**  $setsum\ f\ A \leq of\_nat(card\ A) * K$   
**proof** (*cases finite A*)  
**case** *True*  
**thus** *?thesis* **using** *le* *setsum-mono* [**where**  $K=A$  **and**  $g = \%x. K$ ] **by** *simp*  
**next**  
**case** *False* **thus** *?thesis* **by** (*simp add: setsum-def*)  
**qed**

### 28.2.2 Cardinality as special case of *setsum*

**lemma** *card-eq-setsum*:  
 $card\ A = setsum\ (\lambda x. 1)\ A$   
**by** (*simp only: card-def setsum-def*)

**lemma** *card-UN-disjoint*:  
 $finite\ I ==> (ALL\ i:I. finite\ (A\ i)) ==>$   
 $(ALL\ i:I. ALL\ j:I. i \neq j \longrightarrow A\ i\ Int\ A\ j = \{\})$   
 $==> card\ (UNION\ I\ A) = (\sum i \in I. card\ (A\ i))$   
**apply** (*simp add: card-eq-setsum del: setsum-constant*)  
**apply** (*subgoal-tac*  
 $setsum\ (\%i. card\ (A\ i))\ I = setsum\ (\%i. (setsum\ (\%x. 1)\ (A\ i)))\ I$ )  
**apply** (*simp add: setsum-UN-disjoint del: setsum-constant*)  
**apply** (*simp cong: setsum-cong*)  
**done**

**lemma** *card-Union-disjoint*:  
 $finite\ C ==> (ALL\ A:C. finite\ A) ==>$   
 $(ALL\ A:C. ALL\ B:C. A \neq B \longrightarrow A\ Int\ B = \{\})$   
 $==> card\ (Union\ C) = setsum\ card\ C$   
**apply** (*frule card-UN-disjoint [of C id]*)  
**apply** (*unfold Union-def id-def, assumption+*)  
**done**

The image of a finite set can be expressed using *fold-image*.

**lemma** *image-eq-fold-image*:  
 $finite\ A ==> f\ ` A = fold\_image\ (op\ Un)\ (\%x. \{f\ x\})\ \{\}\ A$

```

proof (induct rule: finite-induct)
  case empty then show ?case by simp
next
  interpret ab-semigroup-mult op Un
  proof qed auto
  case insert
  then show ?case by simp
qed

```

### 28.2.3 Cardinality of products

```

lemma card-SigmaI [simp]:
  [| finite A; ALL a:A. finite (B a) |]
  ==> card (SIGMA x: A. B x) = (SUM a∈A. card (B a))
by(simp add: card-eq-setsum setsum-Sigma del:setsum-constant)

```

```

lemma card-cartesian-product: card (A <*> B) = card(A) * card(B)
by (cases finite A ∧ finite B)
  (auto simp add: card-eq-0-iff dest: finite-cartesian-productD1 finite-cartesian-productD2)

```

```

lemma card-cartesian-product-singleton: card({x} <*> A) = card(A)
by (simp add: card-cartesian-product)

```

### 28.3 Generalized product over a set

```

definition (in comm-monoid-mult) setprod :: ('b => 'a) => 'b set => 'a where
  setprod f A = (if finite A then fold-image (op *) f 1 A else 1)

```

```

sublocale comm-monoid-mult < setprod!: comm-monoid-big op * 1 setprod proof
qed (fact setprod-def)

```

#### abbreviation

```

Setprod (Π - [1000] 999) where
  Π A == setprod (%x. x) A

```

#### syntax

```

-setprod :: pttrn => 'a set => 'b => 'b::comm-monoid-mult ((3PROD -::-. -)
[0, 51, 10] 10)

```

#### syntax (xsymbols)

```

-setprod :: pttrn => 'a set => 'b => 'b::comm-monoid-mult ((3Π -∈-. -) [0,
51, 10] 10)

```

#### syntax (HTML output)

```

-setprod :: pttrn => 'a set => 'b => 'b::comm-monoid-mult ((3Π -∈-. -) [0,
51, 10] 10)

```

**translations** — Beware of argument permutation!

```

PROD i:A. b == CONST setprod (%i. b) A
Π i∈A. b == CONST setprod (%i. b) A

```

Instead of  $\prod x \in \{x. P\}. e$  we introduce the shorter  $\prod x | P. e$ .

**syntax**

$-qsetprod :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\exists PROD - | / - / -) [0,0,10] 10)$

**syntax** (*xsymbols*)

$-qsetprod :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\exists \prod - | (-) / -) [0,0,10] 10)$

**syntax** (*HTML output*)

$-qsetprod :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\exists \prod - | (-) / -) [0,0,10] 10)$

**translations**

$PROD\ x | P. t \Rightarrow CONST\ setprod\ (\%x. t)\ \{x. P\}$

$\prod x | P. t \Rightarrow CONST\ setprod\ (\%x. t)\ \{x. P\}$

**lemma** *setprod-empty*:  $setprod\ f\ \{\} = 1$

**by** (*fact setprod.empty*)

**lemma** *setprod-insert*:  $[| finite\ A; a \notin A |] \Rightarrow$

$setprod\ f\ (insert\ a\ A) = f\ a * setprod\ f\ A$

**by** (*fact setprod.insert*)

**lemma** *setprod-infinite*:  $\sim finite\ A \Rightarrow setprod\ f\ A = 1$

**by** (*fact setprod.infinite*)

**lemma** *setprod-reindex*:

$inj-on\ f\ B \Rightarrow setprod\ h\ (f\ ' B) = setprod\ (h \circ f)\ B$

**by** (*auto simp: setprod-def fold-image-reindex dest!: finite-imageD*)

**lemma** *setprod-reindex-id*:  $inj-on\ f\ B \Rightarrow setprod\ f\ B = setprod\ id\ (f\ ' B)$

**by** (*auto simp add: setprod-reindex*)

**lemma** *setprod-cong*:

$A = B \Rightarrow (!x. x:B \Rightarrow f\ x = g\ x) \Rightarrow setprod\ f\ A = setprod\ g\ B$

**by** (*fastsimp simp: setprod-def intro: fold-image-cong*)

**lemma** *strong-setprod-cong*[*cong*]:

$A = B \Rightarrow (!x. x:B \Rightarrow f\ x = g\ x) \Rightarrow setprod\ f\ A = setprod\ g\ B$

**by** (*fastsimp simp: simp-implies-def setprod-def intro: fold-image-cong*)

**lemma** *setprod-reindex-cong*:  $inj-on\ f\ A \Rightarrow$

$B = f\ ' A \Rightarrow g = h \circ f \Rightarrow setprod\ h\ B = setprod\ g\ A$

**by** (*frule setprod-reindex, simp*)

**lemma** *strong-setprod-reindex-cong*: **assumes**  $i: inj-on\ f\ A$

**and**  $B: B = f\ ' A$  **and**  $eq: \bigwedge x. x \in A \Rightarrow g\ x = (h \circ f)\ x$

**shows**  $setprod\ h\ B = setprod\ g\ A$

**proof** –

**have**  $setprod\ h\ B = setprod\ (h \circ f)\ A$

**by** (*simp add: B setprod-reindex[OF i, of h]*)

**then show** *?thesis* **apply** *simp*

**apply** (*rule setprod-cong*)



```

    apply simp
    by (simp add: eq)
qed

```

```

lemma setprod-Un-one:
  assumes fS: finite S and fT: finite T
  and I0:  $\forall x \in S \cap T. f x = 1$ 
  shows setprod f (S  $\cup$  T) = setprod f S * setprod f T
  using fS fT
  apply (simp add: setprod-def)
  apply (rule fold-image-Un-one)
  using I0 by auto

```

```

lemma setprod-1: setprod (%i. 1) A = 1
  apply (case-tac finite A)
  apply (erule finite-induct, auto simp add: mult-ac)
  done

```

```

lemma setprod-1': ALL a:F. f a = 1 ==> setprod f F = 1
  apply (subgoal-tac setprod f F = setprod (%x. 1) F)
  apply (erule ssubst, rule setprod-1)
  apply (rule setprod-cong, auto)
  done

```

```

lemma setprod-Un-Int: finite A ==> finite B
  ==> setprod g (A Un B) * setprod g (A Int B) = setprod g A * setprod g B
  by (simp add: setprod-def fold-image-Un-Int[symmetric])

```

```

lemma setprod-Un-disjoint: finite A ==> finite B
  ==> A Int B = {} ==> setprod g (A Un B) = setprod g A * setprod g B
  by (subst setprod-Un-Int [symmetric], auto)

```

```

lemma setprod-mono-one-left:
  assumes fT: finite T and ST: S  $\subseteq$  T
  and z:  $\forall i \in T - S. f i = 1$ 
  shows setprod f S = setprod f T
proof-
  have eq: T = S  $\cup$  (T - S) using ST by blast
  have d: S  $\cap$  (T - S) = {} using ST by blast
  from fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)
  show ?thesis
  by (simp add: setprod-Un-disjoint[OF f d, unfolded eq[symmetric]] setprod-1'[OF z])
qed

```

```

lemmas setprod-mono-one-right = setprod-mono-one-left [THEN sym]

```

```

lemma setprod-delta:

```

**assumes**  $fS$ : *finite*  $S$   
**shows**  $\text{setprod } (\lambda k. \text{ if } k=a \text{ then } b \ k \text{ else } 1) \ S = (\text{if } a \in S \text{ then } b \ a \text{ else } 1)$   
**proof** –  
**let**  $?f = (\lambda k. \text{ if } k=a \text{ then } b \ k \text{ else } 1)$   
**{assume**  $a$ :  $a \notin S$   
**hence**  $\forall k \in S. ?f \ k = 1$  **by** *simp*  
**hence**  $?thesis$  **using**  $a$  **by** (*simp add: setprod-1 cong add: setprod-cong*) **}**  
**moreover**  
**{assume**  $a$ :  $a \in S$   
**let**  $?A = S - \{a\}$   
**let**  $?B = \{a\}$   
**have**  $eq$ :  $S = ?A \cup ?B$  **using**  $a$  **by** *blast*  
**have**  $dj$ :  $?A \cap ?B = \{\}$  **by** *simp*  
**from**  $fS$  **have**  $fAB$ : *finite*  $?A$  *finite*  $?B$  **by** *auto*  
**have**  $fA1$ :  $\text{setprod } ?f \ ?A = 1$  **apply** (*rule setprod-1'*) **by** *auto*  
**have**  $\text{setprod } ?f \ ?A * \text{setprod } ?f \ ?B = \text{setprod } ?f \ S$   
**using** *setprod-Un-disjoint*[*OF*  $fAB \ dj$ , *of*  $?f$ , *unfolded eq[symmetric]*]  
**by** *simp*  
**then have**  $?thesis$  **using**  $a$  **by** (*simp add: fA1 cong add: setprod-cong cong*  
*del: if-weak-cong*)**}**  
**ultimately show**  $?thesis$  **by** *blast*  
**qed**

**lemma** *setprod-delta'*:  
**assumes**  $fS$ : *finite*  $S$  **shows**  
 $\text{setprod } (\lambda k. \text{ if } a = k \text{ then } b \ k \text{ else } 1) \ S =$   
 $(\text{if } a \in S \text{ then } b \ a \text{ else } 1)$   
**using** *setprod-delta*[*OF*  $fS$ , *of*  $a \ b$ , *symmetric*]  
**by** (*auto intro: setprod-cong*)

**lemma** *setprod-UN-disjoint*:  
 $\text{finite } I ==> (\text{ALL } i:I. \text{finite } (A \ i)) ==>$   
 $(\text{ALL } i:I. \text{ALL } j:I. i \neq j \longrightarrow A \ i \text{ Int } A \ j = \{\}) ==>$   
 $\text{setprod } f \ (\text{UNION } I \ A) = \text{setprod } (\%i. \text{setprod } f \ (A \ i)) \ I$   
**by**(*simp add: setprod-def fold-image-UN-disjoint cong: setprod-cong*)

**lemma** *setprod-Union-disjoint*:  
 $[\text{ALL } A:C. \text{finite } A];$   
 $(\text{ALL } A:C. \text{ALL } B:C. A \neq B \longrightarrow A \text{ Int } B = \{\}) \ [\text{ALL } A:C. \text{finite } A];$   
 $==> \text{setprod } f \ (\text{Union } C) = \text{setprod } (\text{setprod } f) \ C$   
**apply** (*cases finite C*)  
**prefer** 2 **apply** (*force dest: finite-UnionD simp add: setprod-def*)  
**apply** (*frule setprod-UN-disjoint [of C id f]*)  
**apply** (*unfold Union-def id-def, assumption+*)  
**done**

**lemma** *setprod-Sigma*:  $\text{finite } A ==> \text{ALL } x:A. \text{finite } (B \ x) ==>$   
 $(\prod x \in A. (\prod y \in B \ x. f \ x \ y)) =$

$(\prod (x,y) \in (\text{SIGMA } x:A. B \ x). f \ x \ y)$   
**by** (*simp add:setprod-def fold-image-Sigma split-def cong:setprod-cong*)

Here we can eliminate the finiteness assumptions, by cases.

**lemma** *setprod-cartesian-product*:

$(\prod x \in A. (\prod y \in B. f \ x \ y)) = (\prod (x,y) \in (A \lt * > B). f \ x \ y)$   
**apply** (*cases finite A*)  
**apply** (*cases finite B*)  
**apply** (*simp add: setprod-Sigma*)  
**apply** (*cases A={}, simp*)  
**apply** (*simp add: setprod-1*)  
**apply** (*auto simp add: setprod-def*  
*dest: finite-cartesian-productD1 finite-cartesian-productD2*)  
**done**

**lemma** *setprod-timesf*:

$\text{setprod } (\%x. f \ x \ * \ g \ x) \ A = (\text{setprod } f \ A \ * \ \text{setprod } g \ A)$   
**by** (*simp add:setprod-def fold-image-distrib*)

### 28.3.1 Properties in more restricted classes of structures

**lemma** *setprod-eq-1-iff* [*simp*]:

$\text{finite } F \implies (\text{setprod } f \ F = 1) = (\text{ALL } a:F. f \ a = (1::\text{nat}))$   
**by** (*induct set: finite*) *auto*

**lemma** *setprod-zero*:

$\text{finite } A \implies \text{EX } x: A. f \ x = (0::'a::\text{comm-semiring-1}) \implies \text{setprod } f \ A = 0$   
**apply** (*induct set: finite, force, clarsimp*)  
**apply** (*erule disjE, auto*)  
**done**

**lemma** *setprod-nonneg* [*rule-format*]:

$(\text{ALL } x: A. (0::'a::\text{linordered-semidom}) \leq f \ x) \implies 0 \leq \text{setprod } f \ A$   
**by** (*cases finite A, induct set: finite, simp-all add: mult-nonneg-nonneg*)

**lemma** *setprod-pos* [*rule-format*]:  $(\text{ALL } x: A. (0::'a::\text{linordered-semidom}) < f \ x)$

$\implies 0 < \text{setprod } f \ A$   
**by** (*cases finite A, induct set: finite, simp-all add: mult-pos-pos*)

**lemma** *setprod-zero-iff* [*simp*]:  $\text{finite } A \implies$

$(\text{setprod } f \ A = (0::'a::\{\text{comm-semiring-1}, \text{no-zero-divisors}\})) =$   
 $(\text{EX } x: A. f \ x = 0)$   
**by** (*erule finite-induct, auto simp:no-zero-divisors*)

**lemma** *setprod-pos-nat*:

$\text{finite } S \implies (\text{ALL } x : S. f \ x > (0::\text{nat})) \implies \text{setprod } f \ S > 0$   
**using** *setprod-zero-iff* **by** (*simp del:neq0-conv add:neq0-conv[symmetric]*)

**lemma** *setprod-pos-nat-iff* [*simp*]:

$finite\ S ==> (setprod\ f\ S > 0) = (ALL\ x : S. f\ x > (0::nat))$   
**using** *setprod-zero-iff* **by**(*simp del:neq0-conv add:neq0-conv[symmetric]*)

**lemma** *setprod-Un*:  $finite\ A ==> finite\ B ==> (ALL\ x: A\ Int\ B. f\ x \neq 0) ==>$   
 $(setprod\ f\ (A\ Un\ B) :: 'a :: \{field\})$   
 $= setprod\ f\ A * setprod\ f\ B / setprod\ f\ (A\ Int\ B)$   
**by** (*subst setprod-Un-Int [symmetric]*, *auto*)

**lemma** *setprod-diff1*:  $finite\ A ==> f\ a \neq 0 ==>$   
 $(setprod\ f\ (A - \{a\}) :: 'a :: \{field\}) =$   
 $(if\ a:A\ then\ setprod\ f\ A / f\ a\ else\ setprod\ f\ A)$   
**by** (*erule finite-induct*) (*auto simp add: insert-Diff-if*)

**lemma** *setprod-inversef*:  
**fixes**  $f :: 'b \Rightarrow 'a :: field-inverse-zero$   
**shows**  $finite\ A ==> setprod\ (inverse \circ f)\ A = inverse\ (setprod\ f\ A)$   
**by** (*erule finite-induct*) *auto*

**lemma** *setprod-dividef*:  
**fixes**  $f :: 'b \Rightarrow 'a :: field-inverse-zero$   
**shows**  $finite\ A$   
 $==> setprod\ (\%x. f\ x / g\ x)\ A = setprod\ f\ A / setprod\ g\ A$   
**apply** (*subgoal-tac*  
 $setprod\ (\%x. f\ x / g\ x)\ A = setprod\ (\%x. f\ x * (inverse \circ g)\ x)\ A$ )  
**apply** (*erule ssubst*)  
**apply** (*subst divide-inverse*)  
**apply** (*subst setprod-timesf*)  
**apply** (*subst setprod-inversef, assumption+, rule refl*)  
**apply** (*rule setprod-cong, rule refl*)  
**apply** (*subst divide-inverse, auto*)  
**done**

**lemma** *setprod-dvd-setprod* [*rule-format*]:  
 $(ALL\ x : A. f\ x\ dvd\ g\ x) \longrightarrow setprod\ f\ A\ dvd\ setprod\ g\ A$   
**apply** (*cases finite A*)  
**apply** (*induct set: finite*)  
**apply** (*auto simp add: dvd-def*)  
**apply** (*rule-tac x = k \* ka in exI*)  
**apply** (*simp add: algebra-simps*)  
**done**

**lemma** *setprod-dvd-setprod-subset*:  
 $finite\ B \Longrightarrow A \leq B \Longrightarrow setprod\ f\ A\ dvd\ setprod\ f\ B$   
**apply** (*subgoal-tac setprod f B = setprod f A \* setprod f (B - A)*)  
**apply** (*unfold dvd-def, blast*)  
**apply** (*subst setprod-Un-disjoint [symmetric]*)  
**apply** (*auto elim: finite-subset intro: setprod-cong*)  
**done**

**lemma** *setprod-dvd-setprod-subset2*:

*finite B  $\implies A \leq B \implies \text{ALL } x : A. (f\ x :: 'a :: \text{comm-semiring-1}) \text{ dvd } g\ x \implies$*   
*setprod f A dvd setprod g B*  
**apply** (*rule dvd-trans*)  
**apply** (*rule setprod-dvd-setprod, erule (1) bspec*)  
**apply** (*erule (1) setprod-dvd-setprod-subset*)  
**done**

**lemma** *dvd-setprod*: *finite A  $\implies i:A \implies$*

*(f i :: 'a :: comm-semiring-1) dvd setprod f A*  
**by** (*induct set: finite*) (*auto intro: dvd-mult*)

**lemma** *dvd-setsum* [*rule-format*]: (*ALL i : A. d dvd f i*)  $\longrightarrow$

*(d :: 'a :: comm-semiring-1) dvd (SUM x : A. f x)*  
**apply** (*cases finite A*)  
**apply** (*induct set: finite*)  
**apply** *auto*  
**done**

**lemma** *setprod-mono*:

**fixes** *f :: 'a  $\Rightarrow$  'b :: linordered-semidom*  
**assumes**  $\forall i \in A. 0 \leq f\ i \wedge f\ i \leq g\ i$   
**shows** *setprod f A  $\leq$  setprod g A*  
**proof** (*cases finite A*)  
**case** *True*  
**hence** *?thesis setprod f A  $\geq 0$  using subset-refl[of A]*  
**proof** (*induct A rule: finite-subset-induct*)  
**case** (*insert a F*)  
**thus** *setprod f (insert a F)  $\leq$  setprod g (insert a F)  $0 \leq$  setprod f (insert a F)*  
**unfolding** *setprod-insert[OF insert(1,3)]*  
**using** *assms[rule-format, OF insert(2)] insert*  
**by** (*auto intro: mult-mono mult-nonneg-nonneg*)  
**qed** *auto*  
**thus** *?thesis by simp*  
**qed** *auto*

**lemma** *abs-setprod*:

**fixes** *f :: 'a  $\Rightarrow$  'b :: {linordered-field, abs}*  
**shows** *abs (setprod f A) = setprod ( $\lambda x. \text{abs } (f\ x)$ ) A*  
**proof** (*cases finite A*)  
**case** *True* **thus** *?thesis*  
**by** *induct (auto simp add: field-simps abs-mult)*  
**qed** *auto*

**lemma** *setprod-constant*: *finite A  $\implies (\prod x \in A. (y :: 'a :: \{\text{comm-monoid-mult}\}))$*

*= y<sup>^(card A)</sup>*  
**apply** (*erule finite-induct*)  
**apply** *auto*  
**done**

```

lemma setprod-gen-delta:
  assumes fS: finite S
  shows setprod ( $\lambda k. \text{if } k=a \text{ then } b \text{ } k \text{ else } c$ ) S = ( $\text{if } a \in S \text{ then } (b \text{ } a :: 'a :: \{\text{comm-monoid-mult}\})$ 
    *  $c^{\wedge} (\text{card } S - 1) \text{ else } c^{\wedge} \text{card } S$ )
  proof –
    let ?f = ( $\lambda k. \text{if } k=a \text{ then } b \text{ } k \text{ else } c$ )
    {assume a:  $a \notin S$ 
      hence  $\forall k \in S. ?f \text{ } k = c$  by simp
      hence ?thesis using a setprod-constant[OF fS, of c] by (simp add: setprod-1
        cong add: setprod-cong) }
    moreover
    {assume a:  $a \in S$ 
      let ?A =  $S - \{a\}$ 
      let ?B =  $\{a\}$ 
      have eq:  $S = ?A \cup ?B$  using a by blast
      have dj:  $?A \cap ?B = \{\}$  by simp
      from fS have fAB: finite ?A finite ?B by auto
      have fA0: setprod ?f ?A = setprod ( $\lambda i. c$ ) ?A
      apply (rule setprod-cong) by auto
      have cA:  $\text{card } ?A = \text{card } S - 1$  using fS a by auto
      have fA1: setprod ?f ?A =  $c^{\wedge} \text{card } ?A$  unfolding fA0 apply (rule setprod-constant)
    using fS by auto
      have setprod ?f ?A * setprod ?f ?B = setprod ?f S
      using setprod-Un-disjoint[OF fAB dj, of ?f, unfolded eq[symmetric]]
      by simp
      then have ?thesis using a cA
      by (simp add: fA1 field-simps cong add: setprod-cong cong del: if-weak-cong)}
    ultimately show ?thesis by blast
  qed

```

## 28.4 Versions of *inf* and *sup* on non-empty sets

**no-notation** times (infixl \* 70)

**no-notation** Groups.one (1)

**locale** semilattice-big = semilattice +

**fixes** F :: 'a set  $\Rightarrow$  'a

**assumes** F-eq: finite A  $\Longrightarrow$  F A = fold1 (op \*) A

**sublocale** semilattice-big < folding-one-idem **proof**

**qed** (simp-all add: F-eq)

**notation** times (infixl \* 70)

**notation** Groups.one (1)

**context** lattice

**begin**

**definition**  $\text{Inf-fin} :: 'a \text{ set} \Rightarrow 'a \ (\prod_{fin} [900] \ 900)$  **where**  
 $\text{Inf-fin} = \text{fold1 inf}$

**definition**  $\text{Sup-fin} :: 'a \text{ set} \Rightarrow 'a \ (\sqcup_{fin} [900] \ 900)$  **where**  
 $\text{Sup-fin} = \text{fold1 sup}$

**end**

**sublocale**  $\text{lattice} < \text{Inf-fin}!$ : *semilattice-big inf Inf-fin* **proof**  
**qed** (*simp add: Inf-fin-def*)

**sublocale**  $\text{lattice} < \text{Sup-fin}!$ : *semilattice-big sup Sup-fin* **proof**  
**qed** (*simp add: Sup-fin-def*)

**context** *semilattice-inf*  
**begin**

**lemma** *ab-semigroup-idem-mult-inf*:  
*class.ab-semigroup-idem-mult inf*  
**proof** **qed** (*rule inf-assoc inf-commute inf-idem*)**+**

**lemma** *fold-inf-insert[simp]*: *finite A  $\implies \text{fold inf b (insert a A) = inf a (fold inf b A)$*   
**by**(*rule fun-left-comm-idem.fold-insert-idem[OF ab-semigroup-idem-mult.fun-left-comm-idem[OF ab-semigroup-idem-mult-inf]]*)

**lemma** *inf-le-fold-inf*: *finite A  $\implies \text{ALL } a:A. b \leq a \implies \text{inf b c} \leq \text{fold inf c A}$*   
**by** (*induct pred: finite*) (*auto intro: le-infI1*)

**lemma** *fold-inf-le-inf*: *finite A  $\implies a \in A \implies \text{fold inf b A} \leq \text{inf a b}$*   
**proof**(*induct arbitrary: a pred:finite*)  
  *case empty thus ?case by simp*  
**next**  
  *case (insert x A)*  
  **show** *?case*  
  **proof** *cases*  
    *assume A = {} thus ?thesis using insert by simp*  
  **next**  
    *assume A  $\neq \{\}$  thus ?thesis using insert by (auto intro: le-infI2)*  
  **qed**  
**qed**

**lemma** *below-fold1-iff*:  
  *assumes finite A A  $\neq \{\}$*   
  *shows  $x \leq \text{fold1 inf A} \longleftrightarrow (\forall a \in A. x \leq a)$*   
**proof** –  
  **interpret** *ab-semigroup-idem-mult inf*  
  **by** (*rule ab-semigroup-idem-mult-inf*)  
  **show** *?thesis using assms by (induct rule: finite-ne-induct) simp-all*

qed

lemma *fold1-belowI*:

assumes *finite A*

and  $a \in A$

shows  $\text{fold1 inf } A \leq a$

proof –

from *assms* have  $A \neq \{\}$  by *auto*

from  $\langle \text{finite } A \rangle \langle A \neq \{\} \rangle \langle a \in A \rangle$  show *?thesis*

proof (induct rule: *finite-ne-induct*)

case *singleton* thus *?case* by *simp*

next

interpret *ab-semigroup-idem-mult inf*

by (rule *ab-semigroup-idem-mult-inf*)

case (insert  $x F$ )

from *insert*(5) have  $a = x \vee a \in F$  by *simp*

thus *?case*

proof

assume  $a = x$  thus *?thesis* using *insert*

by (*simp add: mult-ac*)

next

assume  $a \in F$

hence *bel*:  $\text{fold1 inf } F \leq a$  by (rule *insert*)

have  $\text{inf } (\text{fold1 inf } (\text{insert } x F)) a = \text{inf } x (\text{inf } (\text{fold1 inf } F) a)$

using *insert* by (*simp add: mult-ac*)

also have  $\text{inf } (\text{fold1 inf } F) a = \text{fold1 inf } F$

using *bel* by (*auto intro: antisym*)

also have  $\text{inf } x \dots = \text{fold1 inf } (\text{insert } x F)$

using *insert* by (*simp add: mult-ac*)

finally have *aux*:  $\text{inf } (\text{fold1 inf } (\text{insert } x F)) a = \text{fold1 inf } (\text{insert } x F)$  .

moreover have  $\text{inf } (\text{fold1 inf } (\text{insert } x F)) a \leq a$  by *simp*

ultimately show *?thesis* by *simp*

qed

qed

qed

end

context *semilattice-sup*

begin

lemma *ab-semigroup-idem-mult-sup*: *class.ab-semigroup-idem-mult sup*

by (rule *semilattice-inf.ab-semigroup-idem-mult-inf*)(rule *dual-semilattice*)

lemma *fold-sup-insert[simp]*:  $\text{finite } A \implies \text{fold sup } b (\text{insert } a A) = \text{sup } a (\text{fold sup } b A)$

by (rule *semilattice-inf.fold-inf-insert*)(rule *dual-semilattice*)

lemma *fold-sup-le-sup*:  $\text{finite } A \implies \text{ALL } a:A. a \leq b \implies \text{fold sup } c A \leq \text{sup } b c$



**by**(*rule semilattice-inf.fold1-le-fold1-inf*)(*rule dual-semilattice*)

**lemma** *sup-le-fold1-sup*: *finite A*  $\implies a \in A \implies \sup a \leq \text{fold1 } \sup \text{ } A$   
**by**(*rule semilattice-inf.fold1-le-fold1-inf*)(*rule dual-semilattice*)

**end**

**context** *lattice*  
**begin**

**lemma** *Inf-le-Sup* [*simp*]:  $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \bigcap_{fin} A \leq \bigcup_{fin} A$   
**apply**(*unfold Sup-fin-def Inf-fin-def*)  
**apply**(*subgoal-tac EX a. a:A*)  
**prefer** 2 **apply** *blast*  
**apply**(*erule exE*)  
**apply**(*rule order-trans*)  
**apply**(*erule (1) fold1-belowI*)  
**apply**(*erule (1) semilattice-inf.fold1-belowI [OF dual-semilattice]*)  
**done**

**lemma** *sup-Inf-absorb* [*simp*]:  
*finite A*  $\implies a \in A \implies \sup a (\bigcap_{fin} A) = a$   
**apply**(*subst sup-commute*)  
**apply**(*simp add: Inf-fin-def sup-absorb2 fold1-belowI*)  
**done**

**lemma** *inf-Sup-absorb* [*simp*]:  
*finite A*  $\implies a \in A \implies \inf a (\bigcup_{fin} A) = a$   
**by** (*simp add: Sup-fin-def inf-absorb1*  
*semilattice-inf.fold1-belowI [OF dual-semilattice]*)

**end**

**context** *distrib-lattice*  
**begin**

**lemma** *sup-Inf1-distrib*:  
**assumes** *finite A*  
**and**  $A \neq \{\}$   
**shows**  $\sup x (\bigcap_{fin} A) = \bigcap_{fin} \{\sup x a \mid a. a \in A\}$   
**proof** –  
**interpret** *ab-semigroup-idem-mult inf*  
**by** (*rule ab-semigroup-idem-mult-inf*)  
**from** *assms* **show** *?thesis*  
**by** (*simp add: Inf-fin-def image-def*  
*hom-fold1-commute [where h=sup x, OF sup-inf-distrib1]*)  
*(rule arg-cong [where f=fold1 inf], blast)*  
**qed**

**lemma** *sup-Inf2-distrib*:  
**assumes**  $A$ : *finite*  $A$   $A \neq \{\}$  **and**  $B$ : *finite*  $B$   $B \neq \{\}$   
**shows**  $\sup (\bigcap_{fin} A) (\bigcap_{fin} B) = \bigcap_{fin} \{\sup a \ b \mid a \in A \wedge b \in B\}$   
**using**  $A$  **proof** (*induct rule: finite-ne-induct*)  
**case** *singleton* **thus** ?case  
**by** (*simp add: sup-Inf1-distrib [OF B] fold1-singleton-def [OF Inf-fin-def]*)  
**next**  
**interpret** *ab-semigroup-idem-mult inf*  
**by** (*rule ab-semigroup-idem-mult-inf*)  
**case** (*insert x A*)  
**have**  $finB$ : *finite*  $\{\sup x \ b \mid b \in B\}$   
**by**(*rule finite-surj[where f = sup x, OF B(1)], auto*)  
**have**  $finAB$ : *finite*  $\{\sup a \ b \mid a \in A \wedge b \in B\}$   
**proof** –  
**have**  $\{\sup a \ b \mid a \in A \wedge b \in B\} = (UN \ a:A. UN \ b:B. \{\sup a \ b\})$   
**by** *blast*  
**thus** ?thesis **by**(*simp add: insert(1) B(1)*)  
**qed**  
**have**  $ne$ :  $\{\sup a \ b \mid a \in A \wedge b \in B\} \neq \{\}$  **using** *insert B* **by** *blast*  
**have**  $\sup (\bigcap_{fin} (insert \ x \ A)) (\bigcap_{fin} B) = \sup (inf \ x \ (\bigcap_{fin} A)) (\bigcap_{fin} B)$   
**using** *insert* **by** (*simp add: fold1-insert-idem-def [OF Inf-fin-def]*)  
**also have**  $\dots = inf (\sup x \ (\bigcap_{fin} B)) (\sup (\bigcap_{fin} A) (\bigcap_{fin} B))$  **by**(*rule sup-inf-distrib2*)  
**also have**  $\dots = inf (\bigcap_{fin} \{\sup x \ b \mid b \in B\}) (\bigcap_{fin} \{\sup a \ b \mid a \in A \wedge b \in B\})$   
**using** *insert* **by**(*simp add: sup-Inf1-distrib[OF B]*)  
**also have**  $\dots = \bigcap_{fin} (\{\sup x \ b \mid b \in B\} \cup \{\sup a \ b \mid a \in A \wedge b \in B\})$   
**(is - =  $\bigcap_{fin} ?M$ )**  
**using**  $B$  *insert*  
**by** (*simp add: Inf-fin-def fold1-Un2 [OF finB - finAB ne]*)  
**also have**  $?M = \{\sup a \ b \mid a \in insert \ x \ A \wedge b \in B\}$   
**by** *blast*  
**finally show** ?case .  
**qed**

**lemma** *inf-Sup1-distrib*:  
**assumes** *finite*  $A$  **and**  $A \neq \{\}$   
**shows**  $inf \ x \ (\bigcup_{fin} A) = \bigcup_{fin} \{inf \ x \ a \mid a \in A\}$   
**proof** –  
**interpret** *ab-semigroup-idem-mult sup*  
**by** (*rule ab-semigroup-idem-mult-sup*)  
**from** *assms* **show** ?thesis  
**by** (*simp add: Sup-fin-def image-def hom-fold1-commute [where h=inf x, OF inf-sup-distrib1]*)  
*(rule arg-cong [where f=fold1 sup], blast)*  
**qed**

**lemma** *inf-Sup2-distrib*:  
**assumes**  $A$ : *finite*  $A$   $A \neq \{\}$  **and**  $B$ : *finite*  $B$   $B \neq \{\}$   
**shows**  $inf (\bigcup_{fin} A) (\bigcup_{fin} B) = \bigcup_{fin} \{inf \ a \ b \mid a \in A \wedge b \in B\}$

```

using  $A$  proof (induct rule: finite-ne-induct)
  case singleton thus ?case
    by(simp add: inf-Sup1-distrib [OF B] fold1-singleton-def [OF Sup-fin-def])
next
  case (insert x A)
  have finB: finite {inf x b | b. b ∈ B}
    by(rule finite-surj[where f = %b. inf x b, OF B(1)], auto)
  have finAB: finite {inf a b | a b. a ∈ A ∧ b ∈ B}
  proof –
    have {inf a b | a b. a ∈ A ∧ b ∈ B} = (UN a:A. UN b:B. {inf a b})
      by blast
    thus ?thesis by(simp add: insert(1) B(1))
  qed
  have ne: {inf a b | a b. a ∈ A ∧ b ∈ B} ≠ {} using insert B by blast
  interpret ab-semigroup-idem-mult sup
    by (rule ab-semigroup-idem-mult-sup)
  have inf ( $\bigsqcup_{fin}(\text{insert } x \ A)$ ) ( $\bigsqcup_{fin} B$ ) = inf (sup x ( $\bigsqcup_{fin} A$ )) ( $\bigsqcup_{fin} B$ )
    using insert by (simp add: fold1-insert-idem-def [OF Sup-fin-def])
  also have ... = sup (inf x ( $\bigsqcup_{fin} B$ )) (inf ( $\bigsqcup_{fin} A$ ) ( $\bigsqcup_{fin} B$ )) by(rule inf-sup-distrib2)
  also have ... = sup ( $\bigsqcup_{fin}\{\text{inf } x \ b | b. b \in B\}$ ) ( $\bigsqcup_{fin}\{\text{inf } a \ b | a \ b. a \in A \wedge b \in B\}$ )
  using insert by(simp add: inf-Sup1-distrib[OF B])
  also have ... =  $\bigsqcup_{fin}(\{\text{inf } x \ b | b. b \in B\} \cup \{\text{inf } a \ b | a \ b. a \in A \wedge b \in B\})$ 
    (is - =  $\bigsqcup_{fin} ?M$ )
    using B insert
    by (simp add: Sup-fin-def fold1-Un2 [OF finB - finAB ne])
  also have ?M = {inf a b | a b. a ∈ insert x A ∧ b ∈ B}
    by blast
  finally show ?case .
qed

end

context complete-lattice
begin

lemma Inf-fin-Inf:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\bigsqcap_{fin} A = \text{Inf } A$ 
proof –
  interpret ab-semigroup-idem-mult inf
    by (rule ab-semigroup-idem-mult-inf)
  from  $\langle A \neq \{\} \rangle$  obtain b B where  $A = \text{insert } b \ B$  by auto
  moreover with  $\langle \text{finite } A \rangle$  have finite B by simp
  ultimately show ?thesis
    by (simp add: Inf-fin-def fold1-eq-fold-idem inf-Inf-fold-inf [symmetric])
    (simp add: Inf-fold-inf)
qed

```

```

lemma Sup-fin-Sup:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\bigsqcup_{fin} A = Sup\ A$ 
proof –
  interpret ab-semigroup-idem-mult sup
    by (rule ab-semigroup-idem-mult-sup)
  from  $\langle A \neq \{\} \rangle$  obtain  $b\ B$  where  $A = insert\ b\ B$  by auto
  moreover with  $\langle finite\ A \rangle$  have finite B by simp
  ultimately show ?thesis
  by (simp add: Sup-fin-def fold1-eq-fold-idem sup-Sup-fold-sup [symmetric])
    (simp add: Sup-fold-sup)
qed

end

```

## 28.5 Versions of *min* and *max* on non-empty sets

```

definition (in linorder) Min :: 'a set  $\Rightarrow$  'a where
  Min = fold1 min

```

```

definition (in linorder) Max :: 'a set  $\Rightarrow$  'a where
  Max = fold1 max

```

```

sublocale linorder < Min!: semilattice-big min Min proof
qed (simp add: Min-def)

```

```

sublocale linorder < Max!: semilattice-big max Max proof
qed (simp add: Max-def)

```

```

context linorder
begin

```

```

lemmas Min-singleton = Min.singleton
lemmas Max-singleton = Max.singleton

```

```

lemma Min-insert:
  assumes finite A and  $A \neq \{\}$ 
  shows  $Min\ (insert\ x\ A) = min\ x\ (Min\ A)$ 
  using assms by simp

```

```

lemma Max-insert:
  assumes finite A and  $A \neq \{\}$ 
  shows  $Max\ (insert\ x\ A) = max\ x\ (Max\ A)$ 
  using assms by simp

```

```

lemma Min-Un:
  assumes finite A and  $A \neq \{\}$  and finite B and  $B \neq \{\}$ 
  shows  $Min\ (A \cup B) = min\ (Min\ A)\ (Min\ B)$ 
  using assms by (rule Min.union-idem)

```

**lemma** *Max-Un*:

assumes *finite A* and  $A \neq \{\}$  and *finite B* and  $B \neq \{\}$   
 shows  $\text{Max } (A \cup B) = \max (\text{Max } A) (\text{Max } B)$   
 using *assms* by (rule *Max.union-idem*)

**lemma** *hom-Min-commute*:

assumes  $\bigwedge x y. h (\min x y) = \min (h x) (h y)$   
 and *finite N* and  $N \neq \{\}$   
 shows  $h (\text{Min } N) = \text{Min } (h ` N)$   
 using *assms* by (rule *Min.hom-commute*)

**lemma** *hom-Max-commute*:

assumes  $\bigwedge x y. h (\max x y) = \max (h x) (h y)$   
 and *finite N* and  $N \neq \{\}$   
 shows  $h (\text{Max } N) = \text{Max } (h ` N)$   
 using *assms* by (rule *Max.hom-commute*)

**lemma** *ab-semigroup-idem-mult-min*:

*class.ab-semigroup-idem-mult min*  
**proof** qed (auto simp add: *min-def*)

**lemma** *ab-semigroup-idem-mult-max*:

*class.ab-semigroup-idem-mult max*  
**proof** qed (auto simp add: *max-def*)

**lemma** *max-lattice*:

*class.semilattice-inf (op  $\geq$ ) (op  $>$ ) max*  
**by** (fact *min-max.dual-semilattice*)

**lemma** *dual-max*:

*ord.max (op  $\geq$ ) = min*  
**by** (auto simp add: *ord.max-def-raw min-def expand-fun-eq*)

**lemma** *dual-min*:

*ord.min (op  $\geq$ ) = max*  
**by** (auto simp add: *ord.min-def-raw max-def expand-fun-eq*)

**lemma** *strict-below-fold1-iff*:

assumes *finite A* and  $A \neq \{\}$   
 shows  $x < \text{fold1 min } A \longleftrightarrow (\forall a \in A. x < a)$

**proof** –

**interpret** *ab-semigroup-idem-mult min*  
**by** (rule *ab-semigroup-idem-mult-min*)  
**from** *assms* **show** ?thesis  
**by** (induct rule: *finite-ne-induct*)  
 (*simp-all* add: *fold1-insert*)

qed

```

lemma fold1-below-iff:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\text{fold1 min } A \leq x \iff (\exists a \in A. a \leq x)$ 
proof –
  interpret ab-semigroup-idem-mult min
    by (rule ab-semigroup-idem-mult-min)
  from assms show ?thesis
  by (induct rule: finite-ne-induct)
    (simp-all add: fold1-insert min-le-iff-disj)
qed

lemma fold1-strict-below-iff:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\text{fold1 min } A < x \iff (\exists a \in A. a < x)$ 
proof –
  interpret ab-semigroup-idem-mult min
    by (rule ab-semigroup-idem-mult-min)
  from assms show ?thesis
  by (induct rule: finite-ne-induct)
    (simp-all add: fold1-insert min-less-iff-disj)
qed

lemma fold1-antimono:
  assumes  $A \neq \{\}$  and  $A \subseteq B$  and finite B
  shows  $\text{fold1 min } B \leq \text{fold1 min } A$ 
proof cases
  assume  $A = B$  thus ?thesis by simp
next
  interpret ab-semigroup-idem-mult min
    by (rule ab-semigroup-idem-mult-min)
  assume  $A \neq B$ 
  have  $B: B = A \cup (B - A)$  using  $\langle A \subseteq B \rangle$  by blast
  have  $\text{fold1 min } B = \text{fold1 min } (A \cup (B - A))$  by (subst B) (rule refl)
  also have  $\dots = \min (\text{fold1 min } A) (\text{fold1 min } (B - A))$ 
  proof –
    have finite A by (rule finite-subset [OF  $\langle A \subseteq B \rangle$   $\langle \text{finite } B \rangle$ ])
    moreover have finite (B - A) by (rule finite-Diff [OF  $\langle \text{finite } B \rangle$ ])
    moreover have  $(B - A) \neq \{\}$  using prems by blast
    moreover have  $A \text{ Int } (B - A) = \{\}$  using prems by blast
    ultimately show ?thesis using  $\langle A \neq \{\} \rangle$  by (rule-tac fold1-Un)
  qed
  also have  $\dots \leq \text{fold1 min } A$  by (simp add: min-le-iff-disj)
  finally show ?thesis .
qed

lemma Min-in [simp]:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\text{Min } A \in A$ 
proof –

```

```

interpret ab-semigroup-idem-mult min
  by (rule ab-semigroup-idem-mult-min)
from assms fold1-in show ?thesis by (fastsimp simp: Min-def min-def)
qed

```

```

lemma Max-in [simp]:
  assumes finite A and A ≠ {}
  shows Max A ∈ A
proof -
  interpret ab-semigroup-idem-mult max
    by (rule ab-semigroup-idem-mult-max)
  from assms fold1-in [of A] show ?thesis by (fastsimp simp: Max-def max-def)
qed

```

```

lemma Min-le [simp]:
  assumes finite A and x ∈ A
  shows Min A ≤ x
  using assms by (simp add: Min-def min-max.fold1-belowI)

```

```

lemma Max-ge [simp]:
  assumes finite A and x ∈ A
  shows x ≤ Max A
proof -
  interpret semilattice-inf op ≥ op > max
    by (rule max-lattice)
  from assms show ?thesis by (simp add: Max-def fold1-belowI)
qed

```

```

lemma Min-ge-iff [simp, no-atp]:
  assumes finite A and A ≠ {}
  shows x ≤ Min A ⟷ (∀ a ∈ A. x ≤ a)
  using assms by (simp add: Min-def min-max.below-fold1-iff)

```

```

lemma Max-le-iff [simp, no-atp]:
  assumes finite A and A ≠ {}
  shows Max A ≤ x ⟷ (∀ a ∈ A. a ≤ x)
proof -
  interpret semilattice-inf op ≥ op > max
    by (rule max-lattice)
  from assms show ?thesis by (simp add: Max-def below-fold1-iff)
qed

```

```

lemma Min-gr-iff [simp, no-atp]:
  assumes finite A and A ≠ {}
  shows x < Min A ⟷ (∀ a ∈ A. x < a)
  using assms by (simp add: Min-def strict-below-fold1-iff)

```

```

lemma Max-less-iff [simp, no-atp]:
  assumes finite A and A ≠ {}

```

```

  shows  $\text{Max } A < x \longleftrightarrow (\forall a \in A. a < x)$ 
proof -
  interpret dual: linorder op  $\geq$  op  $>$ 
  by (rule dual-linorder)
  from assms show ?thesis
  by (simp add: Max-def dual.strict-below-fold1-iff [folded dual.dual-max])
qed

```

```

lemma Min-le-iff [no-atp]:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\text{Min } A \leq x \longleftrightarrow (\exists a \in A. a \leq x)$ 
  using assms by (simp add: Min-def fold1-below-iff)

```

```

lemma Max-ge-iff [no-atp]:
  assumes finite A and  $A \neq \{\}$ 
  shows  $x \leq \text{Max } A \longleftrightarrow (\exists a \in A. x \leq a)$ 
proof -
  interpret dual: linorder op  $\geq$  op  $>$ 
  by (rule dual-linorder)
  from assms show ?thesis
  by (simp add: Max-def dual.fold1-below-iff [folded dual.dual-max])
qed

```

```

lemma Min-less-iff [no-atp]:
  assumes finite A and  $A \neq \{\}$ 
  shows  $\text{Min } A < x \longleftrightarrow (\exists a \in A. a < x)$ 
  using assms by (simp add: Min-def fold1-strict-below-iff)

```

```

lemma Max-gr-iff [no-atp]:
  assumes finite A and  $A \neq \{\}$ 
  shows  $x < \text{Max } A \longleftrightarrow (\exists a \in A. x < a)$ 
proof -
  interpret dual: linorder op  $\geq$  op  $>$ 
  by (rule dual-linorder)
  from assms show ?thesis
  by (simp add: Max-def dual.fold1-strict-below-iff [folded dual.dual-max])
qed

```

```

lemma Min-eqI:
  assumes finite A
  assumes  $\bigwedge y. y \in A \implies y \geq x$ 
  and  $x \in A$ 
  shows  $\text{Min } A = x$ 
proof (rule antisym)
  from  $\langle x \in A \rangle$  have  $A \neq \{\}$  by auto
  with assms show  $\text{Min } A \geq x$  by simp
next
  from assms show  $x \geq \text{Min } A$  by simp
qed

```



```

lemma Max-eqI:
  assumes finite A
  assumes  $\bigwedge y. y \in A \implies y \leq x$ 
  and  $x \in A$ 
  shows  $\text{Max } A = x$ 
proof (rule antisym)
  from  $\langle x \in A \rangle$  have  $A \neq \{\}$  by auto
  with assms show  $\text{Max } A \leq x$  by simp
next
  from assms show  $x \leq \text{Max } A$  by simp
qed

lemma Min-antimono:
  assumes  $M \subseteq N$  and  $M \neq \{\}$  and finite N
  shows  $\text{Min } N \leq \text{Min } M$ 
  using assms by (simp add: Min-def fold1-antimono)

lemma Max-mono:
  assumes  $M \subseteq N$  and  $M \neq \{\}$  and finite N
  shows  $\text{Max } M \leq \text{Max } N$ 
proof -
  interpret dual: linorder op  $\geq$  op  $>$ 
  by (rule dual-linorder)
  from assms show ?thesis
  by (simp add: Max-def dual.fold1-antimono [folded dual.dual-max])
qed

lemma finite-linorder-max-induct[consumes 1, case-names empty insert]:
  assumes fin: finite A
  and empty:  $P \ \{\}$ 
  and insert:  $(!!b \ A. \text{finite } A \implies \text{ALL } a:A. a < b \implies P \ A \implies P(\text{insert } b \ A))$ 
  shows  $P \ A$ 
using fin empty insert
proof (induct rule: finite-psubset-induct)
  case (psubset A)
  have IH:  $\bigwedge B. \llbracket B < A; P \ \{\}; (\bigwedge A \ b. \llbracket \text{finite } A; \forall a \in A. a < b; P \ A \rrbracket \implies P(\text{insert } b \ A)) \rrbracket \implies P \ B$  by fact
  have fin: finite A by fact
  have empty:  $P \ \{\}$  by fact
  have step:  $\bigwedge b \ A. \llbracket \text{finite } A; \forall a \in A. a < b; P \ A \rrbracket \implies P(\text{insert } b \ A)$  by fact
  show  $P \ A$ 
  proof (cases  $A = \{\}$ )
    assume  $A = \{\}$ 
    then show  $P \ A$  using  $\langle P \ \{\} \rangle$  by simp
  next
    let ?B =  $A - \{\text{Max } A\}$ 
    let ?A =  $\text{insert } (\text{Max } A) \ ?B$ 
    have finite ?B using  $\langle \text{finite } A \rangle$  by simp

```

```

    assume  $A \neq \{\}$ 
    with  $\langle \text{finite } A \rangle$  have  $\text{Max } A : A$  by auto
    then have  $A : ?A = A$  using insert-Diff-single insert-absorb by auto
    then have  $P \ ?B$  using  $\langle P \ \{\} \rangle$  step IH[of ?B] by blast
    moreover
    have  $\forall a \in ?B. a < \text{Max } A$  using Max-ge [OF  $\langle \text{finite } A \rangle$ ] by fastsimp
    ultimately show  $P \ A$  using  $A$  insert-Diff-single step[OF  $\langle \text{finite } ?B \rangle$ ] by
fastsimp
  qed
qed

```

**lemma** *finite-linorder-min-induct*[consumes 1, case-names empty insert]:  
 $\llbracket \text{finite } A; P \ \{\}; \bigwedge b \ A. \llbracket \text{finite } A; \forall a \in A. b < a; P \ A \rrbracket \implies P \ (\text{insert } b \ A) \rrbracket \implies P \ A$   
 by (rule linorder.finite-linorder-max-induct[OF dual-linorder])

**end**

**context** *linordered-ab-semigroup-add*  
**begin**

**lemma** *add-Min-commute*:  
 fixes  $k$   
 assumes *finite*  $N$  and  $N \neq \{\}$   
 shows  $k + \text{Min } N = \text{Min } \{k + m \mid m. m \in N\}$   
**proof** –  
 have  $\bigwedge x \ y. k + \min x \ y = \min (k + x) (k + y)$   
 by (simp add: min-def not-le)  
 (blast intro: antisym less-imp-le add-left-mono)  
 with *assms* show ?thesis  
 using hom-Min-commute [of plus  $k \ N$ ]  
 by simp (blast intro: arg-cong [where  $f = \text{Min}$ ])  
**qed**

**lemma** *add-Max-commute*:  
 fixes  $k$   
 assumes *finite*  $N$  and  $N \neq \{\}$   
 shows  $k + \text{Max } N = \text{Max } \{k + m \mid m. m \in N\}$   
**proof** –  
 have  $\bigwedge x \ y. k + \max x \ y = \max (k + x) (k + y)$   
 by (simp add: max-def not-le)  
 (blast intro: antisym less-imp-le add-left-mono)  
 with *assms* show ?thesis  
 using hom-Max-commute [of plus  $k \ N$ ]  
 by simp (blast intro: arg-cong [where  $f = \text{Max}$ ])  
**qed**

**end**

**context** *linordered-ab-group-add*

**begin**

**lemma** *minus-Max-eq-Min* [simp]:

$finite\ S \implies S \neq \{\} \implies - (Max\ S) = Min\ (uminus\ 'S)$

**by** (induct *S* rule: *finite-ne-induct*) (simp-all add: *minus-max-eq-min*)

**lemma** *minus-Min-eq-Max* [simp]:

$finite\ S \implies S \neq \{\} \implies - (Min\ S) = Max\ (uminus\ 'S)$

**by** (induct *S* rule: *finite-ne-induct*) (simp-all add: *minus-min-eq-max*)

**end**

**end**

## 29 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

**theory** *Equiv-Relations*

**imports** *Big-Operators Relation Plain*

**begin**

### 29.1 Equivalence relations

**locale** *equiv* =

**fixes** *A* **and** *r*

**assumes** *refl-on*: *refl-on A r*

**and** *sym*: *sym r*

**and** *trans*: *trans r*

Suppes, Theorem 70: *r* is an equiv relation iff  $r^{-1} \circ r = r$ .

First half: *equiv A r*  $\implies r^{-1} \circ r = r$ .

**lemma** *sym-trans-comp-subset*:

$sym\ r \implies trans\ r \implies r^{-1} \circ r \subseteq r$

**by** (unfold *trans-def sym-def converse-def*) *blast*

**lemma** *refl-on-comp-subset*:  $refl-on\ A\ r \implies r \subseteq r^{-1} \circ r$

**by** (unfold *refl-on-def*) *blast*

**lemma** *equiv-comp-eq*:  $equiv\ A\ r \implies r^{-1} \circ r = r$

**apply** (unfold *equiv-def*)

**apply** *clarify*

**apply** (rule *equalityI*)

**apply** (iprover intro: *sym-trans-comp-subset refl-on-comp-subset*)

**done**

Second half.

**lemma** *comp-equivI*:

```

 $r^{-1} \circ r = r \implies \text{Domain } r = A \implies \text{equiv } A \ r$ 
apply (unfold equiv-def refl-on-def sym-def trans-def)
apply (erule equalityE)
apply (subgoal-tac  $\forall x \ y. (x, y) \in r \longrightarrow (y, x) \in r$ )
apply fast
apply fast
done

```

## 29.2 Equivalence classes

**lemma** *equiv-class-subset*:

```

 $\text{equiv } A \ r \implies (a, b) \in r \implies r''\{a\} \subseteq r''\{b\}$ 
— lemma for the next result
by (unfold equiv-def trans-def sym-def) blast

```

**theorem** *equiv-class-eq*:  $\text{equiv } A \ r \implies (a, b) \in r \implies r''\{a\} = r''\{b\}$

```

apply (assumption | rule equalityI equiv-class-subset)+
apply (unfold equiv-def sym-def)
apply blast
done

```

**lemma** *equiv-class-self*:  $\text{equiv } A \ r \implies a \in A \implies a \in r''\{a\}$

```

by (unfold equiv-def refl-on-def) blast

```

**lemma** *subset-equiv-class*:

```

 $\text{equiv } A \ r \implies r''\{b\} \subseteq r''\{a\} \implies b \in A \implies (a, b) \in r$ 
— lemma for the next result
by (unfold equiv-def refl-on-def) blast

```

**lemma** *eq-equiv-class*:

```

 $r''\{a\} = r''\{b\} \implies \text{equiv } A \ r \implies b \in A \implies (a, b) \in r$ 
by (iprover intro: equalityD2 subset-equiv-class)

```

**lemma** *equiv-class-nondisjoint*:

```

 $\text{equiv } A \ r \implies x \in (r''\{a\} \cap r''\{b\}) \implies (a, b) \in r$ 
by (unfold equiv-def trans-def sym-def) blast

```

**lemma** *equiv-type*:  $\text{equiv } A \ r \implies r \subseteq A \times A$

```

by (unfold equiv-def refl-on-def) blast

```

**theorem** *equiv-class-eq-iff*:

```

 $\text{equiv } A \ r \implies ((x, y) \in r) = (r''\{x\} = r''\{y\} \ \& \ x \in A \ \& \ y \in A)$ 
by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

```

**theorem** *eq-equiv-class-iff*:

```

 $\text{equiv } A \ r \implies x \in A \implies y \in A \implies (r''\{x\} = r''\{y\}) = ((x, y) \in r)$ 
by (blast intro!: equiv-class-eq dest: eq-equiv-class equiv-type)

```

### 29.3 Quotients

**definition** *quotient* :: 'a set  $\Rightarrow$  ('a  $\times$  'a) set  $\Rightarrow$  'a set set (infixl '/' 90) where  
 [code del]:  $A//r = (\bigcup x \in A. \{r''\{x\}\})$  — set of equiv classes

**lemma** *quotientI*:  $x \in A \implies r''\{x\} \in A//r$   
 by (unfold quotient-def) blast

**lemma** *quotientE*:  
 $X \in A//r \implies (!x. X = r''\{x\} \implies x \in A \implies P) \implies P$   
 by (unfold quotient-def) blast

**lemma** *Union-quotient*:  $\text{equiv } A \ r \implies \text{Union } (A//r) = A$   
 by (unfold equiv-def refl-on-def quotient-def) blast

**lemma** *quotient-disj*:  
 $\text{equiv } A \ r \implies X \in A//r \implies Y \in A//r \implies X = Y \mid (X \cap Y = \{\})$   
 apply (unfold quotient-def)  
 apply clarify  
 apply (rule equiv-class-eq)  
 apply assumption  
 apply (unfold equiv-def trans-def sym-def)  
 apply blast  
 done

**lemma** *quotient-eqI*:  
 $[[\text{equiv } A \ r; X \in A//r; Y \in A//r; x \in X; y \in Y; (x,y) \in r]] \implies X = Y$   
 apply (clarify elim!: quotientE)  
 apply (rule equiv-class-eq, assumption)  
 apply (unfold equiv-def sym-def trans-def, blast)  
 done

**lemma** *quotient-eq-iff*:  
 $[[\text{equiv } A \ r; X \in A//r; Y \in A//r; x \in X; y \in Y]] \implies (X = Y) = ((x,y) \in r)$   
 apply (rule iffI)  
 prefer 2 apply (blast del: equalityI intro: quotient-eqI)  
 apply (clarify elim!: quotientE)  
 apply (unfold equiv-def sym-def trans-def, blast)  
 done

**lemma** *eq-equiv-class-iff2*:  
 $[[\text{equiv } A \ r; x \in A; y \in A]] \implies (\{x\}//r = \{y\}//r) = ((x,y) : r)$   
 by(simp add:quotient-def eq-equiv-class-iff)

**lemma** *quotient-empty* [simp]:  $\{\}//r = \{\}$   
 by(simp add: quotient-def)

**lemma** *quotient-is-empty* [iff]:  $(A//r = \{\}) = (A = \{\})$

**by**(*simp add: quotient-def*)

**lemma** *quotient-is-empty2* [*iff*]:  $(\{\} = A//r) = (A = \{\})$   
**by**(*simp add: quotient-def*)

**lemma** *singleton-quotient*:  $\{x\} // r = \{r \text{ `` } \{x\}\}$   
**by**(*simp add: quotient-def*)

**lemma** *quotient-diff1*:  
 $\llbracket \text{inj-on } (\%a. \{a\} // r) \ A; \ a \in A \rrbracket \implies (A - \{a\}) // r = A // r - \{a\} // r$   
**apply**(*simp add: quotient-def inj-on-def*)  
**apply** *blast*  
**done**

## 29.4 Defining unary operations upon equivalence classes

A congruence-preserving function

**locale** *congruent* =  
**fixes** *r* **and** *f*  
**assumes** *congruent*:  $(y, z) \in r \implies f\ y = f\ z$

**abbreviation**  
 $RESPECTS :: ('a \implies 'b) \implies ('a * 'a) \text{ set} \implies \text{bool}$   
 $(\text{infixr } respects\ 80) \text{ where}$   
 $f\ respects\ r == \text{congruent } r\ f$

**lemma** *UN-constant-eq*:  $a \in A \implies \forall y \in A. f\ y = c \implies (\bigcup y \in A. f(y)) = c$   
— lemma required to prove *UN-equiv-class*  
**by** *auto*

**lemma** *UN-equiv-class*:  
 $\text{equiv } A\ r \implies f\ respects\ r \implies a \in A$   
 $\implies (\bigcup x \in r \text{ `` } \{a\}. f\ x) = f\ a$   
— Conversion rule  
**apply** (*rule equiv-class-self [THEN UN-constant-eq], assumption+*)  
**apply** (*unfold equiv-def congruent-def sym-def*)  
**apply** (*blast del: equalityI*)  
**done**

**lemma** *UN-equiv-class-type*:  
 $\text{equiv } A\ r \implies f\ respects\ r \implies X \in A // r \implies$   
 $(!!x. x \in A \implies f\ x \in B) \implies (\bigcup x \in X. f\ x) \in B$   
**apply** (*unfold quotient-def*)  
**apply** *clarify*  
**apply** (*subst UN-equiv-class*)  
**apply** *auto*  
**done**

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be  $!!y. y \in A \implies f y \in B$ .

**lemma** *UN-equiv-class-inject*:

```
equiv A r ==> f respects r ==>
  (⋃ x ∈ X. f x) = (⋃ y ∈ Y. f y) ==> X ∈ A//r ==> Y ∈ A//r
  ==> (!x y. x ∈ A ==> y ∈ A ==> f x = f y ==> (x, y) ∈ r)
  ==> X = Y
```

**apply** (*unfold quotient-def*)

**apply** *clarify*

**apply** (*rule equiv-class-eq*)

**apply** *assumption*

**apply** (*subgoal-tac f x = f xa*)

**apply** *blast*

**apply** (*erule box-equals*)

**apply** (*assumption | rule UN-equiv-class*) +

**done**

## 29.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

**locale** *congruent2* =

**fixes** *r1 and r2 and f*

**assumes** *congruent2*:

$(y1, z1) \in r1 \implies (y2, z2) \in r2 \implies f y1 y2 = f z1 z2$

Abbreviation for the common case where the relations are identical

**abbreviation**

*RESPECTS2*::  $[ 'a \Rightarrow 'a \Rightarrow 'b, ('a * 'a) \text{ set} ] \Rightarrow \text{bool}$

(**infixr** *respects2* 80) **where**

*f respects2 r == congruent2 r r f*

**lemma** *congruent2-implies-congruent*:

$\text{equiv } A \ r1 \implies \text{congruent2 } r1 \ r2 \ f \implies a \in A \implies \text{congruent } r2 \ (f \ a)$

**by** (*unfold congruent-def congruent2-def equiv-def refl-on-def*) *blast*

**lemma** *congruent2-implies-congruent-UN*:

$\text{equiv } A1 \ r1 \implies \text{equiv } A2 \ r2 \implies \text{congruent2 } r1 \ r2 \ f \implies a \in A2 \implies$

$\text{congruent } r1 \ (\lambda x1. \bigcup x2 \in r2^{\{a\}}. f \ x1 \ x2)$

**apply** (*unfold congruent-def*)

**apply** *clarify*

**apply** (*rule equiv-type [THEN subsetD, THEN SigmaE2], assumption+*)

**apply** (*simp add: UN-equiv-class congruent2-implies-congruent*)

**apply** (*unfold congruent2-def equiv-def refl-on-def*)

**apply** (*blast del: equalityI*)

**done**

**lemma** *UN-equiv-class2*:

```

equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a1 ∈ A1 ==> a2
∈ A2
==> (⋃ x1 ∈ r1“{a1}. ⋃ x2 ∈ r2“{a2}. f x1 x2) = f a1 a2
by (simp add: UN-equiv-class congruent2-implies-congruent
congruent2-implies-congruent-UN)

```

**lemma** *UN-equiv-class-type2*:

```

equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f
==> X1 ∈ A1//r1 ==> X2 ∈ A2//r2
==> (!!x1 x2. x1 ∈ A1 ==> x2 ∈ A2 ==> f x1 x2 ∈ B)
==> (⋃ x1 ∈ X1. ⋃ x2 ∈ X2. f x1 x2) ∈ B
apply (unfold quotient-def)
apply clarify
apply (blast intro: UN-equiv-class-type congruent2-implies-congruent-UN
congruent2-implies-congruent quotientI)
done

```

**lemma** *UN-UN-split-split-eq*:

```

(⋃ (x1, x2) ∈ X. ⋃ (y1, y2) ∈ Y. A x1 x2 y1 y2) =
(⋃ x ∈ X. ⋃ y ∈ Y. (λ(x1, x2). (λ(y1, y2). A x1 x2 y1 y2) y) x)
— Allows a natural expression of binary operators,
— without explicit calls to split
by auto

```

**lemma** *congruent2I*:

```

equiv A1 r1 ==> equiv A2 r2
==> (!!y z w. w ∈ A2 ==> (y,z) ∈ r1 ==> f y w = f z w)
==> (!!y z w. w ∈ A1 ==> (y,z) ∈ r2 ==> f w y = f w z)
==> congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— much simpler than the direct proof.
apply (unfold congruent2-def equiv-def refl-on-def)
apply clarify
apply (blast intro: trans)
done

```

**lemma** *congruent2-commuteI*:

```

assumes equivA: equiv A r
and commute: !!y z. y ∈ A ==> z ∈ A ==> f y z = f z y
and cong: !!y z w. w ∈ A ==> (y,z) ∈ r ==> f w y = f w z
shows f respects2 r
apply (rule congruent2I [OF equivA equivA])
apply (rule commute [THEN trans])
apply (rule-tac [3] commute [THEN trans, symmetric])
apply (rule-tac [5] sym)
apply (rule cong | assumption |
erule equivA [THEN equiv-type, THEN subsetD, THEN SigmaE2])
done

```



## 29.6 Quotients and finiteness

Suggested by Florian Kammüller

```

lemma finite-quotient: finite  $A \implies r \subseteq A \times A \implies \text{finite } (A/r)$ 
  — recall equiv  $?A \ ?r \implies ?r \subseteq ?A \times ?A$ 
  apply (rule finite-subset)
    apply (erule-tac [2] finite-Pow-iff [THEN iffD2])
  apply (unfold quotient-def)
  apply blast
done

lemma finite-equiv-class:
  finite  $A \implies r \subseteq A \times A \implies X \in A/r \implies \text{finite } X$ 
  apply (unfold quotient-def)
  apply (rule finite-subset)
  prefer 2 apply assumption
  apply blast
done

lemma equiv-imp-dvd-card:
  finite  $A \implies \text{equiv } A \ r \implies \forall X \in A/r. k \text{ dvd card } X$ 
   $\implies k \text{ dvd card } A$ 
  apply (rule Union-quotient [THEN subst [where  $P = \lambda A. k \text{ dvd card } A$ ]])
  apply assumption
  apply (rule dvd-partition)
  prefer 3 apply (blast dest: quotient-disj)
  apply (simp-all add: Union-quotient equiv-type)
done

lemma card-quotient-disjoint:
   $\llbracket \text{finite } A; \text{inj-on } (\lambda x. \{x\} // r) \ A \rrbracket \implies \text{card}(A/r) = \text{card } A$ 
  apply(simp add:quotient-def)
  apply(subst card-UN-disjoint)
  apply assumption
  apply simp
  apply(fastsimp simp add:inj-on-def)
  apply simp
done

end

```

## 30 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

```

theory Int
imports Equiv-Relations Nat Wellfounded
uses

```

```

(Tools/numeral.ML)
(Tools/numeral-syntax.ML)
(Tools/int-arith.ML)
begin

```

### 30.1 The equivalence relation underlying the integers

**definition** *intrel* ::  $((\text{nat} \times \text{nat}) \times (\text{nat} \times \text{nat})) \text{ set}$  **where**  
 $\text{[code del]: } \text{intrel} = \{(x, y), (u, v) \mid x \text{ y } u \text{ v. } x + v = u + y \}$

**typedef** (*Integ*)  
*int* = *UNIV* // *intrel*  
**by** (*auto simp add: quotient-def*)

**instantiation** *int* ::  $\{\text{zero}, \text{one}, \text{plus}, \text{minus}, \text{uminus}, \text{times}, \text{ord}, \text{abs}, \text{sgn}\}$   
**begin**

**definition**  
*Zero-int-def*  $\text{[code del]: } 0 = \text{Abs-Integ } (\text{intrel } “ \{(0, 0)\} )$

**definition**  
*One-int-def*  $\text{[code del]: } 1 = \text{Abs-Integ } (\text{intrel } “ \{(1, 0)\} )$

**definition**  
*add-int-def*  $\text{[code del]: } z + w = \text{Abs-Integ}$   
 $(\bigcup (x, y) \in \text{Rep-Integ } z. \bigcup (u, v) \in \text{Rep-Integ } w.$   
 $\text{intrel } “ \{(x + u, y + v)\})$

**definition**  
*minus-int-def*  $\text{[code del]:}$   
 $- z = \text{Abs-Integ } (\bigcup (x, y) \in \text{Rep-Integ } z. \text{intrel } “ \{(y, x)\})$

**definition**  
*diff-int-def*  $\text{[code del]: } z - w = z + (-w :: \text{int})$

**definition**  
*mult-int-def*  $\text{[code del]: } z * w = \text{Abs-Integ}$   
 $(\bigcup (x, y) \in \text{Rep-Integ } z. \bigcup (u, v) \in \text{Rep-Integ } w.$   
 $\text{intrel } “ \{(x*u + y*v, x*v + y*u)\})$

**definition**  
*le-int-def*  $\text{[code del]:}$   
 $z \leq w \longleftrightarrow (\exists x \text{ y } u \text{ v. } x + v \leq u + y \wedge (x, y) \in \text{Rep-Integ } z \wedge (u, v) \in \text{Rep-Integ } w)$

**definition**  
*less-int-def*  $\text{[code del]: } (z :: \text{int}) < w \longleftrightarrow z \leq w \wedge z \neq w$

**definition**

*zabs-def*:  $|i::int| = (\text{if } i < 0 \text{ then } -i \text{ else } i)$

**definition**

*zsgn-def*:  $\text{sgn } (i::int) = (\text{if } i=0 \text{ then } 0 \text{ else if } 0 < i \text{ then } 1 \text{ else } -1)$

**instance ..**

**end**

### 30.2 Construction of the Integers

**lemma** *intrel-iff* [simp]:  $((x,y),(u,v)) \in \text{intrel} \Rightarrow (x+v = u+y)$   
**by** (simp add: intrel-def)

**lemma** *equiv-intrel*: *equiv UNIV intrel*

**by** (simp add: intrel-def equiv-def refl-on-def sym-def trans-def)

Reduces equality of equivalence classes to the *intrel* relation:  $(\text{intrel} \text{ “ } \{x\} = \text{intrel} \text{ “ } \{y\}) = ((x, y) \in \text{intrel})$

**lemmas** *equiv-intrel-iff* [simp] = *eq-equiv-class-iff* [OF *equiv-intrel UNIV-I UNIV-I*]

All equivalence classes belong to set of representatives

**lemma** [simp]:  $\text{intrel} \text{ “ } \{(x,y)\} \in \text{Integ}$

**by** (auto simp add: Integ-def intrel-def quotient-def)

Reduces equality on abstractions to equality on representatives:  $\llbracket x \in \text{Integ}; y \in \text{Integ} \rrbracket \implies (\text{Abs-Integ } x = \text{Abs-Integ } y) = (x = y)$

**declare** *Abs-Integ-inject* [simp,no-atp] *Abs-Integ-inverse* [simp,no-atp]

Case analysis on the representation of an integer as an equivalence class of pairs of naturals.

**lemma** *eq-Abs-Integ* [case-names *Abs-Integ*, cases type: *int*]:

$(!!x \ y. z = \text{Abs-Integ}(\text{intrel} \text{ “ } \{(x,y)\}) \implies P) \implies P$

**apply** (rule *Abs-Integ-cases* [of *z*])

**apply** (auto simp add: Integ-def quotient-def)

**done**

### 30.3 Arithmetic Operations

**lemma** *minus*:  $-\text{Abs-Integ}(\text{intrel} \text{ “ } \{(x,y)\}) = \text{Abs-Integ}(\text{intrel} \text{ “ } \{(y,x)\})$

**proof** –

**have**  $(\lambda(x,y). \text{intrel} \text{ “ } \{(y,x)\}) \text{ respects intrel}$

**by** (simp add: congruent-def)

**thus** ?thesis

**by** (simp add: minus-int-def UN-equiv-class [OF *equiv-intrel*])

**qed**

**lemma** *add*:

```

    Abs-Integ (intrel“(x,y)”) + Abs-Integ (intrel“(u,v)”) =
    Abs-Integ (intrel“(x+u, y+v)”)
  proof -
    have (λz w. (λ(x,y). (λ(u,v). intrel“(x+u, y+v)”) w) z)
      respects2 intrel
    by (simp add: congruent2-def)
  thus ?thesis
    by (simp add: add-int-def UN-UN-split-split-eq
      UN-equiv-class2 [OF equiv-intrel equiv-intrel])
qed

```

Congruence property for multiplication

```

lemma mult-congruent2:
  (%p1 p2. (%(x,y). (%(u,v). intrel“(x*u + y*v, x*v + y*u)”) p2) p1)
  respects2 intrel
apply (rule equiv-intrel [THEN congruent2-commuteI])
apply (force simp add: mult-ac, clarify)
apply (simp add: congruent-def mult-ac)
apply (rename-tac u v w x y z)
apply (subgoal-tac u*y + x*y = w*y + v*y & u*z + x*z = w*z + v*z)
apply (simp add: mult-ac)
apply (simp add: add-mult-distrib [symmetric])
done

```

```

lemma mult:
  Abs-Integ((intrel“(x,y)”)) * Abs-Integ((intrel“(u,v)”)) =
  Abs-Integ(intrel“(x*u + y*v, x*v + y*u)”)
by (simp add: mult-int-def UN-UN-split-split-eq mult-congruent2
  UN-equiv-class2 [OF equiv-intrel equiv-intrel])

```

The integers form a *comm-ring-1*

```

instance int :: comm-ring-1
proof
  fix i j k :: int
  show (i + j) + k = i + (j + k)
    by (cases i, cases j, cases k) (simp add: add-add-assoc)
  show i + j = j + i
    by (cases i, cases j) (simp add: add-ac add)
  show 0 + i = i
    by (cases i) (simp add: Zero-int-def add)
  show - i + i = 0
    by (cases i) (simp add: Zero-int-def minus add)
  show i - j = i + - j
    by (simp add: diff-int-def)
  show (i * j) * k = i * (j * k)
    by (cases i, cases j, cases k) (simp add: mult-algebra-simps)
  show i * j = j * i
    by (cases i, cases j) (simp add: mult-algebra-simps)
  show 1 * i = i

```

```

    by (cases i) (simp add: One-int-def mult)
  show (i + j) * k = i * k + j * k
    by (cases i, cases j, cases k) (simp add: add mult algebra-simps)
  show 0 ≠ (1::int)
    by (simp add: Zero-int-def One-int-def)
qed

```

```

lemma int-def: of-nat m = Abs-Integ (intrel “ {(m, 0)}”)
by (induct m, simp-all add: Zero-int-def One-int-def add)

```

### 30.4 The $\leq$ Ordering

```

lemma le:
  (Abs-Integ(intrel“{(x,y)}”) ≤ Abs-Integ(intrel“{(u,v)}”)) = (x+v ≤ u+y)
by (force simp add: le-int-def)

```

```

lemma less:
  (Abs-Integ(intrel“{(x,y)}”) < Abs-Integ(intrel“{(u,v)}”)) = (x+v < u+y)
by (simp add: less-int-def le order-less-le)

```

```

instance int :: linorder

```

```

proof
  fix i j k :: int
  show antisym: i ≤ j ⟹ j ≤ i ⟹ i = j
    by (cases i, cases j) (simp add: le)
  show (i < j) = (i ≤ j ∧ ¬ j ≤ i)
    by (auto simp add: less-int-def dest: antisym)
  show i ≤ i
    by (cases i) (simp add: le)
  show i ≤ j ⟹ j ≤ k ⟹ i ≤ k
    by (cases i, cases j, cases k) (simp add: le)
  show i ≤ j ∨ j ≤ i
    by (cases i, cases j) (simp add: le linorder-linear)
qed

```

```

instantiation int :: distrib-lattice
begin

```

```

definition
  (inf :: int ⇒ int ⇒ int) = min

```

```

definition
  (sup :: int ⇒ int ⇒ int) = max

```

```

instance
  by intro-classes
  (auto simp add: inf-int-def sup-int-def min-max.sup-inf-distrib1)

```

```

end

```

```

instance int :: ordered-cancel-ab-semigroup-add
proof
  fix i j k :: int
  show  $i \leq j \implies k + i \leq k + j$ 
    by (cases i, cases j, cases k) (simp add: le add)
qed

```

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on  $k \geq 0$

```

lemma zmult-zless-mono2-lemma:
   $(i::int) < j \implies 0 < k \implies \text{of-nat } k * i < \text{of-nat } k * j$ 
apply (induct k, simp)
apply (simp add: left-distrib)
apply (case-tac k=0)
apply (simp-all add: add-strict-mono)
done

```

```

lemma zero-le-imp-eq-int:  $(0::int) \leq k \implies \exists n. k = \text{of-nat } n$ 
apply (cases k)
apply (auto simp add: le add int-def Zero-int-def)
apply (rule-tac  $x=x-y$  in exI, simp)
done

```

```

lemma zero-less-imp-eq-int:  $(0::int) < k \implies \exists n > 0. k = \text{of-nat } n$ 
apply (cases k)
apply (simp add: less int-def Zero-int-def)
apply (rule-tac  $x=x-y$  in exI, simp)
done

```

```

lemma zmult-zless-mono2:  $[i < j; (0::int) < k] \implies k*i < k*j$ 
apply (drule zero-less-imp-eq-int)
apply (auto simp add: zmult-zless-mono2-lemma)
done

```

The integers form an ordered integral domain

```

instance int :: linordered-idom
proof
  fix i j k :: int
  show  $i < j \implies 0 < k \implies k * i < k * j$ 
    by (rule zmult-zless-mono2)
  show  $|i| = (\text{if } i < 0 \text{ then } -i \text{ else } i)$ 
    by (simp only: zabs-def)
  show  $\text{sgn } (i::int) = (\text{if } i=0 \text{ then } 0 \text{ else if } 0 < i \text{ then } 1 \text{ else } -1)$ 
    by (simp only: zsgn-def)
qed

```

```

lemma zless-imp-add1-zle:  $w < z \implies w + (1::int) \leq z$ 

```

```

apply (cases w, cases z)
apply (simp add: less le add One-int-def)
done

```

```

lemma zless-iff-Suc-zadd:
  (w :: int) < z  $\longleftrightarrow$  ( $\exists n. z = w + \text{of-nat } (\text{Suc } n)$ )
apply (cases z, cases w)
apply (auto simp add: less add int-def)
apply (rename-tac a b c d)
apply (rule-tac x=a+d - Suc(c+b) in exI)
apply arith
done

```

```

lemmas int-distrib =
  left-distrib [of z1::int z2 w, standard]
  right-distrib [of w::int z1 z2, standard]
  left-diff-distrib [of z1::int z2 w, standard]
  right-diff-distrib [of w::int z1 z2, standard]

```

### 30.5 Embedding of the Integers into any *ring-1*: *of-int*

```

context ring-1
begin

```

```

definition of-int :: int  $\Rightarrow$  'a where
  [code del]: of-int z = contents ( $\bigcup (i, j) \in \text{Rep-Integ } z. \{ \text{of-nat } i - \text{of-nat } j \}$ )

```

```

lemma of-int: of-int (Abs-Integ (intrel “ {(i,j)} )) = of-int i - of-int j

```

```

proof -
  have ( $\lambda(i,j). \{ \text{of-nat } i - (\text{of-nat } j :: 'a) \}$ ) respects intrel
  by (simp add: congruent-def algebra-simps of-nat-add [symmetric]
    del: of-nat-add)
  thus ?thesis
  by (simp add: of-int-def UN-equiv-class [OF equiv-intrel])
qed

```

```

lemma of-int-0 [simp]: of-int 0 = 0
by (simp add: of-int Zero-int-def)

```

```

lemma of-int-1 [simp]: of-int 1 = 1
by (simp add: of-int One-int-def)

```

```

lemma of-int-add [simp]: of-int (w+z) = of-int w + of-int z
by (cases w, cases z, simp add: algebra-simps of-int add)

```

```

lemma of-int-minus [simp]: of-int (-z) = - (of-int z)
by (cases z, simp add: algebra-simps of-int minus)

```

```

lemma of-int-diff [simp]: of-int (w - z) = of-int w - of-int z

```

**by** (*simp add: diff-minus Groups.diff-minus*)

**lemma** *of-int-mult* [*simp*]:  $\text{of-int } (w * z) = \text{of-int } w * \text{of-int } z$   
**apply** (*cases w, cases z*)  
**apply** (*simp add: algebra-simps of-int mult of-nat-mult*)  
**done**

Collapse nested embeddings

**lemma** *of-int-of-nat-eq* [*simp*]:  $\text{of-int } (\text{of-nat } n) = \text{of-nat } n$   
**by** (*induct n*) *auto*

**lemma** *of-int-power*:  
 $\text{of-int } (z ^ n) = \text{of-int } z ^ n$   
**by** (*induct n*) *simp-all*

**end**

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

**class** *ring-char-0* = *ring-1* + *semiring-char-0*  
**begin**

**lemma** *of-int-eq-iff* [*simp*]:  
 $\text{of-int } w = \text{of-int } z \longleftrightarrow w = z$   
**apply** (*cases w, cases z, simp add: of-int*)  
**apply** (*simp only: diff-eq-eq diff-add-eq eq-diff-eq*)  
**apply** (*simp only: of-nat-add [symmetric] of-nat-eq-iff*)  
**done**

Special cases where either operand is zero

**lemma** *of-int-eq-0-iff* [*simp*]:  
 $\text{of-int } z = 0 \longleftrightarrow z = 0$   
**using** *of-int-eq-iff [of z 0]* **by** *simp*

**lemma** *of-int-0-eq-iff* [*simp*]:  
 $0 = \text{of-int } z \longleftrightarrow z = 0$   
**using** *of-int-eq-iff [of 0 z]* **by** *simp*

**end**

**context** *linordered-idom*  
**begin**

Every *linordered-idom* has characteristic zero.

**subclass** *ring-char-0* ..

**lemma** *of-int-le-iff* [*simp*]:  
 $\text{of-int } w \leq \text{of-int } z \longleftrightarrow w \leq z$



**by** (*cases w, cases z, simp add: of-int le minus algebra-simps of-nat-add [symmetric]*  
*del: of-nat-add*)

**lemma** *of-int-less-iff [simp]:*  
 $of-int\ w < of-int\ z \iff w < z$   
**by** (*simp add: less-le order-less-le*)

**lemma** *of-int-0-le-iff [simp]:*  
 $0 \leq of-int\ z \iff 0 \leq z$   
**using** *of-int-le-iff [of 0 z]* **by** *simp*

**lemma** *of-int-le-0-iff [simp]:*  
 $of-int\ z \leq 0 \iff z \leq 0$   
**using** *of-int-le-iff [of z 0]* **by** *simp*

**lemma** *of-int-0-less-iff [simp]:*  
 $0 < of-int\ z \iff 0 < z$   
**using** *of-int-less-iff [of 0 z]* **by** *simp*

**lemma** *of-int-less-0-iff [simp]:*  
 $of-int\ z < 0 \iff z < 0$   
**using** *of-int-less-iff [of z 0]* **by** *simp*

**end**

**lemma** *of-int-eq-id [simp]: of-int = id*  
**proof**  
**fix** *z* **show** *of-int z = id z*  
**by** (*cases z*) (*simp add: of-int add minus int-def diff-minus*)  
**qed**

### 30.6 Magnitude of an Integer, as a Natural Number: *nat*

**definition**

*nat :: int  $\Rightarrow$  nat*

**where**

[*code del*]: *nat z = contents ( $\bigcup (x, y) \in Rep-Integ\ z. \{x-y\}$ )*

**lemma** *nat: nat (Abs-Integ (intrel“{(x,y)})) = x-y*

**proof** –

**have**  $(\lambda(x,y). \{x-y\})$  *respects intrel*

**by** (*simp add: congruent-def*) *arith*

**thus** *?thesis*

**by** (*simp add: nat-def UN-equiv-class [OF equiv-intrel]*)

**qed**

**lemma** *nat-int [simp]: nat (of-nat n) = n*  
**by** (*simp add: nat int-def*)

**lemma** *nat-zero* [*simp*]:  $\text{nat } 0 = 0$   
**by** (*simp add: Zero-int-def nat*)

**lemma** *int-nat-eq* [*simp*]:  $\text{of-nat } (\text{nat } z) = (\text{if } 0 \leq z \text{ then } z \text{ else } 0)$   
**by** (*cases z, simp add: nat le int-def Zero-int-def*)

**corollary** *nat-0-le*:  $0 \leq z \implies \text{of-nat } (\text{nat } z) = z$   
**by** *simp*

**lemma** *nat-le-0* [*simp*]:  $z \leq 0 \implies \text{nat } z = 0$   
**by** (*cases z, simp add: nat le Zero-int-def*)

**lemma** *nat-le-eq-zle*:  $0 < w \mid 0 \leq z \implies (\text{nat } w \leq \text{nat } z) = (w \leq z)$   
**apply** (*cases w, cases z*)  
**apply** (*simp add: nat le linorder-not-le [symmetric] Zero-int-def, arith*)  
**done**

An alternative condition is  $(0::'a) \leq w$

**corollary** *nat-mono-iff*:  $0 < z \implies (\text{nat } w < \text{nat } z) = (w < z)$   
**by** (*simp add: nat-le-eq-zle linorder-not-le [symmetric]*)

**corollary** *nat-less-eq-zless*:  $0 \leq w \implies (\text{nat } w < \text{nat } z) = (w < z)$   
**by** (*simp add: nat-le-eq-zle linorder-not-le [symmetric]*)

**lemma** *zless-nat-conj* [*simp*]:  $(\text{nat } w < \text{nat } z) = (0 < z \ \& \ w < z)$   
**apply** (*cases w, cases z*)  
**apply** (*simp add: nat le Zero-int-def linorder-not-le [symmetric], arith*)  
**done**

**lemma** *nonneg-eq-int*:  
**fixes**  $z :: \text{int}$   
**assumes**  $0 \leq z$  **and**  $\bigwedge m. z = \text{of-nat } m \implies P$   
**shows**  $P$   
**using** *assms* **by** (*blast dest: nat-0-le sym*)

**lemma** *nat-eq-iff*:  $(\text{nat } w = m) = (\text{if } 0 \leq w \text{ then } w = \text{of-nat } m \text{ else } m=0)$   
**by** (*cases w, simp add: nat le int-def Zero-int-def, arith*)

**corollary** *nat-eq-iff2*:  $(m = \text{nat } w) = (\text{if } 0 \leq w \text{ then } w = \text{of-nat } m \text{ else } m=0)$   
**by** (*simp only: eq-commute [of m] nat-eq-iff*)

**lemma** *nat-less-iff*:  $0 \leq w \implies (\text{nat } w < m) = (w < \text{of-nat } m)$   
**apply** (*cases w*)  
**apply** (*simp add: nat le int-def Zero-int-def linorder-not-le[symmetric], arith*)  
**done**

**lemma** *nat-0-iff* [*simp*]:  $\text{nat}(i::\text{int}) = 0 \iff i \leq 0$   
**by**(*simp add: nat-eq-iff*) *arith*

**lemma** *int-eq-iff*:  $(\text{of-nat } m = z) = (m = \text{nat } z \ \& \ 0 \leq z)$   
**by** (*auto simp add: nat-eq-iff2*)

**lemma** *zero-less-nat-eq* [*simp*]:  $(0 < \text{nat } z) = (0 < z)$   
**by** (*insert zless-nat-conj [of 0], auto*)

**lemma** *nat-add-distrib*:  
 $[(0 :: \text{int}) \leq z; \ 0 \leq z'] \implies \text{nat } (z + z') = \text{nat } z + \text{nat } z'$   
**by** (*cases z, cases z', simp add: nat add le Zero-int-def*)

**lemma** *nat-diff-distrib*:  
 $[(0 :: \text{int}) \leq z'; \ z' \leq z] \implies \text{nat } (z - z') = \text{nat } z - \text{nat } z'$   
**by** (*cases z, cases z', simp add: nat add minus diff-minus le Zero-int-def*)

**lemma** *nat-zminus-int* [*simp*]:  $\text{nat } (- (\text{of-nat } n)) = 0$   
**by** (*simp add: int-def minus nat Zero-int-def*)

**lemma** *zless-nat-eq-int-zless*:  $(m < \text{nat } z) = (\text{of-nat } m < z)$   
**by** (*cases z, simp add: nat less int-def, arith*)

**context** *ring-1*  
**begin**

**lemma** *of-nat-nat*:  $0 \leq z \implies \text{of-nat } (\text{nat } z) = \text{of-int } z$   
**by** (*cases z rule: eq-Abs-Integ*)  
*(simp add: nat le of-int Zero-int-def of-nat-diff)*

**end**

For termination proofs:

**lemma** *measure-function-int*[*measure-function*]: *is-measure* (*nat o abs*) ..

### 30.7 Lemmas about the Function *of-nat* and Orderings

**lemma** *negative-zless-0*:  $-(\text{of-nat } (\text{Suc } n)) < (0 :: \text{int})$   
**by** (*simp add: order-less-le del: of-nat-Suc*)

**lemma** *negative-zless* [*iff*]:  $-(\text{of-nat } (\text{Suc } n)) < (\text{of-nat } m :: \text{int})$   
**by** (*rule negative-zless-0 [THEN order-less-le-trans], simp*)

**lemma** *negative-zle-0*:  $-(\text{of-nat } n) \leq (0 :: \text{int})$   
**by** (*simp add: minus-le-iff*)

**lemma** *negative-zle* [*iff*]:  $-(\text{of-nat } n) \leq (\text{of-nat } m :: \text{int})$   
**by** (*rule order-trans [OF negative-zle-0 of-nat-0-le-iff]*)

**lemma** *not-zle-0-negative* [*simp*]:  $\sim (0 \leq - (\text{of-nat } (\text{Suc } n) :: \text{int}))$

**by** (*subst le-minus-iff*, *simp del: of-nat-Suc*)

**lemma** *int-zle-neg*:  $((\text{of-nat } n :: \text{int}) \leq - \text{of-nat } m) = (n = 0 \ \& \ m = 0)$   
**by** (*simp add: int-def le minus Zero-int-def*)

**lemma** *not-int-zless-negative* [*simp*]:  $\sim ((\text{of-nat } n :: \text{int}) < - \text{of-nat } m)$   
**by** (*simp add: linorder-not-less*)

**lemma** *negative-eq-positive* [*simp*]:  $((- \text{of-nat } n :: \text{int}) = \text{of-nat } m) = (n = 0 \ \& \ m = 0)$   
**by** (*force simp add: order-eq-iff [of - of-nat n] int-zle-neg*)

**lemma** *zle-iff-zadd*:  $(w :: \text{int}) \leq z \iff (\exists n. z = w + \text{of-nat } n)$   
**proof** –

**have**  $(w \leq z) = (0 \leq z - w)$   
   **by** (*simp only: le-diff-eq add-0-left*)  
  **also have**  $\dots = (\exists n. z - w = \text{of-nat } n)$   
   **by** (*auto elim: zero-le-imp-eq-int*)  
  **also have**  $\dots = (\exists n. z = w + \text{of-nat } n)$   
   **by** (*simp only: algebra-simps*)  
  **finally show** ?thesis .

**qed**

**lemma** *zadd-int-left*:  $\text{of-nat } m + (\text{of-nat } n + z) = \text{of-nat } (m + n) + (z :: \text{int})$   
**by** *simp*

**lemma** *int-Suc0-eq-1*:  $\text{of-nat } (\text{Suc } 0) = (1 :: \text{int})$   
**by** *simp*

This version is proved for all ordered rings, not just integers! It is proved here because attribute *arith-split* is not available in theory *Rings*. But is it really better than just rewriting with *abs-if*?

**lemma** *abs-split* [*arith-split, no-atp*]:  
 $P(\text{abs}(a :: 'a :: \text{linordered-idom})) = ((0 \leq a \implies P \ a) \ \& \ (a < 0 \implies P(-a)))$   
**by** (*force dest: order-less-le-trans simp add: abs-if linorder-not-less*)

**lemma** *negD*:  $(x :: \text{int}) < 0 \implies \exists n. x = - (\text{of-nat } (\text{Suc } n))$   
**apply** (*cases x*)  
**apply** (*auto simp add: le minus Zero-int-def int-def order-less-le*)  
**apply** (*rule-tac x=y - Suc x in exI, arith*)  
**done**

### 30.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

**theorem** *int-cases* [*cases type: int, case-names nonneg neg*]:

```

  [|!! n. (z :: int) = of-nat n ==> P; !! n. z = - (of-nat (Suc n)) ==> P |]
==> P
apply (cases z < 0, blast dest!: negD)
apply (simp add: linorder-not-less del: of-nat-Suc)
apply auto
apply (blast dest: nat-0-le [THEN sym])
done

```

```

theorem int-of-nat-induct [induct type: int, case-names nonneg neg]:
  [|!! n. P (of-nat n :: int); !!n. P (- (of-nat (Suc n))) |] ==> P z
  by (cases z rule: int-cases) auto

```

Contributed by Brian Huffman

```

theorem int-diff-cases:
  obtains (diff) m n where (z::int) = of-nat m - of-nat n
apply (cases z rule: eq-Abs-Integ)
apply (rule-tac m=x and n=y in diff)
apply (simp add: int-def diff-def minus add)
done

```

### 30.9 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos-compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that  $(m \bmod 2)$  is 0 or 1, even if  $m$  is negative; For instance,  $-5 \div 2 = -3$  and  $-5 \bmod 2 = 1$ ; thus  $-5 = (-3)*2 + 1$ .

This two’s complement binary representation derives from the paper “An Efficient Representation of Arithmetic for Term Rewriting” by Dave Cohen and Phil Watson, *Rewriting Techniques and Applications*, Springer LNCS 488 (240-251), 1991.

#### 30.9.1 The constructors *Bit0*, *Bit1*, *Pls* and *Min*

##### definition

```

  Pls :: int where
  [code del]: Pls = 0

```

##### definition

```

  Min :: int where
  [code del]: Min = - 1

```

##### definition

```

  Bit0 :: int => int where
  [code del]: Bit0 k = k + k

```

**definition**

$Bit1 :: int \Rightarrow int$  **where**  
 $[code\ del]: Bit1\ k = 1 + k + k$

**class** *number* = — for numeric types: nat, int, real, ...

**fixes** *number-of* ::  $int \Rightarrow 'a$

**use** *Tools/numeral.ML*

**syntax**

$-Numeral :: num-const \Rightarrow 'a \quad (-)$

**use** *Tools/numeral-syntax.ML*

**setup** *Numerals-Syntax.setup*

**abbreviation**

$Numerals0 \equiv number-of\ Pls$

**abbreviation**

$Numerals1 \equiv number-of\ (Bit1\ Pls)$

**lemma** *Let-number-of [simp]: Let (number-of v) f = f (number-of v)*

— Unfold all *lets* involving constants

**unfolding** *Let-def ..*

**definition**

$succ :: int \Rightarrow int$  **where**  
 $[code\ del]: succ\ k = k + 1$

**definition**

$pred :: int \Rightarrow int$  **where**  
 $[code\ del]: pred\ k = k - 1$

**lemmas**

$max-number-of\ [simp] = max-def$   
 $[of\ number-of\ u\ number-of\ v,\ standard]$

**and**

$min-number-of\ [simp] = min-def$   
 $[of\ number-of\ u\ number-of\ v,\ standard]$   
 — unfolding *minx* and *max* on numerals

**lemmas** *numeral-simps =*

*succ-def pred-def Pls-def Min-def Bit0-def Bit1-def*

Removal of leading zeroes

**lemma** *Bit0-Pls [simp, code-post]:*

$Bit0\ Pls = Pls$

**unfolding** *numeral-simps by simp*

**lemma** *Bit1-Min* [*simp*, *code-post*]:  
 $\text{Bit1 } \text{Min} = \text{Min}$   
**unfolding** *numeral-simps* **by** *simp*

**lemmas** *normalize-bin-simps* =  
 $\text{Bit0-Pls } \text{Bit1-Min}$

### 30.9.2 Successor and predecessor functions

Successor

**lemma** *succ-Pls*:  
 $\text{succ } \text{Pls} = \text{Bit1 } \text{Pls}$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *succ-Min*:  
 $\text{succ } \text{Min} = \text{Pls}$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *succ-Bit0*:  
 $\text{succ } (\text{Bit0 } k) = \text{Bit1 } k$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *succ-Bit1*:  
 $\text{succ } (\text{Bit1 } k) = \text{Bit0 } (\text{succ } k)$   
**unfolding** *numeral-simps* **by** *simp*

**lemmas** *succ-bin-simps* [*simp*] =  
 $\text{succ-Pls } \text{succ-Min } \text{succ-Bit0 } \text{succ-Bit1}$

Predecessor

**lemma** *pred-Pls*:  
 $\text{pred } \text{Pls} = \text{Min}$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *pred-Min*:  
 $\text{pred } \text{Min} = \text{Bit0 } \text{Min}$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *pred-Bit0*:  
 $\text{pred } (\text{Bit0 } k) = \text{Bit1 } (\text{pred } k)$   
**unfolding** *numeral-simps* **by** *simp*

**lemma** *pred-Bit1*:  
 $\text{pred } (\text{Bit1 } k) = \text{Bit0 } k$   
**unfolding** *numeral-simps* **by** *simp*

**lemmas** *pred-bin-simps* [*simp*] =  
 $\text{pred-Pls } \text{pred-Min } \text{pred-Bit0 } \text{pred-Bit1}$

### 30.9.3 Binary arithmetic

Addition

**lemma** *add-Pls*:

$$Pls + k = k$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Min*:

$$Min + k = pred\ k$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit0-Bit0*:

$$(Bit0\ k) + (Bit0\ l) = Bit0\ (k + l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit0-Bit1*:

$$(Bit0\ k) + (Bit1\ l) = Bit1\ (k + l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit1-Bit0*:

$$(Bit1\ k) + (Bit0\ l) = Bit1\ (k + l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Bit1-Bit1*:

$$(Bit1\ k) + (Bit1\ l) = Bit0\ (k + succ\ l)$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Pls-right*:

$$k + Pls = k$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *add-Min-right*:

$$k + Min = pred\ k$$

**unfolding** *numeral-simps* **by** *simp*

**lemmas** *add-bin-simps* [*simp*] =

*add-Pls add-Min add-Pls-right add-Min-right*

*add-Bit0-Bit0 add-Bit0-Bit1 add-Bit1-Bit0 add-Bit1-Bit1*

Negation

**lemma** *minus-Pls*:

$$- Pls = Pls$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *minus-Min*:

$$- Min = Bit1\ Pls$$

**unfolding** *numeral-simps* **by** *simp*

**lemma** *minus-Bit0*:



–  $(\text{Bit0 } k) = \text{Bit0 } (-k)$   
**unfolding numeral-simps by simp**

**lemma minus-Bit1:**  
 –  $(\text{Bit1 } k) = \text{Bit1 } (\text{pred } (-k))$   
**unfolding numeral-simps by simp**

**lemmas minus-bin-simps [simp] =**  
 minus-Pls minus-Min minus-Bit0 minus-Bit1

### Subtraction

**lemma diff-bin-simps [simp]:**  
 $k - \text{Pls} = k$   
 $k - \text{Min} = \text{succ } k$   
 $\text{Pls} - (\text{Bit0 } l) = \text{Bit0 } (\text{Pls} - l)$   
 $\text{Pls} - (\text{Bit1 } l) = \text{Bit1 } (\text{Min} - l)$   
 $\text{Min} - (\text{Bit0 } l) = \text{Bit1 } (\text{Min} - l)$   
 $\text{Min} - (\text{Bit1 } l) = \text{Bit0 } (\text{Min} - l)$   
 $(\text{Bit0 } k) - (\text{Bit0 } l) = \text{Bit0 } (k - l)$   
 $(\text{Bit0 } k) - (\text{Bit1 } l) = \text{Bit1 } (\text{pred } k - l)$   
 $(\text{Bit1 } k) - (\text{Bit0 } l) = \text{Bit1 } (k - l)$   
 $(\text{Bit1 } k) - (\text{Bit1 } l) = \text{Bit0 } (k - l)$   
**unfolding numeral-simps by simp-all**

### Multiplication

**lemma mult-Pls:**  
 $\text{Pls} * w = \text{Pls}$   
**unfolding numeral-simps by simp**

**lemma mult-Min:**  
 $\text{Min} * k = -k$   
**unfolding numeral-simps by simp**

**lemma mult-Bit0:**  
 $(\text{Bit0 } k) * l = \text{Bit0 } (k * l)$   
**unfolding numeral-simps int-distrib by simp**

**lemma mult-Bit1:**  
 $(\text{Bit1 } k) * l = (\text{Bit0 } (k * l)) + l$   
**unfolding numeral-simps int-distrib by simp**

**lemmas mult-bin-simps [simp] =**  
 mult-Pls mult-Min mult-Bit0 mult-Bit1

## 30.9.4 Binary comparisons

### Preliminaries

**lemma even-less-0-iff:**

```

  a + a < 0  $\longleftrightarrow$  a < (0::'a::linordered-idom)
proof -
  have a + a < 0  $\longleftrightarrow$  (1+1)*a < 0 by (simp add: left-distrib)
  also have (1+1)*a < 0  $\longleftrightarrow$  a < 0
    by (simp add: mult-less-0-iff zero-less-two
              order-less-not-sym [OF zero-less-two])
  finally show ?thesis .
qed

lemma le-imp-0-less:
  assumes le: 0  $\leq$  z
  shows (0::int) < 1 + z
proof -
  have 0  $\leq$  z by fact
  also have ... < z + 1 by (rule less-add-one)
  also have ... = 1 + z by (simp add: add-ac)
  finally show 0 < 1 + z .
qed

lemma odd-less-0-iff:
  (1 + z + z < 0) = (z < (0::int))
proof (cases z rule: int-cases)
  case (nonneg n)
  thus ?thesis by (simp add: linorder-not-less add-assoc add-increasing
                        le-imp-0-less [THEN order-less-imp-le])
next
  case (neg n)
  thus ?thesis by (simp del: of-nat-Suc of-nat-add of-nat-1
                    add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])
qed

lemma bin-less-0-simps:
  Pls < 0  $\longleftrightarrow$  False
  Min < 0  $\longleftrightarrow$  True
  Bit0 w < 0  $\longleftrightarrow$  w < 0
  Bit1 w < 0  $\longleftrightarrow$  w < 0
  unfolding numeral-simps
  by (simp-all add: even-less-0-iff odd-less-0-iff)

lemma less-bin-lemma: k < l  $\longleftrightarrow$  k - l < (0::int)
  by simp

lemma le-iff-pred-less: k  $\leq$  l  $\longleftrightarrow$  pred k < l
  unfolding numeral-simps
  proof
    have k - 1 < k by simp
    also assume k  $\leq$  l
    finally show k - 1 < l .
  next

```

**assume**  $k - 1 < l$   
**hence**  $(k - 1) + 1 \leq l$  **by** (rule *zless-imp-add1-zle*)  
**thus**  $k \leq l$  **by** *simp*  
**qed**

**lemma** *succ-pred*:  $\text{succ} (\text{pred } x) = x$   
**unfolding** *numeral-simps* **by** *simp*

Less-than

**lemma** *less-bin-simps* [*simp*]:  
 $\text{Pls} < \text{Pls} \longleftrightarrow \text{False}$   
 $\text{Pls} < \text{Min} \longleftrightarrow \text{False}$   
 $\text{Pls} < \text{Bit0 } k \longleftrightarrow \text{Pls} < k$   
 $\text{Pls} < \text{Bit1 } k \longleftrightarrow \text{Pls} \leq k$   
 $\text{Min} < \text{Pls} \longleftrightarrow \text{True}$   
 $\text{Min} < \text{Min} \longleftrightarrow \text{False}$   
 $\text{Min} < \text{Bit0 } k \longleftrightarrow \text{Min} < k$   
 $\text{Min} < \text{Bit1 } k \longleftrightarrow \text{Min} < k$   
 $\text{Bit0 } k < \text{Pls} \longleftrightarrow k < \text{Pls}$   
 $\text{Bit0 } k < \text{Min} \longleftrightarrow k \leq \text{Min}$   
 $\text{Bit1 } k < \text{Pls} \longleftrightarrow k < \text{Pls}$   
 $\text{Bit1 } k < \text{Min} \longleftrightarrow k < \text{Min}$   
 $\text{Bit0 } k < \text{Bit0 } l \longleftrightarrow k < l$   
 $\text{Bit0 } k < \text{Bit1 } l \longleftrightarrow k \leq l$   
 $\text{Bit1 } k < \text{Bit0 } l \longleftrightarrow k < l$   
 $\text{Bit1 } k < \text{Bit1 } l \longleftrightarrow k < l$   
**unfolding** *le-iff-pred-less*  
*less-bin-lemma* [*of Pls*]  
*less-bin-lemma* [*of Min*]  
*less-bin-lemma* [*of k*]  
*less-bin-lemma* [*of Bit0 k*]  
*less-bin-lemma* [*of Bit1 k*]  
*less-bin-lemma* [*of pred Pls*]  
*less-bin-lemma* [*of pred k*]  
**by** (*simp-all add: bin-less-0-simps succ-pred*)

Less-than-or-equal

**lemma** *le-bin-simps* [*simp*]:  
 $\text{Pls} \leq \text{Pls} \longleftrightarrow \text{True}$   
 $\text{Pls} \leq \text{Min} \longleftrightarrow \text{False}$   
 $\text{Pls} \leq \text{Bit0 } k \longleftrightarrow \text{Pls} \leq k$   
 $\text{Pls} \leq \text{Bit1 } k \longleftrightarrow \text{Pls} \leq k$   
 $\text{Min} \leq \text{Pls} \longleftrightarrow \text{True}$   
 $\text{Min} \leq \text{Min} \longleftrightarrow \text{True}$   
 $\text{Min} \leq \text{Bit0 } k \longleftrightarrow \text{Min} < k$   
 $\text{Min} \leq \text{Bit1 } k \longleftrightarrow \text{Min} \leq k$   
 $\text{Bit0 } k \leq \text{Pls} \longleftrightarrow k \leq \text{Pls}$   
 $\text{Bit0 } k \leq \text{Min} \longleftrightarrow k \leq \text{Min}$   
 $\text{Bit1 } k \leq \text{Pls} \longleftrightarrow k < \text{Pls}$

```

    Bit1 k ≤ Min ↔ k ≤ Min
    Bit0 k ≤ Bit0 l ↔ k ≤ l
    Bit0 k ≤ Bit1 l ↔ k ≤ l
    Bit1 k ≤ Bit0 l ↔ k < l
    Bit1 k ≤ Bit1 l ↔ k ≤ l
unfolding not-less [symmetric]
by (simp-all add: not-le)

```

Equality

**lemma** *eq-bin-simps* [simp]:

```

    Pls = Pls ↔ True
    Pls = Min ↔ False
    Pls = Bit0 l ↔ Pls = l
    Pls = Bit1 l ↔ False
    Min = Pls ↔ False
    Min = Min ↔ True
    Min = Bit0 l ↔ False
    Min = Bit1 l ↔ Min = l
    Bit0 k = Pls ↔ k = Pls
    Bit0 k = Min ↔ False
    Bit1 k = Pls ↔ False
    Bit1 k = Min ↔ k = Min
    Bit0 k = Bit0 l ↔ k = l
    Bit0 k = Bit1 l ↔ False
    Bit1 k = Bit0 l ↔ False
    Bit1 k = Bit1 l ↔ k = l
unfolding order-eq-iff [where 'a=int]
by (simp-all add: not-less)

```

### 30.10 Converting Numerals to Rings: *number-of*

```

class number-ring = number + comm-ring-1 +
  assumes number-of-eq: number-of k = of-int k

```

self-embedding of the integers

```

instantiation int :: number-ring
begin

```

```

definition int-number-of-def [code del]:
  number-of w = (of-int w :: int)

```

```

instance proof
qed (simp only: int-number-of-def)

```

**end**

```

lemma number-of-is-id:
  number-of (k::int) = k
unfolding int-number-of-def by simp

```

**lemma** *number-of-succ*:

*number-of* (*succ* *k*) = (*1* + *number-of* *k* :: 'a::number-ring)

**unfolding** *number-of-eq* *numeral-simps* **by** *simp*

**lemma** *number-of-pred*:

*number-of* (*pred* *w*) = (*- 1* + *number-of* *w* :: 'a::number-ring)

**unfolding** *number-of-eq* *numeral-simps* **by** *simp*

**lemma** *number-of-minus*:

*number-of* (*uminus* *w*) = (*-* (*number-of* *w*) :: 'a::number-ring)

**unfolding** *number-of-eq* **by** (*rule of-int-minus*)

**lemma** *number-of-add*:

*number-of* (*v* + *w*) = (*number-of* *v* + *number-of* *w* :: 'a::number-ring)

**unfolding** *number-of-eq* **by** (*rule of-int-add*)

**lemma** *number-of-diff*:

*number-of* (*v* - *w*) = (*number-of* *v* - *number-of* *w* :: 'a::number-ring)

**unfolding** *number-of-eq* **by** (*rule of-int-diff*)

**lemma** *number-of-mult*:

*number-of* (*v* \* *w*) = (*number-of* *v* \* *number-of* *w* :: 'a::number-ring)

**unfolding** *number-of-eq* **by** (*rule of-int-mult*)

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

**lemma** *double-number-of-Bit0*:

(*1* + *1*) \* *number-of* *w* = (*number-of* (*Bit0* *w*) :: 'a::number-ring)

**unfolding** *number-of-eq* *numeral-simps* *left-distrib* **by** *simp*

Converting numerals 0 and 1 to their abstract versions.

**lemma** *numeral-0-eq-0* [*simp*, *code-post*]:

*Numeral0* = (*0* :: 'a::number-ring)

**unfolding** *number-of-eq* *numeral-simps* **by** *simp*

**lemma** *numeral-1-eq-1* [*simp*, *code-post*]:

*Numeral1* = (*1* :: 'a::number-ring)

**unfolding** *number-of-eq* *numeral-simps* **by** *simp*

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simp rule until later: it is *number-of-Min* re-oriented!

**lemma** *numeral-m1-eq-minus-1*:

(*-1* :: 'a::number-ring) = *- 1*

**unfolding** *number-of-eq* *numeral-simps* **by** *simp*

**lemma** *mult-minus1* [*simp*]:

$-1 * z = -(z :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *mult-minus1-right* [*simp*]:  
 $z * -1 = -(z :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *minus-number-of-mult* [*simp*]:  
 $-(\text{number-of } w) * z = \text{number-of } (\text{uminus } w) * (z :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq* **by** *simp*

Subtraction

**lemma** *diff-number-of-eq*:  
 $\text{number-of } v - \text{number-of } w =$   
 $(\text{number-of } (v + \text{uminus } w) :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq* **by** *simp*

**lemma** *number-of-Pls*:  
 $\text{number-of } \text{Pls} = (0 :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-Min*:  
 $\text{number-of } \text{Min} = (-1 :: 'a :: \text{number-ring})$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-Bit0*:  
 $\text{number-of } (\text{Bit0 } w) = (0 :: 'a :: \text{number-ring}) + (\text{number-of } w) + (\text{number-of } w)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

**lemma** *number-of-Bit1*:  
 $\text{number-of } (\text{Bit1 } w) = (1 :: 'a :: \text{number-ring}) + (\text{number-of } w) + (\text{number-of } w)$   
**unfolding** *number-of-eq numeral-simps* **by** *simp*

### 30.10.1 Equality of Binary Numbers

First version by Norbert Voelker

**definition** *iszero* ::  $'a :: \text{semiring-1} \Rightarrow \text{bool}$  **where**  
 $\text{iszero } z \longleftrightarrow z = 0$

**lemma** *iszero-0*: *iszero 0*  
**by** (*simp add: iszero-def*)

**lemma** *iszero-Numeral0*: *iszero (Numeral0 :: 'a :: number-ring)*  
**by** (*simp add: iszero-0*)

**lemma** *not-iszero-1*:  $\neg \text{iszero } 1$   
**by** (*simp add: iszero-def*)

**lemma** *not-iszero-Numeral1*:  $\neg \text{iszero } (\text{Numeral1} :: 'a::\text{number-ring})$   
**by** (*simp add: not-iszero-1*)

**lemma** *eq-number-of-eq [simp]*:  
 $((\text{number-of } x :: 'a::\text{number-ring}) = \text{number-of } y) =$   
 $\text{iszero } (\text{number-of } (x + \text{uminus } y) :: 'a)$   
**unfolding** *iszero-def number-of-add number-of-minus*  
**by** (*simp add: algebra-simps*)

**lemma** *iszero-number-of-Pls*:  
 $\text{iszero } ((\text{number-of } \text{Pls}) :: 'a::\text{number-ring})$   
**unfolding** *iszero-def numeral-0-eq-0 ..*

**lemma** *nonzero-number-of-Min*:  
 $\sim \text{iszero } ((\text{number-of } \text{Min}) :: 'a::\text{number-ring})$   
**unfolding** *iszero-def numeral-m1-eq-minus-1* **by** *simp*

### 30.10.2 Comparisons, for Ordered Rings

**lemmas** *double-eq-0-iff = double-zero*

**lemma** *odd-nonzero*:  
 $1 + z + z \neq (0::\text{int})$   
**proof** (*cases z rule: int-cases*)  
**case** (*nonneg n*)  
**have**  $le: 0 \leq z + z$  **by** (*simp add: nonneg add-increasing*)  
**thus** *?thesis* **using** *le-imp-0-less [OF le]*  
**by** (*auto simp add: add-assoc*)  
**next**  
**case** (*neg n*)  
**show** *?thesis*  
**proof**  
**assume** *eq*:  $1 + z + z = 0$   
**have**  $(0::\text{int}) < 1 + (\text{of-nat } n + \text{of-nat } n)$   
**by** (*simp add: le-imp-0-less add-increasing*)  
**also have**  $\dots = -(1 + z + z)$   
**by** (*simp add: neg add-assoc [symmetric]*)  
**also have**  $\dots = 0$  **by** (*simp add: eq*)  
**finally have**  $0 < 0$  **..**  
**thus** *False* **by** *blast*  
**qed**  
**qed**

**lemma** *iszero-number-of-Bit0*:  
 $\text{iszero } (\text{number-of } (\text{Bit0 } w) :: 'a) =$   
 $\text{iszero } (\text{number-of } w :: 'a::\{\text{ring-char-0}, \text{number-ring}\})$   
**proof** –  
**have**  $(\text{of-int } w + \text{of-int } w = (0::'a)) \implies (w = 0)$   
**proof** –

```

    assume eq: of-int w + of-int w = (0::'a)
    then have of-int (w + w) = (of-int 0 :: 'a) by simp
    then have w + w = 0 by (simp only: of-int-eq-iff)
    then show w = 0 by (simp only: double-eq-0-iff)
  qed
  thus ?thesis
    by (auto simp add: iszero-def number-of-eq numeral-simps)
qed

```

```

lemma iszero-number-of-Bit1:
  ~ iszero (number-of (Bit1 w)::'a::{ring-char-0,number-ring})
proof -
  have 1 + of-int w + of-int w ≠ (0::'a)
  proof
    assume eq: 1 + of-int w + of-int w = (0::'a)
    hence of-int (1 + w + w) = (of-int 0 :: 'a) by simp
    hence 1 + w + w = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
  thus ?thesis
    by (auto simp add: iszero-def number-of-eq numeral-simps)
qed

```

```

lemmas iszero-simps [simp] =
  iszero-0 not-iszero-1
  iszero-number-of-Pls nonzero-number-of-Min
  iszero-number-of-Bit0 iszero-number-of-Bit1

```

### 30.10.3 The Less-Than Relation

```

lemma double-less-0-iff:
  (a + a < 0) = (a < (0::'a::{linordered-idom}))
proof -
  have (a + a < 0) = ((1+1)*a < 0) by (simp add: left-distrib)
  also have ... = (a < 0)
    by (simp add: mult-less-0-iff zero-less-two
      order-less-not-sym [OF zero-less-two])
  finally show ?thesis .
qed

```

```

lemma odd-less-0:
  (1 + z + z < 0) = (z < (0::int))
proof (cases z rule: int-cases)
  case (nonneg n)
  thus ?thesis by (simp add: linorder-not-less add-assoc add-increasing
    le-imp-0-less [THEN order-less-imp-le])
next
  case (neg n)
  thus ?thesis by (simp del: of-nat-Suc of-nat-add of-nat-1

```



*add: algebra-simps of-nat-1 [where 'a=int, symmetric] of-nat-add [symmetric])*  
**qed**

Less-Than or Equals

Reduces  $a \leq b$  to  $\neg b < a$  for ALL numerals.

**lemmas** *le-number-of-eq-not-less* =  
*linorder-not-less [of number-of w number-of v, symmetric,*  
*standard]*

Absolute value (*abs*)

**lemma** *abs-number-of*:  
 $\text{abs}(\text{number-of } x :: 'a :: \{\text{linordered-idom}, \text{number-ring}\}) =$   
 $(\text{if } \text{number-of } x < (0 :: 'a) \text{ then } -\text{number-of } x \text{ else } \text{number-of } x)$   
**by** (*simp add: abs-if*)

Re-orientation of the equation  $\text{nnn} = x$

**lemma** *number-of-reorient*:  
 $(\text{number-of } w = x) = (x = \text{number-of } w)$   
**by** *auto*

### 30.10.4 Simplification of arithmetic operations on integer constants.

**lemmas** *arith-extra-simps* [*standard, simp*] =  
*number-of-add [symmetric]*  
*number-of-minus [symmetric]*  
*numeral-m1-eq-minus-1 [symmetric]*  
*number-of-mult [symmetric]*  
*diff-number-of-eq abs-number-of*

For making a minimal simpset, one must include these default simprules.  
 Also include *simp-thms*.

**lemmas** *arith-simps* =  
*normalize-bin-simps pred-bin-simps succ-bin-simps*  
*add-bin-simps minus-bin-simps mult-bin-simps*  
*abs-zero abs-one arith-extra-simps*

Simplification of relational operations

**lemma** *less-number-of* [*simp*]:  
 $(\text{number-of } x :: 'a :: \{\text{linordered-idom}, \text{number-ring}\}) < \text{number-of } y \longleftrightarrow x < y$   
**unfolding** *number-of-eq* **by** (*rule of-int-less-iff*)

**lemma** *le-number-of* [*simp*]:  
 $(\text{number-of } x :: 'a :: \{\text{linordered-idom}, \text{number-ring}\}) \leq \text{number-of } y \longleftrightarrow x \leq y$   
**unfolding** *number-of-eq* **by** (*rule of-int-le-iff*)

**lemma** *eq-number-of* [*simp*]:

$(\text{number-of } x :: 'a :: \{\text{ring-char-0}, \text{number-ring}\}) = \text{number-of } y \longleftrightarrow x = y$   
**unfolding** *number-of-eq* **by** (*rule of-int-eq-iff*)

**lemmas** *rel-simps* =  
*less-number-of less-bin-simps*  
*le-number-of le-bin-simps*  
*eq-number-of-eq eq-bin-simps*  
*iszero-simps*

### 30.10.5 Simplification of arithmetic when nested to the right.

**lemma** *add-number-of-left* [*simp*]:  
 $\text{number-of } v + (\text{number-of } w + z) =$   
 $(\text{number-of } (v + w) + z :: 'a :: \text{number-ring})$   
**by** (*simp add: add-assoc [symmetric]*)

**lemma** *mult-number-of-left* [*simp*]:  
 $\text{number-of } v * (\text{number-of } w * z) =$   
 $(\text{number-of } (v * w) * z :: 'a :: \text{number-ring})$   
**by** (*simp add: mult-assoc [symmetric]*)

**lemma** *add-number-of-diff1*:  
 $\text{number-of } v + (\text{number-of } w - c) =$   
 $\text{number-of } (v + w) - (c :: 'a :: \text{number-ring})$   
**by** (*simp add: diff-minus*)

**lemma** *add-number-of-diff2* [*simp*]:  
 $\text{number-of } v + (c - \text{number-of } w) =$   
 $\text{number-of } (v + \text{uminus } w) + (c :: 'a :: \text{number-ring})$   
**by** (*simp add: algebra-simps diff-number-of-eq [symmetric]*)

## 30.11 The Set of Integers

**context** *ring-1*  
**begin**

**definition** *Ints* :: 'a set **where**  
 $[\text{code del}]: \text{Ints} = \text{range of-int}$

**notation** (*xsymbols*)  
*Ints* ( $\mathbb{Z}$ )

**lemma** *Ints-of-int* [*simp*]: *of-int*  $z \in \mathbb{Z}$   
**by** (*simp add: Ints-def*)

**lemma** *Ints-of-nat* [*simp*]: *of-nat*  $n \in \mathbb{Z}$   
**apply** (*simp add: Ints-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-int-of-nat-eq [symmetric]*)  
**done**

```

lemma Ints-0 [simp]:  $0 \in \mathbb{Z}$ 
apply (simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-0 [symmetric])
done

```

```

lemma Ints-1 [simp]:  $1 \in \mathbb{Z}$ 
apply (simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-1 [symmetric])
done

```

```

lemma Ints-add [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a + b \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-add [symmetric])
done

```

```

lemma Ints-minus [simp]:  $a \in \mathbb{Z} \implies -a \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-minus [symmetric])
done

```

```

lemma Ints-diff [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-diff [symmetric])
done

```

```

lemma Ints-mult [simp]:  $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a * b \in \mathbb{Z}$ 
apply (auto simp add: Ints-def)
apply (rule range-eqI)
apply (rule of-int-mult [symmetric])
done

```

```

lemma Ints-power [simp]:  $a \in \mathbb{Z} \implies a ^ n \in \mathbb{Z}$ 
by (induct n) simp-all

```

```

lemma Ints-cases [cases set: Ints]:
  assumes  $q \in \mathbb{Z}$ 
  obtains (of-int)  $z$  where  $q = \text{of-int } z$ 
  unfolding Ints-def
proof –
  from  $\langle q \in \mathbb{Z} \rangle$  have  $q \in \text{range of-int}$  unfolding Ints-def .
  then obtain  $z$  where  $q = \text{of-int } z$  ..
  then show thesis ..
qed

```

**lemma** *Ints-induct* [*case-names of-int*, *induct set: Ints*]:

$q \in \mathbb{Z} \implies (\bigwedge z. P \text{ (of-int } z)) \implies P \ q$   
**by** (*rule Ints-cases*) *auto*

**end**

The premise involving  $\mathbb{Z}$  prevents  $a = (1::'a) / (2::'a)$ .

**lemma** *Ints-double-eq-0-iff*:

**assumes** *in-Ints*:  $a \in \text{Ints}$

**shows**  $(a + a = 0) = (a = (0::'a::\text{ring-char-0}))$

**proof** –

**from** *in-Ints* **have**  $a \in \text{range of-int}$  **unfolding** *Ints-def* [*symmetric*] .

**then obtain**  $z$  **where**  $a = \text{of-int } z$  ..

**show** *?thesis*

**proof**

**assume**  $a = 0$

**thus**  $a + a = 0$  **by** *simp*

**next**

**assume** *eq*:  $a + a = 0$

**hence**  $\text{of-int } (z + z) = (\text{of-int } 0 :: 'a)$  **by** (*simp add: a*)

**hence**  $z + z = 0$  **by** (*simp only: of-int-eq-iff*)

**hence**  $z = 0$  **by** (*simp only: double-eq-0-iff*)

**thus**  $a = 0$  **by** (*simp add: a*)

**qed**

**qed**

**lemma** *Ints-odd-nonzero*:

**assumes** *in-Ints*:  $a \in \text{Ints}$

**shows**  $1 + a + a \neq (0::'a::\text{ring-char-0})$

**proof** –

**from** *in-Ints* **have**  $a \in \text{range of-int}$  **unfolding** *Ints-def* [*symmetric*] .

**then obtain**  $z$  **where**  $a = \text{of-int } z$  ..

**show** *?thesis*

**proof**

**assume** *eq*:  $1 + a + a = 0$

**hence**  $\text{of-int } (1 + z + z) = (\text{of-int } 0 :: 'a)$  **by** (*simp add: a*)

**hence**  $1 + z + z = 0$  **by** (*simp only: of-int-eq-iff*)

**with** *odd-nonzero* **show** *False* **by** *blast*

**qed**

**qed**

**lemma** *Ints-number-of* [*simp*]:

$(\text{number-of } w :: 'a::\text{number-ring}) \in \text{Ints}$

**unfolding** *number-of-eq* *Ints-def* **by** *simp*

**lemma** *Nats-number-of* [*simp*]:

$\text{Int.Pls} \leq w \implies (\text{number-of } w :: 'a::\text{number-ring}) \in \text{Nats}$

**unfolding** *Int.Pls-def* *number-of-eq*

by (simp only: of-nat-nat [symmetric] of-nat-in-Nats)

**lemma** *Ints-odd-less-0*:

assumes *in-Ints*:  $a \in \text{Ints}$

shows  $(1 + a + a < 0) = (a < (0::'a::\text{linordered-idom}))$

**proof** –

from *in-Ints* have  $a \in \text{range of-int}$  **unfolding** *Ints-def* [symmetric] .

then obtain  $z$  where  $a = \text{of-int } z$  ..

hence  $((1::'a) + a + a < 0) = (\text{of-int } (1 + z + z) < (\text{of-int } 0 :: 'a))$

by (simp add: a)

also have  $\dots = (z < 0)$  by (simp only: of-int-less-iff odd-less-0)

also have  $\dots = (a < 0)$  by (simp add: a)

finally show ?thesis .

qed

### 30.12 setsum and setprod

**lemma** *of-nat-setsum*:  $\text{of-nat } (\text{setsum } f \ A) = (\sum x \in A. \text{of-nat}(f \ x))$

apply (cases finite A)

apply (erule finite-induct, auto)

done

**lemma** *of-int-setsum*:  $\text{of-int } (\text{setsum } f \ A) = (\sum x \in A. \text{of-int}(f \ x))$

apply (cases finite A)

apply (erule finite-induct, auto)

done

**lemma** *of-nat-setprod*:  $\text{of-nat } (\text{setprod } f \ A) = (\prod x \in A. \text{of-nat}(f \ x))$

apply (cases finite A)

apply (erule finite-induct, auto simp add: of-nat-mult)

done

**lemma** *of-int-setprod*:  $\text{of-int } (\text{setprod } f \ A) = (\prod x \in A. \text{of-int}(f \ x))$

apply (cases finite A)

apply (erule finite-induct, auto)

done

**lemmas** *int-setsum* = *of-nat-setsum* [where 'a=int]

**lemmas** *int-setprod* = *of-nat-setprod* [where 'a=int]

### 30.13 Inequality Reasoning for the Arithmetic Simproc

**lemma** *add-numeral-0*:  $\text{Numeral0} + a = (a::'a::\text{number-ring})$

by simp

**lemma** *add-numeral-0-right*:  $a + \text{Numeral0} = (a::'a::\text{number-ring})$

by simp

**lemma** *mult-numeral-1*:  $\text{Numeral1} * a = (a::'a::\text{number-ring})$

by simp

**lemma** *mult-numeral-1-right*:  $a * \text{Numeral1} = (a::'a::\text{number-ring})$   
**by** *simp*

**lemma** *divide-numeral-1*:  $a / \text{Numeral1} = (a::'a::\{\text{number-ring}, \text{field}\})$   
**by** *simp*

**lemma** *inverse-numeral-1*:  
 $\text{inverse } \text{Numeral1} = (\text{Numeral1}::'a::\{\text{number-ring}, \text{field}\})$   
**by** *simp*

Theorem lists for the cancellation simprocs. The use of binary numerals for 0 and 1 reduces the number of special cases.

**lemmas** *add-0s* = *add-numeral-0 add-numeral-0-right*  
**lemmas** *mult-1s* = *mult-numeral-1 mult-numeral-1-right*  
*mult-minus1 mult-minus1-right*

### 30.14 Special Arithmetic Rules for Abstract 0 and 1

Arithmetic computations are defined for binary literals, which leaves 0 and 1 as special cases. Addition already has rules for 0, but not 1. Multiplication and unary minus already have rules for both 0 and 1.

**lemma** *binop-eq*:  $[[f\ x\ y = g\ x\ y; x = x'; y = y']] ==> f\ x'\ y' = g\ x'\ y'$   
**by** *simp*

**lemmas** *add-number-of-eq* = *number-of-add [symmetric]*

Allow 1 on either or both sides

**lemma** *one-add-one-is-two*:  $1 + 1 = (2::'a::\text{number-ring})$   
**by** (*simp del: numeral-1-eq-1 add: numeral-1-eq-1 [symmetric]*)

**lemmas** *add-special* =  
*one-add-one-is-two*  
*binop-eq [of op +, OF add-number-of-eq numeral-1-eq-1 refl, standard]*  
*binop-eq [of op +, OF add-number-of-eq refl numeral-1-eq-1, standard]*

Allow 1 on either or both sides (1-1 already simplifies to 0)

**lemmas** *diff-special* =  
*binop-eq [of op -, OF diff-number-of-eq numeral-1-eq-1 refl, standard]*  
*binop-eq [of op -, OF diff-number-of-eq refl numeral-1-eq-1, standard]*

Allow 0 or 1 on either side with a binary numeral on the other

**lemmas** *eq-special* =  
*binop-eq [of op =, OF eq-number-of-eq numeral-0-eq-0 refl, standard]*  
*binop-eq [of op =, OF eq-number-of-eq numeral-1-eq-1 refl, standard]*  
*binop-eq [of op =, OF eq-number-of-eq refl numeral-0-eq-0, standard]*

*binop-eq [of op =, OF eq-number-of-eq refl numeral-1-eq-1, standard]*

Allow 0 or 1 on either side with a binary numeral on the other

**lemmas** *less-special* =

*binop-eq [of op <, OF less-number-of numeral-0-eq-0 refl, standard]*  
*binop-eq [of op <, OF less-number-of numeral-1-eq-1 refl, standard]*  
*binop-eq [of op <, OF less-number-of refl numeral-0-eq-0, standard]*  
*binop-eq [of op <, OF less-number-of refl numeral-1-eq-1, standard]*

Allow 0 or 1 on either side with a binary numeral on the other

**lemmas** *le-special* =

*binop-eq [of op ≤, OF le-number-of numeral-0-eq-0 refl, standard]*  
*binop-eq [of op ≤, OF le-number-of numeral-1-eq-1 refl, standard]*  
*binop-eq [of op ≤, OF le-number-of refl numeral-0-eq-0, standard]*  
*binop-eq [of op ≤, OF le-number-of refl numeral-1-eq-1, standard]*

**lemmas** *arith-special*[simp] =

*add-special diff-special eq-special less-special le-special*

Legacy theorems

**lemmas** *zle-int* = *of-nat-le-iff* [where 'a=int]

**lemmas** *int-int-eq* = *of-nat-eq-iff* [where 'a=int]

### 30.15 Setting up simplification procedures

**lemmas** *int-arith-rules* =

*neg-le-iff-le numeral-0-eq-0 numeral-1-eq-1*  
*minus-zero diff-minus left-minus right-minus*  
*mult-zero-left mult-zero-right mult-Bit1 mult-1-left mult-1-right*  
*mult-minus-left mult-minus-right*  
*minus-add-distrib minus-minus mult-assoc*  
*of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult*  
*of-int-0 of-int-1 of-int-add of-int-mult*

**use** *Tools/int-arith.ML*

**setup** *« Int-Arith.global-setup »*

**declaration** *« K Int-Arith.setup »*

**setup** *«*

*Reorient-Proc.add*

*(fn Const (@{const-name number-of}, -) \$ - => true | - => false)*

*»*

**simproc-setup** *reorient-numeral (number-of w = x) = Reorient-Proc.proc*

### 30.16 Lemmas About Small Numerals

**lemma** *of-int-m1* [simp]: *of-int -1 = (-1 :: 'a :: number-ring)*

**proof** —

```

have (of-int -1 :: 'a) = of-int (- 1) by simp
also have ... = - of-int 1 by (simp only: of-int-minus)
also have ... = -1 by simp
finally show ?thesis .
qed

```

```

lemma abs-minus-one [simp]: abs (-1) = (1::'a::{linordered-idom,number-ring})
by (simp add: abs-if)

```

```

lemma abs-power-minus-one [simp]:
  abs(-1 ^ n) = (1::'a::{linordered-idom,number-ring})
by (simp add: power-abs)

```

```

lemma of-int-number-of-eq [simp]:
  of-int (number-of v) = (number-of v :: 'a :: number-ring)
by (simp add: number-of-eq)

```

Lemmas for specialist use, NOT as default simprules

```

lemma mult-2: 2 * z = (z+z::'a::number-ring)
unfolding one-add-one-is-two [symmetric] left-distrib by simp

```

```

lemma mult-2-right: z * 2 = (z+z::'a::number-ring)
by (subst mult-commute, rule mult-2)

```

### 30.17 More Inequality Reasoning

```

lemma zless-add1-eq: (w < z + (1::int)) = (w < z | w = z)
by arith

```

```

lemma add1-zle-eq: (w + (1::int) ≤ z) = (w < z)
by arith

```

```

lemma zle-diff1-eq [simp]: (w ≤ z - (1::int)) = (w < z)
by arith

```

```

lemma zle-add1-eq-le [simp]: (w < z + (1::int)) = (w ≤ z)
by arith

```

```

lemma int-one-le-iff-zero-less: ((1::int) ≤ z) = (0 < z)
by arith

```

### 30.18 The functions *nat* and *int*

Simplify the terms *int* (*0*::'a), *int* (*Suc 0*) and *w* + - *z*

```

declare Zero-int-def [symmetric, simp]
declare One-int-def [symmetric, simp]

```

```

lemmas diff-int-def-symmetric = diff-int-def [symmetric, simp]

```



```
lemma nat-0: nat 0 = 0
by (simp add: nat-eq-iff)
```

```
lemma nat-1: nat 1 = Suc 0
by (subst nat-eq-iff, simp)
```

```
lemma nat-2: nat 2 = Suc (Suc 0)
by (subst nat-eq-iff, simp)
```

```
lemma one-less-nat-eq [simp]: (Suc 0 < nat z) = (1 < z)
apply (insert zless-nat-conj [of 1 z])
apply (auto simp add: nat-1)
done
```

This simplifies expressions of the form  $\text{int } n = z$  where  $z$  is an integer literal.

```
lemmas int-eq-iff-number-of [simp] = int-eq-iff [of - number-of v, standard]
```

```
lemma split-nat [arith-split]:
  P(nat(i::int)) = (( $\forall n. i = \text{of-nat } n \longrightarrow P n$ ) & ( $i < 0 \longrightarrow P 0$ ))
  (is ?P = (?L & ?R))
proof (cases i < 0)
  case True thus ?thesis by auto
next
  case False
  have ?P = ?L
  proof
    assume ?P thus ?L using False by clarsimp
  next
    assume ?L thus ?P using False by simp
  qed
  with False show ?thesis by simp
qed
```

```
context ring-1
begin
```

```
lemma of-int-of-nat [nitpick-simp]:
  of-int k = (if k < 0 then - of-nat (nat (- k)) else of-nat (nat k))
proof (cases k < 0)
  case True then have  $0 \leq - k$  by simp
  then have of-nat (nat (- k)) = of-int (- k) by (rule of-nat-nat)
  with True show ?thesis by simp
next
  case False then show ?thesis by (simp add: not-less of-nat-nat)
qed

end
```

```

lemma nat-mult-distrib:
  fixes  $z\ z' :: \text{int}$ 
  assumes  $0 \leq z$ 
  shows  $\text{nat } (z * z') = \text{nat } z * \text{nat } z'$ 
proof (cases  $0 \leq z'$ )
  case False with assms have  $z * z' \leq 0$ 
    by (simp add: not-le mult-le-0-iff)
  then have  $\text{nat } (z * z') = 0$  by simp
  moreover from False have  $\text{nat } z' = 0$  by simp
  ultimately show ?thesis by simp
next
  case True with assms have ge-0:  $z * z' \geq 0$  by (simp add: zero-le-mult-iff)
  show ?thesis
    by (rule injD [of of-nat :: nat  $\Rightarrow$  int, OF inj-of-nat])
      (simp only: of-nat-mult of-nat-nat [OF True])
      (of-nat-nat [OF assms] of-nat-nat [OF ge-0], simp)
qed

lemma nat-mult-distrib-neg:  $z \leq (0::\text{int}) \implies \text{nat}(z * z') = \text{nat}(-z) * \text{nat}(-z')$ 
apply (rule trans)
apply (rule-tac [2] nat-mult-distrib, auto)
done

lemma nat-abs-mult-distrib:  $\text{nat } (\text{abs } (w * z)) = \text{nat } (\text{abs } w) * \text{nat } (\text{abs } z)$ 
apply (cases  $z=0 \mid w=0$ )
apply (auto simp add: abs-if nat-mult-distrib [symmetric])
      (nat-mult-distrib-neg [symmetric] mult-less-0-iff)
done

```

### 30.19 Induction principles for int

Well-founded segments of the integers

**definition**

*int-ge-less-than* ::  $\text{int} \Rightarrow (\text{int} * \text{int}) \text{ set}$

**where**

*int-ge-less-than*  $d = \{(z', z). d \leq z' \ \& \ z' < z\}$

**theorem** *wf-int-ge-less-than*:  $\text{wf } (\text{int-ge-less-than } d)$

**proof** –

**have** *int-ge-less-than*  $d \subseteq \text{measure } (\%z. \text{nat } (z - d))$

**by** (*auto simp add: int-ge-less-than-def*)

**thus** *?thesis*

**by** (*rule wf-subset [OF wf-measure]*)

**qed**

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

**definition**

$int\text{-}ge\text{-}less\text{-}than2 :: int \Rightarrow (int * int) \text{ set}$   
**where**  
 $int\text{-}ge\text{-}less\text{-}than2\ d = \{(z', z). d \leq z \ \& \ z' < z\}$   
**theorem**  $wf\text{-}int\text{-}ge\text{-}less\text{-}than2$ :  $wf\ (int\text{-}ge\text{-}less\text{-}than2\ d)$   
**proof** –  
 have  $int\text{-}ge\text{-}less\text{-}than2\ d \subseteq measure\ (\%z. nat\ (1 + z - d))$   
 by  $(auto\ simp\ add: int\text{-}ge\text{-}less\text{-}than2\text{-}def)$   
 thus  $?thesis$   
 by  $(rule\ wf\text{-}subset\ [OF\ wf\text{-}measure])$   
**qed**

**abbreviation**

$int :: nat \Rightarrow int$   
**where**  
 $int \equiv of\text{-}nat$

**theorem**  $int\text{-}ge\text{-}induct\ [case\text{-}names\ base\ step, induct\ set: int]$ :  
 fixes  $i :: int$   
 assumes  $ge: k \leq i$  **and**  
    $base: P\ k$  **and**  
    $step: \bigwedge i. k \leq i \implies P\ i \implies P\ (i + 1)$   
 shows  $P\ i$   
**proof** –  
 { fix  $n$  have  $\bigwedge i::int. n = nat(i - k) \implies k \leq i \implies P\ i$   
   **proof**  $(induct\ n)$   
     **case**  $0$   
       hence  $i = k$  **by**  $arith$   
       thus  $P\ i$  **using**  $base$  **by**  $simp$   
     **next**  
       **case**  $(Suc\ n)$   
       **then** have  $n = nat((i - 1) - k)$  **by**  $arith$   
       **moreover**  
       have  $ki1: k \leq i - 1$  **using**  $Suc.prem$ s **by**  $arith$   
       **ultimately**  
       have  $P(i - 1)$  **by**  $(rule\ Suc.hyps)$   
       **from**  $step[OF\ ki1\ this]$  **show**  $?case$  **by**  $simp$   
     **qed**  
   }  
 with  $ge$  **show**  $?thesis$  **by**  $fast$   
**qed**

**theorem**  $int\text{-}gr\text{-}induct\ [case\text{-}names\ base\ step, induct\ set: int]$ :  
 assumes  $gr: k < (i::int)$  **and**  
    $base: P(k + 1)$  **and**  
    $step: \bigwedge i. \llbracket k < i; P\ i \rrbracket \implies P(i + 1)$   
 shows  $P\ i$

```

apply(rule int-ge-induct[of  $k + 1$ ])
  using gr apply arith
  apply(rule base)
apply (rule step, simp+)
done

```

```

theorem int-le-induct[consumes 1, case-names base step]:
  assumes le:  $i \leq (k :: \text{int})$  and
    base:  $P(k)$  and
    step:  $\bigwedge i. \llbracket i \leq k; P\ i \rrbracket \implies P(i - 1)$ 
  shows  $P\ i$ 
proof –
  { fix n have  $\bigwedge i :: \text{int}. n = \text{nat}(k - i) \implies i \leq k \implies P\ i$ 
    proof (induct n)
      case 0
      hence  $i = k$  by arith
      thus  $P\ i$  using base by simp
    next
      case (Suc n)
      hence  $n = \text{nat}(k - (i + 1))$  by arith
      moreover
      have  $ki1: i + 1 \leq k$  using Suc.prems by arith
      ultimately
      have  $P(i + 1)$  by(rule Suc.hyps)
      from step[OF ki1 this] show ?case by simp
    qed
  }
  with le show ?thesis by fast
qed

```

```

theorem int-less-induct [consumes 1, case-names base step]:
  assumes less:  $(i :: \text{int}) < k$  and
    base:  $P(k - 1)$  and
    step:  $\bigwedge i. \llbracket i < k; P\ i \rrbracket \implies P(i - 1)$ 
  shows  $P\ i$ 
apply(rule int-le-induct[of  $k - 1$ ])
  using less apply arith
  apply(rule base)
apply (rule step, simp+)
done

```

```

theorem int-induct [case-names base step1 step2]:
  fixes  $k :: \text{int}$ 
  assumes base:  $P\ k$ 
    and step1:  $\bigwedge i. k \leq i \implies P\ i \implies P\ (i + 1)$ 
    and step2:  $\bigwedge i. k \geq i \implies P\ i \implies P\ (i - 1)$ 
  shows  $P\ i$ 
proof –
  have  $i \leq k \vee i \geq k$  by arith

```

```

then show ?thesis proof
  assume  $i \geq k$  then show ?thesis using base
    by (rule int-ge-induct) (fact step1)
  next
    assume  $i \leq k$  then show ?thesis using base
      by (rule int-le-induct) (fact step2)
qed
qed

```

### 30.20 Intermediate value theorems

```

lemma int-val-lemma:
  ( $\forall i < n::nat. \text{abs}(f(i+1) - f i) \leq 1$ )  $\longrightarrow$ 
   $f 0 \leq k \longrightarrow k \leq f n \longrightarrow (\exists i \leq n. f i = (k::int))$ 
unfolding One-nat-def
apply (induct n, simp)
apply (intro strip)
apply (erule impE, simp)
apply (erule-tac  $x = n$  in allE, simp)
apply (case-tac  $k = f (Suc n)$ )
apply force
apply (erule impE)
  apply (simp add: abs-if split add: split-if-asm)
  apply (blast intro: le-SucI)
done

```

```

lemmas nat0-intermed-int-val = int-val-lemma [rule-format (no-asm)]

```

```

lemma nat-intermed-int-val:
  [|  $\forall i. m \leq i \ \& \ i < n \longrightarrow \text{abs}(f(i+1::nat) - f i) \leq 1$ ;  $m < n$ ;
     $f m \leq k$ ;  $k \leq f n$  |]  $\implies ? i. m \leq i \ \& \ i \leq n \ \& \ f i = (k::int)$ 
apply (cut-tac  $n = n - m$  and  $f = \%i. f (i+m)$  and  $k = k$ 
  in int-val-lemma)
unfolding One-nat-def
apply simp
apply (erule exE)
apply (rule-tac  $x = i+m$  in exI, arith)
done

```

### 30.21 Products and 1, by T. M. Rasmussen

```

lemma zabs-less-one-iff [simp]: ( $|z| < 1$ ) = ( $z = (0::int)$ )
by arith

```

```

lemma abs-zmult-eq-1:
  assumes  $mn: |m * n| = 1$ 
  shows  $|m| = (1::int)$ 
proof -
  have  $0: m \neq 0 \ \& \ n \neq 0$  using mn
  by auto

```

```

have ~ (2 ≤ |m|)
proof
  assume 2 ≤ |m|
  hence 2*|n| ≤ |m|*|n|
    by (simp add: mult-mono 0)
  also have ... = |m*n|
    by (simp add: abs-mult)
  also have ... = 1
    by (simp add: mn)
  finally have 2*|n| ≤ 1 .
  thus False using 0
    by auto
qed
thus ?thesis using 0
  by auto
qed

```

```

lemma pos-zmult-eq-1-iff-lemma: (m * n = 1) ==> m = (1::int) | m = -1
by (insert abs-zmult-eq-1 [of m n], arith)

```

```

lemma pos-zmult-eq-1-iff:
  assumes 0 < (m::int) shows (m * n = 1) = (m = 1 & n = 1)
proof -
  from assms have m * n = 1 ==> m = 1 by (auto dest: pos-zmult-eq-1-iff-lemma)
  thus ?thesis by (auto dest: pos-zmult-eq-1-iff-lemma)
qed

```

```

lemma zmult-eq-1-iff: (m*n = (1::int)) = ((m = 1 & n = 1) | (m = -1 & n =
-1))
apply (rule iffI)
apply (frule pos-zmult-eq-1-iff-lemma)
apply (simp add: mult-commute [of m])
apply (frule pos-zmult-eq-1-iff-lemma, auto)
done

```

```

lemma infinite-UNIV-int: ¬ finite (UNIV::int set)
proof
  assume finite (UNIV::int set)
  moreover have inj (λi::int. 2 * i)
    by (rule injI) simp
  ultimately have surj (λi::int. 2 * i)
    by (rule finite-UNIV-inj-surj)
  then obtain i :: int where 1 = 2 * i by (rule surjE)
  then show False by (simp add: pos-zmult-eq-1-iff)
qed

```

## 30.22 Further theorems on numerals

### 30.22.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

**lemmas** *left-distrib-number-of [simp] =*  
*left-distrib [of - - number-of v, standard]*

**lemmas** *right-distrib-number-of [simp] =*  
*right-distrib [of number-of v, standard]*

**lemmas** *left-diff-distrib-number-of [simp] =*  
*left-diff-distrib [of - - number-of v, standard]*

**lemmas** *right-diff-distrib-number-of [simp] =*  
*right-diff-distrib [of number-of v, standard]*

These are actually for fields, like real: but where else to put them?

**lemmas** *zero-less-divide-iff-number-of [simp, no-atp] =*  
*zero-less-divide-iff [of number-of w, standard]*

**lemmas** *divide-less-0-iff-number-of [simp, no-atp] =*  
*divide-less-0-iff [of number-of w, standard]*

**lemmas** *zero-le-divide-iff-number-of [simp, no-atp] =*  
*zero-le-divide-iff [of number-of w, standard]*

**lemmas** *divide-le-0-iff-number-of [simp, no-atp] =*  
*divide-le-0-iff [of number-of w, standard]*

Replaces *inverse #nn* by  $1/\#nn$ . It looks strange, but then other *simprocs* simplify the quotient.

**lemmas** *inverse-eq-divide-number-of [simp] =*  
*inverse-eq-divide [of number-of w, standard]*

These laws simplify inequalities, moving unary minus from a term into the literal.

**lemmas** *less-minus-iff-number-of [simp, no-atp] =*  
*less-minus-iff [of number-of v, standard]*

**lemmas** *le-minus-iff-number-of [simp, no-atp] =*  
*le-minus-iff [of number-of v, standard]*

**lemmas** *equation-minus-iff-number-of [simp, no-atp] =*  
*equation-minus-iff [of number-of v, standard]*

**lemmas** *minus-less-iff-number-of [simp, no-atp] =*  
*minus-less-iff [of - number-of v, standard]*

**lemmas** *minus-le-iff-number-of* [*simp*, *no-atp*] =  
*minus-le-iff* [*of* - *number-of* *v*, *standard*]

**lemmas** *minus-equation-iff-number-of* [*simp*, *no-atp*] =  
*minus-equation-iff* [*of* - *number-of* *v*, *standard*]

To Simplify Inequalities Where One Side is the Constant 1

**lemma** *less-minus-iff-1* [*simp*,*no-atp*]:  
**fixes** *b::'b::*{*linordered-idom*,*number-ring*}  
**shows**  $(1 < - b) = (b < -1)$   
**by** *auto*

**lemma** *le-minus-iff-1* [*simp*,*no-atp*]:  
**fixes** *b::'b::*{*linordered-idom*,*number-ring*}  
**shows**  $(1 \leq - b) = (b \leq -1)$   
**by** *auto*

**lemma** *equation-minus-iff-1* [*simp*,*no-atp*]:  
**fixes** *b::'b::number-ring*  
**shows**  $(1 = - b) = (b = -1)$   
**by** (*subst equation-minus-iff*, *auto*)

**lemma** *minus-less-iff-1* [*simp*,*no-atp*]:  
**fixes** *a::'a::*{*linordered-idom*,*number-ring*}  
**shows**  $(- a < 1) = (-1 < a)$   
**by** *auto*

**lemma** *minus-le-iff-1* [*simp*,*no-atp*]:  
**fixes** *a::'a::*{*linordered-idom*,*number-ring*}  
**shows**  $(- a \leq 1) = (-1 \leq a)$   
**by** *auto*

**lemma** *minus-equation-iff-1* [*simp*,*no-atp*]:  
**fixes** *a::'a::number-ring*  
**shows**  $(- a = 1) = (a = -1)$   
**by** (*subst minus-equation-iff*, *auto*)

Cancellation of constant factors in comparisons ( $<$  and  $\leq$ )

**lemmas** *mult-less-cancel-left-number-of* [*simp*, *no-atp*] =  
*mult-less-cancel-left* [*of* *number-of* *v*, *standard*]

**lemmas** *mult-less-cancel-right-number-of* [*simp*, *no-atp*] =  
*mult-less-cancel-right* [*of* - *number-of* *v*, *standard*]

**lemmas** *mult-le-cancel-left-number-of* [*simp*, *no-atp*] =  
*mult-le-cancel-left* [*of* *number-of* *v*, *standard*]

**lemmas** *mult-le-cancel-right-number-of* [*simp*, *no-atp*] =



*mult-le-cancel-right* [of - number-of  $v$ , standard]

Multiplying out constant divisors in comparisons ( $<$ ,  $\leq$  and  $=$ )

**lemmas** *le-divide-eq-number-of1* [simp] = *le-divide-eq* [of - - number-of  $w$ , standard]

**lemmas** *divide-le-eq-number-of1* [simp] = *divide-le-eq* [of - - number-of  $w$ , standard]

**lemmas** *less-divide-eq-number-of1* [simp] = *less-divide-eq* [of - - number-of  $w$ , standard]

**lemmas** *divide-less-eq-number-of1* [simp] = *divide-less-eq* [of - - number-of  $w$ , standard]

**lemmas** *eq-divide-eq-number-of1* [simp] = *eq-divide-eq* [of - - number-of  $w$ , standard]

**lemmas** *divide-eq-eq-number-of1* [simp] = *divide-eq-eq* [of - - number-of  $w$ , standard]

### 30.22.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

**lemmas** *le-divide-eq-number-of* = *le-divide-eq* [of number-of  $w$ , standard]

**lemmas** *divide-le-eq-number-of* = *divide-le-eq* [of - - number-of  $w$ , standard]

**lemmas** *less-divide-eq-number-of* = *less-divide-eq* [of number-of  $w$ , standard]

**lemmas** *divide-less-eq-number-of* = *divide-less-eq* [of - - number-of  $w$ , standard]

**lemmas** *eq-divide-eq-number-of* = *eq-divide-eq* [of number-of  $w$ , standard]

**lemmas** *divide-eq-eq-number-of* = *divide-eq-eq* [of - - number-of  $w$ , standard]

Not good as automatic simprules because they cause case splits.

**lemmas** *divide-const-simps* =

*le-divide-eq-number-of divide-le-eq-number-of less-divide-eq-number-of*  
*divide-less-eq-number-of eq-divide-eq-number-of divide-eq-eq-number-of*  
*le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1*

Division By  $-1$

**lemma** *divide-minus1* [simp]:

$x / -1 = -(x :: 'a :: \{\text{field-inverse-zero, number-ring}\})$

**by** *simp*

**lemma** *minus1-divide* [simp]:

$-1 / (x :: 'a :: \{\text{field-inverse-zero, number-ring}\}) = -(1/x)$

**by** (*simp add: divide-inverse*)

**lemma** *half-gt-zero-iff*:

$(0 < r/2) = (0 < (r :: 'a :: \{\text{linordered-field-inverse-zero, number-ring}\}))$

**by** *auto*

**lemmas** *half-gt-zero* [simp] = *half-gt-zero-iff* [THEN *iffD2*, standard]

**lemma** *divide-Numeral1*:

$(x :: 'a :: \{\text{field, number-ring}\}) / \text{Numeral1} = x$

**by** *simp*

```

lemma divide-Numeral0:
  (x::'a::{field-inverse-zero, number-ring}) / Numeral0 = 0
by simp

```

### 30.23 The divides relation

```

lemma zdvd-antisym-nonneg:
  0 <= m ==> 0 <= n ==> m dvd n ==> n dvd m ==> m = (n::int)
apply (simp add: dvd-def, auto)
apply (auto simp add: mult-assoc zero-le-mult-iff zmult-eq-1-iff)
done

```

```

lemma zdvd-antisym-abs: assumes (a::int) dvd b and b dvd a
shows |a| = |b|
proof cases
  assume a = 0 with assms show ?thesis by simp
next
  assume a ≠ 0
  from ⟨a dvd b⟩ obtain k where k:b = a*k unfolding dvd-def by blast
  from ⟨b dvd a⟩ obtain k' where k':a = b*k' unfolding dvd-def by blast
  from k k' have a = a*k*k' by simp
  with mult-cancel-left1[where c=a and b=k*k']
  have k*k':k*k' = 1 using ⟨a≠0⟩ by (simp add: mult-assoc)
  hence k = 1 ∧ k' = 1 ∨ k = -1 ∧ k' = -1 by (simp add: zmult-eq-1-iff)
  thus ?thesis using k k' by auto
qed

```

```

lemma zdvd-zdiffD: k dvd m - n ==> k dvd n ==> k dvd (m::int)
apply (subgoal-tac m = n + (m - n))
apply (erule ssubst)
apply (blast intro: dvd-add, simp)
done

```

```

lemma zdvd-reduce: (k dvd n + k * m) = (k dvd (n::int))
apply (rule iffI)
apply (erule-tac [2] dvd-add)
apply (subgoal-tac n = (n + k * m) - k * m)
apply (erule ssubst)
apply (erule dvd-diff)
apply (simp-all)
done

```

```

lemma dvd-imp-le-int:
  fixes d i :: int
  assumes i ≠ 0 and d dvd i
  shows |d| ≤ |i|
proof -
  from ⟨d dvd i⟩ obtain k where i = d * k ..

```

```

with  $\langle i \neq 0 \rangle$  have  $k \neq 0$  by auto
then have  $1 \leq |k|$  and  $0 \leq |d|$  by auto
then have  $|d| * 1 \leq |d| * |k|$  by (rule mult-left-mono)
with  $\langle i = d * k \rangle$  show ?thesis by (simp add: abs-mult)
qed

```

```

lemma zdvd-not-zless:
  fixes  $m\ n :: \text{int}$ 
  assumes  $0 < m$  and  $m < n$ 
  shows  $\neg n \text{ dvd } m$ 
proof
  from assms have  $0 < n$  by auto
  assume  $n \text{ dvd } m$  then obtain  $k$  where  $k: m = n * k$  ..
  with  $\langle 0 < m \rangle$  have  $0 < n * k$  by auto
  with  $\langle 0 < n \rangle$  have  $0 < k$  by (simp add: zero-less-mult-iff)
  with  $k \langle 0 < n \rangle \langle m < n \rangle$  have  $n * k < n * 1$  by simp
  with  $\langle 0 < n \rangle \langle 0 < k \rangle$  show False unfolding mult-less-cancel-left by auto
qed

```

```

lemma zdvd-mult-cancel: assumes  $d:k * m \text{ dvd } k * n$  and  $kz:k \neq (0::\text{int})$ 
  shows  $m \text{ dvd } n$ 
proof–
  from  $d$  obtain  $h$  where  $h: k*n = k*m * h$  unfolding dvd-def by blast
  {assume  $n \neq m*h$  hence  $k*n \neq k*(m*h)$  using  $kz$  by simp
   with  $h$  have False by (simp add: mult-assoc)}
  hence  $n = m * h$  by blast
  thus ?thesis by simp
qed

```

```

theorem zdvd-int:  $(x \text{ dvd } y) = (\text{int } x \text{ dvd int } y)$ 
proof –
  have  $\bigwedge k. \text{int } y = \text{int } x * k \implies x \text{ dvd } y$ 
  proof –
    fix  $k$ 
    assume  $A: \text{int } y = \text{int } x * k$ 
    then show  $x \text{ dvd } y$  proof (cases k)
      case  $(1\ n)$  with  $A$  have  $y = x * n$  by (simp add: of-nat-mult [symmetric])
      then show ?thesis ..
    next
      case  $(2\ n)$  with  $A$  have  $\text{int } y = \text{int } x * (- \text{int } (\text{Suc } n))$  by simp
      also have  $\dots = - (\text{int } x * \text{int } (\text{Suc } n))$  by (simp only: mult-minus-right)
      also have  $\dots = - \text{int } (x * \text{Suc } n)$  by (simp only: of-nat-mult [symmetric])
      finally have  $- \text{int } (x * \text{Suc } n) = \text{int } y$  ..
      then show ?thesis by (simp only: negative-eq-positive) auto
    qed
  qed
  then show ?thesis by (auto elim!: dvdE simp only: dvd-triv-left of-nat-mult)
qed

```

```

lemma zdvd1-eq[simp]: (x::int) dvd 1 = (|x| = 1)
proof
  assume d: x dvd 1 hence int (nat |x|) dvd int (nat 1) by simp
  hence nat |x| dvd 1 by (simp add: zdvd-int)
  hence nat |x| = 1 by simp
  thus |x| = 1 by (cases x < 0, auto)
next
  assume |x|=1
  then have x = 1 ∨ x = -1 by auto
  then show x dvd 1 by (auto intro: dvdI)
qed

lemma zdvd-mult-cancel1:
  assumes mp:m ≠(0::int) shows (m * n dvd m) = (|n| = 1)
proof
  assume n1: |n| = 1 thus m * n dvd m
    by (cases n > 0, auto simp add: minus-equation-iff)
next
  assume H: m * n dvd m hence H2: m * n dvd m * 1 by simp
  from zdvd-mult-cancel[OF H2 mp] show |n| = 1 by (simp only: zdvd1-eq)
qed

lemma int-dvd-iff: (int m dvd z) = (m dvd nat (abs z))
  unfolding zdvd-int by (cases z ≥ 0) simp-all

lemma dvd-int-iff: (z dvd int m) = (nat (abs z) dvd m)
  unfolding zdvd-int by (cases z ≥ 0) simp-all

lemma nat-dvd-iff: (nat z dvd m) = (if 0 ≤ z then (z dvd int m) else m = 0)
  by (auto simp add: dvd-int-iff)

lemma eq-nat-nat-iff:
  0 ≤ z ⇒ 0 ≤ z' ⇒ nat z = nat z' ⟷ z = z'
  by (auto elim!: nonneg-eq-int)

lemma nat-power-eq:
  0 ≤ z ⇒ nat (z ^ n) = nat z ^ n
  by (induct n) (simp-all add: nat-mult-distrib)

lemma zdvd-imp-le: [| z dvd n; 0 < n |] ==> z ≤ (n::int)
  apply (rule-tac z=n in int-cases)
  apply (auto simp add: dvd-int-iff)
  apply (rule-tac z=z in int-cases)
  apply (auto simp add: dvd-imp-le)
  done

lemma zdvd-period:
  fixes a d :: int
  assumes a dvd d

```

```

shows a dvd (x + t)  $\longleftrightarrow$  a dvd ((x + c * d) + t)
proof -
  from assms obtain k where d = a * k by (rule dvdE)
  show ?thesis proof
    assume a dvd (x + t)
    then obtain l where x + t = a * l by (rule dvdE)
    then have x = a * l - t by simp
    with (d = a * k) show a dvd x + c * d + t by simp
  next
    assume a dvd x + c * d + t
    then obtain l where x + c * d + t = a * l by (rule dvdE)
    then have x = a * l - c * d - t by simp
    with (d = a * k) show a dvd (x + t) by simp
  qed
qed

```

### 30.24 Configuration of the code generator

**code-datatype** *Pls Min Bit0 Bit1 number-of* :: *int*  $\Rightarrow$  *int*

**lemmas** *pred-succ-numeral-code* [code] =  
*pred-bin-simps succ-bin-simps*

**lemmas** *plus-numeral-code* [code] =  
*add-bin-simps*  
*arith-extra-simps*(1) [where 'a = int]

**lemmas** *minus-numeral-code* [code] =  
*minus-bin-simps*  
*arith-extra-simps*(2) [where 'a = int]  
*arith-extra-simps*(5) [where 'a = int]

**lemmas** *times-numeral-code* [code] =  
*mult-bin-simps*  
*arith-extra-simps*(4) [where 'a = int]

**instantiation** *int* :: *eq*  
**begin**

**definition** [code del]: *eq-class.eq* k l  $\longleftrightarrow$  k - l = (0::int)

**instance** by default (*simp add: eq-int-def*)

**end**

**lemma** *eq-number-of-int-code* [code]:  
*eq-class.eq* (number-of k :: int) (number-of l)  $\longleftrightarrow$  *eq-class.eq* k l  
**unfolding** *eq-int-def number-of-is-id* ..

**lemma** *eq-int-code* [code]:  
 $eq\_class.eq\ Int.Pl\ Int.Pl \longleftrightarrow True$   
 $eq\_class.eq\ Int.Pl\ Int.Min \longleftrightarrow False$   
 $eq\_class.eq\ Int.Pl\ (Int.Bit0\ k2) \longleftrightarrow eq\_class.eq\ Int.Pl\ k2$   
 $eq\_class.eq\ Int.Pl\ (Int.Bit1\ k2) \longleftrightarrow False$   
 $eq\_class.eq\ Int.Min\ Int.Pl \longleftrightarrow False$   
 $eq\_class.eq\ Int.Min\ Int.Min \longleftrightarrow True$   
 $eq\_class.eq\ Int.Min\ (Int.Bit0\ k2) \longleftrightarrow False$   
 $eq\_class.eq\ Int.Min\ (Int.Bit1\ k2) \longleftrightarrow eq\_class.eq\ Int.Min\ k2$   
 $eq\_class.eq\ (Int.Bit0\ k1)\ Int.Pl \longleftrightarrow eq\_class.eq\ k1\ Int.Pl$   
 $eq\_class.eq\ (Int.Bit1\ k1)\ Int.Pl \longleftrightarrow False$   
 $eq\_class.eq\ (Int.Bit0\ k1)\ Int.Min \longleftrightarrow False$   
 $eq\_class.eq\ (Int.Bit1\ k1)\ Int.Min \longleftrightarrow eq\_class.eq\ k1\ Int.Min$   
 $eq\_class.eq\ (Int.Bit0\ k1)\ (Int.Bit0\ k2) \longleftrightarrow eq\_class.eq\ k1\ k2$   
 $eq\_class.eq\ (Int.Bit0\ k1)\ (Int.Bit1\ k2) \longleftrightarrow False$   
 $eq\_class.eq\ (Int.Bit1\ k1)\ (Int.Bit0\ k2) \longleftrightarrow False$   
 $eq\_class.eq\ (Int.Bit1\ k1)\ (Int.Bit1\ k2) \longleftrightarrow eq\_class.eq\ k1\ k2$   
**unfolding** *eq-equals* **by** *simp-all*

**lemma** *eq-int-refl* [code nbe]:  
 $eq\_class.eq\ (k::int)\ k \longleftrightarrow True$   
**by** (rule *HOL.eq-refl*)

**lemma** *less-eq-number-of-int-code* [code]:  
 $(number-of\ k :: int) \leq number-of\ l \longleftrightarrow k \leq l$   
**unfolding** *number-of-is-id* ..

**lemma** *less-eq-int-code* [code]:  
 $Int.Pl \leq Int.Pl \longleftrightarrow True$   
 $Int.Pl \leq Int.Min \longleftrightarrow False$   
 $Int.Pl \leq Int.Bit0\ k \longleftrightarrow Int.Pl \leq k$   
 $Int.Pl \leq Int.Bit1\ k \longleftrightarrow Int.Pl \leq k$   
 $Int.Min \leq Int.Pl \longleftrightarrow True$   
 $Int.Min \leq Int.Min \longleftrightarrow True$   
 $Int.Min \leq Int.Bit0\ k \longleftrightarrow Int.Min < k$   
 $Int.Min \leq Int.Bit1\ k \longleftrightarrow Int.Min \leq k$   
 $Int.Bit0\ k \leq Int.Pl \longleftrightarrow k \leq Int.Pl$   
 $Int.Bit1\ k \leq Int.Pl \longleftrightarrow k < Int.Pl$   
 $Int.Bit0\ k \leq Int.Min \longleftrightarrow k \leq Int.Min$   
 $Int.Bit1\ k \leq Int.Min \longleftrightarrow k \leq Int.Min$   
 $Int.Bit0\ k1 \leq Int.Bit0\ k2 \longleftrightarrow k1 \leq k2$   
 $Int.Bit0\ k1 \leq Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$   
 $Int.Bit1\ k1 \leq Int.Bit0\ k2 \longleftrightarrow k1 < k2$   
 $Int.Bit1\ k1 \leq Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$   
**by** *simp-all*

**lemma** *less-number-of-int-code* [code]:  
 $(number-of\ k :: int) < number-of\ l \longleftrightarrow k < l$   
**unfolding** *number-of-is-id* ..

**lemma** *less-int-code* [code]:  
 $Int.Pls < Int.Pls \longleftrightarrow False$   
 $Int.Pls < Int.Min \longleftrightarrow False$   
 $Int.Pls < Int.Bit0\ k \longleftrightarrow Int.Pls < k$   
 $Int.Pls < Int.Bit1\ k \longleftrightarrow Int.Pls \leq k$   
 $Int.Min < Int.Pls \longleftrightarrow True$   
 $Int.Min < Int.Min \longleftrightarrow False$   
 $Int.Min < Int.Bit0\ k \longleftrightarrow Int.Min < k$   
 $Int.Min < Int.Bit1\ k \longleftrightarrow Int.Min < k$   
 $Int.Bit0\ k < Int.Pls \longleftrightarrow k < Int.Pls$   
 $Int.Bit1\ k < Int.Pls \longleftrightarrow k < Int.Pls$   
 $Int.Bit0\ k < Int.Min \longleftrightarrow k \leq Int.Min$   
 $Int.Bit1\ k < Int.Min \longleftrightarrow k < Int.Min$   
 $Int.Bit0\ k1 < Int.Bit0\ k2 \longleftrightarrow k1 < k2$   
 $Int.Bit0\ k1 < Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$   
 $Int.Bit1\ k1 < Int.Bit0\ k2 \longleftrightarrow k1 < k2$   
 $Int.Bit1\ k1 < Int.Bit1\ k2 \longleftrightarrow k1 < k2$   
**by** *simp-all*

**definition**

$nat\_aux :: int \Rightarrow nat \Rightarrow nat$  **where**  
 $nat\_aux\ i\ n = nat\ i + n$

**lemma** [code]:

$nat\_aux\ i\ n = (if\ i \leq 0\ then\ n\ else\ nat\_aux\ (i - 1)\ (Suc\ n))$  — tail recursive  
**by** (*auto simp add: nat-aux-def nat-eq-iff linorder-not-le order-less-imp-le*  
*dest: zless-imp-add1-zle*)

**lemma** [code]:  $nat\ i = nat\_aux\ i\ 0$   
**by** (*simp add: nat-aux-def*)

**hide-const** (**open**) *nat-aux*

**lemma** *zero-is-num-zero* [code, code-unfold-post]:  
 $(0::int) = Numeral0$   
**by** *simp*

**lemma** *one-is-num-one* [code, code-unfold-post]:  
 $(1::int) = Numeral1$   
**by** *simp*

**code-modulename** *SML*  
*Int Arith*

**code-modulename** *OCaml*  
*Int Arith*

**code-modulename** *Haskell*

*Int Arith*

**types-code**

```

  int (int)
attach (term-of) ⟨⟨
  val term-of-int = HLogic.mk-number HLogic.intT;
  ⟩⟩
attach (test) ⟨⟨
  fun gen-int i =
    let val j = one-of [~1, 1] * random-range 0 i
    in (j, fn () => term-of-int j) end;
  ⟩⟩

```

**setup** ⟨⟨

let

```

  fun strip-number-of (@{term Int.number-of :: int => int} $ t) = t
    | strip-number-of t = t;

```

```

  fun numeral-codegen thy defs dep module b t gr =
    let val i = HLogic.dest-numeral (strip-number-of t)
    in
      SOME (Codegen.str (string-of-int i),
        snd (Codegen.invoke-tycodegen thy defs dep module false HLogic.intT gr))
    end handle TERM - => NONE;

```

in

Codegen.add-codegen numeral-codegen numeral-codegen

end

⟩⟩

**consts-code**

```

  number-of :: int => int    ((-))
  0 :: int                (0)
  1 :: int                (1)
  uminus :: int => int      (~)
  op + :: int => int => int  ((- +/ -))
  op * :: int => int => int  ((- */ -))
  op ≤ :: int => int => bool ((- <=/ -))
  op < :: int => int => bool ((- </ -))

```

**quickcheck-params** [default-type = int]

**hide-const** (open) Pls Min Bit0 Bit1 succ pred



## 30.25 Legacy theorems

**lemmas** *zminus-zminus = minus-minus* [*of z::int, standard*]  
**lemmas** *zminus-0 = minus-zero* [**where** *'a=int*]  
**lemmas** *zminus-zadd-distrib = minus-add-distrib* [*of z::int w, standard*]  
**lemmas** *zadd-commute = add-commute* [*of z::int w, standard*]  
**lemmas** *zadd-assoc = add-assoc* [*of z1::int z2 z3, standard*]  
**lemmas** *zadd-left-commute = add-left-commute* [*of x::int y z, standard*]  
**lemmas** *zadd-ac = zadd-assoc zadd-commute zadd-left-commute*  
**lemmas** *zmult-ac = mult-ac*  
**lemmas** *zadd-0 = add-0-left* [*of z::int, standard*]  
**lemmas** *zadd-0-right = add-0-right* [*of z::int, standard*]  
**lemmas** *zadd-zminus-inverse2 = left-minus* [*of z::int, standard*]  
**lemmas** *zmult-zminus = mult-minus-left* [*of z::int w, standard*]  
**lemmas** *zmult-commute = mult-commute* [*of z::int w, standard*]  
**lemmas** *zmult-assoc = mult-assoc* [*of z1::int z2 z3, standard*]  
**lemmas** *zadd-zmult-distrib = left-distrib* [*of z1::int z2 w, standard*]  
**lemmas** *zadd-zmult-distrib2 = right-distrib* [*of w::int z1 z2, standard*]  
**lemmas** *zdiff-zmult-distrib = left-diff-distrib* [*of z1::int z2 w, standard*]  
**lemmas** *zdiff-zmult-distrib2 = right-diff-distrib* [*of w::int z1 z2, standard*]  
  
**lemmas** *zmult-1 = mult-1-left* [*of z::int, standard*]  
**lemmas** *zmult-1-right = mult-1-right* [*of z::int, standard*]  
  
**lemmas** *zle-refl = order-refl* [*of w::int, standard*]  
**lemmas** *zle-trans = order-trans* [**where** *'a=int and x=i and y=j and z=k,*  
*standard*]  
**lemmas** *zle-antisym = order-antisym* [*of z::int w, standard*]  
**lemmas** *zle-linear = linorder-linear* [*of z::int w, standard*]  
**lemmas** *zless-linear = linorder-less-linear* [**where** *'a = int*]  
  
**lemmas** *zadd-left-mono = add-left-mono* [*of i::int j k, standard*]  
**lemmas** *zadd-strict-right-mono = add-strict-right-mono* [*of i::int j k, standard*]  
**lemmas** *zadd-zless-mono = add-less-le-mono* [*of w::int w z' z, standard*]  
  
**lemmas** *int-0-less-1 = zero-less-one* [**where** *'a=int*]  
**lemmas** *int-0-neq-1 = zero-neq-one* [**where** *'a=int*]  
  
**lemmas** *inj-int = inj-of-nat* [**where** *'a=int*]  
**lemmas** *zadd-int = of-nat-add* [**where** *'a=int, symmetric*]  
**lemmas** *int-mult = of-nat-mult* [**where** *'a=int*]  
**lemmas** *zmult-int = of-nat-mult* [**where** *'a=int, symmetric*]  
**lemmas** *int-eq-0-conv = of-nat-eq-0-iff* [**where** *'a=int and m=n, standard*]  
**lemmas** *zless-int = of-nat-less-iff* [**where** *'a=int*]  
**lemmas** *int-less-0-conv = of-nat-less-0-iff* [**where** *'a=int and m=k, standard*]  
**lemmas** *zero-less-int-conv = of-nat-0-less-iff* [**where** *'a=int*]  
**lemmas** *zero-zle-int = of-nat-0-le-iff* [**where** *'a=int*]  
**lemmas** *int-le-0-conv = of-nat-le-0-iff* [**where** *'a=int and m=n, standard*]  
**lemmas** *int-0 = of-nat-0* [**where** *'a=int*]  
**lemmas** *int-1 = of-nat-1* [**where** *'a=int*]

```

lemmas int-Suc = of-nat-Suc [where 'a=int]
lemmas abs-int-eq = abs-of-nat [where 'a=int and n=m, standard]
lemmas of-int-int-eq = of-int-of-nat-eq [where 'a=int]
lemmas zdifff-int = of-nat-diff [where 'a=int, symmetric]
lemmas zless-le = less-int-def
lemmas int-eq-of-nat = TrueI

```

```

lemma zpower-zadd-distrib:
   $x \wedge (y + z) = ((x \wedge y) * (x \wedge z))::int$ 
by (rule power-add)

```

```

lemma zero-less-zpower-abs-iff:
   $(0 < \text{abs } x \wedge n) \longleftrightarrow (x \neq (0::int) \mid n = 0)$ 
by (rule zero-less-power-abs-iff)

```

```

lemma zero-le-zpower-abs:  $(0::int) \leq \text{abs } x \wedge n$ 
by (rule zero-le-power-abs)

```

```

lemma zpower-zpower:
   $(x \wedge y) \wedge z = (x \wedge (y * z))::int$ 
by (rule power-mult [symmetric])

```

```

lemma int-power:
   $\text{int } (m \wedge n) = \text{int } m \wedge n$ 
by (rule of-nat-power)

```

```

lemmas zpower-int = int-power [symmetric]

```

```

end

```

## 31 Nat-Numeral: Binary numerals for the natural numbers

```

theory Nat-Numeral
imports Int
begin

```

### 31.1 Numerals for natural numbers

Arithmetic for naturals is reduced to that for the non-negative integers.

```

instantiation nat :: number
begin

```

```

definition
  nat-number-of-def [code-unfold, code del]: number-of v = nat (number-of v)

```

```

instance ..

```

**end**

**lemma** *[code-post]*:  
 $\text{nat } (\text{number-of } v) = \text{number-of } v$   
**unfolding** *nat-number-of-def* ..

### 31.2 Special case: squares and cubes

**lemma** *numeral-2-eq-2*:  $2 = \text{Suc } (\text{Suc } 0)$   
**by** (*simp add: nat-number-of-def*)

**lemma** *numeral-3-eq-3*:  $3 = \text{Suc } (\text{Suc } (\text{Suc } 0))$   
**by** (*simp add: nat-number-of-def*)

**context** *power*  
**begin**

**abbreviation** (*xsymbols*)  
 $\text{power2} :: 'a \Rightarrow 'a \ ((-^2) [1000] 999)$  **where**  
 $x^2 \equiv x \wedge 2$

**notation** (*latex output*)  
 $\text{power2} \ ((-^2) [1000] 999)$

**notation** (*HTML output*)  
 $\text{power2} \ ((-^2) [1000] 999)$

**end**

**context** *monoid-mult*  
**begin**

**lemma** *power2-eq-square*:  $a^2 = a * a$   
**by** (*simp add: numeral-2-eq-2*)

**lemma** *power3-eq-cube*:  $a^3 = a * a * a$   
**by** (*simp add: numeral-3-eq-3 mult-assoc*)

**lemma** *power-even-eq*:  
 $a \wedge (2*n) = (a \wedge n) \wedge 2$   
**by** (*subst mult-commute*) (*simp add: power-mult*)

**lemma** *power-odd-eq*:  
 $a \wedge \text{Suc } (2*n) = a * (a \wedge n) \wedge 2$   
**by** (*simp add: power-even-eq*)

**end**

**context** *semiring-1*  
**begin**

**lemma** *zero-power2* [*simp*]:  $0^2 = 0$   
**by** (*simp add: power2-eq-square*)

**lemma** *one-power2* [*simp*]:  $1^2 = 1$   
**by** (*simp add: power2-eq-square*)

**end**

**context** *ring-1*  
**begin**

**lemma** *power2-minus* [*simp*]:  
 $(- a)^2 = a^2$   
**by** (*simp add: power2-eq-square*)

We cannot prove general results about the numeral  $-1::'b$ , so we have to use  $-(1::'a)$  instead.

**lemma** *power-minus1-even* [*simp*]:  
 $(- 1) ^ (2*n) = 1$   
**proof** (*induct n*)  
**case 0 show ?case by simp**  
**next**  
**case (Suc n) then show ?case by (simp add: power-add)**  
**qed**

**lemma** *power-minus1-odd*:  
 $(- 1) ^ Suc (2*n) = - 1$   
**by simp**

**lemma** *power-minus-even* [*simp*]:  
 $(-a) ^ (2*n) = a ^ (2*n)$   
**by** (*simp add: power-minus [of a]*)

**end**

**context** *ring-1-no-zero-divisors*  
**begin**

**lemma** *zero-eq-power2* [*simp*]:  
 $a^2 = 0 \longleftrightarrow a = 0$   
**unfolding** *power2-eq-square* **by simp**

**lemma** *power2-eq-1-iff*:  
 $a^2 = 1 \longleftrightarrow a = 1 \vee a = - 1$   
**unfolding** *power2-eq-square* **by** (*rule square-eq-1-iff*)

**end**

**context** *linordered-ring*  
**begin**

**lemma** *sum-squares-ge-zero*:  
 $0 \leq x * x + y * y$   
**by** (*intro add-nonneg-nonneg zero-le-square*)

**lemma** *not-sum-squares-lt-zero*:  
 $\neg x * x + y * y < 0$   
**by** (*simp add: not-less sum-squares-ge-zero*)

**end**

**context** *linordered-ring-strict*  
**begin**

**lemma** *sum-squares-eq-zero-iff*:  
 $x * x + y * y = 0 \longleftrightarrow x = 0 \wedge y = 0$   
**by** (*simp add: add-nonneg-eq-0-iff*)

**lemma** *sum-squares-le-zero-iff*:  
 $x * x + y * y \leq 0 \longleftrightarrow x = 0 \wedge y = 0$   
**by** (*simp add: le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff*)

**lemma** *sum-squares-gt-zero-iff*:  
 $0 < x * x + y * y \longleftrightarrow x \neq 0 \vee y \neq 0$   
**by** (*simp add: not-le [symmetric] sum-squares-le-zero-iff*)

**end**

**context** *linordered-semidom*  
**begin**

**lemma** *power2-le-imp-le*:  
 $x^2 \leq y^2 \Longrightarrow 0 \leq y \Longrightarrow x \leq y$   
**unfolding** *numeral-2-eq-2* **by** (*rule power-le-imp-le-base*)

**lemma** *power2-less-imp-less*:  
 $x^2 < y^2 \Longrightarrow 0 \leq y \Longrightarrow x < y$   
**by** (*rule power-less-imp-less-base*)

**lemma** *power2-eq-imp-eq*:  
 $x^2 = y^2 \Longrightarrow 0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x = y$   
**unfolding** *numeral-2-eq-2* **by** (*erule (2) power-eq-imp-eq-base*) *simp*

**end**

**context** *linordered-idom*  
**begin**

**lemma** *zero-le-power2* [*simp*]:  
 $0 \leq a^2$   
**by** (*simp add: power2-eq-square*)

**lemma** *zero-less-power2* [*simp*]:  
 $0 < a^2 \iff a \neq 0$   
**by** (*force simp add: power2-eq-square zero-less-mult-iff linorder-neq-iff*)

**lemma** *power2-less-0* [*simp*]:  
 $\neg a^2 < 0$   
**by** (*force simp add: power2-eq-square mult-less-0-iff*)

**lemma** *abs-power2* [*simp*]:  
 $\text{abs } (a^2) = a^2$   
**by** (*simp add: power2-eq-square abs-mult abs-mult-self*)

**lemma** *power2-abs* [*simp*]:  
 $(\text{abs } a)^2 = a^2$   
**by** (*simp add: power2-eq-square abs-mult-self*)

**lemma** *odd-power-less-zero*:  
 $a < 0 \implies a \wedge \text{Suc } (2*n) < 0$   
**proof** (*induct n*)  
**case** 0  
**then show** ?*case* **by** *simp*  
**next**  
**case** (*Suc n*)  
**have**  $a \wedge \text{Suc } (2 * \text{Suc } n) = (a*a) * a \wedge \text{Suc } (2*n)$   
**by** (*simp add: mult-ac power-add power2-eq-square*)  
**thus** ?*case*  
**by** (*simp del: power-Suc add: Suc mult-less-0-iff mult-neg-neg*)  
**qed**

**lemma** *odd-0-le-power-imp-0-le*:  
 $0 \leq a \wedge \text{Suc } (2*n) \implies 0 \leq a$   
**using** *odd-power-less-zero* [*of a n*]  
**by** (*force simp add: linorder-not-less [symmetric]*)

**lemma** *zero-le-even-power'* [*simp*]:  
 $0 \leq a \wedge (2*n)$   
**proof** (*induct n*)  
**case** 0  
**show** ?*case* **by** *simp*  
**next**  
**case** (*Suc n*)  
**have**  $a \wedge (2 * \text{Suc } n) = (a*a) * a \wedge (2*n)$

by (simp add: mult-ac power-add power2-eq-square)  
 thus ?case  
 by (simp add: Suc zero-le-mult-iff)  
 qed

**lemma** sum-power2-ge-zero:  
 $0 \leq x^2 + y^2$   
 unfolding power2-eq-square by (rule sum-squares-ge-zero)

**lemma** not-sum-power2-lt-zero:  
 $\neg x^2 + y^2 < 0$   
 unfolding power2-eq-square by (rule not-sum-squares-lt-zero)

**lemma** sum-power2-eq-zero-iff:  
 $x^2 + y^2 = 0 \iff x = 0 \wedge y = 0$   
 unfolding power2-eq-square by (rule sum-squares-eq-zero-iff)

**lemma** sum-power2-le-zero-iff:  
 $x^2 + y^2 \leq 0 \iff x = 0 \wedge y = 0$   
 unfolding power2-eq-square by (rule sum-squares-le-zero-iff)

**lemma** sum-power2-gt-zero-iff:  
 $0 < x^2 + y^2 \iff x \neq 0 \vee y \neq 0$   
 unfolding power2-eq-square by (rule sum-squares-gt-zero-iff)

end

**lemma** power2-sum:  
 fixes  $x y :: 'a::\text{number-ring}$   
 shows  $(x + y)^2 = x^2 + y^2 + 2 * x * y$   
 by (simp add: ring-distrib power2-eq-square mult-2) (rule mult-commute)

**lemma** power2-diff:  
 fixes  $x y :: 'a::\text{number-ring}$   
 shows  $(x - y)^2 = x^2 + y^2 - 2 * x * y$   
 by (simp add: ring-distrib power2-eq-square mult-2) (rule mult-commute)

### 31.3 Predicate for negative binary numbers

**definition** neg ::  $\text{int} \Rightarrow \text{bool}$  where  
 $\text{neg } Z \iff Z < 0$

**lemma** not-neg-int [simp]:  $\sim \text{neg } (\text{of-nat } n)$   
 by (simp add: neg-def)

**lemma** neg-zminus-int [simp]:  $\text{neg } (- (\text{of-nat } (\text{Suc } n)))$   
 by (simp add: neg-def del: of-nat-Suc)

**lemmas** neg-eq-less-0 = neg-def

**lemma** *not-neg-eq-ge-0*:  $(\sim \text{neg } x) = (0 \leq x)$   
**by** (*simp add: neg-def linorder-not-less*)

To simplify inequalities when *Numeral1* can get simplified to 1

**lemma** *not-neg-0*:  $\sim \text{neg } 0$   
**by** (*simp add: One-int-def neg-def*)

**lemma** *not-neg-1*:  $\sim \text{neg } 1$   
**by** (*simp add: neg-def linorder-not-less*)

**lemma** *neg-nat*:  $\text{neg } z ==> \text{nat } z = 0$   
**by** (*simp add: neg-def order-less-imp-le*)

**lemma** *not-neg-nat*:  $\sim \text{neg } z ==> \text{of-nat } (\text{nat } z) = z$   
**by** (*simp add: linorder-not-less neg-def*)

If *Numeral0* is rewritten to 0 then this rule can’t be applied: *Numeral0* IS  
*number-of Pls*

**lemma** *not-neg-number-of-Pls*:  $\sim \text{neg } (\text{number-of Int.Pl})$   
**by** (*simp add: neg-def*)

**lemma** *neg-number-of-Min*:  $\text{neg } (\text{number-of Int.Min})$   
**by** (*simp add: neg-def*)

**lemma** *neg-number-of-Bit0*:  
 $\text{neg } (\text{number-of } (\text{Int.Bit0 } w)) = \text{neg } (\text{number-of } w)$   
**by** (*simp add: neg-def*)

**lemma** *neg-number-of-Bit1*:  
 $\text{neg } (\text{number-of } (\text{Int.Bit1 } w)) = \text{neg } (\text{number-of } w)$   
**by** (*simp add: neg-def*)

**lemmas** *neg-simps* [*simp*] =  
*not-neg-0 not-neg-1*  
*not-neg-number-of-Pls neg-number-of-Min*  
*neg-number-of-Bit0 neg-number-of-Bit1*

### 31.4 Function *nat*: Coercion from Type *int* to *nat*

**declare** *nat-1* [*simp*]

**lemma** *nat-number-of* [*simp*]:  $\text{nat } (\text{number-of } w) = \text{number-of } w$   
**by** (*simp add: nat-number-of-def*)

**lemma** *nat-numeral-0-eq-0* [*simp*, *code-post*]: *Numeral0* =  $(0::\text{nat})$   
**by** (*simp add: nat-number-of-def*)

**lemma** *nat-numeral-1-eq-1* [*simp*]: *Numeral1* =  $(1::\text{nat})$



**by** (*simp add: nat-number-of-def*)

**lemma** *Numeral1-eq1-nat*:

(*1::nat*) = *Numeral1*

**by** *simp*

**lemma** *numeral-1-eq-Suc-0* [*code-post*]: *Numeral1* = *Suc 0*

**by** (*simp only: nat-numeral-1-eq-1 One-nat-def*)

### 31.5 Function *int*: Coercion from Type *nat* to *int*

**lemma** *int-nat-number-of* [*simp*]:

*int* (*number-of v*) =  
 (if *neg* (*number-of v :: int*) then 0  
 else (*number-of v :: int*))

**unfolding** *nat-number-of-def number-of-is-id neg-def*

**by** *simp*

#### 31.5.1 Successor

**lemma** *Suc-nat-eq-nat-zadd1*: (*0::int*) <= *z* ==> *Suc* (*nat z*) = *nat* (*1 + z*)

**apply** (*rule sym*)

**apply** (*simp add: nat-eq-iff int-Suc*)

**done**

**lemma** *Suc-nat-number-of-add*:

*Suc* (*number-of v + n*) =  
 (if *neg* (*number-of v :: int*) then *1+n* else *number-of* (*Int.succ v*) + *n*)

**unfolding** *nat-number-of-def number-of-is-id neg-def numeral-simps*

**by** (*simp add: Suc-nat-eq-nat-zadd1 add-ac*)

**lemma** *Suc-nat-number-of* [*simp*]:

*Suc* (*number-of v*) =  
 (if *neg* (*number-of v :: int*) then 1 else *number-of* (*Int.succ v*))

**apply** (*cut-tac n = 0 in Suc-nat-number-of-add*)

**apply** (*simp cong del: if-weak-cong*)

**done**

#### 31.5.2 Addition

**lemma** *add-nat-number-of* [*simp*]:

(*number-of v :: nat*) + *number-of v'* =  
 (if *v < Int.Pls* then *number-of v'*  
 else if *v' < Int.Pls* then *number-of v*  
 else *number-of* (*v + v'*))

**unfolding** *nat-number-of-def number-of-is-id numeral-simps*

**by** (*simp add: nat-add-distrib*)

**lemma** *nat-number-of-add-1* [*simp*]:

*number-of v + (1::nat)* =

(if  $v < \text{Int.Pls}$  then 1 else number-of ( $\text{Int.succ } v$ ))  
**unfolding** nat-number-of-def number-of-is-id numeral-simps  
**by** (simp add: nat-add-distrib)

**lemma** nat-1-add-number-of [simp]:  
 $(1::\text{nat}) + \text{number-of } v =$   
 (if  $v < \text{Int.Pls}$  then 1 else number-of ( $\text{Int.succ } v$ ))  
**unfolding** nat-number-of-def number-of-is-id numeral-simps  
**by** (simp add: nat-add-distrib)

**lemma** nat-1-add-1 [simp]:  $1 + 1 = (2::\text{nat})$   
**by** (rule int-int-eq [THEN iffD1]) simp

### 31.5.3 Subtraction

**lemma** diff-nat-eq-if:  
 $\text{nat } z - \text{nat } z' =$   
 (if  $\text{neg } z'$  then  $\text{nat } z$   
 else let  $d = z - z'$  in  
   if  $\text{neg } d$  then 0 else  $\text{nat } d$ )  
**by** (simp add: Let-def nat-diff-distrib [symmetric] neg-eq-less-0 not-neg-eq-ge-0)

**lemma** diff-nat-number-of [simp]:  
 $(\text{number-of } v :: \text{nat}) - \text{number-of } v' =$   
 (if  $v' < \text{Int.Pls}$  then  $\text{number-of } v$   
 else let  $d = \text{number-of } (v + \text{uminus } v')$  in  
   if  $\text{neg } d$  then 0 else  $\text{nat } d$ )  
**unfolding** nat-number-of-def number-of-is-id numeral-simps neg-def  
**by** auto

**lemma** nat-number-of-diff-1 [simp]:  
 $\text{number-of } v - (1::\text{nat}) =$   
 (if  $v \leq \text{Int.Pls}$  then 0 else  $\text{number-of } (\text{Int.pred } v)$ )  
**unfolding** nat-number-of-def number-of-is-id numeral-simps  
**by** auto

### 31.5.4 Multiplication

**lemma** mult-nat-number-of [simp]:  
 $(\text{number-of } v :: \text{nat}) * \text{number-of } v' =$   
 (if  $v < \text{Int.Pls}$  then 0 else  $\text{number-of } (v * v')$ )  
**unfolding** nat-number-of-def number-of-is-id numeral-simps  
**by** (simp add: nat-mult-distrib)

## 31.6 Comparisons

### 31.6.1 Equals (=)

**lemma** eq-nat-number-of [simp]:

```

((number-of v :: nat) = number-of v') =
  (if neg (number-of v :: int) then (number-of v' :: int) ≤ 0
   else if neg (number-of v' :: int) then (number-of v :: int) = 0
   else v = v')
unfolding nat-number-of-def number-of-is-id neg-def
by auto

```

### 31.6.2 Less-than (i)

```

lemma less-nat-number-of [simp]:
  (number-of v :: nat) < number-of v' ⟷
    (if v < v' then Int.Pls < v' else False)
unfolding nat-number-of-def number-of-is-id numeral-simps
by auto

```

### 31.6.3 Less-than-or-equal

```

lemma le-nat-number-of [simp]:
  (number-of v :: nat) ≤ number-of v' ⟷
    (if v ≤ v' then True else v ≤ Int.Pls)
unfolding nat-number-of-def number-of-is-id numeral-simps
by auto

```

**lemmas** numerals = nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2

## 31.7 Powers with Numeric Exponents

Squares of literal numerals will be evaluated.

```

lemmas power2-eq-square-number-of [simp] =
  power2-eq-square [of number-of w, standard]

```

Simprules for comparisons where common factors can be cancelled.

```

lemmas zero-compare-simps =
  add-strict-increasing add-strict-increasing2 add-increasing
  zero-le-mult-iff zero-le-divide-iff
  zero-less-mult-iff zero-less-divide-iff
  mult-le-0-iff divide-le-0-iff
  mult-less-0-iff divide-less-0-iff
  zero-le-power2 power2-less-0

```

### 31.7.1 Nat

```

lemma Suc-pred': 0 < n ==> n = Suc(n - 1)
by simp

```

**lemmas** expand-Suc = Suc-pred' [of number-of v, standard]

### 31.7.2 Arith

**lemma** *Suc-eq-plus1*:  $Suc\ n = n + 1$   
**unfolding** *One-nat-def* **by** *simp*

**lemma** *Suc-eq-plus1-left*:  $Suc\ n = 1 + n$   
**unfolding** *One-nat-def* **by** *simp*

**lemma** *add-eq-if*:  $(m::nat) + n = (if\ m=0\ then\ n\ else\ Suc\ ((m - 1) + n))$   
**unfolding** *One-nat-def* **by** *(cases m) simp-all*

**lemma** *mult-eq-if*:  $(m::nat) * n = (if\ m=0\ then\ 0\ else\ n + ((m - 1) * n))$   
**unfolding** *One-nat-def* **by** *(cases m) simp-all*

**lemma** *power-eq-if*:  $(p \wedge m :: nat) = (if\ m=0\ then\ 1\ else\ p * (p \wedge (m - 1)))$   
**unfolding** *One-nat-def* **by** *(cases m) simp-all*

### 31.8 Comparisons involving (0::nat)

Simplification already does  $n < (0::'a)$ ,  $n \leq (0::'a)$  and  $(0::'a) \leq n$ .

**lemma** *eq-number-of-0* [*simp*]:  
 $number-of\ v = (0::nat) \longleftrightarrow v \leq Int.Pls$   
**unfolding** *nat-number-of-def number-of-is-id numeral-simps*  
**by** *auto*

**lemma** *eq-0-number-of* [*simp*]:  
 $(0::nat) = number-of\ v \longleftrightarrow v \leq Int.Pls$   
**by** *(rule trans [OF eq-sym-conv eq-number-of-0])*

**lemma** *less-0-number-of* [*simp*]:  
 $(0::nat) < number-of\ v \longleftrightarrow Int.Pls < v$   
**unfolding** *nat-number-of-def number-of-is-id numeral-simps*  
**by** *simp*

**lemma** *neg-imp-number-of-eq-0*:  $neg\ (number-of\ v :: int) ==> number-of\ v = (0::nat)$   
**by** *(simp del: nat-numeral-0-eq-0 add: nat-numeral-0-eq-0 [symmetric])*

### 31.9 Comparisons involving Suc

**lemma** *eq-number-of-Suc* [*simp*]:  
 $(number-of\ v = Suc\ n) =$   
 $(let\ pv = number-of\ (Int.pred\ v)\ in$   
 $if\ neg\ pv\ then\ False\ else\ nat\ pv = n)$   
**apply** *(simp only: simp-thms Let-def neg-eq-less-0 linorder-not-less*  
 $number-of-pred\ nat-number-of-def$   
 $split\ add: split-if)$

```

apply (rule-tac  $x = \text{number-of } v$  in spec)
apply (auto simp add: nat-eq-iff)
done

```

```

lemma Suc-eq-number-of [simp]:
  (Suc  $n = \text{number-of } v$ ) =
    (let  $pv = \text{number-of } (\text{Int.pred } v)$  in
      if neg  $pv$  then False else nat  $pv = n$ )
by (rule trans [OF eq-sym-conv eq-number-of-Suc])

```

```

lemma less-number-of-Suc [simp]:
  (number-of  $v < \text{Suc } n$ ) =
    (let  $pv = \text{number-of } (\text{Int.pred } v)$  in
      if neg  $pv$  then True else nat  $pv < n$ )
apply (simp only: simp-thms Let-def neg-eq-less-0 linorder-not-less
  number-of-pred nat-number-of-def
  split add: split-if)
apply (rule-tac  $x = \text{number-of } v$  in spec)
apply (auto simp add: nat-less-iff)
done

```

```

lemma less-Suc-number-of [simp]:
  (Suc  $n < \text{number-of } v$ ) =
    (let  $pv = \text{number-of } (\text{Int.pred } v)$  in
      if neg  $pv$  then False else  $n < \text{nat } pv$ )
apply (simp only: simp-thms Let-def neg-eq-less-0 linorder-not-less
  number-of-pred nat-number-of-def
  split add: split-if)
apply (rule-tac  $x = \text{number-of } v$  in spec)
apply (auto simp add: zless-nat-eq-int-zless)
done

```

```

lemma le-number-of-Suc [simp]:
  (number-of  $v \leq \text{Suc } n$ ) =
    (let  $pv = \text{number-of } (\text{Int.pred } v)$  in
      if neg  $pv$  then True else nat  $pv \leq n$ )
by (simp add: Let-def linorder-not-less [symmetric])

```

```

lemma le-Suc-number-of [simp]:
  (Suc  $n \leq \text{number-of } v$ ) =
    (let  $pv = \text{number-of } (\text{Int.pred } v)$  in
      if neg  $pv$  then False else  $n \leq \text{nat } pv$ )
by (simp add: Let-def linorder-not-less [symmetric])

```

```

lemma eq-number-of-Pls-Min: (Numeral0 :: int)  $\sim = \text{number-of Int.Min}$ 
by auto

```

### 31.10 Max and Min Combined with *Suc*

```

lemma max-number-of-Suc [simp]:
  max (Suc n) (number-of v) =
    (let pv = number-of (Int.pred v) in
      if neg pv then Suc n else Suc(max n (nat pv)))
apply (simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def
  split add: split-if nat.split)
apply (rule-tac x = number-of v in spec)
apply auto
done

```

```

lemma max-Suc-number-of [simp]:
  max (number-of v) (Suc n) =
    (let pv = number-of (Int.pred v) in
      if neg pv then Suc n else Suc(max (nat pv) n))
apply (simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def
  split add: split-if nat.split)
apply (rule-tac x = number-of v in spec)
apply auto
done

```

```

lemma min-number-of-Suc [simp]:
  min (Suc n) (number-of v) =
    (let pv = number-of (Int.pred v) in
      if neg pv then 0 else Suc(min n (nat pv)))
apply (simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def
  split add: split-if nat.split)
apply (rule-tac x = number-of v in spec)
apply auto
done

```

```

lemma min-Suc-number-of [simp]:
  min (number-of v) (Suc n) =
    (let pv = number-of (Int.pred v) in
      if neg pv then 0 else Suc(min (nat pv) n))
apply (simp only: Let-def neg-eq-less-0 number-of-pred nat-number-of-def
  split add: split-if nat.split)
apply (rule-tac x = number-of v in spec)
apply auto
done

```

### 31.11 Literal arithmetic involving powers

```

lemma power-nat-number-of:
  (number-of v :: nat) ^ n =
    (if neg (number-of v :: int) then 0 ^ n else nat ((number-of v :: int) ^ n))
by (simp only: simp-thms neg-nat not-neg-eq-ge-0 nat-number-of-def nat-power-eq
  split add: split-if cong: imp-cong)

```

```

lemmas power-nat-number-of-number-of = power-nat-number-of [of - number-of
w, standard]
declare power-nat-number-of-number-of [simp]

```

For arbitrary rings

```

lemma power-number-of-even:
  fixes z :: 'a::number-ring
  shows z ^ number-of (Int.Bit0 w) = (let w = z ^ (number-of w) in w * w)
by (cases w ≥ 0) (auto simp add: Let-def Bit0-def nat-number-of-def number-of-is-id
nat-add-distrib power-add simp del: nat-number-of)

```

```

lemma power-number-of-odd:
  fixes z :: 'a::number-ring
  shows z ^ number-of (Int.Bit1 w) = (if (0::int) ≤ number-of w
then (let w = z ^ (number-of w) in z * w * w) else 1)
unfolding Let-def Bit1-def nat-number-of-def number-of-is-id
apply (cases 0 ≤ w)
apply (simp only: mult-assoc nat-add-distrib power-add, simp)
apply (simp add: not-le mult-2 [symmetric] add-assoc)
done

```

```

lemmas zpower-number-of-even = power-number-of-even [where 'a=int]
lemmas zpower-number-of-odd = power-number-of-odd [where 'a=int]

```

```

lemmas power-number-of-even-number-of [simp] =
power-number-of-even [of number-of v, standard]

```

```

lemmas power-number-of-odd-number-of [simp] =
power-number-of-odd [of number-of v, standard]

```

```

lemma nat-number-of-Pls: Numeral0 = (0::nat)
by (simp add: nat-number-of-def)

```

```

lemma nat-number-of-Min: number-of Int.Min = (0::nat)
apply (simp only: number-of-Min nat-number-of-def nat-zminus-int)
done

```

```

lemma nat-number-of-Bit0:
  number-of (Int.Bit0 w) = (let n::nat = number-of w in n + n)
by (cases w ≥ 0) (auto simp add: Let-def Bit0-def nat-number-of-def number-of-is-id
nat-add-distrib simp del: nat-number-of)

```

```

lemma nat-number-of-Bit1:
  number-of (Int.Bit1 w) =
    (if neg (number-of w :: int) then 0
else let n = number-of w in Suc (n + n))
unfolding Let-def Bit1-def nat-number-of-def number-of-is-id neg-def
apply (cases w < 0)

```

```

apply (simp add: mult-2 [symmetric] add-assoc)
apply (simp only: nat-add-distrib, simp)
done

```

```

lemmas nat-number =
  nat-number-of-Pls nat-number-of-Min
  nat-number-of-Bit0 nat-number-of-Bit1

```

```

lemmas nat-number' =
  nat-number-of-Bit0 nat-number-of-Bit1

```

```

lemmas nat-arith =
  add-nat-number-of
  diff-nat-number-of
  mult-nat-number-of
  eq-nat-number-of
  less-nat-number-of

```

```

lemmas semiring-norm =
  Let-def arith-simps nat-arith rel-simps neg-simps if-False
  if-True add-0 add-Suc add-number-of-left mult-number-of-left
  numeral-1-eq-1 [symmetric] Suc-eq-plus1
  numeral-0-eq-0 [symmetric] numerals [symmetric]
  not-iszero-Numeral1

```

```

lemma Let-Suc [simp]: Let (Suc n) f == f (Suc n)
by (fact Let-def)

```

```

lemma power-m1-even:  $(-1) ^ (2*n) = (1::'a::\{number-ring\})$ 
by (simp only: number-of-Min power-minus1-even)

```

```

lemma power-m1-odd:  $(-1) ^ Suc(2*n) = (-1::'a::\{number-ring\})$ 
by (simp only: number-of-Min power-minus1-odd)

```

```

lemma nat-number-of-add-left:
  number-of v + (number-of v' + (k::nat)) =
    (if neg (number-of v :: int) then number-of v' + k
     else if neg (number-of v' :: int) then number-of v + k
     else number-of (v + v') + k)
by (auto simp add: neg-def)

```

```

lemma nat-number-of-mult-left:
  number-of v * (number-of v' * (k::nat)) =
    (if v < Int.Pls then 0
     else number-of (v * v') * k)
by (auto simp add: not-less Pls-def nat-number-of-def number-of-is-id
  nat-mult-distrib simp del: nat-number-of)

```



**31.12 Literal arithmetic and *of-nat*****lemma** *of-nat-double*:

$$0 \leq x \implies \text{of-nat } (\text{nat } (2 * x)) = \text{of-nat } (\text{nat } x) + \text{of-nat } (\text{nat } x)$$

**by** (*simp only: mult-2 nat-add-distrib of-nat-add*)**lemma** *nat-numeral-m1-eq-0*:  $-1 = (0::\text{nat})$ **by** (*simp only: nat-number-of-def*)**lemma** *of-nat-number-of-lemma*:

$$\begin{aligned} \text{of-nat } (\text{number-of } v :: \text{nat}) = \\ \text{(if } 0 \leq (\text{number-of } v :: \text{int}) \\ \text{then } (\text{number-of } v :: 'a :: \text{number-ring}) \\ \text{else } 0) \end{aligned}$$

**by** (*simp add: int-number-of-def nat-number-of-def number-of-eq of-nat-nat*)**lemma** *of-nat-number-of-eq [simp]*:

$$\begin{aligned} \text{of-nat } (\text{number-of } v :: \text{nat}) = \\ \text{(if neg } (\text{number-of } v :: \text{int}) \text{ then } 0 \\ \text{else } (\text{number-of } v :: 'a :: \text{number-ring})) \end{aligned}$$

**by** (*simp only: of-nat-number-of-lemma neg-def, simp*)**31.12.1 For simplifying  $\text{Suc } m - K$  and  $K - \text{Suc } m$** 

Where K above is a literal

**lemma** *Suc-diff-eq-diff-pred*:  $\text{Numeral0} < n \implies \text{Suc } m - n = m - (n - \text{Numeral1})$ **by** (*simp split: nat-diff-split*)Now just instantiating  $n$  to *number-of v* does the right simplification, but with some redundant inequality tests.**lemma** *neg-number-of-pred-iff-0*:

$$\text{neg } (\text{number-of } (\text{Int.pred } v)::\text{int}) = (\text{number-of } v = (0::\text{nat}))$$

**apply** (*subgoal-tac neg (number-of (Int.pred v)) = (number-of v < Suc 0)*)**apply** (*simp only: less-Suc-eq-le le-0-eq*)**apply** (*subst less-number-of-Suc, simp*)**done**No longer required as a simprule because of the *inverse-fold* simproc**lemma** *Suc-diff-number-of*:

$$\text{Int.Pls} < v \implies$$

$$\text{Suc } m - (\text{number-of } v) = m - (\text{number-of } (\text{Int.pred } v))$$

**apply** (*subst Suc-diff-eq-diff-pred*)**apply** *simp***apply** (*simp del: nat-numeral-1-eq-1*)**apply** (*auto simp only: diff-nat-number-of less-0-number-of [symmetric] neg-number-of-pred-iff-0*)**done**

**lemma** *diff-Suc-eq-diff-pred*:  $m - \text{Suc } n = (m - 1) - n$   
**by** (*simp split: nat-diff-split*)

### 31.12.2 For *nat-case* and *nat-rec*

**lemma** *nat-case-number-of* [*simp*]:  
 $\text{nat-case } a \ f \ (\text{number-of } v) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if } \text{neg } pv \text{ then } a \text{ else } f \ (\text{nat } pv))$   
**by** (*simp split add: nat.split add: Let-def neg-number-of-pred-iff-0*)

**lemma** *nat-case-add-eq-if* [*simp*]:  
 $\text{nat-case } a \ f \ ((\text{number-of } v) + n) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if } \text{neg } pv \text{ then } \text{nat-case } a \ f \ n \text{ else } f \ (\text{nat } pv + n))$   
**apply** (*subst add-eq-if*)  
**apply** (*simp split add: nat.split*  
 $\text{del: nat-numeral-1-eq-1}$   
 $\text{add: nat-numeral-1-eq-1 [symmetric]}$   
 $\text{numeral-1-eq-Suc-0 [symmetric]}$   
 $\text{neg-number-of-pred-iff-0}$ )  
**done**

**lemma** *nat-rec-number-of* [*simp*]:  
 $\text{nat-rec } a \ f \ (\text{number-of } v) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if } \text{neg } pv \text{ then } a \text{ else } f \ (\text{nat } pv) \ (\text{nat-rec } a \ f \ (\text{nat } pv)))$   
**apply** (*case-tac (number-of v)::nat*)  
**apply** (*simp-all (no-asm-simp) add: Let-def neg-number-of-pred-iff-0*)  
**apply** (*simp split add: split-if-asm*)  
**done**

**lemma** *nat-rec-add-eq-if* [*simp*]:  
 $\text{nat-rec } a \ f \ (\text{number-of } v + n) =$   
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$   
 $\text{if } \text{neg } pv \text{ then } \text{nat-rec } a \ f \ n$   
 $\text{else } f \ (\text{nat } pv + n) \ (\text{nat-rec } a \ f \ (\text{nat } pv + n)))$   
**apply** (*subst add-eq-if*)  
**apply** (*simp split add: nat.split*  
 $\text{del: nat-numeral-1-eq-1}$   
 $\text{add: nat-numeral-1-eq-1 [symmetric]}$   
 $\text{numeral-1-eq-Suc-0 [symmetric]}$   
 $\text{neg-number-of-pred-iff-0}$ )  
**done**

### 31.12.3 Various Other Lemmas

**lemma** *card-UNIV-bool*[*simp*]:  $\text{card } (\text{UNIV} :: \text{bool set}) = 2$   
**by**(*simp add: UNIV-bool*)

Evens and Odds, for Mutilated Chess Board

Lemmas for specialist use, NOT as default simprules

**lemma** *nat-mult-2*:  $2 * z = (z + z :: \text{nat})$

**unfolding** *nat-1-add-1* [*symmetric*] *left-distrib* **by** *simp*

**lemma** *nat-mult-2-right*:  $z * 2 = (z + z :: \text{nat})$

**by** (*subst mult-commute*, *rule nat-mult-2*)

Case analysis on  $n < (2 :: 'a)$

**lemma** *less-2-cases*:  $(n :: \text{nat}) < 2 ==> n = 0 \mid n = \text{Suc } 0$

**by** (*auto simp add: nat-1-add-1* [*symmetric*])

Removal of Small Numerals: 0, 1 and (in additive positions) 2

**lemma** *add-2-eq-Suc* [*simp*]:  $2 + n = \text{Suc } (\text{Suc } n)$

**by** *simp*

**lemma** *add-2-eq-Suc'* [*simp*]:  $n + 2 = \text{Suc } (\text{Suc } n)$

**by** *simp*

Can be used to eliminate long strings of Sucs, but not by default

**lemma** *Suc3-eq-add-3*:  $\text{Suc } (\text{Suc } (\text{Suc } n)) = 3 + n$

**by** *simp*

**end**

## 32 Nat-Transfer: Generic transfer machinery; specific transfer from nats to ints and back.

**theory** *Nat-Transfer*

**imports** *Nat-Numeral*

**uses** (*Tools/transfer.ML*)

**begin**

### 32.1 Generic transfer machinery

**definition** *transfer-morphism*::  $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow \text{bool}$

**where** *transfer-morphism*  $f A \longleftrightarrow (\forall P. (\forall x. P x) \longrightarrow (\forall y. A y \longrightarrow P (f y)))$

**lemma** *transfer-morphismI*:

**assumes**  $\bigwedge^P y. (\bigwedge x. P x) \Longrightarrow A y \Longrightarrow P (f y)$

**shows** *transfer-morphism*  $f A$

**using** *assms* **by** (*auto simp add: transfer-morphism-def*)

**use** *Tools/transfer.ML*

**setup** *Transfer.setup*

### 32.2 Set up transfer from nat to int

set up transfer direction

**lemma** *transfer-morphism-nat-int*: *transfer-morphism nat (op <= (0::int))*  
**by** (*rule transfer-morphismI*) *simp*

**declare** *transfer-morphism-nat-int* [*transfer add*  
*mode: manual*  
*return: nat-0-le*  
*labels: nat-int*  
 ]

basic functions and relations

**lemma** *transfer-nat-int-numerals* [*transfer key: transfer-morphism-nat-int*]:  
 (*0::nat*) = *nat 0*  
 (*1::nat*) = *nat 1*  
 (*2::nat*) = *nat 2*  
 (*3::nat*) = *nat 3*  
**by** *auto*

**definition**

*tsub :: int ⇒ int ⇒ int*

**where**

*tsub x y* = (*if x >= y then x - y else 0*)

**lemma** *tsub-eq*: *x >= y ⇒ tsub x y = x - y*  
**by** (*simp add: tsub-def*)

**lemma** *transfer-nat-int-functions* [*transfer key: transfer-morphism-nat-int*]:  
 (*x::int*) >= 0 ⇒ *y >= 0 ⇒ (nat x) + (nat y) = nat (x + y)*  
 (*x::int*) >= 0 ⇒ *y >= 0 ⇒ (nat x) \* (nat y) = nat (x \* y)*  
 (*x::int*) >= 0 ⇒ *y >= 0 ⇒ (nat x) - (nat y) = nat (tsub x y)*  
 (*x::int*) >= 0 ⇒ *(nat x) ^ n = nat (x ^ n)*  
**by** (*auto simp add: eq-nat-nat-iff nat-mult-distrib*  
*nat-power-eq tsub-def*)

**lemma** *transfer-nat-int-function-closures* [*transfer key: transfer-morphism-nat-int*]:  
 (*x::int*) >= 0 ⇒ *y >= 0 ⇒ x + y >= 0*  
 (*x::int*) >= 0 ⇒ *y >= 0 ⇒ x \* y >= 0*  
 (*x::int*) >= 0 ⇒ *y >= 0 ⇒ tsub x y >= 0*  
 (*x::int*) >= 0 ⇒ *x ^ n >= 0*  
 (*0::int*) >= 0  
 (*1::int*) >= 0  
 (*2::int*) >= 0  
 (*3::int*) >= 0  
*int z >= 0*  
**by** (*auto simp add: zero-le-mult-iff tsub-def*)

**lemma** *transfer-nat-int-relations* [*transfer key: transfer-morphism-nat-int*]:

```

 $x \geq 0 \implies y \geq 0 \implies$ 
   $(\text{nat } (x::\text{int}) = \text{nat } y) = (x = y)$ 
 $x \geq 0 \implies y \geq 0 \implies$ 
   $(\text{nat } (x::\text{int}) < \text{nat } y) = (x < y)$ 
 $x \geq 0 \implies y \geq 0 \implies$ 
   $(\text{nat } (x::\text{int}) \leq \text{nat } y) = (x \leq y)$ 
 $x \geq 0 \implies y \geq 0 \implies$ 
   $(\text{nat } (x::\text{int}) \text{ dvd } \text{nat } y) = (x \text{ dvd } y)$ 
by (auto simp add: zdvd-int)

```

first-order quantifiers

```

lemma all-nat:  $(\forall x. P x) \longleftrightarrow (\forall x \geq 0. P (\text{nat } x))$ 
by (simp split add: split-nat)

```

```

lemma ex-nat:  $(\exists x. P x) \longleftrightarrow (\exists x. 0 \leq x \wedge P (\text{nat } x))$ 

```

**proof**

```

  assume  $\exists x. P x$ 
  then obtain  $x$  where  $P x$  ..
  then have  $\text{int } x \geq 0 \wedge P (\text{nat } (\text{int } x))$  by simp
  then show  $\exists x \geq 0. P (\text{nat } x)$  ..

```

**next**

```

  assume  $\exists x \geq 0. P (\text{nat } x)$ 
  then show  $\exists x. P x$  by auto

```

**qed**

```

lemma transfer-nat-int-quantifiers [transfer key: transfer-morphism-nat-int]:

```

```

   $(\text{ALL } (x::\text{nat}). P x) = (\text{ALL } (x::\text{int}). x \geq 0 \longrightarrow P (\text{nat } x))$ 
   $(\text{EX } (x::\text{nat}). P x) = (\text{EX } (x::\text{int}). x \geq 0 \ \& \ P (\text{nat } x))$ 

```

```

by (rule all-nat, rule ex-nat)

```

```

lemma all-cong:  $(\bigwedge x. Q x \implies P x = P' x) \implies$ 
   $(\text{ALL } x. Q x \longrightarrow P x) = (\text{ALL } x. Q x \longrightarrow P' x)$ 
by auto

```

```

lemma ex-cong:  $(\bigwedge x. Q x \implies P x = P' x) \implies$ 
   $(\text{EX } x. Q x \wedge P x) = (\text{EX } x. Q x \wedge P' x)$ 
by auto

```

```

declare transfer-morphism-nat-int [transfer add
  cong: all-cong ex-cong]

```

**if**

```

lemma nat-if-cong [transfer key: transfer-morphism-nat-int]:
   $(\text{if } P \text{ then } (\text{nat } x) \text{ else } (\text{nat } y)) = \text{nat } (\text{if } P \text{ then } x \text{ else } y)$ 
by auto

```

operations with sets

**definition**

```

nat-set :: int set ⇒ bool
where
  nat-set S = (ALL x:S. x >= 0)

lemma transfer-nat-int-set-functions:
  card A = card (int ‘ A)
  {} = nat ‘ ({}::int set)
  A Un B = nat ‘ (int ‘ A Un int ‘ B)
  A Int B = nat ‘ (int ‘ A Int int ‘ B)
  {x. P x} = nat ‘ {x. x >= 0 & P(nat x)}
apply (rule card-image [symmetric])
apply (auto simp add: inj-on-def image-def)
apply (rule-tac x = int x in bexI)
apply auto
apply (rule-tac x = int x in bexI)
apply auto
apply (rule-tac x = int x in bexI)
apply auto
apply (rule-tac x = int x in exI)
apply auto
done

lemma transfer-nat-int-set-function-closures:
  nat-set {}
  nat-set A ⇒ nat-set B ⇒ nat-set (A Un B)
  nat-set A ⇒ nat-set B ⇒ nat-set (A Int B)
  nat-set {x. x >= 0 & P x}
  nat-set (int ‘ C)
  nat-set A ⇒ x : A ⇒ x >= 0
unfolding nat-set-def apply auto
done

lemma transfer-nat-int-set-relations:
  (finite A) = (finite (int ‘ A))
  (x : A) = (int x : int ‘ A)
  (A = B) = (int ‘ A = int ‘ B)
  (A < B) = (int ‘ A < int ‘ B)
  (A <= B) = (int ‘ A <= int ‘ B)
apply (rule iffI)
apply (erule finite-imageI)
apply (erule finite-imageD)
apply (auto simp add: image-def expand-set-eq inj-on-def)
apply (drule-tac x = int x in spec, auto)
apply (drule-tac x = int x in spec, auto)
apply (drule-tac x = int x in spec, auto)
done

lemma transfer-nat-int-set-return-embed: nat-set A ⇒
  (int ‘ nat ‘ A = A)

```

```

by (auto simp add: nat-set-def image-def)

lemma transfer-nat-int-set-cong: (!x. x >= 0 ==> P x = P' x) ==>
  {(x::int). x >= 0 & P x} = {x. x >= 0 & P' x}
by auto

declare transfer-morphism-nat-int [transfer add
  return: transfer-nat-int-set-functions
  transfer-nat-int-set-function-closures
  transfer-nat-int-set-relations
  transfer-nat-int-set-return-embed
  cong: transfer-nat-int-set-cong
]

setsum and setprod

lemma transfer-nat-int-sum-prod:
  setsum f A = setsum (%x. f (nat x)) (int ' A)
  setprod f A = setprod (%x. f (nat x)) (int ' A)
apply (subst setsum-reindex)
apply (unfold inj-on-def, auto)
apply (subst setprod-reindex)
apply (unfold inj-on-def o-def, auto)
done

lemma transfer-nat-int-sum-prod2:
  setsum f A = nat(setsum (%x. int (f x)) A)
  setprod f A = nat(setprod (%x. int (f x)) A)
apply (subst int-setsum [symmetric])
apply auto
apply (subst int-setprod [symmetric])
apply auto
done

lemma transfer-nat-int-sum-prod-closure:
  nat-set A ==> (!x. x >= 0 ==> f x >= (0::int)) ==> setsum f A >= 0
  nat-set A ==> (!x. x >= 0 ==> f x >= (0::int)) ==> setprod f A >= 0
unfolding nat-set-def
apply (rule setsum-nonneg)
apply auto
apply (rule setprod-nonneg)
apply auto
done

lemma transfer-nat-int-sum-prod-cong:

```

```

  A = B  $\implies$  nat-set B  $\implies$  (!x. x >= 0  $\implies$  f x = g x)  $\implies$ 
    setsum f A = setsum g B
  A = B  $\implies$  nat-set B  $\implies$  (!x. x >= 0  $\implies$  f x = g x)  $\implies$ 
    setprod f A = setprod g B
unfolding nat-set-def
apply (subst setsum-cong, assumption)
apply auto [2]
apply (subst setprod-cong, assumption, auto)
done

```

```

declare transfer-morphism-nat-int [transfer add
  return: transfer-nat-int-sum-prod transfer-nat-int-sum-prod2
  transfer-nat-int-sum-prod-closure
  cong: transfer-nat-int-sum-prod-cong]

```

### 32.3 Set up transfer from int to nat

set up transfer direction

```

lemma transfer-morphism-int-nat: transfer-morphism int ( $\lambda n.$  True)
by (rule transfer-morphismI) simp

```

```

declare transfer-morphism-int-nat [transfer add
  mode: manual
  return: nat-int
  labels: int-nat
]

```

basic functions and relations

#### definition

```

  is-nat :: int  $\Rightarrow$  bool
where
  is-nat x = (x >= 0)

```

**lemma** transfer-int-nat-numerals:

```

  0 = int 0
  1 = int 1
  2 = int 2
  3 = int 3

```

**by** auto

**lemma** transfer-int-nat-functions:

```

  (int x) + (int y) = int (x + y)
  (int x) * (int y) = int (x * y)
  tsub (int x) (int y) = int (x - y)
  (int x) ^ n = int (x ^ n)

```

**by** (auto simp add: int-mult tsub-def int-power)

**lemma** transfer-int-nat-function-closures:

```

  is-nat x  $\implies$  is-nat y  $\implies$  is-nat (x + y)

```



```

is-nat x  $\implies$  is-nat y  $\implies$  is-nat (x * y)
is-nat x  $\implies$  is-nat y  $\implies$  is-nat (tsub x y)
is-nat x  $\implies$  is-nat (x^n)
is-nat 0
is-nat 1
is-nat 2
is-nat 3
is-nat (int z)
by (simp-all only: is-nat-def transfer-nat-int-function-closures)

```

**lemma** transfer-int-nat-relations:

```

(int x = int y) = (x = y)
(int x < int y) = (x < y)
(int x <= int y) = (x <= y)
(int x dvd int y) = (x dvd y)
by (auto simp add: zdvd-int)

```

**declare** transfer-morphism-int-nat [transfer add return:

```

transfer-int-nat-numerals
transfer-int-nat-functions
transfer-int-nat-function-closures
transfer-int-nat-relations

```

]

first-order quantifiers

**lemma** transfer-int-nat-quantifiers:

```

(ALL (x::int) >= 0. P x) = (ALL (x::nat). P (int x))
(EX (x::int) >= 0. P x) = (EX (x::nat). P (int x))
apply (subst all-nat)
apply auto [1]
apply (subst ex-nat)
apply auto
done

```

**declare** transfer-morphism-int-nat [transfer add  
return: transfer-int-nat-quantifiers]

if

```

lemma int-if-cong: (if P then (int x) else (int y)) =
  int (if P then x else y)
by auto

```

**declare** transfer-morphism-int-nat [transfer add return: int-if-cong]

operations with sets

**lemma** transfer-int-nat-set-functions:

```

nat-set A  $\implies$  card A = card (nat ‘ A)
{} = int ‘ ({}::nat set)
nat-set A  $\implies$  nat-set B  $\implies$  A Un B = int ‘ (nat ‘ A Un nat ‘ B)

```

$$\text{nat-set } A \implies \text{nat-set } B \implies A \text{ Int } B = \text{int } ' ( \text{nat } ' A \text{ Int } \text{nat } ' B )$$

$$\{x. x \geq 0 \ \& \ P \ x\} = \text{int } ' \{x. P(\text{int } x)\}$$

**by** (*simp-all only: is-nat-def transfer-nat-int-set-functions*  
*transfer-nat-int-set-function-closures*  
*transfer-nat-int-set-return-embed nat-0-le*  
*cong: transfer-nat-int-set-cong*)

**lemma** *transfer-int-nat-set-function-closures:*

$$\text{nat-set } \{\}$$

$$\text{nat-set } A \implies \text{nat-set } B \implies \text{nat-set } (A \text{ Un } B)$$

$$\text{nat-set } A \implies \text{nat-set } B \implies \text{nat-set } (A \text{ Int } B)$$

$$\text{nat-set } \{x. x \geq 0 \ \& \ P \ x\}$$

$$\text{nat-set } (\text{int } ' C)$$

$$\text{nat-set } A \implies x : A \implies \text{is-nat } x$$

**by** (*simp-all only: transfer-nat-int-set-function-closures is-nat-def*)

**lemma** *transfer-int-nat-set-relations:*

$$\text{nat-set } A \implies \text{finite } A = \text{finite } (\text{nat } ' A)$$

$$\text{is-nat } x \implies \text{nat-set } A \implies (x : A) = (\text{nat } x : \text{nat } ' A)$$

$$\text{nat-set } A \implies \text{nat-set } B \implies (A = B) = (\text{nat } ' A = \text{nat } ' B)$$

$$\text{nat-set } A \implies \text{nat-set } B \implies (A < B) = (\text{nat } ' A < \text{nat } ' B)$$

$$\text{nat-set } A \implies \text{nat-set } B \implies (A \leq B) = (\text{nat } ' A \leq \text{nat } ' B)$$

**by** (*simp-all only: is-nat-def transfer-nat-int-set-relations*  
*transfer-nat-int-set-return-embed nat-0-le*)

**lemma** *transfer-int-nat-set-return-embed: nat ' int ' A = A*

**by** (*simp only: transfer-nat-int-set-relations*  
*transfer-nat-int-set-function-closures*  
*transfer-nat-int-set-return-embed nat-0-le*)

**lemma** *transfer-int-nat-set-cong: (!x. P x = P' x)  $\implies$*

$$\{(x::\text{nat}). P \ x\} = \{(x). P' \ x\}$$

**by** *auto*

**declare** *transfer-morphism-int-nat [transfer add*

*return: transfer-int-nat-set-functions*  
*transfer-int-nat-set-function-closures*  
*transfer-int-nat-set-relations*  
*transfer-int-nat-set-return-embed*  
*cong: transfer-int-nat-set-cong*

*]*

setsum and setprod

**lemma** *transfer-int-nat-sum-prod:*

$$\text{nat-set } A \implies \text{setsum } f \ A = \text{setsum } (\%x. f \ (\text{int } x)) \ (\text{nat } ' A)$$

$$\text{nat-set } A \implies \text{setprod } f \ A = \text{setprod } (\%x. f \ (\text{int } x)) \ (\text{nat } ' A)$$

**apply** (*subst setsum-reindex*)

**apply** (*unfold inj-on-def nat-set-def, auto simp add: eq-nat-nat-iff*)

```

apply (subst setprod-reindex)
apply (unfold inj-on-def nat-set-def o-def, auto simp add: eq-nat-nat-iff
        cong: setprod-cong)
done

lemma transfer-int-nat-sum-prod2:
  (!!x. x:A  $\implies$  is-nat (f x))  $\implies$  setsum f A = int(setsum (%x. nat (f x)) A)
  (!!x. x:A  $\implies$  is-nat (f x))  $\implies$ 
    setprod f A = int(setprod (%x. nat (f x)) A)
  unfolding is-nat-def
  apply (subst int-setsum, auto)
  apply (subst int-setprod, auto simp add: cong: setprod-cong)
done

declare transfer-morphism-int-nat [transfer add
  return: transfer-int-nat-sum-prod transfer-int-nat-sum-prod2
  cong: setsum-cong setprod-cong]

end

```

### 33 Divides: The division operators div and mod

```

theory Divides
imports Nat-Numeral Nat-Transfer
uses ~~/src/Provers/Arith/cancel-div-mod.ML
begin

```

#### 33.1 Syntactic division operations

```

class div = dvd +
  fixes div :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl div 70)
  and mod :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixl mod 70)

```

#### 33.2 Abstract division in commutative semirings.

```

class semiring-div = comm-semiring-1-cancel + no-zero-divisors + div +
  assumes mod-div-equality: a div b * b + a mod b = a
  and div-by-0 [simp]: a div 0 = 0
  and div-0 [simp]: 0 div a = 0
  and div-mult-self1 [simp]: b  $\neq$  0  $\implies$  (a + c * b) div b = c + a div b
  and div-mult-mult1 [simp]: c  $\neq$  0  $\implies$  (c * a) div (c * b) = a div b
begin

```

*op div and op mod*

```

lemma mod-div-equality2: b * (a div b) + a mod b = a
  unfolding mult-commute [of b]
  by (rule mod-div-equality)

```

```

lemma mod-div-equality':  $a \bmod b + a \operatorname{div} b * b = a$ 
  using mod-div-equality [of a b]
  by (simp only: add-ac)

lemma div-mod-equality:  $((a \operatorname{div} b) * b + a \bmod b) + c = a + c$ 
  by (simp add: mod-div-equality)

lemma div-mod-equality2:  $(b * (a \operatorname{div} b) + a \bmod b) + c = a + c$ 
  by (simp add: mod-div-equality2)

lemma mod-by-0 [simp]:  $a \bmod 0 = a$ 
  using mod-div-equality [of a zero] by simp

lemma mod-0 [simp]:  $0 \bmod a = 0$ 
  using mod-div-equality [of zero a] div-0 by simp

lemma div-mult-self2 [simp]:
  assumes  $b \neq 0$ 
  shows  $(a + b * c) \operatorname{div} b = c + a \operatorname{div} b$ 
  using assms div-mult-self1 [of b a c] by (simp add: mult-commute)

lemma mod-mult-self1 [simp]:  $(a + c * b) \bmod b = a \bmod b$ 
proof (cases b = 0)
  case True then show ?thesis by simp
next
  case False
  have  $a + c * b = (a + c * b) \operatorname{div} b * b + (a + c * b) \bmod b$ 
    by (simp add: mod-div-equality)
  also from False div-mult-self1 [of b a c] have
     $\dots = (c + a \operatorname{div} b) * b + (a + c * b) \bmod b$ 
    by (simp add: algebra-simps)
  finally have  $a = a \operatorname{div} b * b + (a + c * b) \bmod b$ 
    by (simp add: add-commute [of a] add-assoc left-distrib)
  then have  $a \operatorname{div} b * b + (a + c * b) \bmod b = a \operatorname{div} b * b + a \bmod b$ 
    by (simp add: mod-div-equality)
  then show ?thesis by simp
qed

lemma mod-mult-self2 [simp]:  $(a + b * c) \bmod b = a \bmod b$ 
  by (simp add: mult-commute [of b])

lemma div-mult-self1-is-id [simp]:  $b \neq 0 \implies b * a \operatorname{div} b = a$ 
  using div-mult-self2 [of b 0 a] by simp

lemma div-mult-self2-is-id [simp]:  $b \neq 0 \implies a * b \operatorname{div} b = a$ 
  using div-mult-self1 [of b 0 a] by simp

lemma mod-mult-self1-is-0 [simp]:  $b * a \bmod b = 0$ 

```

```

using mod-mult-self2 [of 0 b a] by simp

lemma mod-mult-self2-is-0 [simp]:  $a * b \bmod b = 0$ 
  using mod-mult-self1 [of 0 a b] by simp

lemma div-by-1 [simp]:  $a \operatorname{div} 1 = a$ 
  using div-mult-self2-is-id [of 1 a] zero-neq-one by simp

lemma mod-by-1 [simp]:  $a \bmod 1 = 0$ 
proof –
  from mod-div-equality [of a one] div-by-1 have  $a + a \bmod 1 = a$  by simp
  then have  $a + a \bmod 1 = a + 0$  by simp
  then show ?thesis by (rule add-left-imp-eq)
qed

lemma mod-self [simp]:  $a \bmod a = 0$ 
  using mod-mult-self2-is-0 [of 1] by simp

lemma div-self [simp]:  $a \neq 0 \implies a \operatorname{div} a = 1$ 
  using div-mult-self2-is-id [of - 1] by simp

lemma div-add-self1 [simp]:
  assumes  $b \neq 0$ 
  shows  $(b + a) \operatorname{div} b = a \operatorname{div} b + 1$ 
  using assms div-mult-self1 [of b a 1] by (simp add: add-commute)

lemma div-add-self2 [simp]:
  assumes  $b \neq 0$ 
  shows  $(a + b) \operatorname{div} b = a \operatorname{div} b + 1$ 
  using assms div-add-self1 [of b a] by (simp add: add-commute)

lemma mod-add-self1 [simp]:
   $(b + a) \bmod b = a \bmod b$ 
  using mod-mult-self1 [of a 1 b] by (simp add: add-commute)

lemma mod-add-self2 [simp]:
   $(a + b) \bmod b = a \bmod b$ 
  using mod-mult-self1 [of a 1 b] by simp

lemma mod-div-decomp:
  fixes  $a\ b$ 
  obtains  $q\ r$  where  $q = a \operatorname{div} b$  and  $r = a \bmod b$ 
  and  $a = q * b + r$ 
proof –
  from mod-div-equality have  $a = a \operatorname{div} b * b + a \bmod b$  by simp
  moreover have  $a \operatorname{div} b = a \operatorname{div} b$  ..
  moreover have  $a \bmod b = a \bmod b$  ..
  note that ultimately show thesis by blast
qed

```

**lemma** *dvd-eq-mod-eq-0* [*code*, *code-unfold*, *code-inline del*]:  $a \text{ dvd } b \iff b \bmod a = 0$

**proof**

**assume**  $b \bmod a = 0$

**with** *mod-div-equality* [*of b a*] **have**  $b \text{ div } a * a = b$  **by** *simp*

**then have**  $b = a * (b \text{ div } a)$  **unfolding** *mult-commute* ..

**then have**  $\exists c. b = a * c$  ..

**then show**  $a \text{ dvd } b$  **unfolding** *dvd-def* .

**next**

**assume**  $a \text{ dvd } b$

**then have**  $\exists c. b = a * c$  **unfolding** *dvd-def* .

**then obtain**  $c$  **where**  $b = a * c$  ..

**then have**  $b \bmod a = a * c \bmod a$  **by** *simp*

**then have**  $b \bmod a = c * a \bmod a$  **by** (*simp add: mult-commute*)

**then show**  $b \bmod a = 0$  **by** *simp*

**qed**

**lemma** *mod-div-trivial* [*simp*]:  $a \bmod b \text{ div } b = 0$

**proof** (*cases b = 0*)

**assume**  $b = 0$

**thus** *?thesis* **by** *simp*

**next**

**assume**  $b \neq 0$

**hence**  $a \text{ div } b + a \bmod b \text{ div } b = (a \bmod b + a \text{ div } b * b) \text{ div } b$   
     **by** (*rule div-mult-self1* [*symmetric*])

**also have**  $\dots = a \text{ div } b$

**by** (*simp only: mod-div-equality'*)

**also have**  $\dots = a \text{ div } b + 0$

**by** *simp*

**finally show** *?thesis*

**by** (*rule add-left-imp-eq*)

**qed**

**lemma** *mod-mod-trivial* [*simp*]:  $a \bmod b \bmod b = a \bmod b$

**proof** –

**have**  $a \bmod b \bmod b = (a \bmod b + a \text{ div } b * b) \bmod b$

**by** (*simp only: mod-mult-self1*)

**also have**  $\dots = a \bmod b$

**by** (*simp only: mod-div-equality'*)

**finally show** *?thesis* .

**qed**

**lemma** *dvd-imp-mod-0*:  $a \text{ dvd } b \implies b \bmod a = 0$

**by** (*rule dvd-eq-mod-eq-0* [*THEN iffD1*])

**lemma** *dvd-div-mult-self*:  $a \text{ dvd } b \implies (b \text{ div } a) * a = b$

**by** (*subst* (2) *mod-div-equality* [*of b a*, *symmetric*]) (*simp add: dvd-imp-mod-0*)

**lemma** *dvd-mult-div-cancel*:  $a \text{ dvd } b \implies a * (b \text{ div } a) = b$   
**by** (*drule* *dvd-div-mult-self*) (*simp* *add*: *mult-commute*)

**lemma** *dvd-div-mult*:  $a \text{ dvd } b \implies (b \text{ div } a) * c = b * c \text{ div } a$   
**apply** (*cases*  $a = 0$ )  
**apply** *simp*  
**apply** (*auto* *simp*: *dvd-def* *mult-assoc*)  
**done**

**lemma** *div-dvd-div*[*simp*]:  
 $a \text{ dvd } b \implies a \text{ dvd } c \implies (b \text{ div } a \text{ dvd } c \text{ div } a) = (b \text{ dvd } c)$   
**apply** (*cases*  $a = 0$ )  
**apply** *simp*  
**apply** (*unfold* *dvd-def*)  
**apply** *auto*  
**apply** (*blast* *intro*:*mult-assoc*[*symmetric*])  
**apply** (*fastsimp* *simp* *add*: *mult-assoc*)  
**done**

**lemma** *dvd-mod-imp-dvd*:  $[k \text{ dvd } m \text{ mod } n; k \text{ dvd } n] \implies k \text{ dvd } m$   
**apply** (*subgoal-tac*  $k \text{ dvd } (m \text{ div } n) * n + m \text{ mod } n$ )  
**apply** (*simp* *add*: *mod-div-equality*)  
**apply** (*simp* *only*: *dvd-add* *dvd-mult*)  
**done**

Addition respects modular equivalence.

**lemma** *mod-add-left-eq*:  $(a + b) \text{ mod } c = (a \text{ mod } c + b) \text{ mod } c$   
**proof** –  
**have**  $(a + b) \text{ mod } c = (a \text{ div } c * c + a \text{ mod } c + b) \text{ mod } c$   
**by** (*simp* *only*: *mod-div-equality*)  
**also have**  $\dots = (a \text{ mod } c + b + a \text{ div } c * c) \text{ mod } c$   
**by** (*simp* *only*: *add-ac*)  
**also have**  $\dots = (a \text{ mod } c + b) \text{ mod } c$   
**by** (*rule* *mod-mult-self1*)  
**finally show** *?thesis* .  
**qed**

**lemma** *mod-add-right-eq*:  $(a + b) \text{ mod } c = (a + b \text{ mod } c) \text{ mod } c$   
**proof** –  
**have**  $(a + b) \text{ mod } c = (a + (b \text{ div } c * c + b \text{ mod } c)) \text{ mod } c$   
**by** (*simp* *only*: *mod-div-equality*)  
**also have**  $\dots = (a + b \text{ mod } c + b \text{ div } c * c) \text{ mod } c$   
**by** (*simp* *only*: *add-ac*)  
**also have**  $\dots = (a + b \text{ mod } c) \text{ mod } c$   
**by** (*rule* *mod-mult-self1*)  
**finally show** *?thesis* .  
**qed**

**lemma** *mod-add-eq*:  $(a + b) \text{ mod } c = (a \text{ mod } c + b \text{ mod } c) \text{ mod } c$

by (rule trans [OF mod-add-left-eq mod-add-right-eq])

**lemma** *mod-add-cong*:

assumes  $a \bmod c = a' \bmod c$

assumes  $b \bmod c = b' \bmod c$

shows  $(a + b) \bmod c = (a' + b') \bmod c$

**proof** –

have  $(a \bmod c + b \bmod c) \bmod c = (a' \bmod c + b' \bmod c) \bmod c$

unfolding *assms* ..

thus ?thesis

by (simp only: mod-add-eq [symmetric])

qed

**lemma** *div-add* [simp]:  $z \text{ dvd } x \implies z \text{ dvd } y$

$\implies (x + y) \text{ div } z = x \text{ div } z + y \text{ div } z$

by (cases  $z = 0$ , simp, unfold dvd-def, auto simp add: algebra-simps)

Multiplication respects modular equivalence.

**lemma** *mod-mult-left-eq*:  $(a * b) \bmod c = ((a \bmod c) * b) \bmod c$

**proof** –

have  $(a * b) \bmod c = ((a \text{ div } c * c + a \bmod c) * b) \bmod c$

by (simp only: mod-div-equality)

also have  $\dots = (a \bmod c * b + a \text{ div } c * b * c) \bmod c$

by (simp only: algebra-simps)

also have  $\dots = (a \bmod c * b) \bmod c$

by (rule mod-mult-self1)

finally show ?thesis .

qed

**lemma** *mod-mult-right-eq*:  $(a * b) \bmod c = (a * (b \bmod c)) \bmod c$

**proof** –

have  $(a * b) \bmod c = (a * (b \text{ div } c * c + b \bmod c)) \bmod c$

by (simp only: mod-div-equality)

also have  $\dots = (a * (b \bmod c) + a * (b \text{ div } c) * c) \bmod c$

by (simp only: algebra-simps)

also have  $\dots = (a * (b \bmod c)) \bmod c$

by (rule mod-mult-self1)

finally show ?thesis .

qed

**lemma** *mod-mult-eq*:  $(a * b) \bmod c = ((a \bmod c) * (b \bmod c)) \bmod c$

by (rule trans [OF mod-mult-left-eq mod-mult-right-eq])

**lemma** *mod-mult-cong*:

assumes  $a \bmod c = a' \bmod c$

assumes  $b \bmod c = b' \bmod c$

shows  $(a * b) \bmod c = (a' * b') \bmod c$

**proof** –

have  $(a \bmod c * (b \bmod c)) \bmod c = (a' \bmod c * (b' \bmod c)) \bmod c$



```

    unfolding assms ..
  thus ?thesis
    by (simp only: mod-mult-eq [symmetric])
qed

```

**lemma** *mod-mod-cancel*:

```

  assumes  $c \text{ dvd } b$ 
  shows  $a \bmod b \bmod c = a \bmod c$ 
proof -
  from  $\langle c \text{ dvd } b \rangle$  obtain  $k$  where  $b = c * k$ 
  by (rule dvdE)
  have  $a \bmod b \bmod c = a \bmod (c * k) \bmod c$ 
  by (simp only:  $\langle b = c * k \rangle$ )
  also have  $\dots = (a \bmod (c * k) + a \text{ div } (c * k) * k * c) \bmod c$ 
  by (simp only: mod-mult-self1)
  also have  $\dots = (a \text{ div } (c * k) * (c * k) + a \bmod (c * k)) \bmod c$ 
  by (simp only: add-ac mult-ac)
  also have  $\dots = a \bmod c$ 
  by (simp only: mod-div-equality)
  finally show ?thesis .
qed

```

**lemma** *div-mult-div-if-dvd*:

```

 $y \text{ dvd } x \implies z \text{ dvd } w \implies (x \text{ div } y) * (w \text{ div } z) = (x * w) \text{ div } (y * z)$ 
apply (cases  $y = 0$ , simp)
apply (cases  $z = 0$ , simp)
apply (auto elim!: dvdE simp add: algebra-simps)
apply (subst mult-assoc [symmetric])
apply (simp add: no-zero-divisors)
done

```

**lemma** *div-mult-swap*:

```

  assumes  $c \text{ dvd } b$ 
  shows  $a * (b \text{ div } c) = (a * b) \text{ div } c$ 
proof -
  from assms have  $b \text{ div } c * (a \text{ div } 1) = b * a \text{ div } (c * 1)$ 
  by (simp only: div-mult-div-if-dvd one-dvd)
  then show ?thesis by (simp add: mult-commute)
qed

```

**lemma** *div-mult-mult2* [simp]:

```

 $c \neq 0 \implies (a * c) \text{ div } (b * c) = a \text{ div } b$ 
by (drule div-mult-mult1) (simp add: mult-commute)

```

**lemma** *div-mult-mult1-if* [simp]:

```

 $(c * a) \text{ div } (c * b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a \text{ div } b)$ 
by simp-all

```

**lemma** *mod-mult-mult1*:

$(c * a) \bmod (c * b) = c * (a \bmod b)$   
**proof** (cases  $c = 0$ )  
   **case** *True* **then show** ?thesis **by** *simp*  
**next**  
   **case** *False*  
   **from** *mod-div-equality*  
   **have**  $((c * a) \operatorname{div} (c * b)) * (c * b) + (c * a) \bmod (c * b) = c * a$  .  
   **with** *False* **have**  $c * ((a \operatorname{div} b) * b + a \bmod b) + (c * a) \bmod (c * b)$   
      $= c * a + c * (a \bmod b)$  **by** (*simp add: algebra-simps*)  
   **with** *mod-div-equality* **show** ?thesis **by** *simp*  
**qed**

**lemma** *mod-mult-mult2*:  
 $(a * c) \bmod (b * c) = (a \bmod b) * c$   
**using** *mod-mult-mult1* [of  $c$   $a$   $b$ ] **by** (*simp add: mult-commute*)

**lemma** *dvd-mod*:  $k \operatorname{dvd} m \implies k \operatorname{dvd} n \implies k \operatorname{dvd} (m \bmod n)$   
**unfolding** *dvd-def* **by** (*auto simp add: mod-mult-mult1*)

**lemma** *dvd-mod-iff*:  $k \operatorname{dvd} n \implies k \operatorname{dvd} (m \bmod n) \longleftrightarrow k \operatorname{dvd} m$   
**by** (*blast intro: dvd-mod-imp-dvd dvd-mod*)

**lemma** *div-power*:  
 $y \operatorname{dvd} x \implies (x \operatorname{div} y) ^ n = x ^ n \operatorname{div} y ^ n$   
**apply** (*induct n*)  
**apply** *simp*  
**apply** (*simp add: div-mult-div-if-dvd dvd-power-same*)  
**done**

**lemma** *dvd-div-eq-mult*:  
**assumes**  $a \neq 0$  **and**  $a \operatorname{dvd} b$   
**shows**  $b \operatorname{div} a = c \longleftrightarrow b = c * a$   
**proof**  
**assume**  $b = c * a$   
**then show**  $b \operatorname{div} a = c$  **by** (*simp add: assms*)  
**next**  
**assume**  $b \operatorname{div} a = c$   
**then have**  $b \operatorname{div} a * a = c * a$  **by** *simp*  
**moreover from**  $\langle a \operatorname{dvd} b \rangle$  **have**  $b \operatorname{div} a * a = b$  **by** (*simp add: dvd-div-mult-self*)  
**ultimately show**  $b = c * a$  **by** *simp*  
**qed**

**lemma** *dvd-div-div-eq-mult*:  
**assumes**  $a \neq 0$   $c \neq 0$  **and**  $a \operatorname{dvd} b$   $c \operatorname{dvd} d$   
**shows**  $b \operatorname{div} a = d \operatorname{div} c \longleftrightarrow b * c = a * d$   
**using** *assms* **by** (*auto simp add: mult-commute [of -  $a$ ] dvd-div-mult-self dvd-div-eq-mult div-mult-swap intro: sym*)

**end**

```
class ring-div = semiring-div + comm-ring-1
begin
```

```
subclass ring-1-no-zero-divisors ..
```

Negation respects modular equivalence.

```
lemma mod-minus-eq:  $(- a) \bmod b = (- (a \bmod b)) \bmod b$ 
proof -
  have  $(- a) \bmod b = (- (a \operatorname{div} b * b + a \bmod b)) \bmod b$ 
    by (simp only: mod-div-equality)
  also have  $\dots = (- (a \bmod b) + - (a \operatorname{div} b) * b) \bmod b$ 
    by (simp only: minus-add-distrib minus-mult-left add-ac)
  also have  $\dots = (- (a \bmod b)) \bmod b$ 
    by (rule mod-mult-self1)
  finally show ?thesis .
qed
```

```
lemma mod-minus-cong:
  assumes  $a \bmod b = a' \bmod b$ 
  shows  $(- a) \bmod b = (- a') \bmod b$ 
proof -
  have  $(- (a \bmod b)) \bmod b = (- (a' \bmod b)) \bmod b$ 
    unfolding assms ..
  thus ?thesis
    by (simp only: mod-minus-eq [symmetric])
qed
```

Subtraction respects modular equivalence.

```
lemma mod-diff-left-eq:  $(a - b) \bmod c = (a \bmod c - b) \bmod c$ 
  unfolding diff-minus
  by (intro mod-add-cong mod-minus-cong) simp-all
```

```
lemma mod-diff-right-eq:  $(a - b) \bmod c = (a - b \bmod c) \bmod c$ 
  unfolding diff-minus
  by (intro mod-add-cong mod-minus-cong) simp-all
```

```
lemma mod-diff-eq:  $(a - b) \bmod c = (a \bmod c - b \bmod c) \bmod c$ 
  unfolding diff-minus
  by (intro mod-add-cong mod-minus-cong) simp-all
```

```
lemma mod-diff-cong:
  assumes  $a \bmod c = a' \bmod c$ 
  assumes  $b \bmod c = b' \bmod c$ 
  shows  $(a - b) \bmod c = (a' - b') \bmod c$ 
  unfolding diff-minus using assms
  by (intro mod-add-cong mod-minus-cong)
```

```
lemma dvd-neg-div:  $y \operatorname{dvd} x \implies -x \operatorname{div} y = - (x \operatorname{div} y)$ 
```

```

apply (case-tac  $y = 0$ ) apply simp
apply (auto simp add: dvd-def)
apply (subgoal-tac  $-(y * k) = y * -k$ )
  apply (erule ssubst)
  apply (erule div-mult-self1-is-id)
apply simp
done

lemma dvd-div-neg:  $y \text{ dvd } x \implies x \text{ div } -y = -(x \text{ div } y)$ 
apply (case-tac  $y = 0$ ) apply simp
apply (auto simp add: dvd-def)
apply (subgoal-tac  $y * k = -y * -k$ )
  apply (erule ssubst)
  apply (rule div-mult-self1-is-id)
apply simp
apply simp
done

end

```

### 33.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments *m*, *n* and two output arguments *q*(uotient) and *r*(emainder).

**definition** *divmod-nat-rel* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*  $\times$  *nat*  $\Rightarrow$  *bool* **where**  
 $\text{divmod-nat-rel } m \ n \ qr \longleftrightarrow$   
 $m = \text{fst } qr * n + \text{snd } qr \wedge$   
 $(\text{if } n = 0 \text{ then } \text{fst } qr = 0 \text{ else if } n > 0 \text{ then } 0 \leq \text{snd } qr \wedge \text{snd } qr < n \text{ else } n < \text{snd } qr \wedge \text{snd } qr \leq 0)$

*divmod-nat-rel* is total:

**lemma** *divmod-nat-rel-ex*:  
**obtains** *q r* **where** *divmod-nat-rel* *m n* (*q*, *r*)  
**proof** (cases  $n = 0$ )  
**case** *True* **with** *that* **show** *thesis*  
**by** (auto simp add: divmod-nat-rel-def)  
**next**  
**case** *False*  
**have**  $\exists q \ r. m = q * n + r \wedge r < n$   
**proof** (induct *m*)  
**case** 0 **with**  $\langle n \neq 0 \rangle$   
**have**  $(0::\text{nat}) = 0 * n + 0 \wedge 0 < n$  **by** simp  
**then show** ?case **by** blast  
**next**  
**case** (Suc *m*) **then obtain** *q' r'*  
**where**  $m = q' * n + r'$  **and**  $n: r' < n$  **by** auto  
**then show** ?case **proof** (cases Suc *r'* < *n*)

```

    case True
    from m n have Suc m = q' * n + Suc r' by simp
    with True show ?thesis by blast
  next
    case False then have n ≤ Suc r' by auto
    moreover from n have Suc r' ≤ n by auto
    ultimately have n = Suc r' by auto
    with m have Suc m = Suc q' * n + 0 by simp
    with ⟨n ≠ 0⟩ show ?thesis by blast
  qed
qed
with that show thesis
  using ⟨n ≠ 0⟩ by (auto simp add: divmod-nat-rel-def)
qed

divmod-nat-rel is injective:
lemma divmod-nat-rel-unique:
  assumes divmod-nat-rel m n qr
    and divmod-nat-rel m n qr'
  shows qr = qr'
proof (cases n = 0)
  case True with assms show ?thesis
    by (cases qr, cases qr')
      (simp add: divmod-nat-rel-def)
next
  case False
  have aux:  $\bigwedge q r q' r'. q' * n + r' = q * n + r \implies r < n \implies q' \leq (q::nat)$ 
  apply (rule leI)
  apply (subst less-iff-Suc-add)
  apply (auto simp add: add-mult-distrib)
  done
  from ⟨n ≠ 0⟩ assms have fst qr = fst qr'
    by (auto simp add: divmod-nat-rel-def intro: order-antisym dest: aux sym)
  moreover from this assms have snd qr = snd qr'
    by (simp add: divmod-nat-rel-def)
  ultimately show ?thesis by (cases qr, cases qr') simp
qed

```

We instantiate divisibility on the natural numbers by means of *divmod-nat-rel*:

```

instantiation nat :: semiring-div
begin

```

```

definition divmod-nat :: nat ⇒ nat ⇒ nat × nat where
  [code del]: divmod-nat m n = (THE qr. divmod-nat-rel m n qr)

```

```

lemma divmod-nat-rel-divmod-nat:
  divmod-nat-rel m n (divmod-nat m n)
proof –
  from divmod-nat-rel-ex

```

**obtain** *qr* **where** *rel*: *divmod-nat-rel* *m n qr* .  
**then show** *?thesis*  
**by** (*auto simp add: divmod-nat-def intro: theI elim: divmod-nat-rel-unique*)  
**qed**

**lemma** *divmod-nat-eq*:  
**assumes** *divmod-nat-rel m n qr*  
**shows** *divmod-nat m n = qr*  
**using** *assms* **by** (*auto intro: divmod-nat-rel-unique divmod-nat-rel-divmod-nat*)

**definition** *div-nat* **where**  
 $m \text{ div } n = \text{fst } (\text{divmod-nat } m \ n)$

**definition** *mod-nat* **where**  
 $m \text{ mod } n = \text{snd } (\text{divmod-nat } m \ n)$

**lemma** *divmod-nat-div-mod*:  
 $\text{divmod-nat } m \ n = (m \text{ div } n, m \text{ mod } n)$   
**unfolding** *div-nat-def mod-nat-def* **by** *simp*

**lemma** *div-eq*:  
**assumes** *divmod-nat-rel m n (q, r)*  
**shows**  $m \text{ div } n = q$   
**using** *assms* **by** (*auto dest: divmod-nat-eq simp add: divmod-nat-div-mod*)

**lemma** *mod-eq*:  
**assumes** *divmod-nat-rel m n (q, r)*  
**shows**  $m \text{ mod } n = r$   
**using** *assms* **by** (*auto dest: divmod-nat-eq simp add: divmod-nat-div-mod*)

**lemma** *divmod-nat-rel*: *divmod-nat-rel m n (m div n, m mod n)*  
**by** (*simp add: div-nat-def mod-nat-def divmod-nat-rel-divmod-nat*)

**lemma** *divmod-nat-zero*:  
 $\text{divmod-nat } m \ 0 = (0, m)$   
**proof** –  
**from** *divmod-nat-rel [of m 0]* **show** *?thesis*  
**unfolding** *divmod-nat-div-mod divmod-nat-rel-def* **by** *simp*  
**qed**

**lemma** *divmod-nat-base*:  
**assumes**  $m < n$   
**shows**  $\text{divmod-nat } m \ n = (0, m)$   
**proof** –  
**from** *divmod-nat-rel [of m n]* **show** *?thesis*  
**unfolding** *divmod-nat-div-mod divmod-nat-rel-def*  
**using** *assms* **by** (*cases m div n = 0*)  
*(auto simp add: gr0-conv-Suc [of m div n])*  
**qed**

**lemma** *divmod-nat-step*:  
**assumes**  $0 < n$  **and**  $n \leq m$   
**shows**  $\text{divmod-nat } m \ n = (\text{Suc } ((m - n) \text{ div } n), (m - n) \text{ mod } n)$   
**proof** –  
**from** *divmod-nat-rel* **have** *divmod-nat-m-n*:  $\text{divmod-nat-rel } m \ n \ (m \text{ div } n, m \text{ mod } n)$  .  
**with** *assms* **have** *m-div-n*:  $m \text{ div } n \geq 1$   
**by** (*cases*  $m \text{ div } n$ ) (*auto simp add: divmod-nat-rel-def*)  
**have**  $\text{divmod-nat-rel } (m - n) \ n \ (m \text{ div } n - \text{Suc } 0, m \text{ mod } n)$   
**proof** –  
**from** *assms* **have**  
 $n \neq 0$   
 $\bigwedge k. m = \text{Suc } k * n + m \text{ mod } n \implies m - n = (\text{Suc } k - \text{Suc } 0) * n + m \text{ mod } n$   
**by** *simp-all*  
**then show** *?thesis* **using** *assms divmod-nat-m-n*  
**by** (*cases*  $m \text{ div } n$ )  
(*simp-all only: divmod-nat-rel-def fst-conv snd-conv, simp-all*)  
**qed**  
**with** *divmod-nat-eq* **have**  $\text{divmod-nat } (m - n) \ n = (m \text{ div } n - \text{Suc } 0, m \text{ mod } n)$  **by** *simp*  
**moreover from** *divmod-nat-div-mod* **have**  $\text{divmod-nat } (m - n) \ n = ((m - n) \text{ div } n, (m - n) \text{ mod } n)$  .  
**ultimately have**  $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$   
**and**  $m \text{ mod } n = (m - n) \text{ mod } n$  **using** *m-div-n* **by** *simp-all*  
**then show** *?thesis* **using** *divmod-nat-div-mod* **by** *simp*  
**qed**

The “recursion” equations for *op div* and *op mod*

**lemma** *div-less [simp]*:  
**fixes**  $m \ n :: \text{nat}$   
**assumes**  $m < n$   
**shows**  $m \text{ div } n = 0$   
**using** *assms divmod-nat-base divmod-nat-div-mod* **by** *simp*

**lemma** *le-div-geq*:  
**fixes**  $m \ n :: \text{nat}$   
**assumes**  $0 < n$  **and**  $n \leq m$   
**shows**  $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$   
**using** *assms divmod-nat-step divmod-nat-div-mod* **by** *simp*

**lemma** *mod-less [simp]*:  
**fixes**  $m \ n :: \text{nat}$   
**assumes**  $m < n$   
**shows**  $m \text{ mod } n = m$   
**using** *assms divmod-nat-base divmod-nat-div-mod* **by** *simp*

**lemma** *le-mod-geq*:

```

fixes  $m\ n :: \text{nat}$ 
assumes  $n \leq m$ 
shows  $m \bmod n = (m - n) \bmod n$ 
using assms divmod-nat-step divmod-nat-div-mod by (cases  $n = 0$ ) simp-all

instance proof –
  have [simp]:  $\bigwedge n :: \text{nat}. n \text{ div } 0 = 0$ 
    by (simp add: div-nat-def divmod-nat-zero)
  have [simp]:  $\bigwedge n :: \text{nat}. 0 \text{ div } n = 0$ 
  proof –
    fix  $n :: \text{nat}$ 
    show  $0 \text{ div } n = 0$ 
    by (cases  $n = 0$ ) simp-all
  qed
show OFCLASS(nat, semiring-div-class) proof
  fix  $m\ n :: \text{nat}$ 
  show  $m \text{ div } n * n + m \bmod n = m$ 
    using divmod-nat-rel [of m n] by (simp add: divmod-nat-rel-def)
next
  fix  $m\ n\ q :: \text{nat}$ 
  assume  $n \neq 0$ 
  then show  $(q + m * n) \text{ div } n = m + q \text{ div } n$ 
    by (induct m) (simp-all add: le-div-geq)
next
  fix  $m\ n\ q :: \text{nat}$ 
  assume  $m \neq 0$ 
  then show  $(m * n) \text{ div } (m * q) = n \text{ div } q$ 
  proof (cases  $n \neq 0 \wedge q \neq 0$ )
    case False then show ?thesis by auto
  next
    case True with  $\langle m \neq 0 \rangle$ 
      have  $m > 0$  and  $n > 0$  and  $q > 0$  by auto
      then have  $\bigwedge a\ b. \text{divmod-nat-rel } n\ q\ (a, b) \implies \text{divmod-nat-rel } (m * n)\ (m * q)\ (a, m * b)$ 
        by (auto simp add: divmod-nat-rel-def) (simp-all add: algebra-simps)
      moreover from divmod-nat-rel have divmod-nat-rel  $n\ q\ (n \text{ div } q, n \bmod q)$  .
      ultimately have divmod-nat-rel  $(m * n)\ (m * q)\ (n \text{ div } q, m * (n \bmod q))$  .
      then show ?thesis by (simp add: div-eq)
    qed
  qed simp-all
qed

end

lemma divmod-nat-if [code]: divmod-nat  $m\ n = (\text{if } n = 0 \vee m < n \text{ then } (0, m)$ 
  else
    let  $(q, r) = \text{divmod-nat } (m - n)\ n \text{ in } (\text{Suc } q, r))$ 
by (simp add: divmod-nat-zero divmod-nat-base divmod-nat-step)
  (simp add: divmod-nat-div-mod)

```



Simproc for cancelling *op div* and *op mod*

**ML**  $\ll$   
*local*

```
structure CancelDivMod = CancelDivModFun(struct

  val div-name = @{const-name div};
  val mod-name = @{const-name mod};
  val mk-binop = HOLogic.mk-binop;
  val mk-sum = Nat-Arith.mk-sum;
  val dest-sum = Nat-Arith.dest-sum;

  val div-mod-egs = map mk-meta-eq [ @{thm div-mod-equality}, @{thm div-mod-equality2} ];

  val trans = trans;

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac
    (@{thm add-0-left} :: @{thm add-0-right} :: @{thms add-ac}))

end)

in

val cancel-div-mod-nat-proc = Simplifier.simproc @{theory}
  cancel-div-mod [(m::nat) + n] (K CancelDivMod.proc);

val - = Addsimprocs [cancel-div-mod-nat-proc];

end
 $\gg$ 
```

### 33.3.1 Quotient

**lemma** *div-geq*:  $0 < n \implies \neg m < n \implies m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$   
**by** (*simp add: le-div-geq linorder-not-less*)

**lemma** *div-if*:  $0 < n \implies m \text{ div } n = (\text{if } m < n \text{ then } 0 \text{ else } \text{Suc } ((m - n) \text{ div } n))$   
**by** (*simp add: div-geq*)

**lemma** *div-mult-self-is-m* [*simp*]:  $0 < n \implies (m * n) \text{ div } n = (m :: \text{nat})$   
**by** *simp*

**lemma** *div-mult-self1-is-m* [*simp*]:  $0 < n \implies (n * m) \text{ div } n = (m :: \text{nat})$   
**by** *simp*

### 33.3.2 Remainder

**lemma** *mod-less-divisor* [*simp*]:  
**fixes**  $m \ n :: \text{nat}$

```

assumes  $n > 0$ 
shows  $m \bmod n < (n::nat)$ 
using assms divmod-nat-rel [of  $m\ n$ ] unfolding divmod-nat-rel-def by auto

```

```

lemma mod-less-eq-dividend [simp]:
  fixes  $m\ n :: nat$ 
  shows  $m \bmod n \leq m$ 
proof (rule add-leD2)
  from mod-div-equality have  $m \operatorname{div} n * n + m \bmod n = m$  .
  then show  $m \operatorname{div} n * n + m \bmod n \leq m$  by auto
qed

```

```

lemma mod-geq:  $\neg m < (n::nat) \implies m \bmod n = (m - n) \bmod n$ 
by (simp add: le-mod-geq linorder-not-less)

```

```

lemma mod-if:  $m \bmod (n::nat) = (\text{if } m < n \text{ then } m \text{ else } (m - n) \bmod n)$ 
by (simp add: le-mod-geq)

```

```

lemma mod-1 [simp]:  $m \bmod \operatorname{Suc} 0 = 0$ 
by (induct m) (simp-all add: mod-geq)

```

```

lemma mod-mult-distrib:  $(m \bmod n) * (k::nat) = (m * k) \bmod (n * k)$ 
apply (cases n = 0, simp)
apply (cases k = 0, simp)
apply (induct m rule: nat-less-induct)
apply (subst mod-if, simp)
apply (simp add: mod-geq diff-mult-distrib)
done

```

```

lemma mod-mult-distrib2:  $(k::nat) * (m \bmod n) = (k * m) \bmod (k * n)$ 
by (simp add: mult-commute [of k] mod-mult-distrib)

```

```

lemma mult-div-cancel:  $(n::nat) * (m \operatorname{div} n) = m - (m \bmod n)$ 
by (cut-tac a = m and b = n in mod-div-equality2, arith)

```

```

lemma mod-le-divisor[simp]:  $0 < n \implies m \bmod n \leq (n::nat)$ 
apply (drule mod-less-divisor [where m = m])
apply simp
done

```

### 33.3.3 Quotient and Remainder

```

lemma divmod-nat-rel-mult1-eq:
   $\operatorname{divmod-nat-rel} b\ c\ (q, r) \implies c > 0$ 
   $\implies \operatorname{divmod-nat-rel} (a * b)\ c\ (a * q + a * r \operatorname{div} c, a * r \bmod c)$ 
by (auto simp add: split-ifs divmod-nat-rel-def algebra-simps)

```

```

lemma div-mult1-eq:

```

```

  (a * b) div c = a * (b div c) + a * (b mod c) div (c::nat)
apply (cases c = 0, simp)
apply (blast intro: divmod-nat-rel [THEN divmod-nat-rel-mult1-eq, THEN div-eq])
done

```

**lemma** *divmod-nat-rel-add1-eq*:

```

  divmod-nat-rel a c (aq, ar) ==> divmod-nat-rel b c (bq, br) ==> c > 0
  ==> divmod-nat-rel (a + b) c (aq + bq + (ar + br) div c, (ar + br) mod c)
by (auto simp add: split-ifs divmod-nat-rel-def algebra-simps)

```

**lemma** *div-add1-eq*:

```

  (a+b) div (c::nat) = a div c + b div c + ((a mod c + b mod c) div c)
apply (cases c = 0, simp)
apply (blast intro: divmod-nat-rel-add1-eq [THEN div-eq] divmod-nat-rel)
done

```

**lemma** *mod-lemma*:  $[ (0::nat) < c; r < b ] ==> b * (q \bmod c) + r < b * c$

```

apply (cut-tac m = q and n = c in mod-less-divisor)
apply (drule-tac [2] m = q mod c in less-imp-Suc-add, auto)
apply (erule-tac P = %x. ?lhs < ?rhs x in ssubst)
apply (simp add: add-mult-distrib2)
done

```

**lemma** *divmod-nat-rel-mult2-eq*:

```

  divmod-nat-rel a b (q, r) ==> 0 < b ==> 0 < c
  ==> divmod-nat-rel a (b * c) (q div c, b *(q mod c) + r)
by (auto simp add: mult-ac divmod-nat-rel-def add-mult-distrib2 [symmetric] mod-lemma)

```

**lemma** *div-mult2-eq*:  $a \text{ div } (b*c) = (a \text{ div } b) \text{ div } (c::nat)$

```

apply (cases b = 0, simp)
apply (cases c = 0, simp)
apply (force simp add: divmod-nat-rel [THEN divmod-nat-rel-mult2-eq, THEN
div-eq])
done

```

**lemma** *mod-mult2-eq*:  $a \bmod (b*c) = b*(a \text{ div } b \bmod c) + a \bmod (b::nat)$

```

apply (cases b = 0, simp)
apply (cases c = 0, simp)
apply (auto simp add: mult-commute divmod-nat-rel [THEN divmod-nat-rel-mult2-eq,
THEN mod-eq])
done

```

### 33.3.4 Further Facts about Quotient and Remainder

**lemma** *div-1* [simp]:  $m \text{ div } \text{Suc } 0 = m$

**by** (induct m) (simp-all add: div-geq)

```

lemma div-le-mono [rule-format (no-asm)]:
   $\forall m::nat. m \leq n \longrightarrow (m \text{ div } k) \leq (n \text{ div } k)$ 
apply (case-tac  $k=0$ , simp)
apply (induct n rule: nat-less-induct, clarify)
apply (case-tac  $n < k$ )

```

```

apply simp

```

```

apply (case-tac  $m < k$ )

```

```

apply simp

```

```

apply (simp add: div-geq diff-le-mono)
done

```

```

lemma div-le-mono2:  $!!m::nat. [| 0 < m; m \leq n |] \Longrightarrow (k \text{ div } n) \leq (k \text{ div } m)$ 
apply (subgoal-tac  $0 < n$ )
  prefer 2 apply simp
apply (induct-tac k rule: nat-less-induct)
apply (rename-tac k)
apply (case-tac  $k < n$ , simp)
apply (subgoal-tac  $\sim (k < m)$ )
  prefer 2 apply simp
apply (simp add: div-geq)
apply (subgoal-tac  $(k-n) \text{ div } n \leq (k-m) \text{ div } n$ )
  prefer 2
    apply (blast intro: div-le-mono diff-le-mono2)
apply (rule le-trans, simp)
apply (simp)
done

```

```

lemma div-le-dividend [simp]:  $m \text{ div } n \leq (m::nat)$ 
apply (case-tac  $n=0$ , simp)
apply (subgoal-tac  $m \text{ div } n \leq m \text{ div } 1$ , simp)
apply (rule div-le-mono2)
apply (simp-all (no-asm-simp))
done

```

```

lemma div-less-dividend [rule-format]:
   $!!n::nat. 1 < n \Longrightarrow 0 < m \longrightarrow m \text{ div } n < m$ 
apply (induct-tac m rule: nat-less-induct)
apply (rename-tac m)
apply (case-tac  $m < n$ , simp)
apply (subgoal-tac  $0 < n$ )
  prefer 2 apply simp
apply (simp add: div-geq)

```

```

apply (case-tac  $n < m$ )
  apply (subgoal-tac  $(m - n) \text{ div } n < (m - n)$ )
    apply (rule impI less-trans-Suc)+
apply assumption
  apply (simp-all)
done

```

```

declare div-less-dividend [simp]

```

A fact for the mutilated chess board

```

lemma mod-Suc:  $\text{Suc}(m) \bmod n = (\text{if } \text{Suc}(m \bmod n) = n \text{ then } 0 \text{ else } \text{Suc}(m \bmod n))$ 
apply (case-tac  $n = 0$ , simp)
apply (induct m rule: nat-less-induct)
apply (case-tac  $\text{Suc } (na) < n$ )

```

```

apply (frule lessI [THEN less-trans], simp add: less-not-refl3)

```

```

apply (simp add: linorder-not-less le-Suc-eq mod-geq)
apply (auto simp add: Suc-diff-le le-mod-geq)
done

```

```

lemma mod-eq-0-iff:  $(m \bmod d = 0) = (\exists q::\text{nat}. m = d * q)$ 
by (auto simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)

```

```

lemmas mod-eq-0D [dest!] = mod-eq-0-iff [THEN iffD1]

```

```

lemma mod-eqD:  $(m \bmod d = r) ==> \exists q::\text{nat}. m = r + q * d$ 
apply (cut-tac  $a = m$  in mod-div-equality)
apply (simp only: add-ac)
apply (blast intro: sym)
done

```

```

lemma split-div:

```

```

   $P(n \text{ div } k :: \text{nat}) =$ 
   $((k = 0 \longrightarrow P\ 0) \wedge (k \neq 0 \longrightarrow (!i. !j < k. n = k * i + j \longrightarrow P\ i)))$ 
   $(\text{is } ?P = ?Q \text{ is } - = (- \wedge (- \longrightarrow ?R)))$ 

```

```

proof

```

```

  assume P: ?P

```

```

  show ?Q

```

```

  proof (cases)

```

```

    assume  $k = 0$ 

```

```

    with P show ?Q by simp

```

```

  next

```

```

    assume not0:  $k \neq 0$ 

```

```

    thus ?Q

```

```

    proof (simp, intro allI impI)

```

```

      fix i j

```

```

    assume n: n = k*i + j and j: j < k
    show P i
    proof (cases)
      assume i = 0
      with n j P show P i by simp
    next
      assume i ≠ 0
      with not0 n j P show P i by (simp add: add-ac)
    qed
  qed
next
  assume Q: ?Q
  show ?P
  proof (cases)
    assume k = 0
    with Q show ?P by simp
  next
    assume not0: k ≠ 0
    with Q have R: ?R by simp
    from not0 R [THEN spec, of n div k, THEN spec, of n mod k]
    show ?P by simp
  qed
qed

lemma split-div-lemma:
  assumes 0 < n
  shows n * q ≤ m ∧ m < n * Suc q ⟷ q = ((m::nat) div n) (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs
  with mult-div-cancel have nq: n * q = m - (m mod n) by simp
  then have A: n * q ≤ m by simp
  have n - (m mod n) > 0 using mod-less-divisor assms by auto
  then have m < m + (n - (m mod n)) by simp
  then have m < n + (m - (m mod n)) by simp
  with nq have m < n + n * q by simp
  then have B: m < n * Suc q by simp
  from A B show ?lhs ..
next
  assume P: ?lhs
  then have divmod-nat-rel m n (q, m - n * q)
    unfolding divmod-nat-rel-def by (auto simp add: mult-ac)
  with divmod-nat-rel-unique divmod-nat-rel [of m n]
  have (q, m - n * q) = (m div n, m mod n) by auto
  then show ?rhs by simp
qed

theorem split-div':
  P ((m::nat) div n) = ((n = 0 ∧ P 0) ∨

```

```

  (∃ q. (n * q ≤ m ∧ m < n * (Suc q)) ∧ P q))
apply (case-tac 0 < n)
apply (simp only: add: split-div-lemma)
apply simp-all
done

lemma split-mod:
  P(n mod k :: nat) =
  ((k = 0 → P n) ∧ (k ≠ 0 → (!i. !j < k. n = k*i + j → P j)))
  (is ?P = ?Q is - = (- ∧ (- → ?R)))
proof
  assume P: ?P
  show ?Q
  proof (cases)
    assume k = 0
    with P show ?Q by simp
  next
    assume not0: k ≠ 0
    thus ?Q
    proof (simp, intro allI impI)
      fix i j
      assume n = k*i + j j < k
      thus P j using not0 P by (simp add: add-ac mult-ac)
    qed
  qed
next
  assume Q: ?Q
  show ?P
  proof (cases)
    assume k = 0
    with Q show ?P by simp
  next
    assume not0: k ≠ 0
    with Q have R: ?R by simp
    from not0 R [THEN spec, of n div k, THEN spec, of n mod k]
    show ?P by simp
  qed
qed

theorem mod-div-equality': (m::nat) mod n = m - (m div n) * n
apply (rule-tac P=%x. m mod n = x - (m div n) * n in
  subst [OF mod-div-equality [of - n]])
apply arith
done

lemma div-mod-equality':
  fixes m n :: nat
  shows m div n * n = m - m mod n
proof -

```

```

have  $m \bmod n \leq m \bmod n$  ..
from div-mod-equality have
   $m \operatorname{div} n * n + m \bmod n - m \bmod n = m - m \bmod n$  by simp
with diff-add-assoc [OF  $\langle m \bmod n \leq m \bmod n \rangle$ , of  $m \operatorname{div} n * n$ ] have
   $m \operatorname{div} n * n + (m \bmod n - m \bmod n) = m - m \bmod n$ 
  by simp
then show ?thesis by simp
qed

```

### 33.3.5 An “induction” law for modulus arithmetic.

```

lemma mod-induct-0:
  assumes step:  $\forall i < p. P\ i \longrightarrow P\ ((\operatorname{Suc}\ i) \bmod p)$ 
  and base:  $P\ i$  and  $i: i < p$ 
  shows  $P\ 0$ 
proof (rule ccontr)
  assume contra:  $\neg(P\ 0)$ 
  from  $i$  have  $p: 0 < p$  by simp
  have  $\forall k. 0 < k \longrightarrow \neg P\ (p-k)$  (is  $\forall k. ?A\ k$ )
  proof
    fix  $k$ 
    show  $?A\ k$ 
    proof (induct k)
      show  $?A\ 0$  by simp — by contradiction
    next
      fix  $n$ 
      assume ih:  $?A\ n$ 
      show  $?A\ (\operatorname{Suc}\ n)$ 
      proof (clarsimp)
        assume  $y: P\ (p - \operatorname{Suc}\ n)$ 
        have  $n: \operatorname{Suc}\ n < p$ 
        proof (rule ccontr)
          assume  $\neg(\operatorname{Suc}\ n < p)$ 
          hence  $p - \operatorname{Suc}\ n = 0$ 
            by simp
          with  $y$  contra show False
            by simp
        qed
        hence  $n2: \operatorname{Suc}\ (p - \operatorname{Suc}\ n) = p - n$  by arith
        from  $p$  have  $p - \operatorname{Suc}\ n < p$  by arith
        with  $y$  step have  $z: P\ ((\operatorname{Suc}\ (p - \operatorname{Suc}\ n)) \bmod p)$ 
          by blast
        show False
        proof (cases n=0)
          case True
            with  $z\ n2$  contra show ?thesis by simp
          next
            case False
              with  $p$  have  $p - n < p$  by arith

```



```

      with  $z \text{ n2 } \textit{False}$   $ih$  show  $?thesis$  by simp
    qed
  qed
  qed
  qed
  moreover
  from  $i$  obtain  $k$  where  $0 < k \wedge i + k = p$ 
    by (blast dest: less-imp-add-positive)
  hence  $0 < k \wedge i = p - k$  by auto
  moreover
  note base
  ultimately
  show  $\textit{False}$  by blast
qed

```

```

lemma mod-induct:
  assumes step:  $\forall i < p. P\ i \longrightarrow P\ ((\textit{Suc}\ i) \bmod p)$ 
  and base:  $P\ i$  and  $i: i < p$  and  $j: j < p$ 
  shows  $P\ j$ 
proof -
  have  $\forall j < p. P\ j$ 
  proof
    fix  $j$ 
    show  $j < p \longrightarrow P\ j$  (is  $?A\ j$ )
    proof (induct j)
      from step base i show  $?A\ 0$ 
        by (auto elim: mod-induct-0)
    next
      fix  $k$ 
      assume  $ih: ?A\ k$ 
      show  $?A\ (\textit{Suc}\ k)$ 
      proof
        assume suc:  $\textit{Suc}\ k < p$ 
        hence  $k: k < p$  by simp
        with  $ih$  have  $P\ k$  ..
        with step k have  $P\ (\textit{Suc}\ k \bmod p)$ 
          by blast
        moreover
        from suc have  $\textit{Suc}\ k \bmod p = \textit{Suc}\ k$ 
          by simp
        ultimately
        show  $P\ (\textit{Suc}\ k)$  by simp
      qed
    qed
  qed
  with  $j$  show  $?thesis$  by blast
qed

```

```

lemma div2-Suc-Suc [simp]:  $\textit{Suc}\ (\textit{Suc}\ m) \textit{div}\ 2 = \textit{Suc}\ (m \textit{div}\ 2)$ 

```

**by** (*auto simp add: numeral-2-eq-2 le-div-geq*)

**lemma** *add-self-div-2* [*simp*]:  $(m + m) \text{ div } 2 = (m :: \text{nat})$   
**by** (*simp add: nat-mult-2 [symmetric]*)

**lemma** *mod2-Suc-Suc* [*simp*]:  $\text{Suc}(\text{Suc}(m)) \text{ mod } 2 = m \text{ mod } 2$   
**apply** (*subgoal-tac m mod 2 < 2*)  
**apply** (*erule less-2-cases [THEN disjE]*)  
**apply** (*simp-all (no-asm-simp) add: Let-def mod-Suc*)  
**done**

**lemma** *mod2-gr-0* [*simp*]:  $0 < (m :: \text{nat}) \text{ mod } 2 \longleftrightarrow m \text{ mod } 2 = 1$   
**proof** –  
 { **fix**  $n :: \text{nat}$  **have**  $(n :: \text{nat}) < 2 \implies n = 0 \vee n = 1$  **by** (*cases n*) *simp-all* }  
**moreover** **have**  $m \text{ mod } 2 < 2$  **by** *simp*  
**ultimately** **have**  $m \text{ mod } 2 = 0 \vee m \text{ mod } 2 = 1$  .  
**then show** *?thesis* **by** *auto*  
**qed**

These lemmas collapse some needless occurrences of *Suc*: at least three *Sucs*, since two and fewer are rewritten back to *Suc* again! We already have some rules to simplify operands smaller than 3.

**lemma** *div-Suc-eq-div-add3* [*simp*]:  $m \text{ div } (\text{Suc } (\text{Suc } (\text{Suc } n))) = m \text{ div } (3+n)$   
**by** (*simp add: Suc3-eq-add-3*)

**lemma** *mod-Suc-eq-mod-add3* [*simp*]:  $m \text{ mod } (\text{Suc } (\text{Suc } (\text{Suc } n))) = m \text{ mod } (3+n)$   
**by** (*simp add: Suc3-eq-add-3*)

**lemma** *Suc-div-eq-add3-div*:  $(\text{Suc } (\text{Suc } (\text{Suc } m))) \text{ div } n = (3+m) \text{ div } n$   
**by** (*simp add: Suc3-eq-add-3*)

**lemma** *Suc-mod-eq-add3-mod*:  $(\text{Suc } (\text{Suc } (\text{Suc } m))) \text{ mod } n = (3+m) \text{ mod } n$   
**by** (*simp add: Suc3-eq-add-3*)

**lemmas** *Suc-div-eq-add3-div-number-of* =  
     *Suc-div-eq-add3-div* [*of - number-of v, standard*]  
**declare** *Suc-div-eq-add3-div-number-of* [*simp*]

**lemmas** *Suc-mod-eq-add3-mod-number-of* =  
     *Suc-mod-eq-add3-mod* [*of - number-of v, standard*]  
**declare** *Suc-mod-eq-add3-mod-number-of* [*simp*]

**lemma** *Suc-times-mod-eq*:  $1 < k \implies \text{Suc } (k * m) \text{ mod } k = 1$   
**apply** (*induct m*)  
**apply** (*simp-all add: mod-Suc*)  
**done**

**declare** *Suc-times-mod-eq* [*of number-of w, standard, simp*]

**lemma** [simp]:  $n \text{ div } k \leq (\text{Suc } n) \text{ div } k$   
**by** (simp add: div-le-mono)

**lemma** Suc-n-div-2-gt-zero [simp]:  $(0::\text{nat}) < n \implies 0 < (n + 1) \text{ div } 2$   
**by** (cases n) simp-all

**lemma** div-2-gt-zero [simp]: **assumes**  $A: (1::\text{nat}) < n$  **shows**  $0 < n \text{ div } 2$   
**proof** –  
**from**  $A$  **have**  $B: 0 < n - 1$  **and**  $C: n - 1 + 1 = n$  **by** simp-all  
**from** Suc-n-div-2-gt-zero [OF  $B$ ]  $C$  **show** ?thesis **by** simp  
**qed**

**lemma** mod-mult-self3 [simp]:  $(k * n + m) \text{ mod } n = m \text{ mod } (n::\text{nat})$   
**by** (simp add: mult-ac add-ac)

**lemma** mod-mult-self4 [simp]:  $\text{Suc } (k * n + m) \text{ mod } n = \text{Suc } m \text{ mod } n$   
**proof** –  
**have**  $\text{Suc } (k * n + m) \text{ mod } n = (k * n + \text{Suc } m) \text{ mod } n$  **by** simp  
**also have**  $\dots = \text{Suc } m \text{ mod } n$  **by** (rule mod-mult-self3)  
**finally show** ?thesis .  
**qed**

**lemma** mod-Suc-eq-Suc-mod:  $\text{Suc } m \text{ mod } n = \text{Suc } (m \text{ mod } n) \text{ mod } n$   
**apply** (subst mod-Suc [of  $m$ ])  
**apply** (subst mod-Suc [of  $m \text{ mod } n$ ], simp)  
**done**

### 33.4 Division on *int*

**definition** divmod-int-rel ::  $\text{int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{int} \Rightarrow \text{bool}$  **where**  
 — definition of quotient and remainder  
 [code]:  $\text{divmod-int-rel } a \ b = (\lambda(q, r). a = b * q + r \wedge$   
 $(\text{if } 0 < b \text{ then } 0 \leq r \wedge r < b \text{ else } b < r \wedge r \leq 0))$

**definition** adjust ::  $\text{int} \Rightarrow \text{int} \times \text{int} \Rightarrow \text{int} \times \text{int}$  **where**  
 — for the division algorithm  
 [code]:  $\text{adjust } b = (\lambda(q, r). \text{if } 0 \leq r - b \text{ then } (2 * q + 1, r - b)$   
 $\text{else } (2 * q, r))$

algorithm for the case  $a \geq 0, b > 0$

**function** posDivAlg ::  $\text{int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{int}$  **where**  
 $\text{posDivAlg } a \ b = (\text{if } a < b \vee b \leq 0 \text{ then } (0, a)$   
 $\text{else adjust } b (\text{posDivAlg } a \ (2 * b)))$   
**by** auto  
**termination by** (relation measure  $(\lambda(a, b). \text{nat } (a - b + 1)))$   
 $(\text{auto simp add: mult-2})$

algorithm for the case  $a < 0, b > 0$

**function** *negDivAlg* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\times$  *int* **where**  
     *negDivAlg* *a* *b* = (if  $0 \leq a + b \vee b \leq 0$  then  $(-1, a + b)$   
     else *adjust* *b* (*negDivAlg* *a* ( $2 * b$ )))  
**by** *auto*  
**termination by** (*relation measure* ( $\lambda(a, b). \text{nat } (-a - b)$ ))  
     (*auto simp add: mult-2*)

algorithm for the general case  $b \neq (0::'a)$

**definition** *negateSnd* :: *int*  $\times$  *int*  $\Rightarrow$  *int*  $\times$  *int* **where**  
     [*code-unfold*]: *negateSnd* = *apsnd uminus*

**definition** *divmod-int* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\times$  *int* **where**  
     — The full division algorithm considers all possible signs for *a*, *b* including the  
     special case  $a=0, b<0$  because *negDivAlg* requires  $a < (0::'a)$ .  
     *divmod-int* *a* *b* = (if  $0 \leq a$  then if  $0 \leq b$  then *posDivAlg* *a* *b*  
     else if  $a = 0$  then  $(0, 0)$   
     else *negateSnd* (*negDivAlg*  $(-a)$   $(-b)$ )  
     else  
     if  $0 < b$  then *negDivAlg* *a* *b*  
     else *negateSnd* (*posDivAlg*  $(-a)$   $(-b)$ ))

**instantiation** *int* :: *Divides.div*  
**begin**

**definition**  
     *a div b* = *fst* (*divmod-int* *a* *b*)

**definition**  
     *a mod b* = *snd* (*divmod-int* *a* *b*)

**instance** ..

**end**

**lemma** *divmod-int-mod-div*:  
     *divmod-int* *p* *q* = (*p div q*, *p mod q*)  
**by** (*auto simp add: div-int-def mod-int-def*)

Here is the division algorithm in ML:

```
fun posDivAlg (a,b) =
  if a<b then (0,a)
  else let val (q,r) = posDivAlg(a, 2*b)
        in if 0<le>r-b then (2*q+1, r-b) else (2*q, r)
        end

fun negDivAlg (a,b) =
  if 0<le>a+b then (~1,a+b)
```

```

    else let val (q,r) = negDivAlg(a, 2*b)
        in if 0 < r-b then (2*q+1, r-b) else (2*q, r)
        end;

fun negateSnd (q,r:int) = (q,~r);

fun divmod (a,b) = if 0 < a then
    if b>0 then posDivAlg (a,b)
    else if a=0 then (0,0)
    else negateSnd (negDivAlg (~a,~b))
else
    if 0 < b then negDivAlg (a,b)
    else negateSnd (posDivAlg (~a,~b));

```

### 33.4.1 Uniqueness and Monotonicity of Quotients and Remainders

**lemma** *unique-quotient-lemma*:

$[| b*q' + r' \leq b*q + r; \ 0 \leq r'; \ r' < b; \ r < b |]$   
 $\implies q' \leq (q::int)$

**apply** (*subgoal-tac*  $r' + b * (q' - q) \leq r$ )  
**prefer** 2 **apply** (*simp add: right-diff-distrib*)  
**apply** (*subgoal-tac*  $0 < b * (1 + q - q')$ )  
**apply** (*erule-tac* [2] *order-le-less-trans*)  
**prefer** 2 **apply** (*simp add: right-diff-distrib right-distrib*)  
**apply** (*subgoal-tac*  $b * q' < b * (1 + q)$ )  
**prefer** 2 **apply** (*simp add: right-diff-distrib right-distrib*)  
**apply** (*simp add: mult-less-cancel-left*)  
**done**

**lemma** *unique-quotient-lemma-neg*:

$[| b*q' + r' \leq b*q + r; \ r \leq 0; \ b < r; \ b < r' |]$   
 $\implies q \leq (q'::int)$

**by** (*rule-tac*  $b = -b$  **and**  $r = -r'$  **and**  $r' = -r$  **in** *unique-quotient-lemma*,  
*auto*)

**lemma** *unique-quotient*:

$[| \text{divmod-int-rel } a \ b \ (q, r); \ \text{divmod-int-rel } a \ b \ (q', r'); \ b \neq 0 |]$   
 $\implies q = q'$

**apply** (*simp add: divmod-int-rel-def linorder-neq-iff split: split-if-asm*)  
**apply** (*blast intro: order-antisym*  
*dest: order-eq-refl [THEN unique-quotient-lemma]*  
*order-eq-refl [THEN unique-quotient-lemma-neg] sym*)  
**done**

**lemma** *unique-remainder*:

```

    [| divmod-int-rel a b (q, r); divmod-int-rel a b (q', r'); b ≠ 0 |]
    ==> r = r'
  apply (subgoal-tac q = q')
  apply (simp add: divmod-int-rel-def)
  apply (blast intro: unique-quotient)
done

```

### 33.4.2 Correctness of *posDivAlg*, the Algorithm for Non-Negative Dividends

And positive divisors

```

lemma adjust-eq [simp]:
  adjust b (q,r) =
    (let diff = r-b in
     if 0 ≤ diff then (2*q + 1, diff)
     else (2*q, r))
by (simp add: Let-def adjust-def)

declare posDivAlg.simps [simp del]

use with a simproc to avoid repeatedly proving the premise

lemma posDivAlg-eqn:
  0 < b ==>
    posDivAlg a b = (if a < b then (0,a) else adjust b (posDivAlg a (2*b)))
by (rule posDivAlg.simps [THEN trans], simp)

```

Correctness of *posDivAlg*: it computes quotients correctly

```

theorem posDivAlg-correct:
  assumes 0 ≤ a and 0 < b
  shows divmod-int-rel a b (posDivAlg a b)
using prems apply (induct a b rule: posDivAlg.induct)
apply auto
apply (simp add: divmod-int-rel-def)
apply (subst posDivAlg-eqn, simp add: right-distrib)
apply (case-tac a < b)
apply simp-all
apply (erule splitE)
apply (auto simp add: right-distrib Let-def mult-ac mult-2-right)
done

```

### 33.4.3 Correctness of *negDivAlg*, the Algorithm for Negative Dividends

And positive divisors

```

declare negDivAlg.simps [simp del]

use with a simproc to avoid repeatedly proving the premise

```

**lemma** *negDivAlg-eqn*:  
 $0 < b \implies$   
 $\text{negDivAlg } a \ b =$   
 $(\text{if } 0 \leq a+b \text{ then } (-1, a+b) \text{ else adjust } b (\text{negDivAlg } a \ (2*b)))$   
**by** (rule *negDivAlg.simps* [THEN trans], simp)

**lemma** *negDivAlg-correct*:  
**assumes**  $a < 0$  **and**  $b > 0$   
**shows** *divmod-int-rel*  $a \ b$  (*negDivAlg*  $a \ b$ )  
**using** *prems* **apply** (*induct*  $a \ b$  rule: *negDivAlg.induct*)  
**apply** (*auto* simp add: *linorder-not-le*)  
**apply** (simp add: *divmod-int-rel-def*)  
**apply** (subst *negDivAlg-eqn*, assumption)  
**apply** (case-tac  $a + b < (0::\text{int})$ )  
**apply** *simp-all*  
**apply** (*erule* *splitE*)  
**apply** (*auto* simp add: *right-distrib* *Let-def* *mult-ac* *mult-2-right*)  
**done**

### 33.4.4 Existence Shown by Proving the Division Algorithm to be Correct

**lemma** *divmod-int-rel-0*:  $b \neq 0 \implies \text{divmod-int-rel } 0 \ b \ (0, 0)$   
**by** (*auto* simp add: *divmod-int-rel-def* *linorder-neq-iff*)

**lemma** *posDivAlg-0* [*simp*]:  $\text{posDivAlg } 0 \ b = (0, 0)$   
**by** (subst *posDivAlg.simps*, *auto*)

**lemma** *negDivAlg-minus1* [*simp*]:  $\text{negDivAlg } -1 \ b = (-1, b - 1)$   
**by** (subst *negDivAlg.simps*, *auto*)

**lemma** *negateSnd-eq* [*simp*]:  $\text{negateSnd}(q, r) = (q, -r)$   
**by** (simp add: *negateSnd-def*)

**lemma** *divmod-int-rel-neg*:  $\text{divmod-int-rel } (-a) \ (-b) \ qr \implies \text{divmod-int-rel } a \ b$   
 $(\text{negateSnd } qr)$   
**by** (*auto* simp add: *split-ifs* *divmod-int-rel-def*)

**lemma** *divmod-int-correct*:  $b \neq 0 \implies \text{divmod-int-rel } a \ b \ (\text{divmod-int } a \ b)$   
**by** (force simp add: *linorder-neq-iff* *divmod-int-rel-0* *divmod-int-def* *divmod-int-rel-neg*  
*posDivAlg-correct* *negDivAlg-correct*)

Arbitrary definitions for division by zero. Useful to simplify certain equations.

**lemma** *DIVISION-BY-ZERO* [*simp*]:  $a \text{ div } (0::\text{int}) = 0 \ \& \ a \text{ mod } (0::\text{int}) = a$   
**by** (simp add: *div-int-def* *mod-int-def* *divmod-int-def* *posDivAlg.simps*)

Basic laws about division and remainder

```

lemma zmod-zdiv-equality: (a::int) = b * (a div b) + (a mod b)
apply (case-tac b = 0, simp)
apply (cut-tac a = a and b = b in divmod-int-correct)
apply (auto simp add: divmod-int-rel-def div-int-def mod-int-def)
done

```

```

lemma zdiv-zmod-equality: (b * (a div b) + (a mod b)) + k = (a::int)+k
by(simp add: zmod-zdiv-equality[symmetric])

```

```

lemma zdiv-zmod-equality2: ((a div b) * b + (a mod b)) + k = (a::int)+k
by(simp add: mult-commute zmod-zdiv-equality[symmetric])

```

Tool setup

```

ML <<
local

```

```

structure CancelDivMod = CancelDivModFun(struct

  val div-name = @{const-name div};
  val mod-name = @{const-name mod};
  val mk-binop = HOLogic.mk-binop;
  val mk-sum = Arith-Data.mk-sum HOLogic.intT;
  val dest-sum = Arith-Data.dest-sum;

  val div-mod-eqs = map mk-meta-eq [@{thm zdiv-zmod-equality}, @{thm zdiv-zmod-equality2}];

  val trans = trans;

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac
    (@{thm diff-minus} :: @{thms add-0s} @ @{thms add-ac}))

end)

in

val cancel-div-mod-int-proc = Simplifier.simproc @{theory}
  cancel-zdiv-zmod [(k::int) + l] (K CancelDivMod.proc);

val - = Addsimprocs [cancel-div-mod-int-proc];

end
>>

```

```

lemma pos-mod-conj : (0::int) < b ==> 0 ≤ a mod b & a mod b < b
apply (cut-tac a = a and b = b in divmod-int-correct)
apply (auto simp add: divmod-int-rel-def mod-int-def)
done

```

```

lemmas pos-mod-sign [simp] = pos-mod-conj [THEN conjunct1, standard]

```



**and** *pos-mod-bound* [*simp*] = *pos-mod-conj* [*THEN conjunct2, standard*]

**lemma** *neg-mod-conj* :  $b < (0::int) \implies a \bmod b \leq 0 \ \& \ b < a \bmod b$   
**apply** (*cut-tac*  $a = a$  **and**  $b = b$  **in** *divmod-int-correct*)  
**apply** (*auto simp add: divmod-int-rel-def div-int-def mod-int-def*)  
**done**

**lemmas** *neg-mod-sign* [*simp*] = *neg-mod-conj* [*THEN conjunct1, standard*]  
**and** *neg-mod-bound* [*simp*] = *neg-mod-conj* [*THEN conjunct2, standard*]

### 33.4.5 General Properties of div and mod

**lemma** *divmod-int-rel-div-mod*:  $b \neq 0 \implies \text{divmod-int-rel } a \ b \ (a \text{ div } b, a \bmod b)$   
**apply** (*cut-tac*  $a = a$  **and**  $b = b$  **in** *zmod-zdiv-equality*)  
**apply** (*force simp add: divmod-int-rel-def linorder-neg-iff*)  
**done**

**lemma** *divmod-int-rel-div*:  $[\text{divmod-int-rel } a \ b \ (q, r); \ b \neq 0] \implies a \text{ div } b = q$   
**by** (*simp add: divmod-int-rel-div-mod [THEN unique-quotient]*)

**lemma** *divmod-int-rel-mod*:  $[\text{divmod-int-rel } a \ b \ (q, r); \ b \neq 0] \implies a \bmod b = r$   
**by** (*simp add: divmod-int-rel-div-mod [THEN unique-remainder]*)

**lemma** *div-pos-pos-trivial*:  $[(0::int) \leq a; \ a < b] \implies a \text{ div } b = 0$   
**apply** (*rule divmod-int-rel-div*)  
**apply** (*auto simp add: divmod-int-rel-def*)  
**done**

**lemma** *div-neg-neg-trivial*:  $[a \leq (0::int); \ b < a] \implies a \text{ div } b = 0$   
**apply** (*rule divmod-int-rel-div*)  
**apply** (*auto simp add: divmod-int-rel-def*)  
**done**

**lemma** *div-pos-neg-trivial*:  $[(0::int) < a; \ a+b \leq 0] \implies a \text{ div } b = -1$   
**apply** (*rule divmod-int-rel-div*)  
**apply** (*auto simp add: divmod-int-rel-def*)  
**done**

**lemma** *mod-pos-pos-trivial*:  $[(0::int) \leq a; \ a < b] \implies a \bmod b = a$   
**apply** (*rule-tac*  $q = 0$  **in** *divmod-int-rel-mod*)  
**apply** (*auto simp add: divmod-int-rel-def*)  
**done**

**lemma** *mod-neg-neg-trivial*:  $[a \leq (0::int); \ b < a] \implies a \bmod b = a$   
**apply** (*rule-tac*  $q = 0$  **in** *divmod-int-rel-mod*)  
**apply** (*auto simp add: divmod-int-rel-def*)

done

**lemma** *mod-pos-neg-trivial*:  $[ (0::int) < a; a+b \leq 0 ] \implies a \bmod b = a+b$   
**apply** (*rule-tac*  $q = -1$  **in** *divmod-int-rel-mod*)  
**apply** (*auto simp add: divmod-int-rel-def*)  
**done**

There is no *mod-neg-pos-trivial*.

**lemma** *zdiv-zminus-zminus* [*simp*]:  $(-a) \operatorname{div} (-b) = a \operatorname{div} (b::int)$   
**apply** (*case-tac*  $b = 0$ , *simp*)  
**apply** (*simp add: divmod-int-rel-div-mod [THEN divmod-int-rel-neg, simplified,*  
*THEN divmod-int-rel-div, THEN sym]*)

done

**lemma** *zmod-zminus-zminus* [*simp*]:  $(-a) \bmod (-b) = -(a \bmod (b::int))$   
**apply** (*case-tac*  $b = 0$ , *simp*)  
**apply** (*subst divmod-int-rel-div-mod [THEN divmod-int-rel-neg, simplified, THEN*  
*divmod-int-rel-mod],*  
*auto*)  
**done**

### 33.4.6 Laws for div and mod with Unary Minus

**lemma** *zminus1-lemma*:  
 $\operatorname{divmod-int-rel} a b (q, r)$   
 $\implies \operatorname{divmod-int-rel} (-a) b$  (*if*  $r=0$  *then*  $-q$  *else*  $-q - 1$ ,  
*if*  $r=0$  *then*  $0$  *else*  $b-r$ )  
**by** (*force simp add: split-ifs divmod-int-rel-def linorder-neq-iff right-diff-distrib*)

**lemma** *zdiv-zminus1-eq-if*:  
 $b \neq (0::int)$   
 $\implies (-a) \operatorname{div} b =$   
 $(\text{if } a \bmod b = 0 \text{ then } -(a \operatorname{div} b) \text{ else } -(a \operatorname{div} b) - 1)$   
**by** (*blast intro: divmod-int-rel-div-mod [THEN zminus1-lemma, THEN divmod-int-rel-div]*)

**lemma** *zmod-zminus1-eq-if*:  
 $(-a::int) \bmod b = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } b - (a \bmod b))$   
**apply** (*case-tac*  $b = 0$ , *simp*)  
**apply** (*blast intro: divmod-int-rel-div-mod [THEN zminus1-lemma, THEN divmod-int-rel-mod]*)  
**done**

**lemma** *zmod-zminus1-not-zero*:  
**fixes**  $k l :: int$   
**shows**  $-k \bmod l \neq 0 \implies k \bmod l \neq 0$   
**unfolding** *zmod-zminus1-eq-if* **by** *auto*

**lemma** *zdiv-zminus2*:  $a \text{ div } (-b) = (-a::\text{int}) \text{ div } b$   
**by** (*cut-tac*  $a = -a$  **in** *zdiv-zminus-zminus*, *auto*)

**lemma** *zmod-zminus2*:  $a \text{ mod } (-b) = -((-a::\text{int}) \text{ mod } b)$   
**by** (*cut-tac*  $a = -a$  **and**  $b = b$  **in** *zmod-zminus-zminus*, *auto*)

**lemma** *zdiv-zminus2-eq-if*:  
 $b \neq (0::\text{int})$   
 $\implies a \text{ div } (-b) =$   
 $(\text{if } a \text{ mod } b = 0 \text{ then } -(a \text{ div } b) \text{ else } -(a \text{ div } b) - 1)$   
**by** (*simp add*: *zdiv-zminus1-eq-if* *zdiv-zminus2*)

**lemma** *zmod-zminus2-eq-if*:  
 $a \text{ mod } (-b::\text{int}) = (\text{if } a \text{ mod } b = 0 \text{ then } 0 \text{ else } (a \text{ mod } b) - b)$   
**by** (*simp add*: *zmod-zminus1-eq-if* *zmod-zminus2*)

**lemma** *zmod-zminus2-not-zero*:  
**fixes**  $k \ l :: \text{int}$   
**shows**  $k \text{ mod } -l \neq 0 \implies k \text{ mod } l \neq 0$   
**unfolding** *zmod-zminus2-eq-if* **by** *auto*

### 33.4.7 Division of a Number by Itself

**lemma** *self-quotient-aux1*:  $[(0::\text{int}) < a; a = r + a*q; r < a] \implies 1 \leq q$   
**apply** (*subgoal-tac*  $0 < a*q$ )  
**apply** (*simp add*: *zero-less-mult-iff*, *arith*)  
**done**

**lemma** *self-quotient-aux2*:  $[(0::\text{int}) < a; a = r + a*q; 0 \leq r] \implies q \leq 1$   
**apply** (*subgoal-tac*  $0 \leq a*(1-q)$ )  
**apply** (*simp add*: *zero-le-mult-iff*)  
**apply** (*simp add*: *right-diff-distrib*)  
**done**

**lemma** *self-quotient*:  $[\text{divmod-int-rel } a \ a \ (q, r); a \neq (0::\text{int})] \implies q = 1$   
**apply** (*simp add*: *split-ifs* *divmod-int-rel-def* *linorder-neq-iff*)  
**apply** (*rule* *order-antisym*, *safe*, *simp-all*)  
**apply** (*rule-tac* [3]  $a = -a$  **and**  $r = -r$  **in** *self-quotient-aux1*)  
**apply** (*rule-tac*  $a = -a$  **and**  $r = -r$  **in** *self-quotient-aux2*)  
**apply** (*force intro*: *self-quotient-aux1* *self-quotient-aux2* *simp add*: *add-commute*)  
**done**

**lemma** *self-remainder*:  $[\text{divmod-int-rel } a \ a \ (q, r); a \neq (0::\text{int})] \implies r = 0$   
**apply** (*frule* *self-quotient*, *assumption*)  
**apply** (*simp add*: *divmod-int-rel-def*)  
**done**

**lemma** *zdiv-self* [*simp*]:  $a \neq 0 \implies a \text{ div } a = (1::\text{int})$   
**by** (*simp add*: *divmod-int-rel-div-mod* [*THEN* *self-quotient*])

```

lemma zmod-self [simp]:  $a \bmod a = (0::int)$ 
apply (case-tac  $a = 0$ , simp)
apply (simp add: divmod-int-rel-div-mod [THEN self-remainder])
done

```

### 33.4.8 Computation of Division and Remainder

```

lemma zdiv-zero [simp]:  $(0::int) \div b = 0$ 
by (simp add: div-int-def divmod-int-def)

```

```

lemma div-eq-minus1:  $(0::int) < b \implies -1 \div b = -1$ 
by (simp add: div-int-def divmod-int-def)

```

```

lemma zmod-zero [simp]:  $(0::int) \bmod b = 0$ 
by (simp add: mod-int-def divmod-int-def)

```

```

lemma zmod-minus1:  $(0::int) < b \implies -1 \bmod b = b - 1$ 
by (simp add: mod-int-def divmod-int-def)

```

a positive, b positive

```

lemma div-pos-pos:  $[| 0 < a; 0 \leq b |] \implies a \div b = \text{fst } (\text{posDivAlg } a \ b)$ 
by (simp add: div-int-def divmod-int-def)

```

```

lemma mod-pos-pos:  $[| 0 < a; 0 \leq b |] \implies a \bmod b = \text{snd } (\text{posDivAlg } a \ b)$ 
by (simp add: mod-int-def divmod-int-def)

```

a negative, b positive

```

lemma div-neg-pos:  $[| a < 0; 0 < b |] \implies a \div b = \text{fst } (\text{negDivAlg } a \ b)$ 
by (simp add: div-int-def divmod-int-def)

```

```

lemma mod-neg-pos:  $[| a < 0; 0 < b |] \implies a \bmod b = \text{snd } (\text{negDivAlg } a \ b)$ 
by (simp add: mod-int-def divmod-int-def)

```

a positive, b negative

```

lemma div-pos-neg:
   $[| 0 < a; b < 0 |] \implies a \div b = \text{fst } (\text{negateSnd } (\text{negDivAlg } (-a) \ (-b)))$ 
by (simp add: div-int-def divmod-int-def)

```

```

lemma mod-pos-neg:
   $[| 0 < a; b < 0 |] \implies a \bmod b = \text{snd } (\text{negateSnd } (\text{negDivAlg } (-a) \ (-b)))$ 
by (simp add: mod-int-def divmod-int-def)

```

a negative, b negative

```

lemma div-neg-neg:
   $[| a < 0; b < 0 |] \implies a \div b = \text{fst } (\text{negateSnd } (\text{posDivAlg } (-a) \ (-b)))$ 
by (simp add: div-int-def divmod-int-def)

```

**lemma** *mod-neg-neg*:

$\llbracket a < 0; \ b \leq 0 \rrbracket \implies a \bmod b = \text{snd } (\text{negateSnd } (\text{posDivAlg } (-a) (-b)))$

**by** (*simp add: mod-int-def divmod-int-def*)

Simplify expresions in which div and mod combine numerical constants

**lemma** *divmod-int-relI*:

$\llbracket a == b * q + r; \text{ if } 0 < b \text{ then } 0 \leq r \wedge r < b \text{ else } b < r \wedge r \leq 0 \rrbracket$

$\implies \text{divmod-int-rel } a \ b \ (q, r)$

**unfolding** *divmod-int-rel-def* **by** *simp*

**lemmas** *divmod-int-rel-div-eq = divmod-int-relI* [*THEN divmod-int-rel-div, THEN eq-reflection*]

**lemmas** *divmod-int-rel-mod-eq = divmod-int-relI* [*THEN divmod-int-rel-mod, THEN eq-reflection*]

**lemmas** *arithmetic-simps =*

*arith-simps*

*add-special*

*add-0-left*

*add-0-right*

*mult-zero-left*

*mult-zero-right*

*mult-1-left*

*mult-1-right*

**ML**  $\ll$

*local*

*val mk-number = HOLogic.mk-number HOLogic.intT;*

*fun mk-cert u k l = @{term plus :: int  $\Rightarrow$  int  $\Rightarrow$  int} \$*

*(@{term times :: int  $\Rightarrow$  int  $\Rightarrow$  int} \$ u \$ mk-number k) \$*

*mk-number l;*

*fun prove ctxt prop = Goal.prove ctxt [] [] prop*

*(K (ALLGOALS (full-simp-tac (HOL-basic-ss addsimps @{thms arithmetic-simps}))));*

*fun binary-proc proc ss ct =*

*(case Thm.term-of ct of*

*- \$ t \$ u =>*

*(case try (pairself ('(snd o HOLogic.dest-number))) (t, u) of*

*SOME args => proc (Simplifier.the-context ss) args*

*| NONE => NONE)*

*| - => NONE);*

*in*

*fun divmod-proc rule = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>*

*if n = 0 then NONE*

*else let val (k, l) = Integer.div-mod m n;*

*in SOME (rule OF [prove ctxt (Logic.mk-equals (t, mk-cert u k l))] end);*

*end*

$\gg$

**simproc-setup** *binary-int-div (number-of m div number-of n :: int) =*

⟨⟨ *K* (*divmod-proc* (@{*thm divmod-int-rel-div-eq*})) ⟩⟩

**simproc-setup** *binary-int-mod* (*number-of m mod number-of n :: int*) =  
 ⟨⟨ *K* (*divmod-proc* (@{*thm divmod-int-rel-mod-eq*})) ⟩⟩

**lemmas** *posDivAlg-eqn-number-of* [*simp*] =  
*posDivAlg-eqn* [*of number-of v number-of w, standard*]

**lemmas** *negDivAlg-eqn-number-of* [*simp*] =  
*negDivAlg-eqn* [*of number-of v number-of w, standard*]

Special-case simplification

**lemma** *zmod-minus1-right* [*simp*]: *a mod (-1::int) = 0*  
**apply** (*cut-tac a = a and b = -1 in neg-mod-sign*)  
**apply** (*cut-tac [2] a = a and b = -1 in neg-mod-bound*)  
**apply** (*auto simp del: neg-mod-sign neg-mod-bound*)  
**done**

**lemma** *zdiv-minus1-right* [*simp*]: *a div (-1::int) = -a*  
**by** (*cut-tac a = a and b = -1 in zmod-zdiv-equality, auto*)

**lemmas** *div-pos-pos-1-number-of* [*simp*] =  
*div-pos-pos* [*OF int-0-less-1, of number-of w, standard*]

**lemmas** *div-pos-neg-1-number-of* [*simp*] =  
*div-pos-neg* [*OF int-0-less-1, of number-of w, standard*]

**lemmas** *mod-pos-pos-1-number-of* [*simp*] =  
*mod-pos-pos* [*OF int-0-less-1, of number-of w, standard*]

**lemmas** *mod-pos-neg-1-number-of* [*simp*] =  
*mod-pos-neg* [*OF int-0-less-1, of number-of w, standard*]

**lemmas** *posDivAlg-eqn-1-number-of* [*simp*] =  
*posDivAlg-eqn* [*of concl: 1 number-of w, standard*]

**lemmas** *negDivAlg-eqn-1-number-of* [*simp*] =  
*negDivAlg-eqn* [*of concl: 1 number-of w, standard*]

### 33.4.9 Monotonicity in the First Argument (Dividend)

**lemma** *zdiv-mono1*: [*a ≤ a'; 0 < (b::int)*] ==> *a div b ≤ a' div b*  
**apply** (*cut-tac a = a and b = b in zmod-zdiv-equality*)  
**apply** (*cut-tac a = a' and b = b in zmod-zdiv-equality*)  
**apply** (*rule unique-quotient-lemma*)  
**apply** (*erule subst*)

```

apply (erule subst, simp-all)
done

```

```

lemma zdiv-mono1-neg: [| a ≤ a'; (b::int) < 0 |] ==> a' div b ≤ a div b
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a' and b = b in zmod-zdiv-equality)
apply (rule unique-quotient-lemma-neg)
apply (erule subst)
apply (erule subst, simp-all)
done

```

### 33.4.10 Monotonicity in the Second Argument (Divisor)

```

lemma q-pos-lemma:
  [| 0 ≤ b'*q' + r'; r' < b'; 0 < b' |] ==> 0 ≤ (q'::int)
apply (subgoal-tac 0 < b'* (q' + 1) )
  apply (simp add: zero-less-mult-iff)
apply (simp add: right-distrib)
done

```

```

lemma zdiv-mono2-lemma:
  [| b*q + r = b'*q' + r'; 0 ≤ b'*q' + r';
    r' < b'; 0 ≤ r; 0 < b'; b' ≤ b |]
  ==> q ≤ (q'::int)
apply (frule q-pos-lemma, assumption+)
apply (subgoal-tac b*q < b* (q' + 1) )
  apply (simp add: mult-less-cancel-left)
apply (subgoal-tac b*q = r' - r + b'*q')
  prefer 2 apply simp
apply (simp (no-asm-simp) add: right-distrib)
apply (subst add-commute, rule zadd-zless-mono, arith)
apply (rule mult-right-mono, auto)
done

```

```

lemma zdiv-mono2:
  [| (0::int) ≤ a; 0 < b'; b' ≤ b |] ==> a div b ≤ a div b'
apply (subgoal-tac b ≠ 0)
  prefer 2 apply arith
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a and b = b' in zmod-zdiv-equality)
apply (rule zdiv-mono2-lemma)
apply (erule subst)
apply (erule subst, simp-all)
done

```

```

lemma q-neg-lemma:
  [| b'*q' + r' < 0; 0 ≤ r'; 0 < b' |] ==> q' ≤ (0::int)
apply (subgoal-tac b'*q' < 0)
  apply (simp add: mult-less-0-iff, arith)

```

done

**lemma** *zdiv-mono2-neg-lemma*:

$[[\ b * q + r = b' * q' + r'; \ b' * q' + r' < 0;$   
 $\quad r < b; \ 0 \leq r'; \ 0 < b'; \ b' \leq b \ ]]$   
 $\implies q' \leq (q :: \text{int})$

**apply** (*frule* *q-neg-lemma*, *assumption* +)

**apply** (*subgoal-tac*  $b * q' < b * (q + 1)$  )

**apply** (*simp* *add*: *mult-less-cancel-left*)

**apply** (*simp* *add*: *right-distrib*)

**apply** (*subgoal-tac*  $b * q' \leq b' * q'$ )

**prefer** 2 **apply** (*simp* *add*: *mult-right-mono-neg*, *arith*)

done

**lemma** *zdiv-mono2-neg*:

$[[\ a < (0 :: \text{int}); \ 0 < b'; \ b' \leq b \ ]] \implies a \text{ div } b' \leq a \text{ div } b$

**apply** (*cut-tac*  $a = a$  **and**  $b = b'$  **in** *zmod-zdiv-equality*)

**apply** (*cut-tac*  $a = a$  **and**  $b = b'$  **in** *zmod-zdiv-equality*)

**apply** (*rule* *zdiv-mono2-neg-lemma*)

**apply** (*erule* *subst*)

**apply** (*erule* *subst*, *simp-all*)

done

### 33.4.11 More Algebraic Laws for div and mod

proving  $(a * b) \text{ div } c = a * (b \text{ div } c) + a * (b \text{ mod } c)$

**lemma** *zmult1-lemma*:

$[[\ \text{divmod-int-rel } b \ c \ (q, r); \ c \neq 0 \ ]]$   
 $\implies \text{divmod-int-rel } (a * b) \ c \ (a * q + a * r \text{ div } c, a * r \text{ mod } c)$

**by** (*auto* *simp* *add*: *split-ifs* *divmod-int-rel-def* *linorder-neg-iff* *right-distrib* *mult-ac*)

**lemma** *zdiv-zmult1-eq*:  $(a * b) \text{ div } c = a * (b \text{ div } c) + a * (b \text{ mod } c) \text{ div } (c :: \text{int})$

**apply** (*case-tac*  $c = 0$ , *simp*)

**apply** (*blast* *intro*: *divmod-int-rel-div-mod* [*THEN* *zmult1-lemma*, *THEN* *divmod-int-rel-div*])

done

**lemma** *zmod-zmult1-eq*:  $(a * b) \text{ mod } c = a * (b \text{ mod } c) \text{ mod } (c :: \text{int})$

**apply** (*case-tac*  $c = 0$ , *simp*)

**apply** (*blast* *intro*: *divmod-int-rel-div-mod* [*THEN* *zmult1-lemma*, *THEN* *divmod-int-rel-mod*])

done

**lemma** *zmod-zdiv-trivial*:  $(a \text{ mod } b) \text{ div } b = (0 :: \text{int})$

**apply** (*case-tac*  $b = 0$ , *simp*)

**apply** (*auto* *simp* *add*: *linorder-neg-iff* *div-pos-pos-trivial* *div-neg-neg-trivial*)

done

proving  $(a + b) \text{ div } c = a \text{ div } c + b \text{ div } c + ((a \text{ mod } c + b \text{ mod } c) \text{ div } c)$

**lemma** *zadd1-lemma*:

$[[\ \text{divmod-int-rel } a \ c \ (aq, ar); \ \text{divmod-int-rel } b \ c \ (bq, br); \ c \neq 0 \ ]]$



$\Rightarrow \text{divmod-int-rel } (a+b) \ c \ (aq + bq + (ar+br) \text{ div } c, (ar+br) \text{ mod } c)$   
**by** (*force simp add: split-ifs divmod-int-rel-def linorder-neq-iff right-distrib*)

**lemma** *zdiv-zadd1-eq*:

$(a+b) \text{ div } (c::\text{int}) = a \text{ div } c + b \text{ div } c + ((a \text{ mod } c + b \text{ mod } c) \text{ div } c)$

**apply** (*case-tac c = 0, simp*)

**apply** (*blast intro: zadd1-lemma [OF divmod-int-rel-div-mod divmod-int-rel-div-mod] divmod-int-rel-div*)

**done**

**instance** *int :: ring-div*

**proof**

**fix** *a b c :: int*

**assume** *not0: b ≠ 0*

**show**  $(a + c * b) \text{ div } b = c + a \text{ div } b$

**unfolding** *zdiv-zadd1-eq [of a c \* b] using not0*

**by** (*simp add: zmod-zmult1-eq zmod-zdiv-trivial zdiv-zmult1-eq*)

**next**

**fix** *a b c :: int*

**assume** *a ≠ 0*

**then show**  $(a * b) \text{ div } (a * c) = b \text{ div } c$

**proof** (*cases b ≠ 0 ∧ c ≠ 0*)

**case** *False* **then show** *?thesis* **by** *auto*

**next**

**case** *True* **then have** *b ≠ 0* **and** *c ≠ 0* **by** *auto*

**with**  $\langle a \neq 0 \rangle$

**have**  $\bigwedge q \ r. \text{divmod-int-rel } b \ c \ (q, r) \Rightarrow \text{divmod-int-rel } (a * b) \ (a * c) \ (q, a *$

*r)*

**apply** (*auto simp add: divmod-int-rel-def*)

**apply** (*auto simp add: algebra-simps*)

**apply** (*auto simp add: zero-less-mult-iff zero-le-mult-iff mult-le-0-iff mult-commute [of a] mult-less-cancel-right*)

**done**

**moreover with**  $\langle c \neq 0 \rangle$  *divmod-int-rel-div-mod* **have** *divmod-int-rel* *b c (b div c, b mod c)* **by** *auto*

**ultimately have** *divmod-int-rel*  $(a * b) \ (a * c) \ (b \text{ div } c, a * (b \text{ mod } c))$  .

**moreover from**  $\langle a \neq 0 \rangle \langle c \neq 0 \rangle$  **have**  $a * c \neq 0$  **by** *simp*

**ultimately show** *?thesis* **by** (*rule divmod-int-rel-div*)

**qed**

**qed** *auto*

**lemma** *posDivAlg-div-mod*:

**assumes**  $k \geq 0$

**and**  $l \geq 0$

**shows**  $\text{posDivAlg } k \ l = (k \text{ div } l, k \text{ mod } l)$

**proof** (*cases l = 0*)

**case** *True* **then show** *?thesis* **by** (*simp add: posDivAlg.simps*)

**next**

```

case False with assms posDivAlg-correct
  have divmod-int-rel k l (fst (posDivAlg k l), snd (posDivAlg k l))
  by simp
  from divmod-int-rel-div [OF this  $\langle l \neq 0 \rangle$ ] divmod-int-rel-mod [OF this  $\langle l \neq 0 \rangle$ ]
  show ?thesis by simp
qed

```

```

lemma negDivAlg-div-mod:
  assumes  $k < 0$ 
  and  $l > 0$ 
  shows negDivAlg k l = (k div l, k mod l)
proof –
  from assms have  $l \neq 0$  by simp
  from assms negDivAlg-correct
    have divmod-int-rel k l (fst (negDivAlg k l), snd (negDivAlg k l))
    by simp
  from divmod-int-rel-div [OF this  $\langle l \neq 0 \rangle$ ] divmod-int-rel-mod [OF this  $\langle l \neq 0 \rangle$ ]
  show ?thesis by simp
qed

```

```

lemma zmod-eq-0-iff: ( $m \bmod d = 0$ ) = (EX q::int.  $m = d * q$ )
by (simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)

```

```

lemmas zmod-eq-0D [dest!] = zmod-eq-0-iff [THEN iffD1]

```

### 33.4.12 Proving $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$

first, four lemmas to bound the remainder for the cases  $b \nmid 0$  and  $b \mid 0$

```

lemma zmult2-lemma-aux1: [ $(0::\text{int}) < c$ ;  $b < r$ ;  $r \leq 0$ ] ==>  $b * c < b * (q \bmod c) + r$ 
apply (subgoal-tac  $b * (c - q \bmod c) < r * 1$ )
apply (simp add: algebra-simps)
apply (rule order-le-less-trans)
apply (erule-tac [2] mult-strict-right-mono)
apply (rule mult-left-mono-neg)
using add1-zle-eq[of  $q \bmod c$ ]apply(simp add: algebra-simps)
apply (simp)
apply (simp)
done

```

```

lemma zmult2-lemma-aux2:
  [ $(0::\text{int}) < c$ ;  $b < r$ ;  $r \leq 0$ ] ==>  $b * (q \bmod c) + r \leq 0$ 
apply (subgoal-tac  $b * (q \bmod c) \leq 0$ )
apply arith
apply (simp add: mult-le-0-iff)
done

```

```

lemma zmult2-lemma-aux3: [ $(0::\text{int}) < c$ ;  $0 \leq r$ ;  $r < b$ ] ==>  $0 \leq b * (q \bmod c) + r$ 

```

```

mod c) + r
apply (subgoal-tac  $0 \leq b * (q \text{ mod } c)$  )
apply arith
apply (simp add: zero-le-mult-iff)
done

```

```

lemma zmult2-lemma-aux4: [ $(0::\text{int}) < c; 0 \leq r; r < b$ ]  $\implies b * (q \text{ mod } c)$ 
+  $r < b * c$ 
apply (subgoal-tac  $r * 1 < b * (c - q \text{ mod } c)$  )
apply (simp add: right-diff-distrib)
apply (rule order-less-le-trans)
apply (erule mult-strict-right-mono)
apply (rule-tac [2] mult-left-mono)
apply simp
using add1-zle-eq[of  $q \text{ mod } c$ ] apply (simp add: algebra-simps)
apply simp
done

```

```

lemma zmult2-lemma: [ $\text{divmod-int-rel } a \ b \ (q, r); b \neq 0; 0 < c$ ]
 $\implies \text{divmod-int-rel } a \ (b * c) \ (q \text{ div } c, b*(q \text{ mod } c) + r)$ 
by (auto simp add: mult-ac divmod-int-rel-def linorder-neq-iff
zero-less-mult-iff right-distrib [symmetric]
zmult2-lemma-aux1 zmult2-lemma-aux2 zmult2-lemma-aux3
zmult2-lemma-aux4)

```

```

lemma zdiv-zmult2-eq:  $(0::\text{int}) < c \implies a \text{ div } (b*c) = (a \text{ div } b) \text{ div } c$ 
apply (case-tac  $b = 0$ , simp)
apply (force simp add: divmod-int-rel-div-mod [THEN zmult2-lemma, THEN divmod-int-rel-div])
done

```

```

lemma zmod-zmult2-eq:
 $(0::\text{int}) < c \implies a \text{ mod } (b*c) = b*(a \text{ div } b \text{ mod } c) + a \text{ mod } b$ 
apply (case-tac  $b = 0$ , simp)
apply (force simp add: divmod-int-rel-div-mod [THEN zmult2-lemma, THEN divmod-int-rel-mod])
done

```

### 33.4.13 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

```

lemma split-pos-lemma:
 $0 < k \implies$ 
 $P(n \text{ div } k :: \text{int})(n \text{ mod } k) = (\forall i \ j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P \ i \ j)$ 
apply (rule iffI, clarify)
apply (erule-tac  $P = P \ ?x \ ?y$  in rev-mp)
apply (subst mod-add-eq)
apply (subst zdiv-zadd1-eq)
apply (simp add: div-pos-pos-trivial mod-pos-pos-trivial)

```

converse direction

```

apply (drule-tac  $x = n \text{ div } k$  in spec)
apply (drule-tac  $x = n \text{ mod } k$  in spec, simp)
done

```

**lemma** *split-neg-lemma*:

```

 $k < 0 ==>$ 
   $P(n \text{ div } k :: \text{int})(n \text{ mod } k) = (\forall i j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P \ i \ j)$ 
apply (rule iffI, clarify)
apply (erule-tac  $P = P \ ?x \ ?y$  in rev-mp)
apply (subst mod-add-eq)
apply (subst zdiv-zadd1-eq)
apply (simp add: div-neg-neg-trivial mod-neg-neg-trivial)

```

converse direction

```

apply (drule-tac  $x = n \text{ div } k$  in spec)
apply (drule-tac  $x = n \text{ mod } k$  in spec, simp)
done

```

**lemma** *split-zdiv*:

```

 $P(n \text{ div } k :: \text{int}) =$ 
   $((k = 0 \longrightarrow P \ 0) \ \&$ 
     $(0 < k \longrightarrow (\forall i j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P \ i)) \ \&$ 
     $(k < 0 \longrightarrow (\forall i j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P \ i)))$ 
apply (case-tac  $k=0$ , simp)
apply (simp only: linorder-neq-iff)
apply (erule disjE)
apply (simp-all add: split-pos-lemma [of concl:  $\%x \ y. P \ x$ ]
  split-neg-lemma [of concl:  $\%x \ y. P \ x$ ])
done

```

**lemma** *split-zmod*:

```

 $P(n \text{ mod } k :: \text{int}) =$ 
   $((k = 0 \longrightarrow P \ n) \ \&$ 
     $(0 < k \longrightarrow (\forall i j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P \ j)) \ \&$ 
     $(k < 0 \longrightarrow (\forall i j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P \ j)))$ 
apply (case-tac  $k=0$ , simp)
apply (simp only: linorder-neq-iff)
apply (erule disjE)
apply (simp-all add: split-pos-lemma [of concl:  $\%x \ y. P \ y$ ]
  split-neg-lemma [of concl:  $\%x \ y. P \ y$ ])
done

```

Enable (lin)arith to deal with *op div* and *op mod* when these are applied to some constant that is of the form *number-of k*:

```

declare split-zdiv [of - - number-of k, standard, arith-split]
declare split-zmod [of - - number-of k, standard, arith-split]

```

### 33.4.14 Speeding up the Division Algorithm with Shifting

computing div by shifting

**lemma** *pos-zdiv-mult-2*:  $(0::int) \leq a \implies (1 + 2*b) \text{ div } (2*a) = b \text{ div } a$

**proof** *cases*

**assume**  $a=0$

**thus** *?thesis* **by** *simp*

**next**

**assume**  $a \neq 0$  **and**  $le\text{-}a: 0 \leq a$

**hence**  $a\text{-pos}: 1 \leq a$  **by** *arith*

**hence**  $one\text{-less}\text{-}a2: 1 < 2 * a$  **by** *arith*

**hence**  $le\text{-}2a: 2 * (1 + b \text{ mod } a) \leq 2 * a$

**unfolding** *mult-le-cancel-left*

**by** (*simp add: add1-zle-eq add-commute [of 1]*)

**with**  $a\text{-pos}$  **have**  $0 \leq b \text{ mod } a$  **by** *simp*

**hence**  $le\text{-}addm: 0 \leq 1 \text{ mod } (2*a) + 2*(b \text{ mod } a)$

**by** (*simp add: mod-pos-pos-trivial one-less-a2*)

**with**  $le\text{-}2a$

**have**  $(1 \text{ mod } (2*a) + 2*(b \text{ mod } a)) \text{ div } (2*a) = 0$

**by** (*simp add: div-pos-pos-trivial le-addm mod-pos-pos-trivial one-less-a2*  
      *right-distrib*)

**thus** *?thesis*

**by** (*subst zdiv-zadd1-eq,*

*simp add: mod-mult-mult1 one-less-a2*

*div-pos-pos-trivial*)

**qed**

**lemma** *neg-zdiv-mult-2*:

**assumes**  $A: a \leq (0::int)$  **shows**  $(1 + 2*b) \text{ div } (2*a) = (b+1) \text{ div } a$

**proof** *–*

**have**  $R: 1 + -(2 * (b + 1)) = -(1 + 2 * b)$  **by** *simp*

**have**  $(1 + 2 * (-b - 1)) \text{ div } (2 * (-a)) = (-b - 1) \text{ div } (-a)$

**by** (*rule pos-zdiv-mult-2, simp add: A*)

**thus** *?thesis*

**by** (*simp only: R zdiv-zminus-zminus diff-minus*

*minus-add-distrib [symmetric] mult-minus-right*)

**qed**

**lemma** *zdiv-number-of-Bit0* [*simp*]:

$\text{number-of } (Int.Bit0\ v) \text{ div } \text{number-of } (Int.Bit0\ w) =$

$\text{number-of } v \text{ div } (\text{number-of } w :: int)$

**by** (*simp only: number-of-eq numeral-simps*) (*simp add: mult-2 [symmetric]*)

**lemma** *zdiv-number-of-Bit1* [*simp*]:

$\text{number-of } (Int.Bit1\ v) \text{ div } \text{number-of } (Int.Bit0\ w) =$

$(\text{if } (0::int) \leq \text{number-of } w$

$\text{then } \text{number-of } v \text{ div } (\text{number-of } w)$

$\text{else } (\text{number-of } v + (1::int)) \text{ div } (\text{number-of } w))$

**apply** (*simp only: number-of-eq numeral-simps UNIV-I split: split-if*)

apply (simp add: pos-zdiv-mult-2 neg-zdiv-mult-2 add-ac mult-2 [symmetric])  
done

### 33.4.15 Computing mod by Shifting (proofs resemble those for div)

lemma pos-zmod-mult-2:  
  fixes  $a\ b :: \text{int}$   
  assumes  $0 \leq a$   
  shows  $(1 + 2 * b) \bmod (2 * a) = 1 + 2 * (b \bmod a)$   
proof (cases  $0 < a$ )  
  case False with assms show ?thesis by simp  
next  
  case True  
  then have  $b \bmod a < a$  by (rule pos-mod-bound)  
  then have  $1 + b \bmod a \leq a$  by simp  
  then have  $A: 2 * (1 + b \bmod a) \leq 2 * a$  by simp  
  from  $\langle 0 < a \rangle$  have  $0 \leq b \bmod a$  by (rule pos-mod-sign)  
  then have  $B: 0 \leq 1 + 2 * (b \bmod a)$  by simp  
  have  $((1::\text{int}) \bmod ((2::\text{int}) * a) + (2::\text{int}) * b \bmod ((2::\text{int}) * a)) \bmod ((2::\text{int}) * a) = (1::\text{int}) + (2::\text{int}) * (b \bmod a)$   
  using  $\langle 0 < a \rangle$  and A  
  by (auto simp add: mod-mult-mult1 mod-pos-pos-trivial ring-distrib intro!: mod-pos-pos-trivial B)  
  then show ?thesis by (subst mod-add-eq)  
qed

lemma neg-zmod-mult-2:  
  fixes  $a\ b :: \text{int}$   
  assumes  $a \leq 0$   
  shows  $(1 + 2 * b) \bmod (2 * a) = 2 * ((b + 1) \bmod a) - 1$   
proof –  
  from assms have  $0 \leq -a$  by auto  
  then have  $(1 + 2 * (-b - 1)) \bmod (2 * (-a)) = 1 + 2 * ((-b - 1) \bmod (-a))$   
  by (rule pos-zmod-mult-2)  
  then show ?thesis by (simp add: zmod-zminus2 algebra-simps)  
  (simp add: diff-minus add-ac)  
qed

lemma zmod-number-of-Bit0 [simp]:  
   $\text{number-of } (\text{Int.Bit0 } v) \bmod \text{number-of } (\text{Int.Bit0 } w) =$   
   $(2::\text{int}) * (\text{number-of } v \bmod \text{number-of } w)$   
apply (simp only: number-of-eq numeral-simps)  
apply (simp add: mod-mult-mult1 pos-zmod-mult-2  
  neg-zmod-mult-2 add-ac mult-2 [symmetric])  
done

lemma zmod-number-of-Bit1 [simp]:

```

    number-of (Int.Bit1 v) mod number-of (Int.Bit0 w) =
      (if (0::int) ≤ number-of w
        then 2 * (number-of v mod number-of w) + 1
        else 2 * ((number-of v + (1::int)) mod number-of w) - 1)
  apply (simp only: number-of-eq numeral-simps)
  apply (simp add: mod-mult-mult1 pos-zmod-mult-2
    neg-zmod-mult-2 add-ac mult-2 [symmetric])
done

```

### 33.4.16 Quotients of Signs

```

lemma div-neg-pos-less0: [| a < (0::int); 0 < b |] ==> a div b < 0
  apply (subgoal-tac a div b ≤ -1, force)
  apply (rule order-trans)
  apply (rule-tac a' = -1 in zdiv-mono1)
  apply (auto simp add: div-eq-minus1)
done

```

```

lemma div-nonneg-neg-le0: [| (0::int) ≤ a; b < 0 |] ==> a div b ≤ 0
  by (drule zdiv-mono1-neg, auto)

```

```

lemma div-nonpos-pos-le0: [| (a::int) ≤ 0; b > 0 |] ==> a div b ≤ 0
  by (drule zdiv-mono1, auto)

```

Now for some equivalences of the form  $a \text{ div } b \geq 0 \iff \dots$  conditional upon the sign of  $a$  or  $b$ . There are many more. They should all be simp rules unless that causes too much search.

```

lemma pos-imp-zdiv-nonneg-iff: (0::int) < b ==> (0 ≤ a div b) = (0 ≤ a)
  apply auto
  apply (drule-tac [2] zdiv-mono1)
  apply (auto simp add: linorder-neg-iff)
  apply (simp (no-asm-use) add: linorder-not-less [symmetric])
  apply (blast intro: div-neg-pos-less0)
done

```

```

lemma neg-imp-zdiv-nonneg-iff:
  b < (0::int) ==> (0 ≤ a div b) = (a ≤ (0::int))
  apply (subst zdiv-zminus-zminus [symmetric])
  apply (subst pos-imp-zdiv-nonneg-iff, auto)
done

```

```

lemma pos-imp-zdiv-neg-iff: (0::int) < b ==> (a div b < 0) = (a < 0)
  by (simp add: linorder-not-le [symmetric] pos-imp-zdiv-nonneg-iff)

```

```

lemma neg-imp-zdiv-neg-iff: b < (0::int) ==> (a div b < 0) = (0 < a)
  by (simp add: linorder-not-le [symmetric] neg-imp-zdiv-nonneg-iff)

```

```

lemma nonneg1-imp-zdiv-pos-iff:
  (0::int) <= a ==> (a div b > 0) = (a >= b & b>0)
apply rule
apply rule
  using div-pos-pos-trivial[of a b]apply arith
apply(cases b=0)apply simp
  using div-nonneg-neg-le0[of a b]apply arith
using int-one-le-iff-zero-less[of a div b] zdiv-mono1[of b a b]apply simp
done

```

### 33.4.17 The Divides Relation

```

lemmas zdvd-iff-zmod-eq-0-number-of [simp] =
  dvd-eq-mod-eq-0 [of number-of x::int number-of y::int, standard]

```

```

lemma zdvd-zmod: f dvd m ==> f dvd (n::int) ==> f dvd m mod n
  by (rule dvd-mod)

```

```

lemma zdvd-zmod-imp-zdvd: k dvd m mod n ==> k dvd n ==> k dvd (m::int)
  by (rule dvd-mod-imp-dvd)

```

```

lemma zmult-div-cancel: (n::int) * (m div n) = m - (m mod n)
  using zmod-zdiv-equality[where a=m and b=n]
  by (simp add: algebra-simps)

```

```

lemma zpower-zmod: ((x::int) mod m) ^ y mod m = x ^ y mod m
apply (induct y, auto)
apply (rule zmod-zmult1-eq [THEN trans])
apply (simp (no-asm-simp))
apply (rule mod-mult-eq [symmetric])
done

```

```

lemma zdiv-int: int (a div b) = (int a) div (int b)
apply (subst split-div, auto)
apply (subst split-zdiv, auto)
apply (rule-tac a=int (b * i) + int j and b=int b and r=int j and r'=ja in
  unique-quotient)
apply (auto simp add: divmod-int-rel-def of-nat-mult)
done

```

```

lemma zmod-int: int (a mod b) = (int a) mod (int b)
apply (subst split-mod, auto)
apply (subst split-zmod, auto)
apply (rule-tac a=int (b * i) + int j and b=int b and q=int i and q'=ia
  in unique-remainder)
apply (auto simp add: divmod-int-rel-def of-nat-mult)
done

```

```

lemma abs-div: (y::int) dvd x ==> abs (x div y) = abs x div abs y

```



by (unfold dvd-def, cases y=0, auto simp add: abs-mult)

**lemma** zdvd-mult-div-cancel:  $(n::int) \text{ dvd } m \implies n * (m \text{ div } n) = m$   
**apply** (subgoal-tac m mod n = 0)  
**apply** (simp add: zmult-div-cancel)  
**apply** (simp only: dvd-eq-mod-eq-0)  
**done**

Suggested by Matthias Daum

**lemma** int-power-div-base:  
 $\llbracket 0 < m; 0 < k \rrbracket \implies k ^ m \text{ div } k = (k::int) ^ (m - \text{Suc } 0)$   
**apply** (subgoal-tac  $k ^ m = k ^ ((m - \text{Suc } 0) + \text{Suc } 0)$ )  
**apply** (erule ssubst)  
**apply** (simp only: power-add)  
**apply** simp-all  
**done**

by Brian Huffman

**lemma** zminus-zmod:  $-(x::int) \text{ mod } m \text{ mod } m = -x \text{ mod } m$   
**by** (rule mod-minus-eq [symmetric])

**lemma** zdiff-zmod-left:  $(x \text{ mod } m - y) \text{ mod } m = (x - y) \text{ mod } (m::int)$   
**by** (rule mod-diff-left-eq [symmetric])

**lemma** zdiff-zmod-right:  $(x - y \text{ mod } m) \text{ mod } m = (x - y) \text{ mod } (m::int)$   
**by** (rule mod-diff-right-eq [symmetric])

**lemmas** zmod-simps =  
 mod-add-left-eq [symmetric]  
 mod-add-right-eq [symmetric]  
 zmod-zmult1-eq [symmetric]  
 mod-mult-left-eq [symmetric]  
 zpower-zmod  
 zminus-zmod zdiff-zmod-left zdiff-zmod-right

Distributive laws for function *nat*.

**lemma** nat-div-distrib:  $0 \leq x \implies \text{nat } (x \text{ div } y) = \text{nat } x \text{ div } \text{nat } y$   
**apply** (rule linorder-cases [of y 0])  
**apply** (simp add: div-nonneg-neg-le0)  
**apply** simp  
**apply** (simp add: nat-eq-iff pos-imp-zdiv-nonneg-iff zdiv-int)  
**done**

**lemma** nat-mod-distrib:  
 $\llbracket 0 \leq x; 0 \leq y \rrbracket \implies \text{nat } (x \text{ mod } y) = \text{nat } x \text{ mod } \text{nat } y$   
**apply** (case-tac y = 0, simp)  
**apply** (simp add: nat-eq-iff zmod-int)  
**done**

transfer setup

**lemma** *transfer-nat-int-functions:*

$(x::int) \geq 0 \implies y \geq 0 \implies (nat\ x) \div (nat\ y) = nat\ (x \div y)$   
 $(x::int) \geq 0 \implies y \geq 0 \implies (nat\ x) \bmod (nat\ y) = nat\ (x \bmod y)$   
**by** (*auto simp add: nat-div-distrib nat-mod-distrib*)

**lemma** *transfer-nat-int-function-closures:*

$(x::int) \geq 0 \implies y \geq 0 \implies x \div y \geq 0$   
 $(x::int) \geq 0 \implies y \geq 0 \implies x \bmod y \geq 0$   
**apply** (*cases y = 0*)  
**apply** (*auto simp add: pos-imp-zdiv-nonneg-iff*)  
**apply** (*cases y = 0*)  
**apply** *auto*

**done**

**declare** *transfer-morphism-nat-int* [*transfer add return:*

*transfer-nat-int-functions*  
*transfer-nat-int-function-closures*

**]**

**lemma** *transfer-int-nat-functions:*

$(int\ x) \div (int\ y) = int\ (x \div y)$   
 $(int\ x) \bmod (int\ y) = int\ (x \bmod y)$   
**by** (*auto simp add: zdiv-int zmod-int*)

**lemma** *transfer-int-nat-function-closures:*

$is\_nat\ x \implies is\_nat\ y \implies is\_nat\ (x \div y)$   
 $is\_nat\ x \implies is\_nat\ y \implies is\_nat\ (x \bmod y)$   
**by** (*simp-all only: is-nat-def transfer-nat-int-function-closures*)

**declare** *transfer-morphism-int-nat* [*transfer add return:*

*transfer-int-nat-functions*  
*transfer-int-nat-function-closures*

**]**

Suggested by Matthias Daum

**lemma** *int-div-less-self:*  $\llbracket 0 < x; 1 < k \rrbracket \implies x \div k < (x::int)$

**apply** (*subgoal-tac nat x div nat k < nat x*)  
**apply** (*simp add: nat-div-distrib [symmetric]*)  
**apply** (*rule Divides.div-less-dividend, simp-all*)  
**done**

**lemma** *zmod-eq-dvd-iff:*  $(x::int) \bmod n = y \bmod n \iff n \text{ dvd } x - y$

**proof**

**assume**  $H: x \bmod n = y \bmod n$   
**hence**  $x \bmod n - y \bmod n = 0$  **by** *simp*  
**hence**  $(x \bmod n - y \bmod n) \bmod n = 0$  **by** *simp*  
**hence**  $(x - y) \bmod n = 0$  **by** (*simp add: mod-diff-eq[symmetric]*)  
**thus**  $n \text{ dvd } x - y$  **by** (*simp add: dvd-eq-mod-eq-0*)

**next**

**assume**  $H: n \text{ dvd } x - y$   
  **then obtain**  $k$  **where**  $k: x - y = n * k$  **unfolding**  $\text{dvd-def}$  **by**  $\text{blast}$   
  **hence**  $x = n * k + y$  **by**  $\text{simp}$   
  **hence**  $x \text{ mod } n = (n * k + y) \text{ mod } n$  **by**  $\text{simp}$   
  **thus**  $x \text{ mod } n = y \text{ mod } n$  **by**  $(\text{simp add: mod-add-left-eq})$   
**qed**

**lemma**  $\text{nat-mod-eq-lemma}$ : **assumes**  $xyn: (x::\text{nat}) \text{ mod } n = y \text{ mod } n$  **and**  $xy:y \leq x$   
**shows**  $\exists q. x = y + n * q$   
**proof** –  
  **from**  $xy$  **have**  $th: \text{int } x - \text{int } y = \text{int } (x - y)$  **by**  $\text{simp}$   
  **from**  $xyn$  **have**  $\text{int } x \text{ mod } \text{int } n = \text{int } y \text{ mod } \text{int } n$   
  **by**  $(\text{simp add: zmod-int[symmetric]})$   
  **hence**  $\text{int } n \text{ dvd } \text{int } x - \text{int } y$  **by**  $(\text{simp only: zmod-eq-dvd-iff[symmetric]})$   
  **hence**  $n \text{ dvd } x - y$  **by**  $(\text{simp add: th zdvd-int})$   
  **then show**  $?thesis$  **using**  $xy$  **unfolding**  $\text{dvd-def}$  **apply**  $\text{clarsimp}$  **apply**  $(\text{rule-tac } x=k \text{ in } exI)$  **by**  $\text{arith}$   
**qed**

**lemma**  $\text{nat-mod-eq-iff}$ :  $(x::\text{nat}) \text{ mod } n = y \text{ mod } n \longleftrightarrow (\exists q1 \ q2. x + n * q1 = y + n * q2)$   
**(is**  $?lhs = ?rhs$ **)**  
**proof**  
  **assume**  $H: x \text{ mod } n = y \text{ mod } n$   
  **{assume**  $xy: x \leq y$   
    **from**  $H$  **have**  $th: y \text{ mod } n = x \text{ mod } n$  **by**  $\text{simp}$   
    **from**  $\text{nat-mod-eq-lemma}[OF \ th \ xy]$  **have**  $?rhs$   
    **apply**  $\text{clarify}$  **apply**  $(\text{rule-tac } x=q \text{ in } exI)$  **by**  $(\text{rule } exI[\text{where } x=0], \text{ simp})$   
  **moreover**  
  **{assume**  $xy: y \leq x$   
    **from**  $\text{nat-mod-eq-lemma}[OF \ H \ xy]$  **have**  $?rhs$   
    **apply**  $\text{clarify}$  **apply**  $(\text{rule-tac } x=0 \text{ in } exI)$  **by**  $(\text{rule-tac } x=q \text{ in } exI, \text{ simp})$   
  **ultimately show**  $?rhs$  **using**  $\text{linear}[of \ x \ y]$  **by**  $\text{blast}$   
**next**  
  **assume**  $?rhs$  **then obtain**  $q1 \ q2$  **where**  $q12: x + n * q1 = y + n * q2$  **by**  $\text{blast}$   
  **hence**  $(x + n * q1) \text{ mod } n = (y + n * q2) \text{ mod } n$  **by**  $\text{simp}$   
  **thus**  $?lhs$  **by**  $\text{simp}$   
**qed**

**lemma**  $\text{div-nat-number-of}$   $[simp]$ :  
   $(\text{number-of } v :: \text{nat}) \text{ div } \text{number-of } v' =$   
   $(\text{if } \text{neg } (\text{number-of } v :: \text{int}) \text{ then } 0$   
   $\text{else } \text{nat } (\text{number-of } v \text{ div } \text{number-of } v'))$   
  **unfolding**  $\text{nat-number-of-def}$   $\text{number-of-is-id}$   $\text{neg-def}$   
  **by**  $(\text{simp add: nat-div-distrib})$

**lemma**  $\text{one-div-nat-number-of}$   $[simp]$ :

$Suc\ 0\ div\ number-of\ v' = nat\ (1\ div\ number-of\ v')$   
**by** (*simp* del: nat-numeral-1-eq-1 add: numeral-1-eq-Suc-0 [symmetric])

**lemma** *mod-nat-number-of* [*simp*]:  
 $(number-of\ v :: nat)\ mod\ number-of\ v' =$   
 $(if\ neg\ (number-of\ v :: int)\ then\ 0$   
 $\quad else\ if\ neg\ (number-of\ v' :: int)\ then\ number-of\ v$   
 $\quad else\ nat\ (number-of\ v\ mod\ number-of\ v'))$   
**unfolding** *nat-number-of-def number-of-is-id neg-def*  
**by** (*simp* add: nat-mod-distrib)

**lemma** *one-mod-nat-number-of* [*simp*]:  
 $Suc\ 0\ mod\ number-of\ v' =$   
 $(if\ neg\ (number-of\ v' :: int)\ then\ Suc\ 0$   
 $\quad else\ nat\ (1\ mod\ number-of\ v'))$   
**by** (*simp* del: nat-numeral-1-eq-1 add: numeral-1-eq-Suc-0 [symmetric])

**lemmas** *dvd-eq-mod-eq-0-number-of* =  
 $dvd-eq-mod-eq-0\ [of\ number-of\ x\ number-of\ y,\ standard]$

**declare** *dvd-eq-mod-eq-0-number-of* [*simp*]

### 33.4.18 Nitpick

**lemma** *zmod-zdiv-equality'*:  
 $(m :: int)\ mod\ n = m - (m\ div\ n) * n$   
**by** (*rule-tac*  $P = \%x.\ m\ mod\ n = x - (m\ div\ n) * n$   
**in** *subst* [*OF mod-div-equality* [*of - n*]])  
*arith*

**lemmas** [*nitpick-def*] = *dvd-eq-mod-eq-0* [*THEN eq-reflection*]  
 $mod-div-equality'\ [THEN\ eq-reflection]$   
 $zmod-zdiv-equality'\ [THEN\ eq-reflection]$

### 33.4.19 Code generation

**definition** *pdivmod* ::  $int \Rightarrow int \Rightarrow int \times int$  **where**  
 $pdivmod\ k\ l = (|k|\ div\ |l|,\ |k|\ mod\ |l|)$

**lemma** *pdivmod-posDivAlg* [*code*]:  
 $pdivmod\ k\ l = (if\ l = 0\ then\ (0,\ |k|)\ else\ posDivAlg\ |k|\ |l|)$   
**by** (*subst posDivAlg-div-mod*) (*simp-all* add: *pdivmod-def*)

**lemma** *divmod-int-pdivmod*:  $divmod-int\ k\ l = (if\ k = 0\ then\ (0,\ 0)\ else\ if\ l = 0$   
 $then\ (0,\ k)\ else$   
 $\quad apsnd\ ((op\ *)\ (sgn\ l))\ (if\ 0 < l \wedge 0 \leq k \vee l < 0 \wedge k < 0$   
 $\quad then\ pdivmod\ k\ l$   
 $\quad else\ (let\ (r,\ s) = pdivmod\ k\ l\ in$   
 $\quad if\ s = 0\ then\ (-\ r,\ 0)\ else\ (-\ r - 1,\ |l| - s))))$

**proof** –

```

have aux:  $\bigwedge q::\text{int}. -k = l * q \longleftrightarrow k = l * -q$  by auto
show ?thesis
  by (simp add: divmod-int-mod-div pdivmod-def)
    (auto simp add: aux not-less not-le zdiv-zminus1-eq-if
      zmod-zminus1-eq-if zdiv-zminus2-eq-if zmod-zminus2-eq-if)
qed

```

```

lemma divmod-int-code [code]: divmod-int k l = (if k = 0 then (0, 0) else if l =
0 then (0, k) else
  apsnd ((op *) (sgn l)) (if sgn k = sgn l
    then pdivmod k l
    else (let (r, s) = pdivmod k l in
      if s = 0 then (- r, 0) else (- r - 1, |l| - s))))
proof -
  have  $k \neq 0 \implies l \neq 0 \implies 0 < l \wedge 0 \leq k \vee l < 0 \wedge k < 0 \longleftrightarrow \text{sgn } k = \text{sgn } l$ 
  by (auto simp add: not-less sgn-if)
  then show ?thesis by (simp add: divmod-int-pdivmod)
qed

```

```

context ring-1
begin

```

```

lemma of-int-num [code]:
  of-int k = (if k = 0 then 0 else if k < 0 then
    - of-int (- k) else let
      (l, m) = divmod-int k 2;
      l' = of-int l
      in if m = 0 then l' + l' else l' + l' + 1)
proof -
  have aux1:  $k \bmod (2::\text{int}) \neq (0::\text{int}) \implies$ 
    of-int k = of-int (k div 2 * 2 + 1)
  proof -
    have  $k \bmod 2 < 2$  by (auto intro: pos-mod-bound)
    moreover have  $0 \leq k \bmod 2$  by (auto intro: pos-mod-sign)
    moreover assume  $k \bmod 2 \neq 0$ 
    ultimately have  $k \bmod 2 = 1$  by arith
    moreover have of-int k = of-int (k div 2 * 2 + k mod 2) by simp
    ultimately show ?thesis by auto
  qed
  have aux2:  $\bigwedge x. \text{of-int } 2 * x = x + x$ 
  proof -
    fix x
    have int2:  $(2::\text{int}) = 1 + 1$  by arith
    show of-int 2 * x = x + x
    unfolding int2 of-int-add left-distrib by simp
  qed
  have aux3:  $\bigwedge x. x * \text{of-int } 2 = x + x$ 
  proof -
    fix x

```

```

    have int2: (2::int) = 1 + 1 by arith
    show x * of-int 2 = x + x
    unfolding int2 of-int-add right-distrib by simp
  qed
  from aux1 show ?thesis by (auto simp add: divmod-int-mod-div Let-def aux2
aux3)
qed
end

code-modulename SML
  Divides Arith

code-modulename OCaml
  Divides Arith

code-modulename Haskell
  Divides Arith

end

```

## 34 Code-Numeral: Type of target language numerals

```

theory Code-Numeral
imports Nat-Numeral Nat-Transfer Divides
begin

```

Code numerals are isomorphic to HOL *nat* but mapped to target-language builtin numerals.

### 34.1 Datatype of target language numerals

```

typedef (open) code-numeral = UNIV :: nat set
morphisms nat-of of-nat by rule

```

```

lemma of-nat-nat-of [simp]:
  of-nat (nat-of k) = k
  by (rule nat-of-inverse)

```

```

lemma nat-of-of-nat [simp]:
  nat-of (of-nat n) = n
  by (rule of-nat-inverse) (rule UNIV-I)

```

```

lemma [measure-function]:
  is-measure nat-of by (rule is-measure-trivial)

```

**lemma** *code-numeral*:

$(\bigwedge n::\text{code-numeral}. \text{PROP } P \ n) \equiv (\bigwedge n::\text{nat}. \text{PROP } P \ (\text{of-nat } n))$

**proof**

**fix**  $n :: \text{nat}$

**assume**  $\bigwedge n::\text{code-numeral}. \text{PROP } P \ n$

**then show**  $\text{PROP } P \ (\text{of-nat } n)$  .

**next**

**fix**  $n :: \text{code-numeral}$

**assume**  $\bigwedge n::\text{nat}. \text{PROP } P \ (\text{of-nat } n)$

**then have**  $\text{PROP } P \ (\text{of-nat } (\text{nat-of } n))$  .

**then show**  $\text{PROP } P \ n$  **by** *simp*

**qed**

**lemma** *code-numeral-case*:

**assumes**  $\bigwedge n. k = \text{of-nat } n \implies P$

**shows**  $P$

**by** (*rule assms [of nat-of k]*) *simp*

**lemma** *code-numeral-induct-raw*:

**assumes**  $\bigwedge n. P \ (\text{of-nat } n)$

**shows**  $P \ k$

**proof** –

**from** *assms* **have**  $P \ (\text{of-nat } (\text{nat-of } k))$  .

**then show** *?thesis* **by** *simp*

**qed**

**lemma** *nat-of-inject* [*simp*]:

$\text{nat-of } k = \text{nat-of } l \longleftrightarrow k = l$

**by** (*rule nat-of-inject*)

**lemma** *of-nat-inject* [*simp*]:

$\text{of-nat } n = \text{of-nat } m \longleftrightarrow n = m$

**by** (*rule of-nat-inject*) (*rule UNIV-I*)+

**instantiation** *code-numeral* :: *zero*

**begin**

**definition** [*simp*, *code del*]:

$0 = \text{of-nat } 0$

**instance** ..

**end**

**definition** [*simp*]:

$\text{Suc-code-numeral } k = \text{of-nat } (\text{Suc } (\text{nat-of } k))$

**rep-datatype**  $0 :: \text{code-numeral}$  *Suc-code-numeral*

**proof** –

```

fix  $P :: \text{code-numeral} \Rightarrow \text{bool}$ 
fix  $k :: \text{code-numeral}$ 
assume  $P\ 0$  then have  $\text{init}: P\ (\text{of-nat}\ 0)$  by simp
assume  $\bigwedge k. P\ k \implies P\ (\text{Suc-code-numeral}\ k)$ 
  then have  $\bigwedge n. P\ (\text{of-nat}\ n) \implies P\ (\text{Suc-code-numeral}\ (\text{of-nat}\ n))$  .
  then have  $\text{step}: \bigwedge n. P\ (\text{of-nat}\ n) \implies P\ (\text{of-nat}\ (\text{Suc}\ n))$  by simp
from init step have  $P\ (\text{of-nat}\ (\text{nat-of}\ k))$ 
  by (induct (nat-of  $k$ )) simp-all
then show  $P\ k$  by simp
qed simp-all

declare code-numeral-case [case-names nat, cases type: code-numeral]
declare code-numeral.induct [case-names nat, induct type: code-numeral]

lemma code-numeral-decr [termination-simp]:
   $k \neq \text{of-nat}\ 0 \implies \text{nat-of}\ k - \text{Suc}\ 0 < \text{nat-of}\ k$ 
  by (cases  $k$ ) simp

lemma [simp, code]:
   $\text{code-numeral-size} = \text{nat-of}$ 
proof (rule ext)
  fix  $k$ 
  have  $\text{code-numeral-size}\ k = \text{nat-size}\ (\text{nat-of}\ k)$ 
    by (induct  $k$  rule: code-numeral.induct) (simp-all del: zero-code-numeral-def
Suc-code-numeral-def, simp-all)
  also have  $\text{nat-size}\ (\text{nat-of}\ k) = \text{nat-of}\ k$  by (induct (nat-of  $k$ )) simp-all
  finally show  $\text{code-numeral-size}\ k = \text{nat-of}\ k$  .
qed

lemma [simp, code]:
   $\text{size} = \text{nat-of}$ 
proof (rule ext)
  fix  $k$ 
  show  $\text{size}\ k = \text{nat-of}\ k$ 
  by (induct  $k$ ) (simp-all del: zero-code-numeral-def Suc-code-numeral-def, simp-all)
qed

lemmas [code del] = code-numeral.recs code-numeral.cases

lemma [code]:
   $\text{eq-class.eq}\ k\ l \longleftrightarrow \text{eq-class.eq}\ (\text{nat-of}\ k)\ (\text{nat-of}\ l)$ 
  by (cases  $k$ , cases  $l$ ) (simp add: eq)

lemma [code nbe]:
   $\text{eq-class.eq}\ (k::\text{code-numeral})\ k \longleftrightarrow \text{True}$ 
  by (rule HOL.eq-refl)

```



### 34.2 Indices as datatype of ints

**instantiation** *code-numeral* :: *number*  
**begin**

**definition**

*number-of* = *of-nat* o *nat*

**instance** ..

**end**

**lemma** *nat-of-number* [*simp*]:

*nat-of* (*number-of* *k*) = *number-of* *k*

**by** (*simp* *add*: *number-of-code-numeral-def* *nat-number-of-def* *number-of-is-id*)

**code-datatype** *number-of* :: *int*  $\Rightarrow$  *code-numeral*

### 34.3 Basic arithmetic

**instantiation** *code-numeral* :: {*minus*, *linordered-semidom*, *semiring-div*, *linorder*}  
**begin**

**definition** [*simp*, *code del*]:

(*1*::*code-numeral*) = *of-nat* 1

**definition** [*simp*, *code del*]:

*n* + *m* = *of-nat* (*nat-of* *n* + *nat-of* *m*)

**definition** [*simp*, *code del*]:

*n* - *m* = *of-nat* (*nat-of* *n* - *nat-of* *m*)

**definition** [*simp*, *code del*]:

*n* \* *m* = *of-nat* (*nat-of* *n* \* *nat-of* *m*)

**definition** [*simp*, *code del*]:

*n* *div* *m* = *of-nat* (*nat-of* *n* *div* *nat-of* *m*)

**definition** [*simp*, *code del*]:

*n* *mod* *m* = *of-nat* (*nat-of* *n* *mod* *nat-of* *m*)

**definition** [*simp*, *code del*]:

*n*  $\leq$  *m*  $\longleftrightarrow$  *nat-of* *n*  $\leq$  *nat-of* *m*

**definition** [*simp*, *code del*]:

*n* < *m*  $\longleftrightarrow$  *nat-of* *n* < *nat-of* *m*

**instance** **proof**

**qed** (*auto* *simp* *add*: *code-numeral* *left-distrib* *intro*: *mult-commute*)

**end**

**lemma** *zero-code-numeral-code* [code, code-unfold]:  
 $(0::\text{code-numeral}) = \text{Numeral0}$   
**by** (*simp add: number-of-code-numeral-def Pls-def*)  
**lemma** [code-post]:  $\text{Numeral0} = (0::\text{code-numeral})$   
**using** *zero-code-numeral-code ..*

**lemma** *one-code-numeral-code* [code, code-unfold]:  
 $(1::\text{code-numeral}) = \text{Numeral1}$   
**by** (*simp add: number-of-code-numeral-def Pls-def Bit1-def*)  
**lemma** [code-post]:  $\text{Numeral1} = (1::\text{code-numeral})$   
**using** *one-code-numeral-code ..*

**lemma** *plus-code-numeral-code* [code nbe]:  
 $\text{of-nat } n + \text{of-nat } m = \text{of-nat } (n + m)$   
**by** *simp*

**definition** *subtract-code-numeral* ::  $\text{code-numeral} \Rightarrow \text{code-numeral} \Rightarrow \text{code-numeral}$   
**where**  
 $[\text{simp}, \text{code del}]: \text{subtract-code-numeral} = \text{op } -$

**lemma** *subtract-code-numeral-code* [code nbe]:  
 $\text{subtract-code-numeral } (\text{of-nat } n) (\text{of-nat } m) = \text{of-nat } (n - m)$   
**by** *simp*

**lemma** *minus-code-numeral-code* [code]:  
 $n - m = \text{subtract-code-numeral } n m$   
**by** *simp*

**lemma** *times-code-numeral-code* [code nbe]:  
 $\text{of-nat } n * \text{of-nat } m = \text{of-nat } (n * m)$   
**by** *simp*

**lemma** *less-eq-code-numeral-code* [code nbe]:  
 $\text{of-nat } n \leq \text{of-nat } m \longleftrightarrow n \leq m$   
**by** *simp*

**lemma** *less-code-numeral-code* [code nbe]:  
 $\text{of-nat } n < \text{of-nat } m \longleftrightarrow n < m$   
**by** *simp*

**lemma** *code-numeral-zero-minus-one*:  
 $(0::\text{code-numeral}) - 1 = 0$   
**by** *simp*

**lemma** *Suc-code-numeral-minus-one*:  
 $\text{Suc-code-numeral } n - 1 = n$   
**by** *simp*

**lemma** *of-nat-code* [code]:

*of-nat* = *Nat.of-nat*

**proof**

**fix** *n* :: *nat*

**have** *Nat.of-nat n* = *of-nat n*

**by** (*induct n*) *simp-all*

**then show** *of-nat n* = *Nat.of-nat n*

**by** (*rule sym*)

**qed**

**lemma** *code-numeral-not-eq-zero*:  $i \neq \text{of-nat } 0 \longleftrightarrow i \geq 1$

**by** (*cases i*) *auto*

**definition** *nat-of-aux* :: *code-numeral*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**

*nat-of-aux i n* = *nat-of i + n*

**lemma** *nat-of-aux-code* [code]:

*nat-of-aux i n* = (*if i* = 0 *then n* *else nat-of-aux (i - 1) (Suc n)*)

**by** (*auto simp add: nat-of-aux-def code-numeral-not-eq-zero*)

**lemma** *nat-of-code* [code]:

*nat-of i* = *nat-of-aux i 0*

**by** (*simp add: nat-of-aux-def*)

**definition** *div-mod-code-numeral* :: *code-numeral*  $\Rightarrow$  *code-numeral*  $\Rightarrow$  *code-numeral*  $\times$  *code-numeral* **where**

[*code del*]: *div-mod-code-numeral n m* = (*n div m*, *n mod m*)

**lemma** [code]:

*div-mod-code-numeral n m* = (*if m* = 0 *then (0, n)* *else (n div m, n mod m)*)

**unfolding** *div-mod-code-numeral-def* **by** *auto*

**lemma** [code]:

*n div m* = *fst (div-mod-code-numeral n m)*

**unfolding** *div-mod-code-numeral-def* **by** *simp*

**lemma** [code]:

*n mod m* = *snd (div-mod-code-numeral n m)*

**unfolding** *div-mod-code-numeral-def* **by** *simp*

**definition** *int-of* :: *code-numeral*  $\Rightarrow$  *int* **where**

*int-of* = *Nat.of-nat o nat-of*

**lemma** *int-of-code* [code]:

*int-of k* = (*if k* = 0 *then 0*

*else (if k mod 2* = 0 *then 2 \* int-of (k div 2)* *else 2 \* int-of (k div 2) + 1)*)

**proof** –

**have** (*nat-of k div 2*) \* 2 + *nat-of k mod 2* = *nat-of k*

```

    by (rule mod-div-equality)
  then have int ((nat-of k div 2) * 2 + nat-of k mod 2) = int (nat-of k)
    by simp
  then have int (nat-of k) = int (nat-of k div 2) * 2 + int (nat-of k mod 2)
    unfolding int-mult zadd-int [symmetric] by simp
  then show ?thesis by (auto simp add: int-of-def mult-ac)
qed

```

```
hide-const (open) of-nat nat-of int-of
```

### 34.3.1 Lazy Evaluation of an indexed function

```

function iterate-upto :: (code-numeral => 'a) => code-numeral => code-numeral
=> 'a Predicate.pred
where
  iterate-upto f n m = Predicate.Seq (%u. if n > m then Predicate.Empty else
    Predicate.Insert (f n) (iterate-upto f (n + 1) m))
by pat-completeness auto

```

```

termination by (relation measure (%(f, n, m). Code-Numeral.nat-of (m + 1 -
n))) auto

```

```
hide-const (open) iterate-upto
```

## 34.4 Code generator setup

Implementation of indices by bounded integers

```

code-type code-numeral
  (SML int)
  (OCaml Big'-int.big'-int)
  (Haskell Int)
  (Scala Int)

```

```

code-instance code-numeral :: eq
  (Haskell -)

```

```

setup <<
  fold (Numeral.add-code @{const-name number-code-numeral-inst.number-of-code-numeral}
    false Code-Printer.literal-naive-numeral) [SML, Haskell]
  #> Numeral.add-code @{const-name number-code-numeral-inst.number-of-code-numeral}
    false Code-Printer.literal-numeral OCaml
  #> Numeral.add-code @{const-name number-code-numeral-inst.number-of-code-numeral}
    false Code-Printer.literal-naive-numeral Scala
  >>

```

```

code-reserved SML Int int
code-reserved Scala Int

```

```

code-const op + :: code-numeral => code-numeral => code-numeral

```

```

(SML Int.+/ ((-),/ (-)))
(OCaml Big'-int.add'-big'-int)
(Haskell infixl 6 +)
(Scala infixl 7 +)

code-const subtract-code-numeral :: code-numeral ⇒ code-numeral ⇒ code-numeral
(SML Int.max/ (-/ -/ -,/ 0 : int))
(OCaml Big'-int.max'-big'-int / (Big'-int.sub'-big'-int / -/ -) / Big'-int.zero'-big'-int)
(Haskell max / (-/ -/ -) / (0 :: Int))
(Scala !(-/ -/ -).max(0))

code-const op * :: code-numeral ⇒ code-numeral ⇒ code-numeral
(SML Int.* / ((-),/ (-)))
(OCaml Big'-int.mult'-big'-int)
(Haskell infixl 7 *)
(Scala infixl 8 *)

code-const div-mod-code-numeral
(SML !(fn n => fn m => / if m = 0 / then (0, n) else / (n div m, n mod m)))
(OCaml Big'-int.quomod'-big'-int / (Big'-int.abs'-big'-int -) / (Big'-int.abs'-big'-int
-))
(Haskell divMod)
(Scala !((n: Int) => (m: Int) => / if (m == 0) / (0, n) else / (n ' / m, n %
m)))

code-const eq-class.eq :: code-numeral ⇒ code-numeral ⇒ bool
(SML !((- : Int.int) = -))
(OCaml Big'-int.eq'-big'-int)
(Haskell infixl 4 ==)
(Scala infixl 5 ==)

code-const op ≤ :: code-numeral ⇒ code-numeral ⇒ bool
(SML Int.<= / ((-),/ (-)))
(OCaml Big'-int.le'-big'-int)
(Haskell infix 4 <=)
(Scala infixl 4 <=)

code-const op < :: code-numeral ⇒ code-numeral ⇒ bool
(SML Int.< / ((-),/ (-)))
(OCaml Big'-int.lt'-big'-int)
(Haskell infix 4 <)
(Scala infixl 4 <)

end

```

### 35 Numeral-Simprocs: Combination and Cancellation Simprocs for Numeral Expressions

**theory** *Numeral-Simprocs*

**imports** *Divides*

**uses**

~~/src/Provers/Arith/assoc-fold.ML  
 ~~/src/Provers/Arith/cancel-numerals.ML  
 ~~/src/Provers/Arith/combine-numerals.ML  
 ~~/src/Provers/Arith/cancel-numeral-factor.ML  
 ~~/src/Provers/Arith/extract-common-term.ML  
 (Tools/numeral-simprocs.ML)  
 (Tools/nat-numeral-simprocs.ML)

**begin**

**declare** *split-div* [of - - number-of *k*, standard, arith-split]

**declare** *split-mod* [of - - number-of *k*, standard, arith-split]

For *combine-numerals*

**lemma** *left-add-mult-distrib*:  $i*u + (j*u + k) = (i+j)*u + (k::nat)$

**by** (*simp add: add-mult-distrib*)

For *cancel-numerals*

**lemma** *nat-diff-add-eq1*:

$j \leq (i::nat) \implies ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)$

**by** (*simp split add: nat-diff-split add: add-mult-distrib*)

**lemma** *nat-diff-add-eq2*:

$i \leq (j::nat) \implies ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))$

**by** (*simp split add: nat-diff-split add: add-mult-distrib*)

**lemma** *nat-eq-add-iff1*:

$j \leq (i::nat) \implies (i*u + m = j*u + n) = ((i-j)*u + m = n)$

**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-eq-add-iff2*:

$i \leq (j::nat) \implies (i*u + m = j*u + n) = (m = (j-i)*u + n)$

**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-less-add-iff1*:

$j \leq (i::nat) \implies (i*u + m < j*u + n) = ((i-j)*u + m < n)$

**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-less-add-iff2*:

$i \leq (j::nat) \implies (i*u + m < j*u + n) = (m < (j-i)*u + n)$

**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-le-add-iff1*:

$j \leq (i::nat) \implies (i*u + m \leq j*u + n) = ((i-j)*u + m \leq n)$

**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

**lemma** *nat-le-add-iff2*:

$i \leq (j::nat) \implies (i*u + m \leq j*u + n) = (m \leq (j-i)*u + n)$

**by** (*auto split add: nat-diff-split simp add: add-mult-distrib*)

For *cancel-numeral-factors*

**lemma** *nat-mult-le-cancel1*:  $(0::nat) < k \implies (k*m \leq k*n) = (m \leq n)$

**by** *auto*

**lemma** *nat-mult-less-cancel1*:  $(0::nat) < k \implies (k*m < k*n) = (m < n)$

**by** *auto*

**lemma** *nat-mult-eq-cancel1*:  $(0::nat) < k \implies (k*m = k*n) = (m = n)$

**by** *auto*

**lemma** *nat-mult-div-cancel1*:  $(0::nat) < k \implies (k*m) \text{ div } (k*n) = (m \text{ div } n)$

**by** *auto*

**lemma** *nat-mult-dvd-cancel-disj[simp]*:

$(k*m) \text{ dvd } (k*n) = (k=0 \mid m \text{ dvd } (n::nat))$

**by**(*auto simp: dvd-eq-mod-eq-0 mod-mult-distrib2[symmetric]*)

**lemma** *nat-mult-dvd-cancel1*:  $0 < k \implies (k*m) \text{ dvd } (k*n::nat) = (m \text{ dvd } n)$

**by**(*auto*)

For *cancel-factor*

**lemma** *nat-mult-le-cancel-disj*:  $(k*m \leq k*n) = ((0::nat) < k \longrightarrow m \leq n)$

**by** *auto*

**lemma** *nat-mult-less-cancel-disj*:  $(k*m < k*n) = ((0::nat) < k \ \& \ m < n)$

**by** *auto*

**lemma** *nat-mult-eq-cancel-disj*:  $(k*m = k*n) = (k = (0::nat) \mid m = n)$

**by** *auto*

**lemma** *nat-mult-div-cancel-disj[simp]*:

$(k*m) \text{ div } (k*n) = (\text{if } k = (0::nat) \text{ then } 0 \text{ else } m \text{ div } n)$

**by** (*simp add: nat-mult-div-cancel1*)

**use** *Tools/numeral-simprocs.ML*

**use** *Tools/nat-numeral-simprocs.ML*

**declaration**  $\ll$

$K \text{ (} \textit{Lin-Arith.add-simps} \text{ (} @\{\textit{thms neg-simps}\} \text{ @ } [ @\{\textit{thm Suc-nat-number-of}\}, @\{\textit{thm int-nat-number-of}\} \})$

$\#> \textit{Lin-Arith.add-simps} \text{ (} @\{\textit{thms ring-distrib}\} \text{ @ } [ @\{\textit{thm Let-number-of}\}, @\{\textit{thm}$

```

Let-0}, @{thm Let-1},
  @{thm nat-0}, @{thm nat-1},
  @{thm add-nat-number-of}, @{thm diff-nat-number-of}, @{thm mult-nat-number-of},
  @{thm eq-nat-number-of}, @{thm less-nat-number-of}, @{thm le-number-of-eq-not-less},
  @{thm le-Suc-number-of}, @{thm le-number-of-Suc},
  @{thm less-Suc-number-of}, @{thm less-number-of-Suc},
  @{thm Suc-eq-number-of}, @{thm eq-number-of-Suc},
  @{thm mult-Suc}, @{thm mult-Suc-right},
  @{thm add-Suc}, @{thm add-Suc-right},
  @{thm eq-number-of-0}, @{thm eq-0-number-of}, @{thm less-0-number-of},
  @{thm of-int-number-of-eq}, @{thm of-nat-number-of-eq}, @{thm nat-number-of},
  @{thm if-True}, @{thm if-False}])
#> Lin-Arith.add-simprocs (Numeral-Simprocs.assoc-fold-simproc
  :: Numeral-Simprocs.combine-numerals
  :: Numeral-Simprocs.cancel-numerals)
#> Lin-Arith.add-simprocs (Nat-Numeral-Simprocs.combine-numerals :: Nat-Numeral-Simprocs.cancel-numerals)
>>

end

```

## 36 Semiring-Normalization: Semiring normalization

```

theory Semiring-Normalization
imports Numeral-Simprocs Nat-Transfer
uses
  Tools/semiring-normalizer.ML
begin

Prelude

class comm-semiring-1-cancel-crossproduct = comm-semiring-1-cancel +
  assumes crossproduct-eq:  $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$ 
begin

lemma crossproduct-noteq:
   $a \neq b \wedge c \neq d \longleftrightarrow a * c + b * d \neq a * d + b * c$ 
  by (simp add: crossproduct-eq)

lemma add-scale-eq-noteq:
   $r \neq 0 \implies a = b \wedge c \neq d \implies a + r * c \neq b + r * d$ 
proof (rule notI)
  assume nz:  $r \neq 0$  and cnd:  $a = b \wedge c \neq d$ 
  and eq:  $a + (r * c) = b + (r * d)$ 
  have  $(0 * d) + (r * c) = (0 * c) + (r * d)$ 
  using add-imp-eq eq mult-zero-left by (simp add: cnd)
  then show False using crossproduct-eq [of 0 d] nz cnd by simp
qed

```



**lemma** *add-0-iff*:

$$b = b + a \longleftrightarrow a = 0$$

**using** *add-imp-eq* [of *b a 0*] **by** *auto*

**end**

**sublocale** *idom* < *comm-semiring-1-cancel-crossproduct*

**proof**

**fix** *w x y z*

$$\text{show } w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$$

**proof**

**assume**  $w * y + x * z = w * z + x * y$

**then have**  $w * y + x * z - w * z - x * y = 0$  **by** (*simp add: algebra-simps*)

**then have**  $w * (y - z) - x * (y - z) = 0$  **by** (*simp add: algebra-simps*)

**then have**  $(y - z) * (w - x) = 0$  **by** (*simp add: algebra-simps*)

**then have**  $y - z = 0 \vee w - x = 0$  **by** (*rule divisors-zero*)

**then show**  $w = x \vee y = z$  **by** *auto*

**qed** (*auto simp add: add-ac*)

**qed**

**instance** *nat* :: *comm-semiring-1-cancel-crossproduct*

**proof**

**fix** *w x y z* :: *nat*

$$\text{have } \text{aux}: \bigwedge y z. y < z \implies w * y + x * z = w * z + x * y \implies w = x$$

**proof** –

**fix** *y z* :: *nat*

**assume**  $y < z$  **then have**  $\exists k. z = y + k \wedge k \neq 0$  **by** (*intro exI [of - z - y]*)

*auto*

**then obtain** *k* **where**  $z = y + k$  **and**  $k \neq 0$  **by** *blast*

**assume**  $w * y + x * z = w * z + x * y$

**then have**  $(w * y + x * y) + x * k = (w * y + x * y) + w * k$  **by** (*simp*

*add: (z = y + k) algebra-simps*)

**then have**  $x * k = w * k$  **by** *simp*

**then show**  $w = x$  **using**  $\langle k \neq 0 \rangle$  **by** *simp*

**qed**

$$\text{show } w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$$

**by** (*auto simp add: neq-iff dest!: aux*)

**qed**

Semiring normalization proper

**setup** *Semiring-Normalizer.setup*

**context** *comm-semiring-1*

**begin**

**lemma** *normalizing-semiring-ops*:

**shows** *TERM* ( $x + y$ ) **and** *TERM* ( $x * y$ ) **and** *TERM* ( $x \wedge n$ )

**and** *TERM* 0 **and** *TERM* 1 .

**lemma** *normalizing-semiring-rules:*

$$\begin{aligned}
 (a * m) + (b * m) &= (a + b) * m \\
 (a * m) + m &= (a + 1) * m \\
 m + (a * m) &= (a + 1) * m \\
 m + m &= (1 + 1) * m \\
 0 + a &= a \\
 a + 0 &= a \\
 a * b &= b * a \\
 (a + b) * c &= (a * c) + (b * c) \\
 0 * a &= 0 \\
 a * 0 &= 0 \\
 1 * a &= a \\
 a * 1 &= a \\
 (lx * ly) * (rx * ry) &= (lx * rx) * (ly * ry) \\
 (lx * ly) * (rx * ry) &= lx * (ly * (rx * ry)) \\
 (lx * ly) * (rx * ry) &= rx * ((lx * ly) * ry) \\
 (lx * ly) * rx &= (lx * rx) * ly \\
 (lx * ly) * rx &= lx * (ly * rx) \\
 lx * (rx * ry) &= (lx * rx) * ry \\
 lx * (rx * ry) &= rx * (lx * ry) \\
 (a + b) + (c + d) &= (a + c) + (b + d) \\
 (a + b) + c &= a + (b + c) \\
 a + (c + d) &= c + (a + d) \\
 (a + b) + c &= (a + c) + b \\
 a + c &= c + a \\
 a + (c + d) &= (a + c) + d \\
 (x \hat{ } p) * (x \hat{ } q) &= x \hat{ } (p + q) \\
 x * (x \hat{ } q) &= x \hat{ } (Suc\ q) \\
 (x \hat{ } q) * x &= x \hat{ } (Suc\ q) \\
 x * x &= x \hat{ } 2 \\
 (x * y) \hat{ } q &= (x \hat{ } q) * (y \hat{ } q) \\
 (x \hat{ } p) \hat{ } q &= x \hat{ } (p * q) \\
 x \hat{ } 0 &= 1 \\
 x \hat{ } 1 &= x \\
 x * (y + z) &= (x * y) + (x * z) \\
 x \hat{ } (Suc\ q) &= x * (x \hat{ } q) \\
 x \hat{ } (2*n) &= (x \hat{ } n) * (x \hat{ } n) \\
 x \hat{ } (Suc\ (2*n)) &= x * ((x \hat{ } n) * (x \hat{ } n)) \\
 \text{by (simp-all add: algebra-simps power-add power2-eq-square power-mult-distrib} \\
 &\text{power-mult)}
 \end{aligned}$$

**lemmas** *normalizing-comm-semiring-1-axioms =*

*comm-semiring-1-axioms* [normalizer  
 semiring ops: normalizing-semiring-ops  
 semiring rules: normalizing-semiring-rules]

**declaration**

⟨⟨ *Semiring-Normalizer.semiring-funs* @{thm *normalizing-comm-semiring-1-axioms*}  
 ⟩⟩

**end**

**context** *comm-ring-1*  
**begin**

**lemma** *normalizing-ring-ops*: **shows**  $TERM\ (x - y)$  **and**  $TERM\ (-\ x)$  .

**lemma** *normalizing-ring-rules*:

$-\ x = (-\ 1) * x$   
 $x - y = x + (-\ y)$   
**by** (*simp-all add: diff-minus*)

**lemmas** *normalizing-comm-ring-1-axioms* =  
*comm-ring-1-axioms* [*normalizer*  
*semiring ops: normalizing-semiring-ops*  
*semiring rules: normalizing-semiring-rules*  
*ring ops: normalizing-ring-ops*  
*ring rules: normalizing-ring-rules*]

**declaration**

⟨⟨ *Semiring-Normalizer.semiring-funs* @{*thm normalizing-comm-ring-1-axioms*}  
 ⟩⟩

**end**

**context** *comm-semiring-1-cancel-crossproduct*  
**begin**

**declare**

*normalizing-comm-semiring-1-axioms* [*normalizer del*]

**lemmas**

*normalizing-comm-semiring-1-cancel-crossproduct-axioms* =  
*comm-semiring-1-cancel-crossproduct-axioms* [*normalizer*  
*semiring ops: normalizing-semiring-ops*  
*semiring rules: normalizing-semiring-rules*  
*idom rules: crossproduct-noteq add-scale-eq-noteq*]

**declaration**

⟨⟨ *Semiring-Normalizer.semiring-funs* @{*thm normalizing-comm-semiring-1-cancel-crossproduct-axioms*}  
 ⟩⟩

**end**

**context** *idom*  
**begin**

**declare** *normalizing-comm-ring-1-axioms* [*normalizer del*]

**lemmas** *normalizing-idom-axioms* = *idom-axioms* [*normalizer*  
*semiring ops: normalizing-semiring-ops*  
*semiring rules: normalizing-semiring-rules*  
*ring ops: normalizing-ring-ops*  
*ring rules: normalizing-ring-rules*  
*idom rules: crossproduct-noteq add-scale-eq-noteq*  
*ideal rules: right-minus-eq add-0-iff*]

**declaration**

⟨⟨ *Semiring-Normalizer.semiring-funs* @{*thm normalizing-idom-axioms*} ⟩⟩

**end**

**context** *field*

**begin**

**lemma** *normalizing-field-ops:*

**shows** *TERM* (*x* / *y*) **and** *TERM* (*inverse x*) .

**lemmas** *normalizing-field-rules* = *divide-inverse inverse-eq-divide*

**lemmas** *normalizing-field-axioms* =

*field-axioms* [*normalizer*  
*semiring ops: normalizing-semiring-ops*  
*semiring rules: normalizing-semiring-rules*  
*ring ops: normalizing-ring-ops*  
*ring rules: normalizing-ring-rules*  
*field ops: normalizing-field-ops*  
*field rules: normalizing-field-rules*  
*idom rules: crossproduct-noteq add-scale-eq-noteq*  
*ideal rules: right-minus-eq add-0-iff*]

**declaration**

⟨⟨ *Semiring-Normalizer.field-funs* @{*thm normalizing-field-axioms*} ⟩⟩

**end**

**hide-fact (open)** *normalizing-comm-semiring-1-axioms*

*normalizing-comm-semiring-1-cancel-crossproduct-axioms normalizing-semiring-ops*  
*normalizing-semiring-rules*

**hide-fact (open)** *normalizing-comm-ring-1-axioms*

*normalizing-idom-axioms normalizing-ring-ops normalizing-ring-rules*

**hide-fact (open)** *normalizing-field-axioms normalizing-field-ops normalizing-field-rules*

**end**

## 37 Groebner-Basis: Groebner bases

```

theory Groebner-Basis
imports Semiring-Normalization
uses
  (Tools/groebner.ML)
begin

```

### 37.1 Groebner Bases

```

lemmas bool-simps = simp-thms(1-34)

```

```

lemma dnf:
   $(P \ \& \ (Q \mid R)) = ((P \ \& \ Q) \mid (P \ \& \ R)) \ ((Q \mid R) \ \& \ P) = ((Q \ \& \ P) \mid (R \ \& \ P))$ 
   $(P \ \wedge \ Q) = (Q \ \wedge \ P) \ (P \ \vee \ Q) = (Q \ \vee \ P)$ 
by blast+

```

```

lemmas weak-dnf-simps = dnf bool-simps

```

```

lemma nnf-simps:
   $(\neg(P \ \wedge \ Q)) = (\neg P \ \vee \ \neg Q) \ (\neg(P \ \vee \ Q)) = (\neg P \ \wedge \ \neg Q) \ (P \longrightarrow Q) = (\neg P \ \vee \ Q)$ 
   $(P = Q) = ((P \ \wedge \ Q) \vee (\neg P \ \wedge \ \neg Q)) \ (\neg \neg(P)) = P$ 
by blast+

```

```

lemma PFalse:
   $P \equiv \text{False} \implies \neg P$ 
   $\neg P \implies (P \equiv \text{False})$ 
by auto

```

```

ML <<
  structure Algebra-Simplification = Named-Thms(
    val name = algebra
    val description = pre-simplification rules for algebraic methods
  )
>>

```

```

setup Algebra-Simplification.setup

```

```

use Tools/groebner.ML

```

```

method-setup algebra = Groebner.algebra-method
  solve polynomial equations over (semi)rings and ideal membership problems using
  Groebner bases

```

```

declare dvd-def[algebra]
declare dvd-eq-mod-eq-0[symmetric, algebra]
declare mod-div-trivial[algebra]
declare mod-mod-trivial[algebra]
declare conjunct1[OF DIVISION-BY-ZERO, algebra]
declare conjunct2[OF DIVISION-BY-ZERO, algebra]

```

```

declare zmod-zdiv-equality [symmetric, algebra]
declare zdiv-zmod-equality [symmetric, algebra]
declare zdiv-zminus-zminus [algebra]
declare zmod-zminus-zminus [algebra]
declare zdiv-zminus2 [algebra]
declare zmod-zminus2 [algebra]
declare zdiv-zero [algebra]
declare zmod-zero [algebra]
declare mod-by-1 [algebra]
declare div-by-1 [algebra]
declare zmod-minus1-right [algebra]
declare zdiv-minus1-right [algebra]
declare mod-div-trivial [algebra]
declare mod-mod-trivial [algebra]
declare mod-mult-self2-is-0 [algebra]
declare mod-mult-self1-is-0 [algebra]
declare zmod-eq-0-iff [algebra]
declare dvd-0-left-iff [algebra]
declare zdvd1-eq [algebra]
declare zmod-eq-dvd-iff [algebra]
declare nat-mod-eq-iff [algebra]

```

```

end

```

### 38 SetInterval: Set intervals

```

theory SetInterval
imports Int Nat-Transfer
begin

```

```

context ord

```

```

begin

```

```

definition

```

```

  lessThan    :: 'a => 'a set ((1{.. $\cdot$ })) where
    {.. $\cdot$  u} == {x. x < u}

```

```

definition

```

```

  atMost      :: 'a => 'a set ((1{.. $\cdot$ })) where
    {.. $\cdot$  u} == {x. x ≤ u}

```

```

definition

```

```

  greaterThan :: 'a => 'a set ((1{ $\cdot$ .. $\cdot$ })) where
    {l.. $\cdot$ } == {x. l < x}

```

```

definition

```

```

  atLeast     :: 'a => 'a set ((1{ $\cdot$ .. $\cdot$ })) where
    {l.. $\cdot$ } == {x. l ≤ x}

```

**definition**

*greaterThanLessThan* :: 'a => 'a => 'a set ((1{-<..<>-})) **where**  
 {l<..} == {l<..} Int {..}

**definition**

*atLeastLessThan* :: 'a => 'a => 'a set ((1{-..<>-})) **where**  
 {l..} == {l..} Int {..}

**definition**

*greaterThanAtMost* :: 'a => 'a => 'a set ((1{-<..<>-})) **where**  
 {l<..} == {l<..} Int {..}

**definition**

*atLeastAtMost* :: 'a => 'a => 'a set ((1{-..<>-})) **where**  
 {l..} == {l..} Int {..}

**end**

A note of warning when using {..n} on type *nat*: it is equivalent to {0..n} but some lemmas involving {m..n} may not exist in {..n}-form as well.

**syntax**

-UNION-le :: 'a => 'a => 'b set => 'b set ((3UN -<=.. / -) [0, 0, 10] 10)  
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3UN -<.. / -) [0, 0, 10] 10)  
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3INT -<=.. / -) [0, 0, 10] 10)  
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3INT -<.. / -) [0, 0, 10] 10)

**syntax** (*xsymbols*)

-UNION-le :: 'a => 'a => 'b set => 'b set ((3UN -<=.. / -) [0, 0, 10] 10)  
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3UN -<.. / -) [0, 0, 10] 10)  
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3INT -<=.. / -) [0, 0, 10] 10)  
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3INT -<.. / -) [0, 0, 10] 10)

**syntax** (*latex output*)

-UNION-le :: 'a => 'a => 'b set => 'b set ((3UN (00- ≤ -) / -) [0, 0, 10] 10)  
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3UN (00- < -) / -) [0, 0, 10] 10)  
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3INT (00- ≤ -) / -) [0, 0, 10] 10)  
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3INT (00- < -) / -) [0, 0, 10] 10)

**translations**

UN *i* ≤ *n*. *A* == UN *i*:{..*n*}. *A*  
 UN *i* < *n*. *A* == UN *i*:{..*n*}. *A*  
 INT *i* ≤ *n*. *A* == INT *i*:{..*n*}. *A*  
 INT *i* < *n*. *A* == INT *i*:{..*n*}. *A*

### 38.1 Various equivalences

**lemma** (in ord) *lessThan-iff* [iff]:  $(i: \text{lessThan } k) = (i < k)$   
**by** (simp add: lessThan-def)

**lemma** *Compl-lessThan* [simp]:  
 $!!k:: 'a::\text{linorder}. \neg \text{lessThan } k = \text{atLeast } k$   
**apply** (auto simp add: lessThan-def atLeast-def)  
**done**

**lemma** *single-Diff-lessThan* [simp]:  $!!k:: 'a::\text{order}. \{k\} - \text{lessThan } k = \{k\}$   
**by** auto

**lemma** (in ord) *greaterThan-iff* [iff]:  $(i: \text{greaterThan } k) = (k < i)$   
**by** (simp add: greaterThan-def)

**lemma** *Compl-greaterThan* [simp]:  
 $!!k:: 'a::\text{linorder}. \neg \text{greaterThan } k = \text{atMost } k$   
**by** (auto simp add: greaterThan-def atMost-def)

**lemma** *Compl-atMost* [simp]:  $!!k:: 'a::\text{linorder}. \neg \text{atMost } k = \text{greaterThan } k$   
**apply** (subst Compl-greaterThan [symmetric])  
**apply** (rule double-complement)  
**done**

**lemma** (in ord) *atLeast-iff* [iff]:  $(i: \text{atLeast } k) = (k \leq i)$   
**by** (simp add: atLeast-def)

**lemma** *Compl-atLeast* [simp]:  
 $!!k:: 'a::\text{linorder}. \neg \text{atLeast } k = \text{lessThan } k$   
**by** (auto simp add: lessThan-def atLeast-def)

**lemma** (in ord) *atMost-iff* [iff]:  $(i: \text{atMost } k) = (i \leq k)$   
**by** (simp add: atMost-def)

**lemma** *atMost-Int-atLeast*:  $!!n:: 'a::\text{order}. \text{atMost } n \text{ Int } \text{atLeast } n = \{n\}$   
**by** (blast intro: order-antisym)

### 38.2 Logical Equivalences for Set Inclusion and Equality

**lemma** *atLeast-subset-iff* [iff]:  
 $(\text{atLeast } x \subseteq \text{atLeast } y) = (y \leq (x::'a::\text{order}))$   
**by** (blast intro: order-trans)

**lemma** *atLeast-eq-iff* [iff]:  
 $(\text{atLeast } x = \text{atLeast } y) = (x = (y::'a::\text{linorder}))$   
**by** (blast intro: order-antisym order-trans)

**lemma** *greaterThan-subset-iff* [iff]:  
 $(\text{greaterThan } x \subseteq \text{greaterThan } y) = (y \leq (x::'a::\text{linorder}))$



```

apply (auto simp add: greaterThan-def)
apply (subst linorder-not-less [symmetric], blast)
done

```

```

lemma greaterThan-eq-iff [iff]:
  (greaterThan x = greaterThan y) = (x = (y::'a::linorder))
apply (rule iffI)
apply (erule equalityE)
apply simp-all
done

```

```

lemma atMost-subset-iff [iff]: (atMost x  $\subseteq$  atMost y) = (x  $\leq$  (y::'a::order))
by (blast intro: order-trans)

```

```

lemma atMost-eq-iff [iff]: (atMost x = atMost y) = (x = (y::'a::linorder))
by (blast intro: order-antisym order-trans)

```

```

lemma lessThan-subset-iff [iff]:
  (lessThan x  $\subseteq$  lessThan y) = (x  $\leq$  (y::'a::linorder))
apply (auto simp add: lessThan-def)
apply (subst linorder-not-less [symmetric], blast)
done

```

```

lemma lessThan-eq-iff [iff]:
  (lessThan x = lessThan y) = (x = (y::'a::linorder))
apply (rule iffI)
apply (erule equalityE)
apply simp-all
done

```

### 38.3 Two-sided intervals

```

context ord
begin

```

```

lemma greaterThanLessThan-iff [simp,no-atp]:
  (i : {l<..}) = (l < i & i < u)
by (simp add: greaterThanLessThan-def)

```

```

lemma atLeastLessThan-iff [simp,no-atp]:
  (i : {l..}) = (l <= i & i < u)
by (simp add: atLeastLessThan-def)

```

```

lemma greaterThanAtMost-iff [simp,no-atp]:
  (i : {l<..}) = (l < i & i <= u)
by (simp add: greaterThanAtMost-def)

```

```

lemma atLeastAtMost-iff [simp,no-atp]:
  (i : {l..}) = (l <= i & i <= u)

```

**by** (*simp add: atLeastAtMost-def*)

The above four lemmas could be declared as *iffs*. Unfortunately this breaks many proofs. Since it only helps *blast*, it is better to leave well alone

**end**

### 38.3.1 Emptiness, singletons, subset

**context** *order*

**begin**

**lemma** *atLeastatMost-empty[*simp*]*:

$b < a \implies \{a..b\} = \{\}$

**by**(*auto simp: atLeastAtMost-def atLeast-def atMost-def*)

**lemma** *atLeastatMost-empty-iff[*simp*]*:

$\{a..b\} = \{\} \longleftrightarrow (\sim a \leq b)$

**by** *auto* (*blast intro: order-trans*)

**lemma** *atLeastatMost-empty-iff2[*simp*]*:

$\{\} = \{a..b\} \longleftrightarrow (\sim a \leq b)$

**by** *auto* (*blast intro: order-trans*)

**lemma** *atLeastLessThan-empty[*simp*]*:

$b \leq a \implies \{a..<b\} = \{\}$

**by**(*auto simp: atLeastLessThan-def*)

**lemma** *atLeastLessThan-empty-iff[*simp*]*:

$\{a..<b\} = \{\} \longleftrightarrow (\sim a < b)$

**by** *auto* (*blast intro: le-less-trans*)

**lemma** *atLeastLessThan-empty-iff2[*simp*]*:

$\{\} = \{a..<b\} \longleftrightarrow (\sim a < b)$

**by** *auto* (*blast intro: le-less-trans*)

**lemma** *greaterThanAtMost-empty[*simp*]*:  $l \leq k \implies \{k<..l\} = \{\}$

**by**(*auto simp: greaterThanAtMost-def greaterThan-def atMost-def*)

**lemma** *greaterThanAtMost-empty-iff[*simp*]*:  $\{k<..l\} = \{\} \longleftrightarrow \sim k < l$

**by** *auto* (*blast intro: less-le-trans*)

**lemma** *greaterThanAtMost-empty-iff2[*simp*]*:  $\{\} = \{k<..l\} \longleftrightarrow \sim k < l$

**by** *auto* (*blast intro: less-le-trans*)

**lemma** *greaterThanLessThan-empty[*simp*]*:  $l \leq k \implies \{k<..$

**by**(*auto simp: greaterThanLessThan-def greaterThan-def lessThan-def*)

**lemma** *atLeastAtMost-singleton [*simp*]*:  $\{a..a\} = \{a\}$

**by** (*auto simp add: atLeastAtMost-def atMost-def atLeast-def*)

**lemma** *atLeastAtMost-singleton'*:  $a = b \implies \{a \dots b\} = \{a\}$  **by** *simp*

**lemma** *atLeastatMost-subset-iff*[*simp*]:

$\{a \dots b\} \leq \{c \dots d\} \iff (\neg a \leq b) \mid c \leq a \ \& \ b \leq d$

**unfolding** *atLeastAtMost-def atLeast-def atMost-def*

**by** (*blast intro: order-trans*)

**lemma** *atLeastatMost-psubset-iff*:

$\{a \dots b\} < \{c \dots d\} \iff$

$((\neg a \leq b) \mid c \leq a \ \& \ b \leq d \ \& \ (c < a \mid b < d)) \ \& \ c \leq d$

**by**(*simp add: psubset-eq expand-set-eq less-le-not-le*)(*blast intro: order-trans*)

**lemma** *atLeastAtMost-singleton-iff*[*simp*]:

$\{a \dots b\} = \{c\} \iff a = b \wedge b = c$

**proof**

**assume**  $\{a \dots b\} = \{c\}$

**hence**  $\neg (\neg a \leq b)$  **unfolding** *atLeastatMost-empty-iff*[*symmetric*] **by** *simp*

**moreover with**  $\langle \{a \dots b\} = \{c\} \rangle$  **have**  $c \leq a \wedge b \leq c$  **by** *auto*

**ultimately show**  $a = b \wedge b = c$  **by** *auto*

**qed** *simp*

**end**

**lemma** (**in** *linorder*) *atLeastLessThan-subset-iff*:

$\{a \dots < b\} \leq \{c \dots < d\} \implies b \leq a \mid c \leq a \ \& \ b \leq d$

**apply** (*auto simp: subset-eq Ball-def*)

**apply**(*frule-tac x=a in spec*)

**apply**(*erule-tac x=d in allE*)

**apply** (*simp add: less-imp-le*)

**done**

### 38.3.2 Intersection

**context** *linorder*

**begin**

**lemma** *Int-atLeastAtMost*[*simp*]:  $\{a \dots b\} \text{ Int } \{c \dots d\} = \{\max a \ c \dots \min b \ d\}$

**by** *auto*

**lemma** *Int-atLeastAtMostR1*[*simp*]:  $\{\dots b\} \text{ Int } \{c \dots d\} = \{c \dots \min b \ d\}$

**by** *auto*

**lemma** *Int-atLeastAtMostR2*[*simp*]:  $\{a \dots\} \text{ Int } \{c \dots d\} = \{\max a \ c \dots d\}$

**by** *auto*

**lemma** *Int-atLeastAtMostL1*[*simp*]:  $\{a \dots b\} \text{ Int } \{\dots d\} = \{a \dots \min b \ d\}$

**by** *auto*

**lemma** *Int-atLeastAtMostL2*[simp]:  $\{a..b\} \text{ Int } \{c..\} = \{\max a \ c \ .. \ b\}$   
**by** *auto*

**lemma** *Int-atLeastLessThan*[simp]:  $\{a..<b\} \text{ Int } \{c..<d\} = \{\max a \ c \ ..< \min b \ d\}$   
**by** *auto*

**lemma** *Int-greaterThanAtMost*[simp]:  $\{a<..b\} \text{ Int } \{c<..d\} = \{\max a \ c \ <.. \min b \ d\}$   
**by** *auto*

**lemma** *Int-greaterThanLessThan*[simp]:  $\{a<..**b\} \text{ Int } \{c<..**d\} = \{\max a \ c \ <..**\min b \ d\}******$   
**by** *auto*

**end**

## 38.4 Intervals of natural numbers

### 38.4.1 The Constant *lessThan*

**lemma** *lessThan-0* [simp]:  $\text{lessThan } (0::\text{nat}) = \{\}$   
**by** (*simp add: lessThan-def*)

**lemma** *lessThan-Suc*:  $\text{lessThan } (\text{Suc } k) = \text{insert } k \ (\text{lessThan } k)$   
**by** (*simp add: lessThan-def less-Suc-eq, blast*)

**lemma** *lessThan-Suc-atMost*:  $\text{lessThan } (\text{Suc } k) = \text{atMost } k$   
**by** (*simp add: lessThan-def atMost-def less-Suc-eq-le*)

**lemma** *UN-lessThan-UNIV*:  $(\text{UN } m::\text{nat}. \text{lessThan } m) = \text{UNIV}$   
**by** *blast*

### 38.4.2 The Constant *greaterThan*

**lemma** *greaterThan-0* [simp]:  $\text{greaterThan } 0 = \text{range } \text{Suc}$   
**apply** (*simp add: greaterThan-def*)  
**apply** (*blast dest: gr0-conv-Suc [THEN iffD1]*)  
**done**

**lemma** *greaterThan-Suc*:  $\text{greaterThan } (\text{Suc } k) = \text{greaterThan } k - \{\text{Suc } k\}$   
**apply** (*simp add: greaterThan-def*)  
**apply** (*auto elim: linorder-neqE*)  
**done**

**lemma** *INT-greaterThan-UNIV*:  $(\text{INT } m::\text{nat}. \text{greaterThan } m) = \{\}$   
**by** *blast*

### 38.4.3 The Constant *atLeast*

**lemma** *atLeast-0* [simp]:  $\text{atLeast } (0::\text{nat}) = \text{UNIV}$

by (unfold atLeast-def UNIV-def, simp)

**lemma** atLeast-Suc: atLeast (Suc k) = atLeast k - {k}  
 apply (simp add: atLeast-def)  
 apply (simp add: Suc-le-eq)  
 apply (simp add: order-le-less, blast)  
 done

**lemma** atLeast-Suc-greaterThan: atLeast (Suc k) = greaterThan k  
 by (auto simp add: greaterThan-def atLeast-def less-Suc-eq-le)

**lemma** UN-atLeast-UNIV: (UN m::nat. atLeast m) = UNIV  
 by blast

#### 38.4.4 The Constant atMost

**lemma** atMost-0 [simp]: atMost (0::nat) = {0}  
 by (simp add: atMost-def)

**lemma** atMost-Suc: atMost (Suc k) = insert (Suc k) (atMost k)  
 apply (simp add: atMost-def)  
 apply (simp add: less-Suc-eq order-le-less, blast)  
 done

**lemma** UN-atMost-UNIV: (UN m::nat. atMost m) = UNIV  
 by blast

#### 38.4.5 The Constant atLeastLessThan

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

**lemma** atLeast0LessThan: {0::nat.. $n$ } = {.. $n$ }  
 by (simp add: lessThan-def atLeastLessThan-def)

**lemma** atLeast0AtMost: {0.. $n$ ::nat} = {.. $n$ }  
 by (simp add: atMost-def atLeastAtMost-def)

**declare** atLeast0LessThan[symmetric, code-unfold]  
 atLeast0AtMost[symmetric, code-unfold]

**lemma** atLeastLessThan0: { $m$ .. $0$ ::nat} = {}  
 by (simp add: atLeastLessThan-def)

### 38.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

**lemma** *atLeastLessThanSuc*:

$\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$

**by** (*auto simp add: atLeastLessThan-def*)

**lemma** *atLeastLessThan-singleton [simp]*:  $\{m..<Suc\ m\} = \{m\}$

**by** (*auto simp add: atLeastLessThan-def*)

**lemma** *atLeastLessThanSuc-atLeastAtMost*:  $\{l..<Suc\ u\} = \{l..u\}$

**by** (*simp add: lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def*)

**lemma** *atLeastSucAtMost-greaterThanAtMost*:  $\{Suc\ l..u\} = \{l<..u\}$

**by** (*simp add: atLeast-Suc-greaterThan atLeastAtMost-def  
greaterThanAtMost-def*)

**lemma** *atLeastSucLessThan-greaterThanLessThan*:  $\{Suc\ l..<u\} = \{l<..<u\}$

**by** (*simp add: atLeast-Suc-greaterThan atLeastLessThan-def  
greaterThanLessThan-def*)

**lemma** *atLeastAtMostSuc-conv*:  $m \leq Suc\ n \implies \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$

**by** (*auto simp add: atLeastAtMost-def*)

**lemma** *atLeastLessThan-add-Un*:  $i \leq j \implies \{i..<j+k\} = \{i..<j\} \cup \{j..<j+k::nat\}$

**apply** (*induct k*)

**apply** (*simp-all add: atLeastLessThanSuc*)

**done**

### 38.4.7 Image

**lemma** *image-add-atLeastAtMost*:

$(\%n::nat. n+k) \text{ ` } \{i..j\} = \{i+k..j+k\} \text{ (is } ?A = ?B)$

**proof**

**show**  $?A \subseteq ?B$  **by** *auto*

**next**

**show**  $?B \subseteq ?A$

**proof**

**fix**  $n$  **assume**  $a: n : ?B$

**hence**  $n - k : \{i..j\}$  **by** *auto*

**moreover** **have**  $n = (n - k) + k$  **using**  $a$  **by** *auto*

**ultimately** **show**  $n : ?A$  **by** *blast*

**qed**

**qed**

**lemma** *image-add-atLeastLessThan*:

$(\%n::nat. n+k) \text{ ` } \{i..<j\} = \{i+k..<j+k\} \text{ (is } ?A = ?B)$

**proof**

```

  show ?A ⊆ ?B by auto
next
  show ?B ⊆ ?A
proof
  fix n assume a: n : ?B
  hence n - k : {i..

```

```

    and image-uminus-atMost[simp]: uminus ‘ {.. $x$ } = {.. $-x$ ..}
  proof -
    have uminus ‘ {.. $x$ } = uminus ‘ uminus ‘ {.. $-x$ ..}
      and uminus ‘ {.. $x$ } = uminus ‘ uminus ‘ {.. $-x$ ..} by simp-all
    thus uminus ‘ {.. $x$ } = {.. $-x$ ..} and uminus ‘ {.. $x$ } = {.. $-x$ ..}
      by (simp-all add: image-image
        del: image-uminus-greaterThan image-uminus-atLeast)
  qed

```

```

lemma
  fixes  $x :: 'a$ 
  shows image-uminus-atLeastAtMost[simp]: uminus ‘ { $x..y$ } = {.. $-y$ .. $-x$ }
  and image-uminus-greaterThanAtMost[simp]: uminus ‘ { $x<..y$ } = {.. $-y$ .. $-x$ }
  and image-uminus-atLeastLessThan[simp]: uminus ‘ { $x..<y$ } = {.. $-y$ .. $-x$ }
  and image-uminus-greaterThanLessThan[simp]: uminus ‘ { $x<.. $y$ } = {.. $-y$ .. $-x$ }
  by (simp-all add: atLeastAtMost-def greaterThanAtMost-def atLeastLessThan-def
    greaterThanLessThan-def image-Int[OF inj-uminus] Int-commute)
end$ 
```

### 38.4.8 Finiteness

```

lemma finite-lessThan [iff]: fixes  $k :: nat$  shows finite {.. $k$ }
  by (induct  $k$ ) (simp-all add: lessThan-Suc)

```

```

lemma finite-atMost [iff]: fixes  $k :: nat$  shows finite {.. $k$ }
  by (induct  $k$ ) (simp-all add: atMost-Suc)

```

```

lemma finite-greaterThanLessThan [iff]:
  fixes  $l :: nat$  shows finite { $l<.. $u$ }
  by (simp add: greaterThanLessThan-def)$ 
```

```

lemma finite-atLeastLessThan [iff]:
  fixes  $l :: nat$  shows finite { $l..<u$ }
  by (simp add: atLeastLessThan-def)

```

```

lemma finite-greaterThanAtMost [iff]:
  fixes  $l :: nat$  shows finite { $l<.. $u$ }
  by (simp add: greaterThanAtMost-def)$ 
```

```

lemma finite-atLeastAtMost [iff]:
  fixes  $l :: nat$  shows finite { $l..u$ }
  by (simp add: atLeastAtMost-def)

```

A bounded set of natural numbers is finite.

```

lemma bounded-nat-set-is-finite:
  (ALL  $i:N. i < (n::nat)$ ) ==> finite  $N$ 
  apply (rule finite-subset)
  apply (rule-tac [2] finite-lessThan, auto)
  done

```



A set of natural numbers is finite iff it is bounded.

**lemma** *finite-nat-set-iff-bounded*:

$finite(N::nat\ set) = (EX\ m.\ ALL\ n:N.\ n < m) \text{ (is } ?F = ?B)$

**proof**

assume  $f: ?F$  show  $?B$

using *Max-ge*[*OF*  $\langle ?F \rangle$ , *simplified less-Suc-eq-le*[*symmetric*]] by *blast*

**next**

assume  $?B$  show  $?F$  using  $\langle ?B \rangle$  by (*blast intro: bounded-nat-set-is-finite*)

**qed**

**lemma** *finite-nat-set-iff-bounded-le*:

$finite(N::nat\ set) = (EX\ m.\ ALL\ n:N.\ n \leq m)$

**apply**(*simp add: finite-nat-set-iff-bounded*)

**apply**(*blast dest: less-imp-le-nat le-imp-less-Suc*)

**done**

**lemma** *finite-less-ub*:

$!!f::nat \Rightarrow nat. (!n. n \leq f\ n) \Rightarrow finite\ \{n. f\ n \leq u\}$

**by** (*rule-tac B = {..u}* in *finite-subset*, *auto intro: order-trans*)

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

**lemma** *subset-card-intvl-is-intvl*:

$A \leq \{k..<k+card\ A\} \Rightarrow A = \{k..<k+card\ A\} \text{ (is } PROP\ ?P)$

**proof** *cases*

assume *finite A*

thus *PROP ?P*

**proof**(*induct A rule: finite-linorder-max-induct*)

case *empty* thus *?case* by *auto*

**next**

case (*insert b A*)

moreover hence  $b \sim: A$  by *auto*

moreover have  $A \leq \{k..<k+card\ A\}$  and  $b = k+card\ A$

using  $\langle b \sim: A \rangle$  *insert* by *fastsimp+*

ultimately show *?case* by *auto*

**qed**

**next**

assume  $\sim finite\ A$  thus *PROP ?P* by *simp*

**qed**

### 38.4.9 Proving Inclusions and Equalities between Unions

**lemma** *UN-le-eq-UN0*:

$(\bigcup i \leq n::nat. M\ i) = (\bigcup i \in \{1..n\}. M\ i) \cup M\ 0 \text{ (is } ?A = ?B)$

**proof**

show  $?A \leq ?B$

**proof**

fix  $x$  assume  $x : ?A$

then obtain  $i$  where  $i: i \leq n\ x : M\ i$  by *auto*

```

    show  $x : ?B$ 
  proof(cases i)
    case 0 with i show ?thesis by simp
  next
    case (Suc j) with i show ?thesis by auto
  qed
qed
next
  show  $?B \leq ?A$  by auto
qed

```

```

lemma UN-le-add-shift:
   $(\bigcup_{i \leq n :: \text{nat}} M(i+k)) = (\bigcup_{i \in \{k..n+k\}} M i)$  (is  $?A = ?B$ )
proof
  show  $?A \leq ?B$  by fastsimp
next
  show  $?B \leq ?A$ 
  proof
    fix x assume  $x : ?B$ 
    then obtain i where  $i : \{k..n+k\} \ x : M(i)$  by auto
    hence  $i-k \leq n$  &  $x : M((i-k)+k)$  by auto
    thus  $x : ?A$  by blast
  qed
qed

```

```

lemma UN-UN-finite-eq:  $(\bigcup_{n :: \text{nat}} \bigcup_{i \in \{0..<n\}} A i) = (\bigcup_n A n)$ 
  by (auto simp add: atLeast0LessThan)

```

```

lemma UN-finite-subset:  $(!!n :: \text{nat}. (\bigcup_{i \in \{0..<n\}} A i) \subseteq C) \implies (\bigcup_n A n) \subseteq C$ 
  by (subst UN-UN-finite-eq [symmetric]) blast

```

```

lemma UN-finite2-subset:
   $(!!n :: \text{nat}. (\bigcup_{i \in \{0..<n\}} A i) \subseteq (\bigcup_{i \in \{0..<n+k\}} B i)) \implies (\bigcup_n A n) \subseteq$ 
 $(\bigcup_n B n)$ 
  apply (rule UN-finite-subset)
  apply (subst UN-UN-finite-eq [symmetric, of B])
  apply blast
done

```

```

lemma UN-finite2-eq:
   $(!!n :: \text{nat}. (\bigcup_{i \in \{0..<n\}} A i) = (\bigcup_{i \in \{0..<n+k\}} B i)) \implies (\bigcup_n A n) = (\bigcup_n$ 
 $B n)$ 
  apply (rule subset-antisym)
  apply (rule UN-finite2-subset, blast)
  apply (rule UN-finite2-subset [where k=k])
  apply (force simp add: atLeastLessThan-add-Un [of 0])
done

```

**38.4.10 Cardinality**

**lemma** *card-lessThan* [simp]:  $\text{card } \{..\} = u$   
**by** (induct *u*, simp-all add: *lessThan-Suc*)

**lemma** *card-atMost* [simp]:  $\text{card } \{..u\} = \text{Suc } u$   
**by** (simp add: *lessThan-Suc-atMost* [THEN *sym*])

**lemma** *card-atLeastLessThan* [simp]:  $\text{card } \{l..\} = u - l$   
**apply** (subgoal-tac  $\text{card } \{l..\} = \text{card } \{..)  
**apply** (erule *ssubst*, rule *card-lessThan*)  
**apply** (subgoal-tac ( $\%x. x + l$ ) ‘  $\{..\}$ ’)  
**apply** (erule *subst*)  
**apply** (rule *card-image*)  
**apply** (simp add: *inj-on-def*)  
**apply** (auto simp add: *image-def atLeastLessThan-def lessThan-def*)  
**apply** (rule-tac  $x = x - l$  in *exI*)  
**apply** *arith*  
**done**$

**lemma** *card-atLeastAtMost* [simp]:  $\text{card } \{l..u\} = \text{Suc } u - l$   
**by** (subst *atLeastLessThanSuc-atLeastAtMost* [THEN *sym*], simp)

**lemma** *card-greaterThanAtMost* [simp]:  $\text{card } \{l<..u\} = u - l$   
**by** (subst *atLeastSucAtMost-greaterThanAtMost* [THEN *sym*], simp)

**lemma** *card-greaterThanLessThan* [simp]:  $\text{card } \{l<..\} = u - \text{Suc } l$   
**by** (subst *atLeastSucLessThan-greaterThanLessThan* [THEN *sym*], simp)

**lemma** *ex-bij-betw-nat-finite*:  
 $\text{finite } M \implies \exists h. \text{bij-betw } h \{0..  
**apply** (drule *finite-imp-nat-seg-image-inj-on*)  
**apply** (auto simp: *atLeast0LessThan*[*symmetric*] *lessThan-def*[*symmetric*] *card-image*  
*bij-betw-def*)  
**done**$

**lemma** *ex-bij-betw-finite-nat*:  
 $\text{finite } M \implies \exists h. \text{bij-betw } h M \{0..  
**by** (blast dest: *ex-bij-betw-nat-finite* *bij-betw-inv*)$

**lemma** *finite-same-card-bij*:  
 $\text{finite } A \implies \text{finite } B \implies \text{card } A = \text{card } B \implies \text{EX } h. \text{bij-betw } h A B$   
**apply** (drule *ex-bij-betw-finite-nat*)  
**apply** (drule *ex-bij-betw-nat-finite*)  
**apply** (auto intro!: *bij-betw-trans*)  
**done**

**lemma** *ex-bij-betw-nat-finite-1*:  
 $\text{finite } M \implies \exists h. \text{bij-betw } h \{1 .. \text{card } M\} M$   
**by** (rule *finite-same-card-bij*) auto

### 38.5 Intervals of integers

**lemma** *atLeastLessThanPlusOne-atLeastAtMost-int*:  $\{l..<u+1\} = \{l..(u::int)\}$   
**by** (*auto simp add: atLeastAtMost-def atLeastLessThan-def*)

**lemma** *atLeastPlusOneAtMost-greaterThanAtMost-int*:  $\{l+1..u\} = \{l<..(u::int)\}$   
**by** (*auto simp add: atLeastAtMost-def greaterThanAtMost-def*)

**lemma** *atLeastPlusOneLessThan-greaterThanLessThan-int*:  
 $\{l+1..<u\} = \{l<..<u::int\}$   
**by** (*auto simp add: atLeastLessThan-def greaterThanLessThan-def*)

#### 38.5.1 Finiteness

**lemma** *image-atLeastZeroLessThan-int*:  $0 \leq u \implies$   
 $\{(0::int)..<u\} = \text{int} \text{ ‘ } \{..<\text{nat } u\}$   
**apply** (*unfold image-def lessThan-def*)  
**apply** *auto*  
**apply** (*rule-tac x = nat x in exI*)  
**apply** (*auto simp add: zless-nat-eq-int-zless [THEN sym]*)  
**done**

**lemma** *finite-atLeastZeroLessThan-int*: *finite*  $\{(0::int)..<u\}$   
**apply** (*case-tac 0 ≤ u*)  
**apply** (*subst image-atLeastZeroLessThan-int, assumption*)  
**apply** (*rule finite-imageI*)  
**apply** *auto*  
**done**

**lemma** *finite-atLeastLessThan-int [iff]*: *finite*  $\{l..<u::int\}$   
**apply** (*subgoal-tac (%x. x + 1) ‘ {0..<u-l} = {l..<u}*)  
**apply** (*erule subst*)  
**apply** (*rule finite-imageI*)  
**apply** (*rule finite-atLeastZeroLessThan-int*)  
**apply** (*rule image-add-int-atLeastLessThan*)  
**done**

**lemma** *finite-atLeastAtMost-int [iff]*: *finite*  $\{l..(u::int)\}$   
**by** (*subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym], simp*)

**lemma** *finite-greaterThanAtMost-int [iff]*: *finite*  $\{l<..(u::int)\}$   
**by** (*subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp*)

**lemma** *finite-greaterThanLessThan-int [iff]*: *finite*  $\{l<..<u::int\}$   
**by** (*subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp*)

#### 38.5.2 Cardinality

**lemma** *card-atLeastZeroLessThan-int*:  $\text{card } \{(0::int)..<u\} = \text{nat } u$   
**apply** (*case-tac 0 ≤ u*)

```

apply (subst image-atLeastZeroLessThan-int, assumption)
apply (subst card-image)
apply (auto simp add: inj-on-def)
done

```

```

lemma card-atLeastLessThan-int [simp]: card {l..} = nat (u - l)
apply (subgoal-tac card {l..} = card {0..-l})
apply (erule ssubst, rule card-atLeastZeroLessThan-int)
apply (subgoal-tac (%x. x + l) ‘ {0..-l} = {l..})
apply (erule ssubst)
apply (rule card-image)
apply (simp add: inj-on-def)
apply (rule image-add-int-atLeastLessThan)
done

```

```

lemma card-atLeastAtMost-int [simp]: card {l..u} = nat (u - l + 1)
apply (subst atLeastLessThanPlusOne-atLeastAtMost-int [THEN sym])
apply (auto simp add: algebra-simps)
done

```

```

lemma card-greaterThanAtMost-int [simp]: card {l<..u} = nat (u - l)
by (subst atLeastPlusOneAtMost-greaterThanAtMost-int [THEN sym], simp)

```

```

lemma card-greaterThanLessThan-int [simp]: card {l<..} = nat (u - (l + 1))
by (subst atLeastPlusOneLessThan-greaterThanLessThan-int [THEN sym], simp)

```

```

lemma finite-M-bounded-by-nat: finite {k. P k ∧ k < (i::nat)}
proof -
  have {k. P k ∧ k < i} ⊆ {..} by auto
  with finite-lessThan[of i] show ?thesis by (simp add: finite-subset)
qed

```

```

lemma card-less:
assumes zero-in-M: 0 ∈ M
shows card {k ∈ M. k < Suc i} ≠ 0
proof -
  from zero-in-M have {k ∈ M. k < Suc i} ≠ {} by auto
  with finite-M-bounded-by-nat show ?thesis by (auto simp add: card-eq-0-iff)
qed

```

```

lemma card-less-Suc2: 0 ∉ M ⇒ card {k. Suc k ∈ M ∧ k < i} = card {k ∈
M. k < Suc i}
apply (rule card-bij-eq [of Suc - - λx. x - Suc 0])
apply simp
apply fastsimp
apply auto
apply (rule inj-on-diff-nat)
apply auto
apply (case-tac x)

```

```

apply auto
apply (case-tac xa)
apply auto
apply (case-tac xa)
apply auto
done

```

**lemma** *card-less-Suc*:

```

  assumes zero-in-M:  $0 \in M$ 
  shows  $Suc\ (card\ \{k.\ Suc\ k \in M \wedge k < i\}) = card\ \{k \in M.\ k < Suc\ i\}$ 
proof –
  from assms have  $a: 0 \in \{k \in M.\ k < Suc\ i\}$  by simp
  hence  $c: \{k \in M.\ k < Suc\ i\} = insert\ 0\ (\{k \in M.\ k < Suc\ i\} - \{0\})$ 
  by (auto simp only: insert-Diff)
  have  $b: \{k \in M.\ k < Suc\ i\} - \{0\} = \{k \in M - \{0\}.\ k < Suc\ i\}$  by auto
  from finite-M-bounded-by-nat[of  $\lambda x. x \in M\ Suc\ i$ ] have  $Suc\ (card\ \{k.\ Suc\ k \in$ 
 $M \wedge k < i\}) = card\ (insert\ 0\ (\{k \in M.\ k < Suc\ i\} - \{0\}))$ 
  apply (subst card-insert)
  apply simp-all
  apply (subst b)
  apply (subst card-less-Suc2[symmetric])
  apply simp-all
  done
  with c show ?thesis by simp
qed

```

## 38.6 Lemmas useful with the summation operator *setsum*

For examples, see `Algebra/poly/UnivPoly2.thy`

### 38.6.1 Disjoint Unions

Singletons and open intervals

**lemma** *ivl-disj-un-singleton*:

```

   $\{l::'a::linorder\} \ Un\ \{l<..\} = \{l..\}$ 
   $\{..\} \ Un\ \{u::'a::linorder\} = \{..u\}$ 
   $(l::'a::linorder) < u ==> \{l\} \ Un\ \{l<..\} = \{l..\}$ 
   $(l::'a::linorder) < u ==> \{l<..\} \ Un\ \{u\} = \{l<..u\}$ 
   $(l::'a::linorder) <= u ==> \{l\} \ Un\ \{l<..u\} = \{l..u\}$ 
   $(l::'a::linorder) <= u ==> \{l..\} \ Un\ \{u\} = \{l..u\}$ 
by auto

```

One- and two-sided intervals

**lemma** *ivl-disj-un-one*:

```

   $(l::'a::linorder) < u ==> \{..l\} \ Un\ \{l<..\} = \{..\}$ 
   $(l::'a::linorder) <= u ==> \{..\} \ Un\ \{l<..\} = \{..\}$ 
   $(l::'a::linorder) <= u ==> \{..l\} \ Un\ \{l<..u\} = \{..u\}$ 
   $(l::'a::linorder) <= u ==> \{..\} \ Un\ \{l..u\} = \{..u\}$ 

```

$(l::'a::linorder) \leq u \implies \{l<..u\} \text{ Un } \{u<..\} = \{l<..\}$   
 $(l::'a::linorder) < u \implies \{l<..\} \text{ Un } \{u..\} = \{l<..\}$   
 $(l::'a::linorder) \leq u \implies \{l..u\} \text{ Un } \{u<..\} = \{l..\}$   
 $(l::'a::linorder) < u \implies \{l..\} \text{ Un } \{u..\} = \{l..\}$   
**by** *auto*

Two- and two-sided intervals

**lemma** *ivl-disj-un-two*:

$\llbracket (l::'a::linorder) < m; m \leq u \rrbracket \implies \{l<..\} \text{ Un } \{m..\} = \{l<..\}$   
 $\llbracket (l::'a::linorder) \leq m; m < u \rrbracket \implies \{l<..\} \text{ Un } \{m<..\} = \{l<..\}$   
 $\llbracket (l::'a::linorder) \leq m; m \leq u \rrbracket \implies \{l<..\} \text{ Un } \{m..\} = \{l<..\}$   
 $\llbracket (l::'a::linorder) < m; m < u \rrbracket \implies \{l..\} \text{ Un } \{m<..\} = \{l..\}$   
 $\llbracket (l::'a::linorder) < m; m \leq u \rrbracket \implies \{l<..\} \text{ Un } \{m..\} = \{l<..\}$   
 $\llbracket (l::'a::linorder) \leq m; m < u \rrbracket \implies \{l<..\} \text{ Un } \{m<..\} = \{l<..\}$   
 $\llbracket (l::'a::linorder) \leq m; m \leq u \rrbracket \implies \{l<..\} \text{ Un } \{m..\} = \{l<..\}$   
 $\llbracket (l::'a::linorder) < m; m < u \rrbracket \implies \{l..\} \text{ Un } \{m<..\} = \{l..\}$   
**by** *auto*

**lemmas** *ivl-disj-un* = *ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two*

### 38.6.2 Disjoint Intersections

One- and two-sided intervals

**lemma** *ivl-disj-int-one*:

$\{..l::'a::order\} \text{ Int } \{l<..\} = \{\}$   
 $\{..\} \text{ Int } \{l..\} = \{\}$   
 $\{..l\} \text{ Int } \{l<..\} = \{\}$   
 $\{..\} \text{ Int } \{l..\} = \{\}$   
 $\{l<..\} \text{ Int } \{u<..\} = \{\}$   
 $\{l<..\} \text{ Int } \{u..\} = \{\}$   
 $\{l..\} \text{ Int } \{u<..\} = \{\}$   
 $\{l..\} \text{ Int } \{u..\} = \{\}$   
**by** *auto*

Two- and two-sided intervals

**lemma** *ivl-disj-int-two*:

$\{l::'a::order<..\} \text{ Int } \{m..\} = \{\}$   
 $\{l<..\} \text{ Int } \{m<..\} = \{\}$   
 $\{l..\} \text{ Int } \{m..\} = \{\}$   
 $\{l..\} \text{ Int } \{m<..\} = \{\}$   
 $\{l<..\} \text{ Int } \{m..\} = \{\}$   
 $\{l<..\} \text{ Int } \{m<..\} = \{\}$   
 $\{l..\} \text{ Int } \{m..\} = \{\}$   
 $\{l..\} \text{ Int } \{m<..\} = \{\}$   
**by** *auto*

**lemmas** *ivl-disj-int* = *ivl-disj-int-one ivl-disj-int-two*

### 38.6.3 Some Differences

**lemma** *ivl-diff* [*simp*]:  
 $i \leq n \implies \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$   
**by**(*auto*)

### 38.6.4 Some Subset Conditions

**lemma** *ivl-subset* [*simp, no-atp*]:  
 $(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \ \& \ j \leq (n::'a::linorder))$   
**apply**(*auto simp: linorder-not-le*)  
**apply**(*rule ccontr*)  
**apply**(*insert linorder-le-less-linear* [of *i n*])  
**apply**(*clarsimp simp: linorder-not-le*)  
**apply**(*fastsimp*)  
**done**

## 38.7 Summation indexed over intervals

### syntax

-*from-to-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* ((*SUM* - = -..*./* -) [*0,0,0,10*] 10)  
 -*from-upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* ((*SUM* - = -..*<./* -) [*0,0,0,10*] 10)  
 -*upt-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* ((*SUM* -<..*./* -) [*0,0,10*] 10)  
 -*upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* ((*SUM* -<=..*./* -) [*0,0,10*] 10)

### syntax (*xsymbols*)

-*from-to-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  - = -..*./* -) [*0,0,0,10*] 10)  
 -*from-upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  - = -..*<./* -) [*0,0,0,10*] 10)  
 -*upt-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  -<..*./* -) [*0,0,10*] 10)  
 -*upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  -<=..*./* -) [*0,0,10*] 10)

### syntax (*HTML output*)

-*from-to-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  - = -..*./* -) [*0,0,0,10*] 10)  
 -*from-upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  - = -..*<./* -) [*0,0,0,10*] 10)  
 -*upt-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  -<..*./* -) [*0,0,10*] 10)  
 -*upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b* (( $\sum$  -<=..*./* -) [*0,0,10*] 10)

### syntax (*latex-sum output*)

-*from-to-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b*  
 (( $\sum$  - = -) [*0,0,0,10*] 10)  
 -*from-upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b*  
 (( $\sum$  -<= -) [*0,0,0,10*] 10)  
 -*upt-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b*  
 (( $\sum$  -< -) [*0,0,10*] 10)  
 -*upto-setsum* :: *idt*  $\Rightarrow$  '*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*b*  
 (( $\sum$  -  $\leq$  -) [*0,0,10*] 10)

### translations

$\sum_{x=a..b} t == \text{CONST setsum } (\%x. t) \{a..b\}$   
 $\sum_{x=a..<b} t == \text{CONST setsum } (\%x. t) \{a..<b\}$



$$\begin{aligned} \sum_{i \leq n}. t &== \text{CONST setsum } (\lambda i. t) \{..n\} \\ \sum_{i < n}. t &== \text{CONST setsum } (\lambda i. t) \{..<n\} \end{aligned}$$

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	L <sup>A</sup> T <sub>E</sub> X
$\sum_{x \in \{a..b\}}. e$	$\sum x = a..b. e$	$\sum_{x=a}^b e$
$\sum_{x \in \{a..<b\}}. e$	$\sum x = a..<b. e$	$\sum_{x=a}^{<b} e$
$\sum_{x \in \{..b\}}. e$	$\sum x \leq b. e$	$\sum_{x \leq b} e$
$\sum_{x \in \{..<b\}}. e$	$\sum x < b. e$	$\sum_{x < b} e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode = latex-sum* in antiquotations). It is not the default L<sup>A</sup>T<sub>E</sub>X output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use  $\sum x = 0..<n. e$  rather than  $\sum x < n. e$ : *setsum* may not provide all lemmas available for  $\{m..<n\}$  also in the special form for  $\{..<n\}$ .

This congruence rule should be used for sums over intervals as the standard theorem *setsum-cong* does not work well with the simplifier who adds the unsimplified premise  $x \in B$  to the context.

**lemma** *setsum-ivl-cong*:

$$\llbracket a = c; b = d; !!x. \llbracket c \leq x; x < d \rrbracket \implies f x = g x \rrbracket \implies \text{setsum } f \{a..<b\} = \text{setsum } g \{c..<d\}$$

**by** (*rule setsum-cong, simp-all*)

**lemma** *setsum-atMost-Suc[simp]*:  $(\sum i \leq \text{Suc } n. f i) = (\sum i \leq n. f i) + f(\text{Suc } n)$   
**by** (*simp add:atMost-Suc add-ac*)

**lemma** *setsum-lessThan-Suc[simp]*:  $(\sum i < \text{Suc } n. f i) = (\sum i < n. f i) + f n$   
**by** (*simp add:lessThan-Suc add-ac*)

**lemma** *setsum-cl-ivl-Suc[simp]*:

$$\text{setsum } f \{m.. \text{Suc } n\} = (\text{if } \text{Suc } n < m \text{ then } 0 \text{ else } \text{setsum } f \{m..n\} + f(\text{Suc } n))$$

**by** (*auto simp:add-ac atLeastAtMostSuc-conv*)

**lemma** *setsum-op-ivl-Suc[simp]*:

$$\text{setsum } f \{m..<\text{Suc } n\} = (\text{if } n < m \text{ then } 0 \text{ else } \text{setsum } f \{m..<n\} + f(n))$$

**by** (*auto simp:add-ac atLeastLessThanSuc*)

```

lemma setsum-head:
  fixes n :: nat
  assumes mn: m ≤ n
  shows (∑ x∈{m..n}. P x) = P m + (∑ x∈{m<..n}. P x) (is ?lhs = ?rhs)
proof -
  from mn
  have {m..n} = {m} ∪ {m<..n}
    by (auto intro: ivl-disj-un-singleton)
  hence ?lhs = (∑ x∈{m} ∪ {m<..n}. P x)
    by (simp add: atLeast0LessThan)
  also have ... = ?rhs by simp
  finally show ?thesis .
qed

lemma setsum-head-Suc:
  m ≤ n ⇒ setsum f {m..n} = f m + setsum f {Suc m..n}
by (simp add: setsum-head atLeastSucAtMost-greaterThanAtMost)

lemma setsum-head-upt-Suc:
  m < n ⇒ setsum f {m..apply (insert setsum-head-Suc [of m n - Suc 0 f])
apply (simp add: atLeastLessThanSuc-atLeastAtMost [symmetric] algebra-simps)
done

lemma setsum-ub-add-nat: assumes (m::nat) ≤ n + 1
  shows setsum f {m..n + p} = setsum f {m..n} + setsum f {n + 1..n + p}
proof -
  have {m .. n+p} = {m..n} ∪ {n+1..n+p} using ⟨m ≤ n+1⟩ by auto
  thus ?thesis by (auto simp: ivl-disj-int setsum-Un-disjoint
    atLeastSucAtMost-greaterThanAtMost)
qed

lemma setsum-add-nat-ivl: [ m ≤ n; n ≤ p ] ⇒
  setsum f {m..by (simp add: setsum-Un-disjoint [symmetric] ivl-disj-int ivl-disj-un)

lemma setsum-diff-nat-ivl:
  fixes f :: nat ⇒ 'a::ab-group-add
  shows [ m ≤ n; n ≤ p ] ⇒
    setsum f {m..using setsum-add-nat-ivl [of m n p f, symmetric]
apply (simp add: add-ac)
done

lemma setsum-natinterval-diff:
  fixes f :: nat ⇒ ('a::ab-group-add)
  shows setsum (λk. f k - f(k + 1)) {(m::nat) .. n} =
    (if m ≤ n then f m - f(n + 1) else 0)
by (induct n, auto simp add: algebra-simps not-le le-Suc-eq)

```

**lemmas** *setsum-restrict-set'* = *setsum-restrict-set*[*unfolded Int-def*]

**lemma** *setsum-setsum-restrict*:

*finite S*  $\implies$  *finite T*  $\implies$  *setsum* ( $\lambda x. \text{setsum } (\lambda y. f\ x\ y) \{y. y \in T \wedge R\ x\ y\}$ ) *S*  
 = *setsum* ( $\lambda y. \text{setsum } (\lambda x. f\ x\ y) \{x. x \in S \wedge R\ x\ y\}$ ) *T*  
**by** (*simp add: setsum-restrict-set'*[*unfolded mem-def*] *mem-def*)  
 (*rule setsum-commute*)

**lemma** *setsum-image-gen*: **assumes** *fS: finite S*

**shows** *setsum g S* = *setsum* ( $\lambda y. \text{setsum } g \{x. x \in S \wedge f\ x = y\}$ ) (*f* ‘ *S*)

**proof** –

**{ fix x assume x  $\in$  S then have  $\{y. y \in f'S \wedge f\ x = y\} = \{f\ x\}$  by auto }**

**hence *setsum g S* = *setsum* ( $\lambda x. \text{setsum } (\lambda y. g\ x) \{y. y \in f'S \wedge f\ x = y\}$ ) *S***

**by *simp***

**also have ... = *setsum* ( $\lambda y. \text{setsum } g \{x. x \in S \wedge f\ x = y\}$ ) (*f* ‘ *S*)**

**by (*rule setsum-setsum-restrict*[*OF fS finite-imageI*[*OF fS*]])**

**finally show ?thesis .**

**qed**

**lemma** *setsum-le-included*:

**fixes** *f* :: 'a  $\Rightarrow$  'b::ordered-comm-monoid-add

**assumes** *finite s finite t*

**and**  $\forall y \in t. 0 \leq g\ y \ (\forall x \in s. \exists y \in t. i\ y = x \wedge f\ x \leq g\ y)$

**shows** *setsum f s*  $\leq$  *setsum g t*

**proof** –

**have *setsum f s*  $\leq$  *setsum* ( $\lambda y. \text{setsum } g \{x. x \in t \wedge i\ x = y\}$ ) *s***

**proof** (*rule setsum-mono*)

**fix y assume y  $\in$  s**

**with *assms* obtain z where z: z  $\in$  t y = i z f y  $\leq$  g z by auto**

**with *assms* show f y  $\leq$  *setsum g* {x  $\in$  t. i x = y} (*is* ?A y  $\leq$  ?B y)**

**using *order-trans*[of ?A (i z) *setsum g* {z} ?B (i z), *intro*]**

**by (*auto intro!*: *setsum-mono2*)**

**qed**

**also have ...  $\leq$  *setsum* ( $\lambda y. \text{setsum } g \{x. x \in t \wedge i\ x = y\}$ ) (*i* ‘ *t*)**

**using *assms*(2-4) by (*auto intro!*: *setsum-mono2 setsum-nonneg*)**

**also have ...  $\leq$  *setsum g t***

**using *assms* by (*auto simp: setsum-image-gen*[*symmetric*])**

**finally show ?thesis .**

**qed**

**lemma** *setsum-multicount-gen*:

**assumes** *finite s finite t*  $\forall j \in t. (\text{card } \{i \in s. R\ i\ j\} = k\ j)$

**shows** *setsum* ( $\lambda i. (\text{card } \{j \in t. R\ i\ j\})$ ) *s* = *setsum k t* (*is* ?l = ?r)

**proof** –

**have ?l = *setsum* ( $\lambda i. \text{setsum } (\lambda x. 1) \{j \in t. R\ i\ j\}$ ) *s* by auto**

**also have ... = ?r unfolding *setsum-setsum-restrict*[*OF assms*(1-2)]**

**using *assms*(3) by auto**

**finally show ?thesis .**

qed

**lemma** *setsum-multicount*:

**assumes** *finite S finite T*  $\forall j \in T. (\text{card } \{i \in S. R \ i \ j\} = k)$

**shows**  $\text{setsum } (\lambda i. \text{card } \{j \in T. R \ i \ j\}) \ S = k * \text{card } T$  (**is**  $?l = ?r$ )

**proof** –

**have**  $?l = \text{setsum } (\lambda i. k) \ T$  **by** (*rule setsum-multicount-gen*)(*auto simp:assms*)

**also have**  $\dots = ?r$  **by** (*simp add: mult-commute*)

**finally show**  $?thesis$  **by** *auto*

qed

### 38.8 Shifting bounds

**lemma** *setsum-shift-bounds-nat-ivl*:

$\text{setsum } f \ \{m+k..<n+k\} = \text{setsum } (\%i. f(i+k)) \ \{m..<n::\text{nat}\}$

**by** (*induct n, auto simp:atLeastLessThanSuc*)

**lemma** *setsum-shift-bounds-cl-nat-ivl*:

$\text{setsum } f \ \{m+k..n+k\} = \text{setsum } (\%i. f(i+k)) \ \{m..n::\text{nat}\}$

**apply** (*insert setsum-reindex[OF inj-on-add-nat, where h=f and B = {m..n}]*)

**apply** (*simp add:image-add-atLeastAtMost o-def*)

**done**

**corollary** *setsum-shift-bounds-cl-Suc-ivl*:

$\text{setsum } f \ \{\text{Suc } m..\text{Suc } n\} = \text{setsum } (\%i. f(\text{Suc } i)) \ \{m..n\}$

**by** (*simp add:setsum-shift-bounds-cl-nat-ivl[where k=Suc 0, simplified]*)

**corollary** *setsum-shift-bounds-Suc-ivl*:

$\text{setsum } f \ \{\text{Suc } m..<\text{Suc } n\} = \text{setsum } (\%i. f(\text{Suc } i)) \ \{m..<n\}$

**by** (*simp add:setsum-shift-bounds-nat-ivl[where k=Suc 0, simplified]*)

**lemma** *setsum-shift-lb-Suc0-0*:

$f(0::\text{nat}) = (0::\text{nat}) \implies \text{setsum } f \ \{\text{Suc } 0..k\} = \text{setsum } f \ \{0..k\}$

**by**(*simp add:setsum-head-Suc*)

**lemma** *setsum-shift-lb-Suc0-0-upt*:

$f(0::\text{nat}) = 0 \implies \text{setsum } f \ \{\text{Suc } 0..<k\} = \text{setsum } f \ \{0..<k\}$

**apply**(*cases k*)**apply** *simp*

**apply**(*simp add:setsum-head-upt-Suc*)

**done**

### 38.9 The formula for geometric sums

**lemma** *geometric-sum*:

**assumes**  $x \neq 1$

**shows**  $(\sum_{i=0..<n.} x^i) = (x^n - 1) / (x - 1::'a::\text{field})$

**proof** –

**from** *assms* **obtain**  $y$  **where**  $y = x - 1$  **and**  $y \neq 0$  **by** *simp-all*

**moreover have**  $(\sum_{i=0..<n.} (y+1)^i) = ((y+1)^n - 1) / y$

**proof** (*induct n*)

```

    case 0 then show ?case by simp
  next
    case (Suc n)
    moreover with ⟨y ≠ 0⟩ have (1 + y) ^ n = (y * inverse y) * (1 + y) ^ n
  by simp
    ultimately show ?case by (simp add: field-simps divide-inverse)
  qed
  ultimately show ?thesis by simp
qed

```

### 38.10 The formula for arithmetic sums

**lemma** *gauss-sum*:

$$((1::'a::\text{comm-semiring-1}) + 1) * (\sum i \in \{1..n\}. \text{of-nat } i) = \text{of-nat } n * ((\text{of-nat } n) + 1)$$

**proof** (*induct n*)

```

  case 0
  show ?case by simp
next
  case (Suc n)
  then show ?case by (simp add: algebra-simps)
qed

```

**theorem** *arith-series-general*:

$$((1::'a::\text{comm-semiring-1}) + 1) * (\sum i \in \{..<n\}. a + \text{of-nat } i * d) = \text{of-nat } n * (a + (a + \text{of-nat } (n - 1) * d))$$

**proof** *cases*

```

  assume ngt1: n > 1
  let ?I = λi. of-nat i and ?n = of-nat n
  have
    (∑ i ∈ {..<n}. a + ?I i * d) =
      ((∑ i ∈ {..<n}. a) + (∑ i ∈ {..<n}. ?I i * d))
    by (rule setsum-addf)
  also from ngt1 have ... = ?n * a + (∑ i ∈ {..<n}. ?I i * d) by simp
  also from ngt1 have ... = (?n * a + d * (∑ i ∈ {1..<n}. ?I i))
    unfolding One-nat-def
    by (simp add: setsum-right-distrib atLeast0LessThan[symmetric] setsum-shift-lb-Suc0-0-upt
mult-ac)
  also have (1+1)*... = (1+1)*?n*a + d*(1+1)*(∑ i ∈ {1..<n}. ?I i)
    by (simp add: left-distrib right-distrib)
  also from ngt1 have {1..<n} = {1..n - 1}
    by (cases n) (auto simp: atLeastLessThanSuc-atLeastAtMost)
  also from ngt1
  have (1+1)*?n*a + d*(1+1)*(∑ i ∈ {1..n - 1}. ?I i) = ((1+1)*?n*a + d*?I
(n - 1)*?I n)
    by (simp only: mult-ac gauss-sum [of n - 1], unfold One-nat-def)
    (simp add: mult-ac trans [OF add-commute of-nat-Suc [symmetric]])
  finally show ?thesis by (simp add: algebra-simps)
next

```

```

assume  $\neg(n > 1)$ 
hence  $n = 1 \vee n = 0$  by auto
thus ?thesis by (auto simp: algebra-simps)
qed

```

**lemma** *arith-series-nat*:

```

   $Suc (Suc\ 0) * (\sum i \in \{..<n\}. a + i * d) = n * (a + (a + (n - 1) * d))$ 
proof –
  have
     $((1::nat) + 1) * (\sum i \in \{..<n::nat\}. a + of\_nat(i) * d) =$ 
     $of\_nat(n) * (a + (a + of\_nat(n - 1) * d))$ 
    by (rule arith-series-general)
  thus ?thesis
    unfolding One-nat-def by auto
qed

```

**lemma** *arith-series-int*:

```

   $(2::int) * (\sum i \in \{..<n\}. a + of\_nat\ i * d) =$ 
   $of\_nat\ n * (a + (a + of\_nat(n - 1) * d))$ 
proof –
  have
     $((1::int) + 1) * (\sum i \in \{..<n\}. a + of\_nat\ i * d) =$ 
     $of\_nat(n) * (a + (a + of\_nat(n - 1) * d))$ 
    by (rule arith-series-general)
  thus ?thesis by simp
qed

```

**lemma** *sum-diff-distrib*:

```

  fixes  $P::nat \Rightarrow nat$ 
  shows
     $\forall x. Q\ x \leq P\ x \implies$ 
     $(\sum x < n. P\ x) - (\sum x < n. Q\ x) = (\sum x < n. P\ x - Q\ x)$ 
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc n)

```

```

  let ?lhs =  $(\sum x < n. P\ x) - (\sum x < n. Q\ x)$ 
  let ?rhs =  $\sum x < n. P\ x - Q\ x$ 

```

```

  from Suc have ?lhs = ?rhs by simp

```

**moreover**

```

  from Suc have ?lhs +  $P\ n - Q\ n = ?rhs + (P\ n - Q\ n)$  by simp

```

**moreover**

**from** *Suc* **have**

```

   $(\sum x < n. P\ x) + P\ n - ((\sum x < n. Q\ x) + Q\ n) = ?rhs + (P\ n - Q\ n)$ 
  by (subst diff-diff-left[symmetric],
    subst diff-add-assoc2)
  (auto simp: diff-add-assoc2 intro: setsum-mono)

```

```

ultimately
show ?case by simp
qed

```

### 38.11 Products indexed over intervals

**syntax**

```

-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((PROD - = ..-/ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((PROD - = ..<-/ -) [0,0,0,10]
10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((PROD -<-/ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((PROD -<= -/ -) [0,0,10] 10)

```

**syntax** (*xsymbols*)

```

-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = ..-/ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = ..<-/ -) [0,0,0,10]
10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -<-/ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -<= -/ -) [0,0,10] 10)

```

**syntax** (*HTML output*)

```

-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = ..-/ -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b ((∏ - = ..<-/ -) [0,0,0,10]
10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -<-/ -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b ((∏ -<= -/ -) [0,0,10] 10)

```

**syntax** (*latex-prod output*)

```

-from-to-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b
((∏ - = -) [0,0,0,10] 10)
-from-upto-setprod :: idt ⇒ 'a ⇒ 'a ⇒ 'b ⇒ 'b
((∏ - < -) [0,0,0,10] 10)
-upt-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b
((∏ - < -) [0,0,10] 10)
-upto-setprod :: idt ⇒ 'a ⇒ 'b ⇒ 'b
((∏ - ≤ -) [0,0,10] 10)

```

**translations**

```

∏ x=a..b. t == CONST setprod (%x. t) {a..b}
∏ x=a..<b. t == CONST setprod (%x. t) {a..<b}
∏ i≤n. t == CONST setprod (λi. t) {..n}
∏ i<n. t == CONST setprod (λi. t) {..<n}

```

### 38.12 Transfer setup

**lemma** *transfer-nat-int-set-functions:*

```

{..n} = nat ‘ {0..int n}
{m..n} = nat ‘ {int m..int n}
apply (auto simp add: image-def)
apply (rule-tac x = int x in bexI)
apply auto
apply (rule-tac x = int x in bexI)
apply auto

```

done

**lemma** *transfer-nat-int-set-function-closures*:

$x \geq 0 \implies \text{nat-set } \{x..y\}$

**by** (*simp add: nat-set-def*)

**declare** *transfer-morphism-nat-int*[*transfer add*

*return: transfer-nat-int-set-functions*

*transfer-nat-int-set-function-closures*

]

**lemma** *transfer-int-nat-set-functions*:

$\text{is-nat } m \implies \text{is-nat } n \implies \{m..n\} = \text{int } ' \{ \text{nat } m.. \text{nat } n \}$

**by** (*simp only: is-nat-def transfer-nat-int-set-functions*

*transfer-nat-int-set-function-closures*

*transfer-nat-int-set-return-embed nat-0-le*

*cong: transfer-nat-int-set-cong*)

**lemma** *transfer-int-nat-set-function-closures*:

$\text{is-nat } x \implies \text{nat-set } \{x..y\}$

**by** (*simp only: transfer-nat-int-set-function-closures is-nat-def*)

**declare** *transfer-morphism-int-nat*[*transfer add*

*return: transfer-int-nat-set-functions*

*transfer-int-nat-set-function-closures*

]

end

## 39 Presburger: Decision Procedure for Presburger Arithmetic

**theory** *Presburger*

**imports** *Groebner-Basis SetInterval*

**uses**

*Tools/Qelim/qelim.ML*

*Tools/Qelim/cooper-procedure.ML*

*(Tools/Qelim/cooper.ML)*

**begin**

### 39.1 The $-\infty$ and $+\infty$ Properties

**lemma** *minf*:

$\llbracket \exists (z :: 'a::\text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket$

$\implies \exists z. \forall x < z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$

$\llbracket \exists (z :: 'a::\text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket$

$\implies \exists z. \forall x < z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$



$\exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x = t) = \text{False}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x \neq t) = \text{True}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x < t) = \text{True}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x \leq t) = \text{True}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x > t) = \text{False}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x \geq t) = \text{False}$   
 $\exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x < z. (d \text{ dvd } x + s) = (d \text{ dvd } x + s)$   
 $\exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x < z. (\neg d \text{ dvd } x + s) = (\neg d \text{ dvd } x + s)$   
 $\exists z. \forall x < z. F = F$   
**by** ((erule exE, erule exE, rule-tac x=min z za **in** exI, simp)+, (rule-tac x=t **in** exI, fastsimp)+) simp-all

**lemma** pinf:

$\llbracket \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. P \ x = P' \ x; \exists z. \forall x > z. Q \ x = Q' \ x \rrbracket$   
 $\implies \exists z. \forall x > z. (P \ x \wedge Q \ x) = (P' \ x \wedge Q' \ x)$   
 $\llbracket \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. P \ x = P' \ x; \exists z. \forall x > z. Q \ x = Q' \ x \rrbracket$   
 $\implies \exists z. \forall x > z. (P \ x \vee Q \ x) = (P' \ x \vee Q' \ x)$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x = t) = \text{False}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \neq t) = \text{True}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x < t) = \text{False}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \leq t) = \text{False}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x > t) = \text{True}$   
 $\exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \geq t) = \text{True}$   
 $\exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x > z. (d \text{ dvd } x + s) = (d \text{ dvd } x + s)$   
 $\exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x > z. (\neg d \text{ dvd } x + s) = (\neg d \text{ dvd } x + s)$   
 $\exists z. \forall x > z. F = F$   
**by** ((erule exE, erule exE, rule-tac x=max z za **in** exI, simp)+, (rule-tac x=t **in** exI, fastsimp)+) simp-all

**lemma** inf-period:

$\llbracket \forall x \ k. P \ x = P \ (x - k * D); \forall x \ k. Q \ x = Q \ (x - k * D) \rrbracket$   
 $\implies \forall x \ k. (P \ x \wedge Q \ x) = (P \ (x - k * D) \wedge Q \ (x - k * D))$   
 $\llbracket \forall x \ k. P \ x = P \ (x - k * D); \forall x \ k. Q \ x = Q \ (x - k * D) \rrbracket$   
 $\implies \forall x \ k. (P \ x \vee Q \ x) = (P \ (x - k * D) \vee Q \ (x - k * D))$   
 $(d :: 'a::\{\text{comm-ring}, \text{Rings.dvd}\}) \text{ dvd } D \implies \forall x \ k. (d \text{ dvd } x + t) = (d \text{ dvd } (x - k * D) + t)$   
 $(d :: 'a::\{\text{comm-ring}, \text{Rings.dvd}\}) \text{ dvd } D \implies \forall x \ k. (\neg d \text{ dvd } x + t) = (\neg d \text{ dvd } (x - k * D) + t)$   
 $\forall x \ k. F = F$

**apply** (auto elim!: dvdE simp add: algebra-simps)

**unfolding** mult-assoc [symmetric] left-distrib [symmetric] left-diff-distrib [symmetric]

**unfolding** dvd-def mult-commute [of d]

**by** auto

## 39.2 The A and B sets

**lemma** bset:

$\llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P \ x \longrightarrow P(x - D) ;$   
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q \ x \longrightarrow Q(x - D) \rrbracket \implies$

$$\begin{aligned}
& \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x - D) \wedge Q(x - D)) \\
& \llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P x \longrightarrow P(x - D) ; \\
& \quad \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q x \longrightarrow Q(x - D) \rrbracket \Longrightarrow \\
& \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x - D) \vee Q(x - D)) \\
& \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t)) \\
& \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t)) \\
& D > 0 \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t)) \\
& D > 0 \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)) \\
& \llbracket D > 0 ; t \in B \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)) \\
& \llbracket D > 0 ; t - 1 \in B \rrbracket \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)) \\
& d \text{ dvd } D \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x - D) + t)) \\
& d \text{ dvd } D \Longrightarrow (\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x - D) + t)) \\
& \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow F \longrightarrow F \\
\text{proof (blast, blast)} \\
& \text{assume } dp: D > 0 \text{ and } tB: t - 1 \in B \\
& \text{show } (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t)) \\
& \text{apply (rule allI, rule impI, erule ballE[where x=1], erule ballE[where x=t-1])} \\
& \text{apply algebra using dp tB by simp-all} \\
\text{next} \\
& \text{assume } dp: D > 0 \text{ and } tB: t \in B \\
& \text{show } (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t)) \\
& \text{apply (rule allI, rule impI, erule ballE[where x=D], erule ballE[where x=t])} \\
& \text{apply algebra} \\
& \text{using dp tB by simp-all} \\
\text{next} \\
& \text{assume } dp: D > 0 \text{ thus } (\forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t)) \text{ by arith} \\
\text{next} \\
& \text{assume } dp: D > 0 \text{ thus } \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t) \text{ by arith} \\
\text{next} \\
& \text{assume } dp: D > 0 \text{ and } tB: t \in B \\
& \{\text{fix } x \text{ assume nob: } \forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j \text{ and } g: x > t \text{ and } ng: \neg (x - D) > t \\
& \text{hence } x - t \leq D \text{ and } 1 \leq x - t \text{ by simp+} \\
& \text{hence } \exists j \in \{1 \dots D\}. x - t = j \text{ by auto} \\
& \text{hence } \exists j \in \{1 \dots D\}. x = t + j \text{ by (simp add: algebra-simps)} \\
& \text{with nob tB have False by simp}\}
\end{aligned}$$

**thus**  $\forall x.(\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)$  **by** *blast*  
**next**  
**assume**  $dp: D > 0$  **and**  $tB: t - 1 \in B$   
**{fix**  $x$  **assume**  $nob: \forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j$  **and**  $g: x \geq t$  **and**  $ng: \neg (x - D) \geq t$   
**hence**  $x - (t - 1) \leq D$  **and**  $1 \leq x - (t - 1)$  **by** *simp+*  
**hence**  $\exists j \in \{1 \dots D\}. x - (t - 1) = j$  **by** *auto*  
**hence**  $\exists j \in \{1 \dots D\}. x = (t - 1) + j$  **by** (*simp add: algebra-simps*)  
**with**  $nob$   $tB$  **have** *False* **by** *simp*  
**thus**  $\forall x.(\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)$  **by** *blast*  
**next**  
**assume**  $d: d \text{ dvd } D$   
**{fix**  $x$  **assume**  $H: d \text{ dvd } x + t$  **with**  $d$  **have**  $d \text{ dvd } (x - D) + t$  **by** *algebra*  
**thus**  $\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x - D) + t)$  **by** *simp*  
**next**  
**assume**  $d: d \text{ dvd } D$   
**{fix**  $x$  **assume**  $H: \neg(d \text{ dvd } x + t)$  **with**  $d$  **have**  $\neg d \text{ dvd } (x - D) + t$   
**by** (*clarsimp simp add: dvd-def,erule-tac x = ka + k in allE,simp add: algebra-simps*)  
**thus**  $\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x - D) + t)$  **by** *auto*  
**qed** *blast*

**lemma** *aset*:

$\llbracket \forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ;$   
 $\forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow$   
 $\forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x + D) \wedge Q(x + D))$   
 $\llbracket \forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ;$   
 $\forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow Q x \longrightarrow Q(x + D) \rrbracket \Longrightarrow$   
 $\forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (P x \vee Q x) \longrightarrow (P(x + D) \vee Q(x + D))$   
 $\llbracket D > 0; t + 1 \in A \rrbracket \Longrightarrow (\forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t))$   
 $\llbracket D > 0; t \in A \rrbracket \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$   
 $\llbracket D > 0; t \in A \rrbracket \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t))$   
 $\llbracket D > 0; t + 1 \in A \rrbracket \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t))$   
 $D > 0 \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$   
 $D > 0 \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t))$   
 $d \text{ dvd } D \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x + D) + t))$   
 $d \text{ dvd } D \Longrightarrow (\forall (x::int).(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x + D) + t))$

```

 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow F \longrightarrow F$ 
proof (blast, blast)
  assume dp:  $D > 0$  and tA:  $t + 1 \in A$ 
  show  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x = t) \longrightarrow (x + D = t))$ 
  apply (rule allI, rule impI, erule ballE[where  $x=1$ ], erule ballE[where  $x=t + 1$ ])
  using dp tA by simp-all
next
  assume dp:  $D > 0$  and tA:  $t \in A$ 
  show  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$ 
  apply (rule allI, rule impI, erule ballE[where  $x=D$ ], erule ballE[where  $x=t$ ])
  using dp tA by simp-all
next
  assume dp:  $D > 0$  thus  $(\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x > t) \longrightarrow (x + D > t))$  by arith
next
  assume dp:  $D > 0$  thus  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \geq t) \longrightarrow (x + D \geq t)$  by arith
next
  assume dp:  $D > 0$  and tA:  $t \in A$ 
  {fix x assume nob:  $\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j$  and g:  $x < t$  and ng:  $\neg (x + D) < t$ 
    hence  $t - x \leq D$  and  $1 \leq t - x$  by simp+
    hence  $\exists j \in \{1 \dots D\}. t - x = j$  by auto
    hence  $\exists j \in \{1 \dots D\}. x = t - j$  by (auto simp add: algebra-simps)
    with nob tA have False by simp}
  thus  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x < t) \longrightarrow (x + D < t)$  by blast
next
  assume dp:  $D > 0$  and tA:  $t + 1 \in A$ 
  {fix x assume nob:  $\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j$  and g:  $x \leq t$  and ng:  $\neg (x + D) \leq t$ 
    hence  $(t + 1) - x \leq D$  and  $1 \leq (t + 1) - x$  by (simp-all add: algebra-simps)
    hence  $\exists j \in \{1 \dots D\}. (t + 1) - x = j$  by auto
    hence  $\exists j \in \{1 \dots D\}. x = (t + 1) - j$  by (auto simp add: algebra-simps)
    with nob tA have False by simp}
  thus  $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (x \leq t) \longrightarrow (x + D \leq t)$  by blast
next
  assume d:  $d \text{ dvd } D$ 
  {fix x assume H:  $d \text{ dvd } x + t$  with d have  $d \text{ dvd } (x + D) + t$ 
    by (clarsimp simp add: dvd-def, rule-tac x = ka + k in exI, simp add: algebra-simps)}
  thus  $\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (d \text{ dvd } x + t) \longrightarrow (d \text{ dvd } (x + D) + t)$  by simp
next
  assume d:  $d \text{ dvd } D$ 
  {fix x assume H:  $\neg (d \text{ dvd } x + t)$  with d have  $\neg d \text{ dvd } (x + D) + t$ 
    by (clarsimp simp add: dvd-def, erule-tac x = ka - k in allE, simp add: algebra-simps)}
  thus  $\forall (x :: \text{int}). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow (\neg d \text{ dvd } (x + D) + t)$ 

```

$(x + D) + t$  **by** *auto*  
**qed** *blast*

### 39.3 Cooper’s Theorem $-\infty$ and $+\infty$ Version

#### 39.3.1 First some trivial facts about periodic sets or predicates

**lemma** *periodic-finite-ex*:

**assumes** *dpos*:  $(0::int) < d$  **and** *modd*:  $ALL\ x\ k.\ P\ x = P(x - k*d)$

**shows**  $(EX\ x.\ P\ x) = (EX\ j : \{1..d\}.\ P\ j)$

(**is** *?LHS* = *?RHS*)

**proof**

**assume** *?LHS*

**then obtain** *x* **where** *P*:  $P\ x$  ..

**have**  $x\ mod\ d = x - (x\ div\ d)*d$  **by**(*simp add:zmod-zdiv-equality mult-ac eq-diff-eq*)

**hence** *Pmod*:  $P\ x = P(x\ mod\ d)$  **using** *modd* **by** *simp*

**show** *?RHS*

**proof** (*cases*)

**assume**  $x\ mod\ d = 0$

**hence**  $P\ 0$  **using** *P Pmod* **by** *simp*

**moreover have**  $P\ 0 = P(0 - (-1)*d)$  **using** *modd* **by** *blast*

**ultimately have**  $P\ d$  **by** *simp*

**moreover have**  $d : \{1..d\}$  **using** *dpos* **by** *simp*

**ultimately show** *?RHS* ..

**next**

**assume** *not0*:  $x\ mod\ d \neq 0$

**have**  $P(x\ mod\ d)$  **using** *dpos P Pmod* **by** *simp*

**moreover have**  $x\ mod\ d : \{1..d\}$

**proof** –

**from** *dpos* **have**  $0 \leq x\ mod\ d$  **by**(*rule pos-mod-sign*)

**moreover from** *dpos* **have**  $x\ mod\ d < d$  **by**(*rule pos-mod-bound*)

**ultimately show** *?thesis* **using** *not0* **by** *simp*

**qed**

**ultimately show** *?RHS* ..

**qed**

**qed** *auto*

#### 39.3.2 The $-\infty$ Version

**lemma** *decr-lemma*:  $0 < (d::int) \implies x - (abs(x-z)+1) * d < z$   
**by**(*induct rule: int-gr-induct, simp-all add:int-distrib*)

**lemma** *incr-lemma*:  $0 < (d::int) \implies z < x + (abs(x-z)+1) * d$   
**by**(*induct rule: int-gr-induct, simp-all add:int-distrib*)

**lemma** *decr-mult-lemma*:

**assumes** *dpos*:  $(0::int) < d$  **and** *minus*:  $\forall x.\ P\ x \longrightarrow P(x - d)$  **and** *knneg*:  $0 \leq k$

**shows**  $ALL\ x.\ P\ x \longrightarrow P(x - k*d)$

**using** *knneg*

```

proof (induct rule:int-ge-induct)
  case base thus ?case by simp
next
  case (step i)
  {fix x
   have  $P\ x \longrightarrow P\ (x - i * d)$  using step.hyps by blast
   also have  $\dots \longrightarrow P(x - (i + 1) * d)$  using minus[THEN spec, of  $x - i * d$ ]
   by (simp add: algebra-simps)
   ultimately have  $P\ x \longrightarrow P(x - (i + 1) * d)$  by blast}
  thus ?case ..
qed

```

```

lemma minusinfinity:
  assumes dpos:  $0 < d$  and
    P1eqP1:  $\text{ALL } x\ k. P1\ x = P1(x - k*d)$  and ePeqP1:  $\text{EX } z::\text{int}. \text{ALL } x. x < z \longrightarrow (P\ x = P1\ x)$ 
  shows  $(\text{EX } x. P1\ x) \longrightarrow (\text{EX } x. P\ x)$ 
proof
  assume eP1:  $\text{EX } x. P1\ x$ 
  then obtain x where P1:  $P1\ x ..$ 
  from ePeqP1 obtain z where P1eqP:  $\text{ALL } x. x < z \longrightarrow (P\ x = P1\ x) ..$ 
  let ?w =  $x - (\text{abs}(x-z)+1) * d$ 
  from dpos have w:  $?w < z$  by (rule decr-lemma)
  have  $P1\ x = P1\ ?w$  using P1eqP1 by blast
  also have  $\dots = P(?w)$  using w P1eqP by blast
  finally have  $P\ ?w$  using P1 by blast
  thus  $\text{EX } x. P\ x ..$ 
qed

```

```

lemma cpmi:
  assumes dp:  $0 < D$  and p1: $\exists z. \forall x < z. P\ x = P'\ x$ 
  and nb: $\forall x. (\forall j \in \{1..D\}. \forall (b::\text{int}) \in B. x \neq b+j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$ 
  and pd: $\forall x\ k. P'\ x = P'\ (x - k*D)$ 
  shows  $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \mid (\exists j \in \{1..D\}. \exists b \in B. P\ (b+j)))$ 
  (is ?L = (?R1  $\vee$  ?R2))
proof–
  {assume ?R2 hence ?L by blast}
  moreover
  {assume H: ?R1 hence ?L using minusinfinity[OF dp pd p1] periodic-finite-ex[OF dp pd] by simp}
  moreover
  { fix x
    assume P:  $P\ x$  and H:  $\neg ?R2$ 
    {fix y assume  $\neg (\exists j \in \{1..D\}. \exists b \in B. P\ (b + j))$  and P:  $P\ y$ 
     hence  $\sim (\text{EX } (j::\text{int}) : \{1..D\}. \text{EX } (b::\text{int}) : B. y = b+j)$  by auto
     with nb P have  $P\ (y - D)$  by auto }
    hence  $\text{ALL } x. \sim (\text{EX } (j::\text{int}) : \{1..D\}. \text{EX } (b::\text{int}) : B. P(b+j)) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$  by blast
  }

```

```

with  $H\ P$  have  $th: \forall x. P\ x \longrightarrow P\ (x - D)$  by auto
from  $p1$  obtain  $z$  where  $z: ALL\ x. x < z \dashv\dashv (P\ x = P'\ x)$  by blast
let  $?y = x - (|x - z| + 1) * D$ 
have  $zp: 0 \leq (|x - z| + 1)$  by arith
from  $dp$  have  $yz: ?y < z$  using decr-lemma[OF dp] by simp
from  $z$ [rule-format, OF yz] decr-mult-lemma[OF dp th zp, rule-format, OF P]
have  $th2: P'\ ?y$  by auto
  with periodic-finite-ex[OF dp pd]
  have  $?R1$  by blast}
ultimately show  $?thesis$  by blast
qed

```

### 39.3.3 The $+\infty$ Version

**lemma** *plusinfinity*:

```

assumes  $dpos: (0::int) < d$  and
   $P1eqP1: \forall x\ k. P'\ x = P'(x - k*d)$  and  $ePeqP1: \exists z. \forall x > z. P\ x = P'\ x$ 
shows  $(\exists x. P'\ x) \longrightarrow (\exists x. P\ x)$ 
proof
  assume  $eP1: EX\ x. P'\ x$ 
  then obtain  $x$  where  $P1: P'\ x ..$ 
  from  $ePeqP1$  obtain  $z$  where  $P1eqP: \forall x > z. P\ x = P'\ x ..$ 
  let  $?w' = x + (abs(x-z)+1) * d$ 
  let  $?w = x - (-(abs(x-z) + 1)) * d$ 
  have  $ww'[simp]: ?w = ?w'$  by (simp add: algebra-simps)
  from  $dpos$  have  $w: ?w > z$  by (simp only: ww' incr-lemma)
  hence  $P'\ x = P'\ ?w$  using  $P1eqP1$  by blast
  also have  $\dots = P(?w)$  using  $w\ P1eqP$  by blast
  finally have  $P\ ?w$  using  $P1$  by blast
  thus  $EX\ x. P\ x ..$ 
qed

```

**lemma** *incr-mult-lemma*:

```

assumes  $dpos: (0::int) < d$  and  $plus: ALL\ x::int. P\ x \longrightarrow P(x + d)$  and  $knneg: 0 \leq k$ 
shows  $ALL\ x. P\ x \longrightarrow P(x + k*d)$ 
using  $knneg$ 
proof (induct rule:int-ge-induct)
  case base thus  $?case$  by simp
next
  case (step i)
  {fix  $x$ 
    have  $P\ x \longrightarrow P(x + i * d)$  using step.hyps by blast
    also have  $\dots \longrightarrow P(x + (i + 1) * d)$  using plus[THEN spec, of x + i * d]
      by (simp add:int-distrib zadd-ac)
    ultimately have  $P\ x \longrightarrow P(x + (i + 1) * d)$  by blast}
  thus  $?case ..$ 
qed

```

```

lemma cppi:
  assumes dp:  $0 < D$  and p1:  $\exists z. \forall x > z. P\ x = P'\ x$ 
  and nb:  $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in A. x \neq b - j) \longrightarrow P\ (x) \longrightarrow P\ (x + D)$ 
  and pd:  $\forall x\ k. P'\ x = P'\ (x - k * D)$ 
  shows  $(\exists x. P\ x) = ((\exists j \in \{1..D\}. P'\ j) \mid (\exists j \in \{1..D\}. \exists b \in A. P\ (b - j)))$ 
(is ?L = (?R1  $\vee$  ?R2))
proof –
  {assume ?R2 hence ?L by blast}
moreover
  {assume H: ?R1 hence ?L using plusinfinity[OF dp pd p1] periodic-finite-ex[OF dp pd] by simp}
moreover
  {fix x
   assume P:  $P\ x$  and H:  $\neg ?R2$ 
   {fix y assume  $\neg (\exists j \in \{1..D\}. \exists b \in A. P\ (b - j))$  and P:  $P\ y$ 
    hence  $\sim (EX\ (j::int) : \{1..D\}. EX\ (b::int) : A. y = b - j)$  by auto
    with nb P have  $P\ (y + D)$  by auto }
    hence  $ALL\ x. \sim (EX\ (j::int) : \{1..D\}. EX\ (b::int) : A. P\ (b - j)) \longrightarrow P\ (x) \longrightarrow P\ (x + D)$  by blast
    with H P have th:  $\forall x. P\ x \longrightarrow P\ (x + D)$  by auto
    from p1 obtain z where  $z: ALL\ x. x > z \longrightarrow (P\ x = P'\ x)$  by blast
    let ?y =  $x + (|x - z| + 1) * D$ 
    have zp:  $0 \leq (|x - z| + 1)$  by arith
    from dp have yz: ?y > z using incr-lemma[OF dp] by simp
    from z[rule-format, OF yz] incr-mult-lemma[OF dp th zp, rule-format, OF P]
  } have th2:  $P'\ ?y$  by auto
  with periodic-finite-ex[OF dp pd]
  have ?R1 by blast}
ultimately show ?thesis by blast
qed

```

```

lemma simp-from-to:  $\{i..j::int\} = (\text{if } j < i \text{ then } \{\} \text{ else insert } i\ \{i+1..j\})$ 
apply (simp add: atLeastAtMost-def atLeast-def atMost-def)
apply (fastsimp)
done

```

```

theorem unity-coeff-ex:  $(\exists (x::'a::\{\text{semiring-0}, \text{Rings.dvd}\}). P\ (l * x)) \equiv (\exists x. l \text{ dvd } (x + 0) \wedge P\ x)$ 
apply (rule eq-reflection [symmetric])
apply (rule iffI)
defer
apply (erule exE)
apply (rule-tac  $x = l * x$  in exI)
apply (simp add: dvd-def)
apply (rule-tac  $x = x$  in exI, simp)
apply (erule exE)
apply (erule conjE)
apply simp

```



```

apply (erule dvdE)
apply (rule-tac  $x = k$  in exI)
apply simp
done

```

```

lemma zdvd-mono: assumes not0:  $(k::int) \neq 0$ 
shows  $((m::int) \text{ dvd } t) \equiv (k*m \text{ dvd } k*t)$ 
using not0 by (simp add: dvd-def)

```

```

lemma uminus-dvd-conv:  $(d \text{ dvd } (t::int)) \equiv (-d \text{ dvd } t) \ (d \text{ dvd } (t::int)) \equiv (d \text{ dvd } -t)$ 
by simp-all

```

Theorems for transforming predicates on nat to predicates on *int*

```

lemma zdiff-int-split:  $P \ (int \ (x - y)) =$ 
 $((y \leq x \longrightarrow P \ (int \ x - int \ y)) \wedge (x < y \longrightarrow P \ 0))$ 
by (cases  $y \leq x$ ) (simp-all add: zdiff-int)

```

```

lemma number-of1:  $(0::int) \leq number-of \ n \implies (0::int) \leq number-of \ (Int.Bit0 \ n) \wedge (0::int) \leq number-of \ (Int.Bit1 \ n)$ 
by simp

```

```

lemma number-of2:  $(0::int) \leq Numeral0$  by simp

```

Specific instances of congruence rules, to prevent simplifier from looping.

```

theorem imp-le-cong:  $(0 \leq x \implies P = P') \implies (0 \leq (x::int) \longrightarrow P) = (0 \leq x \longrightarrow P')$  by simp

```

```

theorem conj-le-cong:  $(0 \leq x \implies P = P') \implies (0 \leq (x::int) \wedge P) = (0 \leq x \wedge P')$ 
by (simp cong: conj-cong)

```

```

use Tools/Qelim/cooper.ML

```

```

setup Cooper.setup

```

```

method-setup presburger = Cooper.method Cooper's algorithm for Presburger arithmetic

```

```

declare dvd-eq-mod-eq-0[symmetric, presburger]
declare mod-1[presburger]
declare mod-0[presburger]
declare mod-by-1[presburger]
declare zmod-zero[presburger]
declare zmod-self[presburger]
declare mod-self[presburger]
declare mod-by-0[presburger]
declare mod-div-trivial[presburger]

```

```

declare div-mod-equality2[presburger]
declare div-mod-equality[presburger]
declare mod-div-equality2[presburger]
declare mod-div-equality[presburger]
declare mod-mult-self1[presburger]
declare mod-mult-self2[presburger]
declare zdiv-zmod-equality2[presburger]
declare zdiv-zmod-equality[presburger]
declare mod2-Suc-Suc[presburger]
lemma [presburger]: (a::int) div 0 = 0 and [presburger]: a mod 0 = a
by simp-all

lemma [presburger, algebra]: m mod 2 = (1::nat)  $\longleftrightarrow$   $\neg$  2 dvd m by presburger
lemma [presburger, algebra]: m mod 2 = Suc 0  $\longleftrightarrow$   $\neg$  2 dvd m by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = (1::nat)  $\longleftrightarrow$   $\neg$  2 dvd m by
presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = Suc 0  $\longleftrightarrow$   $\neg$  2 dvd m by
presburger
lemma [presburger, algebra]: m mod 2 = (1::int)  $\longleftrightarrow$   $\neg$  2 dvd m by presburger

end

```

## 40 Hilbert-Choice: Hilbert’s Epsilon-Operator and the Axiom of Choice

```

theory Hilbert-Choice
imports Nat Wellfounded Plain
uses (Tools/meson.ML) (Tools/choice-specification.ML)
begin

```

### 40.1 Hilbert’s epsilon

```

axiomatization Eps :: ('a => bool) => 'a where
  someI: P x ==> P (Eps P)

syntax (epsilon)
  -Eps      :: [pttrn, bool] => 'a    (( $\exists \epsilon$  -./ -) [0, 10] 10)
syntax (HOL)
  -Eps      :: [pttrn, bool] => 'a    (( $\exists @$  -./ -) [0, 10] 10)
syntax
  -Eps      :: [pttrn, bool] => 'a    (( $\exists \text{SOME}$  -./ -) [0, 10] 10)
translations
  SOME x. P == CONST Eps (%x. P)

print-translation <<
  [(@{const-syntax Eps}, fn [Abs abs] =>
    let val (x, t) = atomic-abs-tr' abs
```

*in Syntax.const @{syntax-const -Eps} \$ x \$ t end)]*  
 >> — to avoid eta-contraction of body

**definition** *inv-into* :: 'a set => ('a => 'b) => ('b => 'a) **where**  
*inv-into* A f == %x. SOME y. y : A & f y = x

**abbreviation** *inv* :: ('a => 'b) => ('b => 'a) **where**  
*inv* == *inv-into* UNIV

## 40.2 Hilbert’s Epsilon-operator

Easier to apply than *someI* if the witness comes from an existential formula

**lemma** *someI-ex* [*elim?*]:  $\exists x. P\ x \implies P\ (SOME\ x.\ P\ x)$   
**apply** (*erule exE*)  
**apply** (*erule someI*)  
**done**

Easier to apply than *someI* because the conclusion has only one occurrence of *P*.

**lemma** *someI2*:  $[P\ a;\ !x. P\ x \implies Q\ x] \implies Q\ (SOME\ x.\ P\ x)$   
**by** (*blast intro: someI*)

Easier to apply than *someI2* if the witness comes from an existential formula

**lemma** *someI2-ex*:  $[P\ a;\ !x. P\ x \implies Q\ x] \implies Q\ (SOME\ x.\ P\ x)$   
**by** (*blast intro: someI2*)

**lemma** *some-equality* [*intro*]:  
 $[P\ a;\ !x. P\ x \implies x=a] \implies (SOME\ x.\ P\ x) = a$   
**by** (*blast intro: someI2*)

**lemma** *some1-equality*:  $[EX!x. P\ x;\ P\ a] \implies (SOME\ x.\ P\ x) = a$   
**by** *blast*

**lemma** *some-eq-ex*:  $P\ (SOME\ x.\ P\ x) = (\exists x. P\ x)$   
**by** (*blast intro: someI*)

**lemma** *some-eq-trivial* [*simp*]:  $(SOME\ y.\ y=x) = x$   
**apply** (*rule some-equality*)  
**apply** (*rule refl, assumption*)  
**done**

**lemma** *some-sym-eq-trivial* [*simp*]:  $(SOME\ y.\ x=y) = x$   
**apply** (*rule some-equality*)  
**apply** (*rule refl*)  
**apply** (*erule sym*)  
**done**

### 40.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

**lemma** *choice*:  $\forall x. \exists y. Q\ x\ y \implies \exists f. \forall x. Q\ x\ (f\ x)$   
**by** (*fast elim: someI*)

**lemma** *bchoice*:  $\forall x \in S. \exists y. Q\ x\ y \implies \exists f. \forall x \in S. Q\ x\ (f\ x)$   
**by** (*fast elim: someI*)

### 40.4 Function Inverse

**lemma** *inv-def*:  $inv\ f = (\%y. SOME\ x. f\ x = y)$   
**by** (*simp add: inv-into-def*)

**lemma** *inv-into-into*:  $x : f\ 'A \implies inv\ into\ A\ f\ x : A$   
**apply** (*simp add: inv-into-def*)  
**apply** (*fast intro: someI2*)  
**done**

**lemma** *inv-id* [*simp*]:  $inv\ id = id$   
**by** (*simp add: inv-into-def id-def*)

**lemma** *inv-into-f-f* [*simp*]:  
 $[| inj\ on\ f\ A; x : A |] \implies inv\ into\ A\ f\ (f\ x) = x$   
**apply** (*simp add: inv-into-def inj-on-def*)  
**apply** (*blast intro: someI2*)  
**done**

**lemma** *inv-f-f*:  $inj\ f \implies inv\ f\ (f\ x) = x$   
**by** *simp*

**lemma** *f-inv-into-f*:  $y : f\ 'A \implies f\ (inv\ into\ A\ f\ y) = y$   
**apply** (*simp add: inv-into-def*)  
**apply** (*fast intro: someI2*)  
**done**

**lemma** *inv-into-f-eq*:  $[| inj\ on\ f\ A; x : A; f\ x = y |] \implies inv\ into\ A\ f\ y = x$   
**apply** (*erule subst*)  
**apply** (*fast intro: inv-into-f-f*)  
**done**

**lemma** *inv-f-eq*:  $[| inj\ f; f\ x = y |] \implies inv\ f\ y = x$   
**by** (*simp add: inv-into-f-eq*)

**lemma** *inj-imp-inv-eq*:  $[| inj\ f; ALL\ x. f\ (g\ x) = x |] \implies inv\ f = g$   
**by** (*blast intro: ext inv-into-f-eq*)

But is it useful?

**lemma** *inj-transfer*:

**assumes** *injf*:  $\text{inj } f$  **and** *minor*:  $\forall y. y \in \text{range}(f) \implies P(\text{inv } f \ y)$

**shows**  $P \ x$

**proof** –

**have**  $f \ x \in \text{range } f$  **by** *auto*

**hence**  $P(\text{inv } f \ (f \ x))$  **by** (*rule minor*)

**thus**  $P \ x$  **by** (*simp add: inv-into-f-f [OF injf]*)

**qed**

**lemma** *inj-iff*:  $(\text{inj } f) = (\text{inv } f \ o \ f = \text{id})$

**apply** (*simp add: o-def expand-fun-eq*)

**apply** (*blast intro: inj-on-inverseI inv-into-f-f*)

**done**

**lemma** *inv-o-cancel[simp]*:  $\text{inj } f \implies \text{inv } f \ o \ f = \text{id}$

**by** (*simp add: inj-iff*)

**lemma** *o-inv-o-cancel[simp]*:  $\text{inj } f \implies g \ o \ \text{inv } f \ o \ f = g$

**by** (*simp add: o-assoc[symmetric]*)

**lemma** *inv-into-image-cancel[simp]*:

*inj-on*  $f \ A \implies S \leq A \implies \text{inv-into } A \ f \ ' \ f \ ' \ S = S$

**by**(*fastsimp simp: image-def*)

**lemma** *inj-imp-surj-inv*:  $\text{inj } f \implies \text{surj } (\text{inv } f)$

**by** (*blast intro: surjI inv-into-f-f*)

**lemma** *surj-f-inv-f*:  $\text{surj } f \implies f(\text{inv } f \ y) = y$

**by** (*simp add: f-inv-into-f surj-range*)

**lemma** *inv-into-injective*:

**assumes** *eq*:  $\text{inv-into } A \ f \ x = \text{inv-into } A \ f \ y$

**and**  $x: x: f'A$

**and**  $y: y: f'A$

**shows**  $x=y$

**proof** –

**have**  $f \ (\text{inv-into } A \ f \ x) = f \ (\text{inv-into } A \ f \ y)$  **using** *eq* **by** *simp*

**thus** *?thesis* **by** (*simp add: f-inv-into-f x y*)

**qed**

**lemma** *inj-on-inv-into*:  $B \leq f'A \implies \text{inj-on } (\text{inv-into } A \ f) \ B$

**by** (*blast intro: inj-onI dest: inv-into-injective injD*)

**lemma** *bij-betw-inv-into*:  $\text{bij-betw } f \ A \ B \implies \text{bij-betw } (\text{inv-into } A \ f) \ B \ A$

**by** (*auto simp add: bij-betw-def inj-on-inv-into*)

**lemma** *surj-imp-inj-inv*:  $\text{surj } f \implies \text{inj } (\text{inv } f)$

**by** (*simp add: inj-on-inv-into surj-range*)

```

lemma surj-iff: (surj f) = (f o inv f = id)
apply (simp add: o-def expand-fun-eq)
apply (blast intro: surjI surj-f-inv-f)
done

```

```

lemma surj-imp-inv-eq: [| surj f;  $\forall x. g(f\ x) = x$  |] ==> inv f = g
apply (rule ext)
apply (drule-tac x = inv f x in spec)
apply (simp add: surj-f-inv-f)
done

```

```

lemma bij-imp-bij-inv: bij f ==> bij (inv f)
by (simp add: bij-def inj-imp-surj-inv surj-imp-inj-inv)

```

```

lemma inv-equality: [|  $\forall x. g(f\ x) = x$ ;  $\forall y. f(g\ y) = y$  |] ==> inv f = g
apply (rule ext)
apply (auto simp add: inv-into-def)
done

```

```

lemma inv-inv-eq: bij f ==> inv (inv f) = f
apply (rule inv-equality)
apply (auto simp add: bij-def surj-f-inv-f)
done

```

```

lemma inv-into-comp:
  [| inj-on f (g ‘ A); inj-on g A; x : f ‘ g ‘ A |] ==>
    inv-into A (f o g) x = inv-into A g o inv-into (g ‘ A) f x
apply (rule inv-into-f-eq)
  apply (fast intro: comp-inj-on)
  apply (simp add: inv-into-into)
apply (simp add: f-inv-into-f inv-into-into)
done

```

```

lemma o-inv-distrib: [| bij f; bij g |] ==> inv (f o g) = inv g o inv f
apply (rule inv-equality)
apply (auto simp add: bij-def surj-f-inv-f)
done

```

```

lemma image-surj-f-inv-f: surj f ==> f ‘ (inv f ‘ A) = A
by (simp add: image-eq-UN surj-f-inv-f)

```

```

lemma image-inv-f-f: inj f ==> (inv f) ‘ (f ‘ A) = A
by (simp add: image-eq-UN)

```

```

lemma inv-image-comp: inj f ==> inv f ‘ (f‘X) = X
by (auto simp add: image-def)

```

```

lemma bij-image-Collect-eq:  $\text{bij } f \implies f \text{ ` Collect } P = \{y. P (\text{inv } f y)\}$ 
apply auto
apply (force simp add: bij-is-inj)
apply (blast intro: bij-is-surj [THEN surj-f-inv-f, symmetric])
done

```

```

lemma bij-vimage-eq-inv-image:  $\text{bij } f \implies f \text{ - ` } A = \text{inv } f \text{ ` } A$ 
apply (auto simp add: bij-is-surj [THEN surj-f-inv-f])
apply (blast intro: bij-is-inj [THEN inv-into-f-f, symmetric])
done

```

```

lemma finite-fun-UNIVD1:
  assumes fin: finite (UNIV :: ('a  $\Rightarrow$  'b) set)
  and card: card (UNIV :: 'b set)  $\neq$  Suc 0
  shows finite (UNIV :: 'a set)
proof -
  from fin have finb: finite (UNIV :: 'b set) by (rule finite-fun-UNIVD2)
  with card have card (UNIV :: 'b set)  $\geq$  Suc (Suc 0)
    by (cases card (UNIV :: 'b set)) (auto simp add: card-eq-0-iff)
  then obtain n where card (UNIV :: 'b set) = Suc (Suc n) n = card (UNIV ::
    'b set) - Suc (Suc 0) by auto
  then obtain b1 b2 where b1b2: (b1 :: 'b)  $\neq$  (b2 :: 'b) by (auto simp add:
card-Suc-eq)
  from fin have finite (range ( $\lambda f :: 'a \Rightarrow 'b. \text{inv } f b1$ )) by (rule finite-imageI)
  moreover have UNIV = range ( $\lambda f :: 'a \Rightarrow 'b. \text{inv } f b1$ )
  proof (rule UNIV-eq-I)
    fix x :: 'a
    from b1b2 have x = inv ( $\lambda y. \text{if } y = x \text{ then } b1 \text{ else } b2$ ) b1 by (simp add:
inv-into-def)
    thus x  $\in$  range ( $\lambda f :: 'a \Rightarrow 'b. \text{inv } f b1$ ) by blast
  qed
  ultimately show finite (UNIV :: 'a set) by simp
qed

```

## 40.5 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the *split* Operator

Looping simprule

```

lemma split-paired-Eps:  $(\text{SOME } x. P x) = (\text{SOME } (a,b). P(a,b))$ 
by simp

```

```

lemma Eps-split:  $\text{Eps } (\text{split } P) = (\text{SOME } xy. P (\text{fst } xy) (\text{snd } xy))$ 
by (simp add: split-def)

```

```

lemma Eps-split-eq [simp]:  $(@ (x',y'). x = x' \ \& \ y = y') = (x,y)$ 
by blast

```

A relation is wellfounded iff it has no infinite descending chain

```

lemma wf-iff-no-infinite-down-chain:
  wf r = ( $\sim(\exists f. \forall i. (f(Suc\ i), f\ i) \in r)$ )
apply (simp only: wf-eq-minimal)
apply (rule iffI)
apply (rule notI)
apply (erule exE)
apply (erule tac x = {w.  $\exists i. w=f\ i$ } in allE, blast)
apply (erule contrapos-np, simp, clarify)
apply (subgoal-tac  $\forall n. \text{nat-rec } x\ (\%i\ y. @z. z:Q \ \&\ (z,y):r)\ n \in Q$ )
apply (rule-tac x = nat-rec x ( $\%i\ y. @z. z:Q \ \&\ (z,y):r$ ) in exI)
apply (rule allI, simp)
apply (rule someI2-ex, blast, blast)
apply (rule allI)
apply (induct-tac n, simp-all)
apply (rule someI2-ex, blast+)
done

```

```

lemma wf-no-infinite-down-chainE:
  assumes wf r obtains k where (f (Suc k), f k)  $\notin$  r
using (wf r) wf-iff-no-infinite-down-chain[of r] by blast

```

A dynamically-scoped fact for TFL

```

lemma tfl-some:  $\forall P\ x. P\ x \dashv\dashv P\ (Eps\ P)$ 
by (blast intro: someI)

```

## 40.6 Least value operator

**definition**

```

LeastM :: [ $'a \Rightarrow 'b::ord, 'a \Rightarrow bool$ ]  $\Rightarrow 'a$  where
LeastM m P == SOME x. P x & ( $\forall y. P\ y \dashv\dashv m\ x \leq m\ y$ )

```

**syntax**

```

-LeastM :: [pttrn,  $'a \Rightarrow 'b::ord, bool$ ]  $\Rightarrow 'a$  (LEAST - WRT -. - [0, 4, 10]
10)

```

**translations**

```

LEAST x WRT m. P == CONST LeastM m ( $\%x. P$ )

```

**lemma** LeastMI2:

```

P x ==> (!y. P y ==> m x <= m y)
==> (!x. P x ==>  $\forall y. P\ y \dashv\dashv m\ x \leq m\ y ==> Q\ x$ )
==> Q (LeastM m P)
apply (simp add: LeastM-def)
apply (rule someI2-ex, blast, blast)
done

```

**lemma** LeastM-equality:

```

P k ==> (!x. P x ==> m k <= m x)
==> m (LEAST x WRT m. P x) = (m k :: 'a::order)
apply (rule LeastMI2, assumption, blast)

```



**apply** (*blast intro!*: *order-antisym*)  
**done**

**lemma** *wf-linord-ex-has-least*:

$wf\ r \implies \forall x\ y. ((x,y):r^+)=((y,x):\sim r^*) \implies P\ k$   
 $\implies \exists x. P\ x \ \& \ (!y. P\ y \longrightarrow (m\ x, m\ y):r^*)$   
**apply** (*drule wf-trancl* [*THEN wf-eq-minimal* [*THEN iffD1*]])  
**apply** (*drule-tac*  $x = m\ 'Collect\ P\ in\ spec, force$ )  
**done**

**lemma** *ex-has-least-nat*:

$P\ k \implies \exists x. P\ x \ \& \ (\forall y. P\ y \longrightarrow m\ x \leq (m\ y::nat))$   
**apply** (*simp only*: *pred-nat-trancl-eq-le* [*symmetric*])  
**apply** (*rule wf-pred-nat* [*THEN wf-linord-ex-has-least*])  
**apply** (*simp add*: *less-eq linorder-not-le pred-nat-trancl-eq-le, assumption*)  
**done**

**lemma** *LeastM-nat-lemma*:

$P\ k \implies P\ (LeastM\ m\ P) \ \& \ (\forall y. P\ y \longrightarrow m\ (LeastM\ m\ P) \leq (m\ y::nat))$   
**apply** (*simp add*: *LeastM-def*)  
**apply** (*rule someI-ex*)  
**apply** (*erule ex-has-least-nat*)  
**done**

**lemmas** *LeastM-natI* = *LeastM-nat-lemma* [*THEN conjunct1, standard*]

**lemma** *LeastM-nat-le*:  $P\ x \implies m\ (LeastM\ m\ P) \leq (m\ x::nat)$

**by** (*rule LeastM-nat-lemma* [*THEN conjunct2, THEN spec, THEN mp*], *assumption, assumption*)

## 40.7 Greatest value operator

**definition**

*GreatestM* :: [*'a* => *'b::ord*, *'a* => *bool*] => *'a* **where**  
*GreatestM* *m* *P* == *SOME* *x*. *P* *x* & ( $\forall y. P\ y \longrightarrow m\ y \leq m\ x$ )

**definition**

*Greatest* :: (*'a::ord* => *bool*) => *'a* (**binder** *GREATEST* 10) **where**  
*Greatest* == *GreatestM* (%*x*. *x*)

**syntax**

*-GreatestM* :: [*pttrn*, *'a* => *'b::ord*, *bool*] => *'a*  
(*GREATEST* - *WRT* -. - [*0*, 4, 10] 10)

**translations**

*GREATEST* *x* *WRT* *m*. *P* == *CONST* *GreatestM* *m* (%*x*. *P*)

**lemma** *GreatestM12*:

$P\ x \implies (!y. P\ y \implies m\ y \leq m\ x)$   
 $\implies (!x. P\ x \implies \forall y. P\ y \longrightarrow m\ y \leq m\ x \implies Q\ x)$

```

==> Q (GreatestM m P)
apply (simp add: GreatestM-def)
apply (rule someI2-ex, blast, blast)
done

```

**lemma** *GreatestM-equality*:

```

P k ==> (!!x. P x ==> m x <= m k)
==> m (GREATEST x WRT m. P x) = (m k::'a::order)
apply (rule-tac m = m in GreatestMI2, assumption, blast)
apply (blast intro!: order-antisym)
done

```

**lemma** *Greatest-equality*:

```

P (k::'a::order) ==> (!!x. P x ==> x <= k) ==> (GREATEST x. P x) = k
apply (simp add: Greatest-def)
apply (erule GreatestM-equality, blast)
done

```

**lemma** *ex-has-greatest-nat-lemma*:

```

P k ==> ∀ x. P x --> (∃ y. P y & ~ ((m y::nat) <= m x))
==> ∃ y. P y & ~ (m y < m k + n)
apply (induct n, force)
apply (force simp add: le-Suc-eq)
done

```

**lemma** *ex-has-greatest-nat*:

```

P k ==> ∀ y. P y --> m y < b
==> ∃ x. P x & (∀ y. P y --> (m y::nat) <= m x)
apply (rule ccontr)
apply (cut-tac P = P and n = b - m k in ex-has-greatest-nat-lemma)
apply (subgoal-tac [3] m k <= b, auto)
done

```

**lemma** *GreatestM-nat-lemma*:

```

P k ==> ∀ y. P y --> m y < b
==> P (GreatestM m P) & (∀ y. P y --> (m y::nat) <= m (GreatestM m
P))
apply (simp add: GreatestM-def)
apply (rule someI-ex)
apply (erule ex-has-greatest-nat, assumption)
done

```

**lemmas** *GreatestM-natI* = *GreatestM-nat-lemma* [THEN conjunct1, standard]

**lemma** *GreatestM-nat-le*:

```

P x ==> ∀ y. P y --> m y < b
==> (m x::nat) <= m (GreatestM m P)
apply (blast dest: GreatestM-nat-lemma [THEN conjunct2, THEN spec, of P])
done

```

Specialization to *GREATEST*.

```
lemma GreatestI:  $P \ (k::nat) \implies \forall y. P \ y \longrightarrow y < b \implies P \ (GREATEST \ x. P \ x)$ 
apply (simp add: Greatest-def)
apply (rule GreatestM-natI, auto)
done
```

**lemma** *Greatest-le*:

```
 $P \ x \implies \forall y. P \ y \longrightarrow y < b \implies (x::nat) \leq (GREATEST \ x. P \ x)$ 
apply (simp add: Greatest-def)
apply (rule GreatestM-nat-le, auto)
done
```

## 40.8 The Meson proof procedure

### 40.8.1 Negation Normal Form

de Morgan laws

```
lemma meson-not-conjD:  $\sim(P \& Q) \implies \sim P \mid \sim Q$ 
and meson-not-disjD:  $\sim(P \mid Q) \implies \sim P \& \sim Q$ 
and meson-not-notD:  $\sim\sim P \implies P$ 
and meson-not-allD:  $!!P. \sim(\forall x. P(x)) \implies \exists x. \sim P(x)$ 
and meson-not-exD:  $!!P. \sim(\exists x. P(x)) \implies \forall x. \sim P(x)$ 
by fast+
```

Removal of  $\longrightarrow$  and  $\longleftarrow$  (positive and negative occurrences)

```
lemma meson-imp-to-disjD:  $P \longrightarrow Q \implies \sim P \mid Q$ 
and meson-not-impD:  $\sim(P \longrightarrow Q) \implies P \& \sim Q$ 
and meson-iff-to-disjD:  $P = Q \implies (\sim P \mid Q) \& (\sim Q \mid P)$ 
and meson-not-iffD:  $\sim(P = Q) \implies (P \mid Q) \& (\sim P \mid \sim Q)$ 
— Much more efficient than  $P \wedge \neg Q \vee Q \wedge \neg P$  for computing CNF
and meson-not-refl-disj-D:  $x \sim = x \mid P \implies P$ 
by fast+
```

### 40.8.2 Pulling out the existential quantifiers

Conjunction

```
lemma meson-conj-exD1:  $!!P \ Q. (\exists x. P(x)) \& Q \implies \exists x. P(x) \& Q$ 
and meson-conj-exD2:  $!!P \ Q. P \& (\exists x. Q(x)) \implies \exists x. P \& Q(x)$ 
by fast+
```

Disjunction

```
lemma meson-disj-exD:  $!!P \ Q. (\exists x. P(x)) \mid (\exists x. Q(x)) \implies \exists x. P(x) \mid Q(x)$ 
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and meson-disj-exD1:  $!!P \ Q. (\exists x. P(x)) \mid Q \implies \exists x. P(x) \mid Q$ 
and meson-disj-exD2:  $!!P \ Q. P \mid (\exists x. Q(x)) \implies \exists x. P \mid Q(x)$ 
by fast+
```

### 40.8.3 Generating clauses for the Meson Proof Procedure

Disjunctions

**lemma** *meson-disj-assoc*:  $(P|Q)|R \implies P|(Q|R)$   
**and** *meson-disj-comm*:  $P|Q \implies Q|P$   
**and** *meson-disj-FalseD1*:  $\text{False}|P \implies P$   
**and** *meson-disj-FalseD2*:  $P|\text{False} \implies P$   
**by** *fast+*

## 40.9 Lemmas for Meson, the Model Elimination Procedure

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

**lemma** *make-neg-rule*:  $\sim P|Q \implies ((\sim P \implies P) \implies Q)$   
**by** *blast*

Version for Plaisted’s “Positive refinement” of the Meson procedure

**lemma** *make-refined-neg-rule*:  $\sim P|Q \implies (P \implies Q)$   
**by** *blast*

P should be a literal

**lemma** *make-pos-rule*:  $P|Q \implies ((P \implies \sim P) \implies Q)$   
**by** *blast*

Versions of *make-neg-rule* and *make-pos-rule* that don’t insert new assumptions, for ordinary resolution.

**lemmas** *make-neg-rule'* = *make-refined-neg-rule*

**lemma** *make-pos-rule'*:  $[|P|Q; \sim P|] \implies Q$   
**by** *blast*

Generation of a goal clause – put away the final literal

**lemma** *make-neg-goal*:  $\sim P \implies ((\sim P \implies P) \implies \text{False})$   
**by** *blast*

**lemma** *make-pos-goal*:  $P \implies ((P \implies \sim P) \implies \text{False})$   
**by** *blast*

### 40.9.1 Lemmas for Forward Proof

There is a similarity to congruence rules

**lemma** *conj-forward*:  $[|P' \& Q'; P' \implies P; Q' \implies Q|] \implies P \& Q$   
**by** *blast*

**lemma** *disj-forward*:  $[[ P' \mid Q'; P' \implies P; Q' \implies Q ]] \implies P \mid Q$   
**by** *blast*

**lemma** *disj-forward2*:  
 $[[ P' \mid Q'; P' \implies P; [[ Q'; P \implies \text{False} ]] \implies Q ]] \implies P \mid Q$   
**apply** *blast*  
**done**

**lemma** *all-forward*:  $[[ \forall x. P'(x); !!x. P'(x) \implies P(x) ]] \implies \forall x. P(x)$   
**by** *blast*

**lemma** *ex-forward*:  $[[ \exists x. P'(x); !!x. P'(x) \implies P(x) ]] \implies \exists x. P(x)$   
**by** *blast*

#### 40.10 Meson package

**use** *Tools/meson.ML*

**setup** *Meson.setup*

#### 40.11 Specification package – Hilbertized version

**lemma** *exE-some*:  $[[ Ex P ; c == Eps P ]] \implies P c$   
**by** (*simp only: someI-ex*)

**use** *Tools/choice-specification.ML*

**end**

## 41 Sledgehammer: Sledgehammer: Isabelle–ATP Linkup

**theory** *Sledgehammer*

**imports** *Plain Hilbert-Choice*

**uses**

*~~/src/Tools/Metis/metis.ML*

*Tools/Sledgehammer/sledgehammer-util.ML*

*(Tools/Sledgehammer/sledgehammer-fol-clause.ML)*

*(Tools/Sledgehammer/sledgehammer-fact-preprocessor.ML)*

*(Tools/Sledgehammer/sledgehammer-hol-clause.ML)*

*(Tools/Sledgehammer/sledgehammer-proof-reconstruct.ML)*

*(Tools/Sledgehammer/sledgehammer-fact-filter.ML)*

*(Tools/ATP-Manager/atp-manager.ML)*

*(Tools/ATP-Manager/atp-systems.ML)*

*(Tools/Sledgehammer/sledgehammer-fact-minimizer.ML)*

```

(Tools/Sledgehammer/sledgehammer-isar.ML)
(Tools/Sledgehammer/meson-tactic.ML)
(Tools/Sledgehammer/metis-tactics.ML)
begin

```

```

definition COMBI :: 'a  $\Rightarrow$  'a where
[no-atp]: COMBI P  $\equiv$  P

```

```

definition COMBK :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'a where
[no-atp]: COMBK P Q  $\equiv$  P

```

```

definition COMBB :: ('b  $\Rightarrow$  'c)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'c where [no-atp]:
COMBB P Q R  $\equiv$  P (Q R)

```

```

definition COMBC :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  'b  $\Rightarrow$  'a  $\Rightarrow$  'c where
[no-atp]: COMBC P Q R  $\equiv$  P R Q

```

```

definition COMBS :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'c)  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'c where
[no-atp]: COMBS P Q R  $\equiv$  P R (Q R)

```

```

definition fequal :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool where [no-atp]:
fequal X Y  $\equiv$  (X = Y)

```

```

lemma fequal-imp-equal [no-atp]: fequal X Y  $\Longrightarrow$  X = Y
by (simp add: fequal-def)

```

```

lemma equal-imp-fequal [no-atp]: X = Y  $\Longrightarrow$  fequal X Y
by (simp add: fequal-def)

```

These two represent the equivalence between Boolean equality and iff. They can't be converted to clauses automatically, as the iff would be expanded...

```

lemma iff-positive: P  $\vee$  Q  $\vee$  P = Q
by blast

```

```

lemma iff-negative:  $\neg$  P  $\vee$   $\neg$  Q  $\vee$  P = Q
by blast

```

Theorems for translation to combinators

```

lemma abs-S [no-atp]:  $\lambda x. (f\ x)\ (g\ x) \equiv$  COMBS f g
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBS-def)
done

```

```

lemma abs-I [no-atp]:  $\lambda x. x \equiv$  COMBI
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBI-def)
done

```

```

lemma abs-K [no-atp]:  $\lambda x. y \equiv \text{COMBK } y$ 
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBK-def)
done

```

```

lemma abs-B [no-atp]:  $\lambda x. a (g x) \equiv \text{COMBB } a g$ 
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBB-def)
done

```

```

lemma abs-C [no-atp]:  $\lambda x. (f x) b \equiv \text{COMBC } f b$ 
apply (rule eq-reflection)
apply (rule ext)
apply (simp add: COMBC-def)
done

```

#### 41.1 Setup of external ATPs

```

use Tools/Sledgehammer/sledgehammer-fol-clause.ML
use Tools/Sledgehammer/sledgehammer-fact-preprocessor.ML
setup Sledgehammer-Fact-Preprocessor.setup
use Tools/Sledgehammer/sledgehammer-hol-clause.ML
use Tools/Sledgehammer/sledgehammer-proof-reconstruct.ML
use Tools/Sledgehammer/sledgehammer-fact-filter.ML
use Tools/ATP-Manager/atp-manager.ML
use Tools/ATP-Manager/atp-systems.ML
setup ATP-Systems.setup
use Tools/Sledgehammer/sledgehammer-fact-minimizer.ML
use Tools/Sledgehammer/sledgehammer-isar.ML
setup Sledgehammer-Isar.setup

```

#### 41.2 The MESON prover

```

use Tools/Sledgehammer/meson-tactic.ML
setup Meson-Tactic.setup

```

#### 41.3 The Metis prover

```

use Tools/Sledgehammer/metis-tactics.ML
setup Metis-Tactics.setup

```

```

end

```

## 42 Recdef: TFL: recursive function definitions

```

theory Recdef
imports FunDef Plain
uses
  (Tools/TFL/casesplit.ML)
  (Tools/TFL/utis.ML)
  (Tools/TFL/usyntax.ML)
  (Tools/TFL/dcterm.ML)
  (Tools/TFL/thms.ML)
  (Tools/TFL/rules.ML)
  (Tools/TFL/thry.ML)
  (Tools/TFL/tfl.ML)
  (Tools/TFL/post.ML)
  (Tools/recdef.ML)
begin

inductive
  wfrec-rel :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b => bool
  for R :: ('a * 'a) set
  and F :: ('a => 'b) => 'a => 'b
where
  wfrecI: ALL z. (z, x) : R --> wfrec-rel R F z (g z) ==>
    wfrec-rel R F x (F g x)

definition
  cut :: ('a => 'b) => ('a * 'a) set => 'a => 'a => 'b where
  cut f r x == (%y. if (y,x):r then f y else undefined)

definition
  adm-wf :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => bool where
  adm-wf R F == ALL f g x.
    (ALL z. (z, x) : R --> f z = g z) --> F f x = F g x

definition
  wfrec :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b where
  [code del]: wfrec R F == %x. THE y. wfrec-rel R (%f x. F (cut f R x) x) x y

```

### 42.1 Well-Founded Recursion

cut

**lemma** cuts-eq: (cut f r x = cut g r x) = (ALL y. (y,x):r --> f(y)=g(y))  
**by** (simp add: expand-fun-eq cut-def)

**lemma** cut-apply: (x,a):r ==> (cut f r a)(x) = f(x)  
**by** (simp add: cut-def)

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"



```

lemma wfrec-unique: [| adm-wf R F; wf R |] ==> EX! y. wfrec-rel R F x y
apply (simp add: adm-wf-def)
apply (erule-tac a=x in wf-induct)
apply (rule ex1I)
apply (rule-tac g = %x. THE y. wfrec-rel R F x y in wfrec-rel.wfrecI)
apply (fast dest!: theI')
apply (erule wfrec-rel.cases, simp)
apply (erule allE, erule allE, erule allE, erule mp)
apply (fast intro: the-equality [symmetric])
done

```

```

lemma adm-lemma: adm-wf R (%f x. F (cut f R x) x)
apply (simp add: adm-wf-def)
apply (intro strip)
apply (rule cuts-eq [THEN iffD2, THEN subst], assumption)
apply (rule refl)
done

```

```

lemma wfrec: wf(r) ==> wfrec r H a = H (cut (wfrec r H) r a) a
apply (simp add: wfrec-def)
apply (rule adm-lemma [THEN wfrec-unique, THEN the1-equality], assumption)
apply (rule wfrec-rel.wfrecI)
apply (intro strip)
apply (erule adm-lemma [THEN wfrec-unique, THEN theI'])
done

```

\* This form avoids giant explosions in proofs. NOTE USE OF ==

```

lemma def-wfrec: [| f==wfrec r H; wf(r) |] ==> f(a) = H (cut f r a) a
apply auto
apply (blast intro: wfrec)
done

```

```

lemma tfl-wf-induct: ALL R. wf R -->
  (ALL P. (ALL x. (ALL y. (y,x):R --> P y) --> P x) --> (ALL x. P
  x))
apply clarify
apply (rule-tac r = R and P = P and a = x in wf-induct, assumption, blast)
done

```

```

lemma tfl-cut-apply: ALL f R. (x,a):R --> (cut f R a)(x) = f(x)
apply clarify
apply (rule cut-apply, assumption)
done

```

```

lemma tfl-wfrec:
  ALL M R f. (f==wfrec R M) --> wf R --> (ALL x. f x = M (cut f R x) x)
apply clarify
apply (erule wfrec)

```

done

**lemma** *tfl-eq-True*:  $(x = \text{True}) \dashv\vdash x$   
**by** *blast*

**lemma** *tfl-rev-eq-mp*:  $(x = y) \dashv\vdash y \dashv\vdash x$   
**by** *blast*

**lemma** *tfl-simp-thm*:  $(x \dashv\vdash y) \dashv\vdash (x = x') \dashv\vdash (x' \dashv\vdash y)$   
**by** *blast*

**lemma** *tfl-P-imp-P-iff-True*:  $P \implies P = \text{True}$   
**by** *blast*

**lemma** *tfl-imp-trans*:  $(A \dashv\vdash B) \implies (B \dashv\vdash C) \implies (A \dashv\vdash C)$   
**by** *blast*

**lemma** *tfl-disj-assoc*:  $(a \vee b) \vee c == a \vee (b \vee c)$   
**by** *simp*

**lemma** *tfl-disjE*:  $P \vee Q \implies P \dashv\vdash R \implies Q \dashv\vdash R \implies R$   
**by** *blast*

**lemma** *tfl-exE*:  $\exists x. P x \implies \forall x. P x \dashv\vdash Q \implies Q$   
**by** *blast*

**use** *Tools/TFL/casesplit.ML*  
**use** *Tools/TFL/utis.ML*  
**use** *Tools/TFL/usyntax.ML*  
**use** *Tools/TFL/dcterm.ML*  
**use** *Tools/TFL/thms.ML*  
**use** *Tools/TFL/rules.ML*  
**use** *Tools/TFL/thry.ML*  
**use** *Tools/TFL/tfl.ML*  
**use** *Tools/TFL/post.ML*  
**use** *Tools/recdef.ML*  
**setup** *Recdef.setup*

Wellfoundedness of *same-fst*

**definition**

*same-fst* ::  $('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow ('b * 'b)\text{set}) \Rightarrow (('a * 'b) * ('a * 'b))\text{set}$

**where**

*same-fst*  $P R == \{((x',y'),(x,y)) . x'=x \ \& \ P x \ \& \ (y',y) : R x\}$

— For *rec-def* declarations where the first n parameters stay unchanged in the recursive call.

**lemma** *same-fstI* [*intro!*]:

$[| P x; (y',y) : R x |] \implies ((x,y'),(x,y)) : \text{same-fst } P R$

**by** (*simp add: same-fst-def*)

```

lemma wf-same-fst:
  assumes prem: ( $\forall x. P\ x \implies wf\ (R\ x)$ )
  shows  $wf\ (same\_fst\ P\ R)$ 
apply (simp cong del: imp-cong add: wf-def same-fst-def)
apply (intro strip)
apply (rename-tac a b)
apply (case-tac wf (R a))
  apply (erule-tac a = b in wf-induct, blast)
apply (blast intro: prem)
done

```

Rule setup

```

lemmas [recdef-simp] =
  inv-image-def
  measure-def
  lex-prod-def
  same-fst-def
  less-Suc-eq [THEN iffD2]

lemmas [recdef-cong] =
  if-cong let-cong image-cong INT-cong UN-cong bex-cong ball-cong imp-cong

lemmas [recdef-wf] =
  wf-trancl
  wf-less-than
  wf-lex-prod
  wf-inv-image
  wf-measure
  wf-pred-nat
  wf-same-fst
  wf-empty

end

```

## 43 List: The datatype of finite lists

```

theory List
imports Plain Presburger Sledgehammer Recdef
uses (Tools/list-code.ML)
begin

datatype 'a list =
  Nil    ( $[]$ )
  | Cons 'a 'a list  (infixr # 65)

syntax
  — list Enumeration

```

$-list :: args \Rightarrow 'a\ list \quad ([(-)])$

#### translations

$[x, xs] == x \# [xs]$   
 $[x] == x \# []$

### 43.1 Basic list processing functions

#### primrec

$hd :: 'a\ list \Rightarrow 'a\ \mathbf{where}$   
 $hd\ (x \# xs) = x$

#### primrec

$tl :: 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$   
 $tl\ [] = []$   
 $| tl\ (x \# xs) = xs$

#### primrec

$last :: 'a\ list \Rightarrow 'a\ \mathbf{where}$   
 $last\ (x \# xs) = (if\ xs = []\ then\ x\ else\ last\ xs)$

#### primrec

$butlast :: 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$   
 $butlast\ [] = []$   
 $| butlast\ (x \# xs) = (if\ xs = []\ then\ []\ else\ x \# butlast\ xs)$

#### primrec

$set :: 'a\ list \Rightarrow 'a\ set\ \mathbf{where}$   
 $set\ [] = \{\}$   
 $| set\ (x \# xs) = insert\ x\ (set\ xs)$

#### primrec

$map :: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list\ \mathbf{where}$   
 $map\ f\ [] = []$   
 $| map\ f\ (x \# xs) = f\ x \# map\ f\ xs$

#### primrec

$append :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list\ (\mathbf{infixr}\ @\ 65)\ \mathbf{where}$   
 $append\ Nil:[]\ @\ ys = ys$   
 $| append\ Cons: (x \# xs)\ @\ ys = x \# xs\ @\ ys$

#### primrec

$rev :: 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$   
 $rev\ [] = []$   
 $| rev\ (x \# xs) = rev\ xs\ @\ [x]$

#### primrec

$filter :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list\ \mathbf{where}$   
 $filter\ P\ [] = []$

|  $\text{filter } P \ (x \# xs) = (\text{if } P \ x \text{ then } x \# \text{filter } P \ xs \text{ else } \text{filter } P \ xs)$

### **syntax**

— Special syntax for filter

$\text{-filter} :: [\text{pttrn}, 'a \text{ list}, \text{bool}] \Rightarrow 'a \text{ list} \quad ((1[-<--./-]))$

### **translations**

$[x < -xs \ . \ P] == \text{CONST filter } (\%x. P) \ xs$

### **syntax** (*xsymbols*)

$\text{-filter} :: [\text{pttrn}, 'a \text{ list}, \text{bool}] \Rightarrow 'a \text{ list} \quad ((1[\leftarrow-./-]))$

### **syntax** (*HTML output*)

$\text{-filter} :: [\text{pttrn}, 'a \text{ list}, \text{bool}] \Rightarrow 'a \text{ list} \quad ((1[\leftarrow-./-]))$

### **primrec**

$\text{foldl} :: ('b \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ where}$

$\text{foldl-Nil: foldl } f \ a \ [] = a$

|  $\text{foldl-Cons: foldl } f \ a \ (x \# xs) = \text{foldl } f \ (f \ a \ x) \ xs$

### **primrec**

$\text{foldr} :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \text{ where}$

$\text{foldr } f \ [] \ a = a$

|  $\text{foldr } f \ (x \# xs) \ a = f \ x \ (\text{foldr } f \ xs \ a)$

### **primrec**

$\text{concat} :: 'a \text{ list list} \Rightarrow 'a \text{ list where}$

$\text{concat } [] = []$

|  $\text{concat } (x \# xs) = x \ @ \ \text{concat } xs$

### **primrec** (*in monoid-add*)

$\text{listsum} :: 'a \text{ list} \Rightarrow 'a \text{ where}$

$\text{listsum } [] = 0$

|  $\text{listsum } (x \# xs) = x + \text{listsum } xs$

### **primrec**

$\text{drop} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list where}$

$\text{drop-Nil: drop } n \ [] = []$

|  $\text{drop-Cons: drop } n \ (x \# xs) = (\text{case } n \text{ of } 0 \Rightarrow x \# xs \mid \text{Suc } m \Rightarrow \text{drop } m \ xs)$

— Warning: simpset does not contain this definition, but separate theorems for  $n = 0$  and  $n = \text{Suc } k$

### **primrec**

$\text{take} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list where}$

$\text{take-Nil: take } n \ [] = []$

|  $\text{take-Cons: take } n \ (x \# xs) = (\text{case } n \text{ of } 0 \Rightarrow [] \mid \text{Suc } m \Rightarrow x \# \text{take } m \ xs)$

— Warning: simpset does not contain this definition, but separate theorems for  $n = 0$  and  $n = \text{Suc } k$

### **primrec**

$nth :: 'a \text{ list} \Rightarrow nat \Rightarrow 'a$  (**infixl** ! 100) **where**  
 $nth\text{-Cons}: (x \# xs) ! n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ k \Rightarrow xs ! k)$   
 — Warning: simpset does not contain this definition, but separate theorems for  $n = 0$  and  $n = Suc\ k$

**primrec**

$list\text{-update} :: 'a \text{ list} \Rightarrow nat \Rightarrow 'a \Rightarrow 'a \text{ list}$  **where**  
 $list\text{-update}\ []\ i\ v = []$   
 $\mid list\text{-update}\ (x \# xs)\ i\ v = (case\ i\ of\ 0 \Rightarrow v \# xs \mid Suc\ j \Rightarrow x \# list\text{-update}\ xs\ j\ v)$

**nonterminals**  $lupdbinds\ lupdbind$

**syntax**

$\text{-lupdbind} :: ['a, 'a] \Rightarrow lupdbind\quad ((2\text{-} := / \text{-}))$   
 $:: lupdbind \Rightarrow lupdbinds\quad (-)$   
 $\text{-lupdbinds} :: [lupdbind, lupdbinds] \Rightarrow lupdbinds\quad (-, / \text{-})$   
 $\text{-LUpdate} :: ['a, lupdbinds] \Rightarrow 'a\quad (-/[(-)]\ [900, 0]\ 900)$

**translations**

$\text{-LUpdate}\ xs\ (\text{-lupdbinds}\ b\ bs) == \text{-LUpdate}\ (\text{-LUpdate}\ xs\ b)\ bs$   
 $xs[i:=x] == \text{CONST}\ list\text{-update}\ xs\ i\ x$

**primrec**

$takeWhile :: ('a \Rightarrow bool) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $takeWhile\ P\ [] = []$   
 $\mid takeWhile\ P\ (x \# xs) = (if\ P\ x\ then\ x \# takeWhile\ P\ xs\ else\ [])$

**primrec**

$dropWhile :: ('a \Rightarrow bool) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $dropWhile\ P\ [] = []$   
 $\mid dropWhile\ P\ (x \# xs) = (if\ P\ x\ then\ dropWhile\ P\ xs\ else\ x \# xs)$

**primrec**

$zip :: 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow ('a \times 'b) \text{ list}$  **where**  
 $zip\ xs\ [] = []$   
 $\mid zip\text{-Cons}: zip\ xs\ (y \# ys) = (case\ xs\ of\ [] \Rightarrow [] \mid z \# zs \Rightarrow (z, y) \# zip\ zs\ ys)$   
 — Warning: simpset does not contain this definition, but separate theorems for  $xs = []$  and  $xs = z \# zs$

**primrec**

$upt :: nat \Rightarrow nat \Rightarrow nat \text{ list}$  ( $(1[-..</-])$ ) **where**  
 $upt\ 0: [i..<0] = []$   
 $\mid upt\text{-Suc}: [i..<(Suc\ j)] = (if\ i\ <= j\ then\ [i..<j]\ @\ [j]\ else\ [])$

**primrec**

$distinct :: 'a \text{ list} \Rightarrow bool$  **where**  
 $distinct\ [] \longleftrightarrow True$

$$| \text{distinct } (x \# xs) \longleftrightarrow x \notin \text{set } xs \wedge \text{distinct } xs$$
**primrec**

$\text{remdups} :: 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $\text{remdups } [] = []$   
 $| \text{remdups } (x \# xs) = (\text{if } x \in \text{set } xs \text{ then } \text{remdups } xs \text{ else } x \# \text{remdups } xs)$

**definition**

$\text{insert} :: 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $\text{insert } x xs = (\text{if } x \in \text{set } xs \text{ then } xs \text{ else } x \# xs)$

**hide-const** (open)  $\text{insert}$ **hide-fact** (open)  $\text{insert-def}$ **primrec**

$\text{remove1} :: 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $\text{remove1 } x [] = []$   
 $| \text{remove1 } x (y \# xs) = (\text{if } x = y \text{ then } xs \text{ else } y \# \text{remove1 } x xs)$

**primrec**

$\text{removeAll} :: 'a \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $\text{removeAll } x [] = []$   
 $| \text{removeAll } x (y \# xs) = (\text{if } x = y \text{ then } \text{removeAll } x xs \text{ else } y \# \text{removeAll } x xs)$

**primrec**

$\text{replicate} :: \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ list}$  **where**  
 $\text{replicate-0: replicate } 0 \ x = []$   
 $| \text{replicate-Suc: replicate } (\text{Suc } n) \ x = x \# \text{replicate } n \ x$

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

**abbreviation**

$\text{length} :: 'a \text{ list} \Rightarrow \text{nat}$  **where**  
 $\text{length} \equiv \text{size}$

**definition**

$\text{rotate1} :: 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $\text{rotate1 } xs = (\text{case } xs \text{ of } [] \Rightarrow [] \mid x \# xs \Rightarrow xs @ [x])$

**definition**

$\text{rotate} :: \text{nat} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list}$  **where**  
 $\text{rotate } n = \text{rotate1} \ ^{\wedge} n$

**definition**

$\text{list-all2} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow \text{bool}$  **where**  
 $[\text{code del}]: \text{list-all2 } P \ xs \ ys =$   
 $(\text{length } xs = \text{length } ys \wedge (\forall (x, y) \in \text{set } (\text{zip } xs \ ys). P \ x \ y))$

**definition**

```

sublist :: 'a list => nat set => 'a list where
sublist xs A = map fst (filter (λp. snd p ∈ A) (zip xs [0..

```

**primrec**

```

splice :: 'a list ⇒ 'a list ⇒ 'a list where
splice [] ys = ys
| splice (x # xs) ys = (if ys = [] then x # xs else x # hd ys # splice xs (tl ys))
— Warning: simpset does not contain the second eqn but a derived one.

```

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient implementations.

**context** *linorder*  
**begin**

```

fun sorted :: 'a list ⇒ bool where
sorted [] ⟷ True |
sorted [x] ⟷ True |
sorted (x#y#zs) ⟷ x ≤ y ∧ sorted (y#zs)

```

```

primrec insort-key :: ('b ⇒ 'a) ⇒ 'b ⇒ 'b list ⇒ 'b list where
insort-key f x [] = [x] |
insort-key f x (y#ys) = (if f x ≤ f y then (x#y#ys) else y#(insort-key f x ys))

```

```

definition sort-key :: ('b ⇒ 'a) ⇒ 'b list ⇒ 'b list where
sort-key f xs = foldr (insort-key f) xs []

```

**abbreviation** *sort* ≡ *sort-key* (λx. x)  
**abbreviation** *insort* ≡ *insort-key* (λx. x)

```

definition insort-insert :: 'a ⇒ 'a list ⇒ 'a list where
insort-insert x xs = (if x ∈ set xs then xs else insort x xs)

```

**end**

**43.1.1 List comprehension**

Input syntax for Haskell-like list comprehension notation. Typical example:  $[(x,y). x \leftarrow xs, y \leftarrow ys, x \neq y]$ , the list of all pairs of distinct elements from  $xs$  and  $ys$ . The syntax is as in Haskell, except that  $|$  becomes a dot (like in Isabelle’s set comprehension):  $[e. x \leftarrow xs, \dots]$  rather than  $[e \mid x \leftarrow xs, \dots]$ .

The qualifiers after the dot are

**generators**  $p \leftarrow xs$ , where  $p$  is a pattern and  $xs$  an expression of list type,  
or



```

[a, b] @ [c, d] = [a, b, c, d]
length [a, b, c] = 3
set [a, b, c] = {a, b, c}
map f [a, b, c] = [f a, f b, f c]
rev [a, b, c] = [c, b, a]
hd [a, b, c, d] = a
tl [a, b, c, d] = [b, c, d]
last [a, b, c, d] = d
butlast [a, b, c, d] = [a, b, c]
filter (λn::nat. n<2) [0,2,1] = [0,1]
concat [[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]
foldl f x [a, b, c] = f (f (f x a) b) c
foldr f [a, b, c] x = f a (f b (f c x))
zip [a, b, c] [x, y, z] = [(a, x), (b, y), (c, z)]
zip [a, b] [x, y, z] = [(a, x), (b, y)]
splice [a, b, c] [x, y, z] = [a, x, b, y, c, z]
splice [a, b, c, d] [x, y] = [a, x, b, y, c, d]
take 2 [a, b, c, d] = [a, b]
take 6 [a, b, c, d] = [a, b, c, d]
drop 2 [a, b, c, d] = [c, d]
drop 6 [a, b, c, d] = []
takeWhile (λn. n < 3) [1, 2, 3, 0] = [1, 2]
dropWhile (λn. n < 3) [1, 2, 3, 0] = [3, 0]
distinct [2, 0, 1]
remdups [2, 0, 2, 1, 2] = [0, 1, 2]
List.insert 2 [0, 1, 2] = [0, 1, 2]
List.insert 3 [0, 1, 2] = [3, 0, 1, 2]
remove1 2 [2, 0, 2, 1, 2] = [0, 2, 1, 2]
removeAll 2 [2, 0, 2, 1, 2] = [0, 1]
[a, b, c, d] ! 2 = c
[a, b, c, d][2 := x] = [a, b, x, d]
sublist [a, b, c, d, e] {0, 2, 3} = [a, c, d]
rotate1 [a, b, c, d] = [b, c, d, a]
rotate 3 [a, b, c, d] = [d, a, b, c]
replicate 4 a = [a, a, a, a]
[2..<5] = [2, 3, 4]
listsum [1, 2, 3] = 6

```

Figure 1: Characteristic examples

**guards**  $b$ , where  $b$  is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of  $[e. x \leftarrow xs]$  is optimized to  $\text{map } (\lambda x. e) xs$ .

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

**nonterminals**  $lc\text{-}qual$   $lc\text{-}quals$

**syntax**

```
-listcompr :: 'a  $\Rightarrow$  lc-qual  $\Rightarrow$  lc-quals  $\Rightarrow$  'a list  ([ - . --)
-lc-gen :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  lc-qual (- <- -)
-lc-test :: bool  $\Rightarrow$  lc-qual (-)

-lc-end :: lc-quals ()
-lc-quals :: lc-qual  $\Rightarrow$  lc-quals  $\Rightarrow$  lc-quals (, --)
-lc-abs :: 'a  $\Rightarrow$  'b list  $\Rightarrow$  'b list
```

**syntax** ( $xsymbols$ )

```
-lc-gen :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  lc-qual (-  $\leftarrow$  -)
```

**syntax** (*HTML output*)

```
-lc-gen :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  lc-qual (-  $\leftarrow$  -)
```

**parse-translation** (**advanced**)  $\ll$

*let*

```
val NilC = Syntax.const @{const-syntax Nil};
val ConsC = Syntax.const @{const-syntax Cons};
val mapC = Syntax.const @{const-syntax map};
val concatC = Syntax.const @{const-syntax concat};
val IfC = Syntax.const @{const-syntax If};
```

```
fun singl x = ConsC $ x $ NilC;
```

```
fun pat-tr ctxt p e opti = (* %x. case x of p => e | - => [] *)
```

*let*

```
val x = Free (Name.variant (fold Term.add-free-names [p, e] []) x, dummyT);
val e = if opti then singl e else e;
val case1 = Syntax.const @{syntax-const -case1} $ p $ e;
val case2 =
  Syntax.const @{syntax-const -case1} $
    Syntax.const @{const-syntax dummy-pattern} $ NilC;
val cs = Syntax.const @{syntax-const -case2} $ case1 $ case2;
val ft = Datatype-Case.case-tr false Datatype.info-of-constr ctxt [x, cs];
```

```

    in lambda x ft end;

fun abs-tr ctxt (p as Free (s, T)) e opti =
  let
    val thy = ProofContext.theory-of ctxt;
    val s' = Sign.intern-const thy s;
  in
    if Sign.declared-const thy s'
    then (pat-tr ctxt p e opti, false)
    else (lambda p e, true)
  end
  | abs-tr ctxt p e opti = (pat-tr ctxt p e opti, false);

fun lc-tr ctxt [e, Const (@{syntax-const -lc-test}, -) $ b, qs] =
  let
    val res =
      (case qs of
        Const (@{syntax-const -lc-end}, -) => singl e
      | Const (@{syntax-const -lc-quals}, -) $ q $ qs => lc-tr ctxt [e, q, qs]);
    in IfC $ b $ res $ NilC end
  | lc-tr ctxt
    [e, Const (@{syntax-const -lc-gen}, -) $ p $ es,
     Const(@{syntax-const -lc-end}, -)] =
    (case abs-tr ctxt p e true of
      (f, true) => mapC $ f $ es
    | (f, false) => concatC $ (mapC $ f $ es))
  | lc-tr ctxt
    [e, Const (@{syntax-const -lc-gen}, -) $ p $ es,
     Const (@{syntax-const -lc-quals}, -) $ q $ qs] =
    let val e' = lc-tr ctxt [e, q, qs];
    in concatC $ (mapC $ (fst (abs-tr ctxt p e' false)) $ es) end;

in [(@{syntax-const -listcompr}, lc-tr)] end
>>

term [(x,y,z). b]
term [(x,y,z). x←xs]
term [e x y. x←xs, y←ys]
term [(x,y,z). x<a, x>b]
term [(x,y,z). x←xs, x>b]
term [(x,y,z). x<a, x←xs]
term [(x,y). Cons True x ← xs]
term [(x,y,z). Cons x [] ← xs]
term [(x,y,z). x<a, x>b, x=d]
term [(x,y,z). x<a, x>b, y←ys]
term [(x,y,z). x<a, x←xs, y>b]
term [(x,y,z). x<a, x←xs, y←ys]
term [(x,y,z). x←xs, x>b, y<a]
term [(x,y,z). x←xs, x>b, y←ys]

```

**term**  $[(x,y,z). x \leftarrow xs, y \leftarrow ys, y > x]$   
**term**  $[(x,y,z). x \leftarrow xs, y \leftarrow ys, z \leftarrow zs]$

### 43.1.2 $[]$ and $op \#$

**lemma** *not-Cons-self* [simp]:

$xs \neq x \# xs$

**by** (*induct xs*) *auto*

**lemmas** *not-Cons-self2* [simp] = *not-Cons-self* [*symmetric*]

**lemma** *neq-Nil-conv*:  $(xs \neq []) = (\exists y \ ys. xs = y \# ys)$

**by** (*induct xs*) *auto*

**lemma** *length-induct*:

$(\bigwedge xs. \forall ys. \text{length } ys < \text{length } xs \longrightarrow P \ ys \Longrightarrow P \ xs) \Longrightarrow P \ xs$

**by** (*rule measure-induct [of length]*) *iprover*

**lemma** *list-nonempty-induct* [*consumes 1*, *case-names single cons*]:

**assumes**  $xs \neq []$

**assumes** *single*:  $\bigwedge x. P \ [x]$

**assumes** *cons*:  $\bigwedge x \ xs. xs \neq [] \Longrightarrow P \ xs \Longrightarrow P \ (x \# xs)$

**shows**  $P \ xs$

**using**  $\langle xs \neq [] \rangle$  **proof** (*induct xs*)

**case** *Nil* **then show** ?*case* **by** *simp*

**next**

**case** (*Cons x xs*) **show** ?*case* **proof** (*cases xs*)

**case** *Nil* **with** *single* **show** ?*thesis* **by** *simp*

**next**

**case** *Cons* **then have**  $xs \neq []$  **by** *simp*

**moreover with** *Cons.hyps* **have**  $P \ xs$  .

**ultimately show** ?*thesis* **by** (*rule cons*)

**qed**

**qed**

### 43.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

**lemma** *length-append* [simp]:  $\text{length } (xs @ ys) = \text{length } xs + \text{length } ys$

**by** (*induct xs*) *auto*

**lemma** *length-map* [simp]:  $\text{length } (\text{map } f \ xs) = \text{length } xs$

**by** (*induct xs*) *auto*

**lemma** *length-rev* [simp]:  $\text{length } (\text{rev } xs) = \text{length } xs$

**by** (*induct xs*) *auto*

**lemma** *length-tl* [simp]:  $\text{length } (\text{tl } xs) = \text{length } xs - 1$

**by** (*cases xs*) *auto*

**lemma** *length-0-conv* [iff]:  $(\text{length } xs = 0) = (xs = [])$   
**by** (*induct xs*) *auto*

**lemma** *length-greater-0-conv* [iff]:  $(0 < \text{length } xs) = (xs \neq [])$   
**by** (*induct xs*) *auto*

**lemma** *length-pos-if-in-set*:  $x : \text{set } xs \implies \text{length } xs > 0$   
**by** *auto*

**lemma** *length-Suc-conv*:  
 $(\text{length } xs = \text{Suc } n) = (\exists y \text{ } ys. xs = y \# ys \wedge \text{length } ys = n)$   
**by** (*induct xs*) *auto*

**lemma** *Suc-length-conv*:  
 $(\text{Suc } n = \text{length } xs) = (\exists y \text{ } ys. xs = y \# ys \wedge \text{length } ys = n)$   
**apply** (*induct xs, simp, simp*)  
**apply** *blast*  
**done**

**lemma** *impossible-Cons*:  $\text{length } xs <= \text{length } ys \implies xs = x \# ys = \text{False}$   
**by** (*induct xs*) *auto*

**lemma** *list-induct2* [consumes 1, case-names Nil Cons]:  
 $\text{length } xs = \text{length } ys \implies P [] [] \implies$   
 $(\bigwedge x \text{ } xs \text{ } y \text{ } ys. \text{length } xs = \text{length } ys \implies P \text{ } xs \text{ } ys \implies P (x \# xs) (y \# ys))$   
 $\implies P \text{ } xs \text{ } ys$   
**proof** (*induct xs arbitrary: ys*)  
**case** Nil **then show** ?case **by** *simp*  
**next**  
**case** (Cons *x xs ys*) **then show** ?case **by** (*cases ys*) *simp-all*  
**qed**

**lemma** *list-induct3* [consumes 2, case-names Nil Cons]:  
 $\text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P [] [] [] \implies$   
 $(\bigwedge x \text{ } xs \text{ } y \text{ } ys \text{ } z \text{ } zs. \text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies P \text{ } xs \text{ } ys \text{ } zs$   
 $\implies P (x \# xs) (y \# ys) (z \# zs))$   
 $\implies P \text{ } xs \text{ } ys \text{ } zs$   
**proof** (*induct xs arbitrary: ys zs*)  
**case** Nil **then show** ?case **by** *simp*  
**next**  
**case** (Cons *x xs ys zs*) **then show** ?case **by** (*cases ys, simp-all*)  
(*cases zs, simp-all*)  
**qed**

**lemma** *list-induct4* [consumes 3, case-names Nil Cons]:  
 $\text{length } xs = \text{length } ys \implies \text{length } ys = \text{length } zs \implies \text{length } zs = \text{length } ws \implies$   
 $P [] [] [] [] \implies (\bigwedge x \text{ } xs \text{ } y \text{ } ys \text{ } z \text{ } zs \text{ } w \text{ } ws. \text{length } xs = \text{length } ys \implies$   
 $\text{length } ys = \text{length } zs \implies \text{length } zs = \text{length } ws \implies P \text{ } xs \text{ } ys \text{ } zs \text{ } ws \implies$

```

    P (x#xs) (y#ys) (z#zs) (w#ws)) ==> P xs ys zs ws
proof (induct xs arbitrary: ys zs ws)
  case Nil then show ?case by simp
next
  case (Cons x xs ys zs ws) then show ?case by ((cases ys, simp-all), (cases
zs,simp-all)) (cases ws, simp-all)
qed

```

```

lemma list-induct2':
  [| P [] |];
  |>x xs. P (x#xs) [|];
  |>y ys. P [|] (y#ys);
  |>x xs y ys. P xs ys ==> P (x#xs) (y#ys) |]
  ==> P xs ys
by (induct xs arbitrary: ys) (case-tac x, auto)+

```

```

lemma neq-if-length-neq: length xs ≠ length ys ==> (xs = ys) == False
by (rule Eq-FalseI) auto

```

```

simproc-setup list-neq ((xs::'a list) = ys) = <<
  (*)
  Reduces xs=ys to False if xs and ys cannot be of the same length.
  This is the case if the atomic sublists of one are a submultiset
  of those of the other list and there are fewer Cons's in one than the other.
  *)

```

```

let

```

```

fun len (Const(@{const-name Nil},-)) acc = acc
| len (Const(@{const-name Cons},-) $ - $ xs) (ts,n) = len xs (ts,n+1)
| len (Const(@{const-name append},-) $ xs $ ys) acc = len xs (len ys acc)
| len (Const(@{const-name rev},-) $ xs) acc = len xs acc
| len (Const(@{const-name map},-) $ - $ xs) acc = len xs acc
| len t (ts,n) = (t::ts,n);

```

```

fun list-neq - ss ct =
  let
    val (Const(-,eqT) $ lhs $ rhs) = Thm.term-of ct;
    val (ls,m) = len lhs ([],0) and (rs,n) = len rhs ([],0);
    fun prove-neq() =
      let
        val Type(-,listT::-) = eqT;
        val size = HOLogic.size-const listT;
        val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
        val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
        val thm = Goal.prove (Simplifier.the-context ss) [|] [|] neq-len
          (K (simp-tac (Simplifier.inherit-context ss @ {simpset}) 1));
        in SOME (thm RS @ {thm neq-if-length-neq}) end
      in

```

```

    if  $m < n$  andalso submultiset (op aconv) (ls,rs) orelse
       $n < m$  andalso submultiset (op aconv) (rs,ls)
    then prove-neq() else NONE
  end;
in list-neq end;
>>

```

#### 43.1.4 @ – append

**lemma** *append-assoc* [simp]:  $(xs @ ys) @ zs = xs @ (ys @ zs)$   
**by** (induct xs) auto

**lemma** *append-Nil2* [simp]:  $xs @ [] = xs$   
**by** (induct xs) auto

**lemma** *append-is-Nil-conv* [iff]:  $(xs @ ys = []) = (xs = [] \wedge ys = [])$   
**by** (induct xs) auto

**lemma** *Nil-is-append-conv* [iff]:  $([] = xs @ ys) = (xs = [] \wedge ys = [])$   
**by** (induct xs) auto

**lemma** *append-self-conv* [iff]:  $(xs @ ys = xs) = (ys = [])$   
**by** (induct xs) auto

**lemma** *self-append-conv* [iff]:  $(xs = xs @ ys) = (ys = [])$   
**by** (induct xs) auto

**lemma** *append-eq-append-conv* [simp, no-atp]:  
 $\text{length } xs = \text{length } ys \vee \text{length } us = \text{length } vs$   
 $\implies (xs @ us = ys @ vs) = (xs = ys \wedge us = vs)$   
**apply** (induct xs arbitrary: ys)  
**apply** (case-tac ys, simp, force)  
**apply** (case-tac ys, force, simp)  
**done**

**lemma** *append-eq-append-conv2*:  $(xs @ ys = zs @ ts) =$   
 $(\exists us. xs = zs @ us \ \& \ us @ ys = ts \mid xs @ us = zs \ \& \ ys = us @ ts)$   
**apply** (induct xs arbitrary: ys zs ts)  
**apply** fastsimp  
**apply** (case-tac zs)  
**apply** simp  
**apply** fastsimp  
**done**

**lemma** *same-append-eq* [iff, induct-simp]:  $(xs @ ys = xs @ zs) = (ys = zs)$   
**by** simp

**lemma** *append1-eq-conv* [iff]:  $(xs @ [x] = ys @ [y]) = (xs = ys \wedge x = y)$   
**by** simp

**lemma** *append-same-eq* [*iff*, *induct-simp*]:  $(ys @ xs = zs @ xs) = (ys = zs)$   
**by** *simp*

**lemma** *append-self-conv2* [*iff*]:  $(xs @ ys = ys) = (xs = [])$   
**using** *append-same-eq* [*of* - - []] **by** *auto*

**lemma** *self-append-conv2* [*iff*]:  $(ys = xs @ ys) = (xs = [])$   
**using** *append-same-eq* [*of* []] **by** *auto*

**lemma** *hd-Cons-tl* [*simp*, *no-atp*]:  $xs \neq [] \implies hd\ xs \# tl\ xs = xs$   
**by** (*induct xs*) *auto*

**lemma** *hd-append*:  $hd\ (xs @ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$   
**by** (*induct xs*) *auto*

**lemma** *hd-append2* [*simp*]:  $xs \neq [] \implies hd\ (xs @ ys) = hd\ xs$   
**by** (*simp add: hd-append split: list.split*)

**lemma** *tl-append*:  $tl\ (xs @ ys) = (case\ xs\ of\ [] \implies tl\ ys \mid z \# zs \implies zs @ ys)$   
**by** (*simp split: list.split*)

**lemma** *tl-append2* [*simp*]:  $xs \neq [] \implies tl\ (xs @ ys) = tl\ xs @ ys$   
**by** (*simp add: tl-append split: list.split*)

**lemma** *Cons-eq-append-conv*:  $x \# xs = ys @ zs =$   
 $(ys = [] \ \& \ x \# xs = zs \mid (EX\ ys'.\ x \# ys' = ys \ \& \ xs = ys' @ zs))$   
**by** (*cases ys*) *auto*

**lemma** *append-eq-Cons-conv*:  $(ys @ zs = x \# xs) =$   
 $(ys = [] \ \& \ zs = x \# xs \mid (EX\ ys'.\ ys = x \# ys' \ \& \ ys' @ zs = xs))$   
**by** (*cases ys*) *auto*

Trivial rules for solving @-equations automatically.

**lemma** *eq-Nil-appendI*:  $xs = ys \implies xs = [] @ ys$   
**by** *simp*

**lemma** *Cons-eq-appendI*:  
 $[| x \# xs1 = ys; xs = xs1 @ zs |] \implies x \# xs = ys @ zs$   
**by** (*drule sym*) *simp*

**lemma** *append-eq-appendI*:  
 $[| xs @ xs1 = zs; ys = xs1 @ us |] \implies xs @ ys = zs @ us$   
**by** (*drule sym*) *simp*

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.



```

ML <<
local

fun last (cons as Const(@{const-name Cons},-) $ - $ xs) =
  (case xs of Const(@{const-name Nil},-) => cons | - => last xs)
  | last (Const(@{const-name append},-) $ - $ ys) = last ys
  | last t = t;

fun list1 (Const(@{const-name Cons},-) $ - $ Const(@{const-name Nil},-)) = true
  | list1 - = false;

fun butlast ((cons as Const(@{const-name Cons},-) $ x) $ xs) =
  (case xs of Const(@{const-name Nil},-) => xs | - => cons $ butlast xs)
  | butlast ((app as Const(@{const-name append},-) $ xs) $ ys) = app $ butlast ys
  | butlast xs = Const(@{const-name Nil},fastype-of xs);

val rearr-ss = HOL-basic-ss addsimps [@{thm append-assoc},
  @{thm append-Nil}, @{thm append-Cons}];

fun list-eq ss (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
  let
    val lastl = last lhs and lastr = last rhs;
    fun rearr conv =
      let
        val lhs1 = butlast lhs and rhs1 = butlast rhs;
        val Type(-,listT::-) = eqT
        val appT = [listT,listT] ----> listT
        val app = Const(@{const-name append},appT)
        val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
        val thm = Goal.prove (Simplifier.the-context ss) [] [] eq
          (K (simp-tac (Simplifier.inherit-context ss rearr-ss) 1));
        in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;
      in
        if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
        else if lastl aconv lastr then rearr @{thm append-same-eq}
        else NONE
      end;
  in

val list-eq-simproc =
  Simplifier.simproc @{theory} list-eq [(xs::'a list) = ys] (K list-eq);

end;

Addsimprocs [list-eq-simproc];
>>

```

**43.1.5** *map*

**lemma** *map-ext*:  $(!!x. x : \text{set } xs \longrightarrow f x = g x) \implies \text{map } f xs = \text{map } g xs$   
**by** (*induct xs*) *simp-all*

**lemma** *map-ident* [*simp*]:  $\text{map } (\lambda x. x) = (\lambda xs. xs)$   
**by** (*rule ext*, *induct-tac xs*) *auto*

**lemma** *map-append* [*simp*]:  $\text{map } f (xs @ ys) = \text{map } f xs @ \text{map } f ys$   
**by** (*induct xs*) *auto*

**lemma** *map-map* [*simp*]:  $\text{map } f (\text{map } g xs) = \text{map } (f \circ g) xs$   
**by** (*induct xs*) *auto*

**lemma** *map-comp-map* [*simp*]:  $((\text{map } f) \circ (\text{map } g)) = \text{map}(f \circ g)$   
**apply**(*rule ext*)  
**apply**(*simp*)  
**done**

**lemma** *rev-map*:  $\text{rev } (\text{map } f xs) = \text{map } f (\text{rev } xs)$   
**by** (*induct xs*) *auto*

**lemma** *map-eq-conv* [*simp*]:  $(\text{map } f xs = \text{map } g xs) = (!x : \text{set } xs. f x = g x)$   
**by** (*induct xs*) *auto*

**lemma** *map-cong* [*fundef-cong*, *recdef-cong*]:  
 $xs = ys \implies (!x. x : \text{set } ys \implies f x = g x) \implies \text{map } f xs = \text{map } g ys$   
— a congruence rule for *map*  
**by** *simp*

**lemma** *map-is-Nil-conv* [*iff*]:  $(\text{map } f xs = []) = (xs = [])$   
**by** (*cases xs*) *auto*

**lemma** *Nil-is-map-conv* [*iff*]:  $([] = \text{map } f xs) = (xs = [])$   
**by** (*cases xs*) *auto*

**lemma** *map-eq-Cons-conv*:  
 $(\text{map } f xs = y \# ys) = (\exists z zs. xs = z \# zs \wedge f z = y \wedge \text{map } f zs = ys)$   
**by** (*cases xs*) *auto*

**lemma** *Cons-eq-map-conv*:  
 $(x \# xs = \text{map } f ys) = (\exists z zs. ys = z \# zs \wedge x = f z \wedge xs = \text{map } f zs)$   
**by** (*cases ys*) *auto*

**lemmas** *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]  
**lemmas** *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]  
**declare** *map-eq-Cons-D* [*dest!*] *Cons-eq-map-D* [*dest!*]

**lemma** *ex-map-conv*:  
 $(EX xs. ys = \text{map } f xs) = (ALL y : \text{set } ys. EX x. y = f x)$

**by**(*induct ys, auto simp add: Cons-eq-map-conv*)

**lemma** *map-eq-imp-length-eq*:

**assumes** *map f xs = map g ys*

**shows** *length xs = length ys*

**using** *assms* **proof** (*induct ys arbitrary: xs*)

**case** *Nil* **then show** *?case* **by** *simp*

**next**

**case** (*Cons y ys*) **then obtain** *z zs* **where** *xs = z # zs* **by** *auto*

**from** *Cons xs* **have** *map f zs = map g ys* **by** *simp*

**moreover with** *Cons* **have** *length zs = length ys* **by** *blast*

**with** *xs* **show** *?case* **by** *simp*

**qed**

**lemma** *map-inj-on*:

**[|** *map f xs = map f ys; inj-on f (set xs Un set ys)* **|]**

**==>** *xs = ys*

**apply**(*frule map-eq-imp-length-eq*)

**apply**(*rotate-tac -1*)

**apply**(*induct rule:list-induct2*)

**apply** *simp*

**apply**(*simp*)

**apply** (*blast intro:sym*)

**done**

**lemma** *inj-on-map-eq-map*:

*inj-on f (set xs Un set ys) ==> (map f xs = map f ys) = (xs = ys)*

**by**(*blast dest:map-inj-on*)

**lemma** *map-injective*:

*map f xs = map f ys ==> inj f ==> xs = ys*

**by** (*induct ys arbitrary: xs*) (*auto dest!:injD*)

**lemma** *inj-map-eq-map[simp]*: *inj f ==> (map f xs = map f ys) = (xs = ys)*

**by**(*blast dest:map-injective*)

**lemma** *inj-mapI*: *inj f ==> inj (map f)*

**by** (*iprover dest: map-injective injD intro: inj-onI*)

**lemma** *inj-mapD*: *inj (map f) ==> inj f*

**apply** (*unfold inj-on-def, clarify*)

**apply** (*erule-tac x = [x] in ballE*)

**apply** (*erule-tac x = [y] in ballE, simp, blast*)

**apply** *blast*

**done**

**lemma** *inj-map[iff]*: *inj (map f) = inj f*

**by** (*blast dest: inj-mapD intro: inj-mapI*)

```

lemma inj-on-mapI: inj-on f (⋃ (set ‘ A))  $\implies$  inj-on (map f) A
apply (rule inj-onI)
apply (erule map-inj-on)
apply (blast intro: inj-onI dest: inj-onD)
done

```

```

lemma map-idI: ( $\bigwedge x. x \in \text{set } xs \implies f\ x = x$ )  $\implies$  map f xs = xs
by (induct xs, auto)

```

```

lemma map-fun-upd [simp]:  $y \notin \text{set } xs \implies \text{map } (f(y:=v))\ xs = \text{map } f\ xs$ 
by (induct xs) auto

```

```

lemma map-fst-zip[simp]:
  length xs = length ys  $\implies$  map fst (zip xs ys) = xs
by (induct rule: list-induct2, simp-all)

```

```

lemma map-snd-zip[simp]:
  length xs = length ys  $\implies$  map snd (zip xs ys) = ys
by (induct rule: list-induct2, simp-all)

```

#### 43.1.6 rev

```

lemma rev-append [simp]: rev (xs @ ys) = rev ys @ rev xs
by (induct xs) auto

```

```

lemma rev-rev-ident [simp]: rev (rev xs) = xs
by (induct xs) auto

```

```

lemma rev-swap: (rev xs = ys) = (xs = rev ys)
by auto

```

```

lemma rev-is-Nil-conv [iff]: (rev xs = []) = (xs = [])
by (induct xs) auto

```

```

lemma Nil-is-rev-conv [iff]: ([] = rev xs) = (xs = [])
by (induct xs) auto

```

```

lemma rev-singleton-conv [simp]: (rev xs = [x]) = (xs = [x])
by (cases xs) auto

```

```

lemma singleton-rev-conv [simp]: ([x] = rev xs) = (xs = [x])
by (cases xs) auto

```

```

lemma rev-is-rev-conv [iff]: (rev xs = rev ys) = (xs = ys)
apply (induct xs arbitrary: ys, force)
apply (case-tac ys, simp, force)
done

```

```

lemma inj-on-rev[iff]: inj-on rev A

```

**by**(*simp add:inj-on-def*)

**lemma** *rev-induct* [*case-names Nil snoc*]:  

$$[\![ P \ ]\!]; \text{!!}x\ xs. P\ xs \implies P\ (xs\ @\ [x])\ ]\implies P\ xs$$
  
**apply**(*simplesubst rev-rev-ident[symmetric]*)  
**apply**(*rule-tac list = rev xs in list.induct, simp-all*)  
**done**

**lemma** *rev-exhaust* [*case-names Nil snoc*]:  

$$(xs = [] \implies P) \implies (\text{!!}ys\ y. xs = ys\ @\ [y] \implies P) \implies P$$
  
**by** (*induct xs rule: rev-induct*) *auto*

**lemmas** *rev-cases = rev-exhaust*

**lemma** *rev-eq-Cons-iff*[*iff*]:  $(rev\ xs = y\ \#ys) = (xs = rev\ ys\ @\ [y])$   
**by**(*rule rev-cases[of xs]*) *auto*

#### 43.1.7 set

**lemma** *finite-set* [*iff*]: *finite* (*set xs*)  
**by** (*induct xs*) *auto*

**lemma** *set-append* [*simp*]:  $set\ (xs\ @\ ys) = (set\ xs \cup set\ ys)$   
**by** (*induct xs*) *auto*

**lemma** *hd-in-set*[*simp*]:  $xs \neq [] \implies hd\ xs : set\ xs$   
**by**(*cases xs*) *auto*

**lemma** *set-subset-Cons*:  $set\ xs \subseteq set\ (x\ \# xs)$   
**by** *auto*

**lemma** *set-ConsD*:  $y \in set\ (x\ \# xs) \implies y=x \vee y \in set\ xs$   
**by** *auto*

**lemma** *set-empty* [*iff*]:  $(set\ xs = \{\}) = (xs = [])$   
**by** (*induct xs*) *auto*

**lemma** *set-empty2*[*iff*]:  $(\{\} = set\ xs) = (xs = [])$   
**by**(*induct xs*) *auto*

**lemma** *set-rev* [*simp*]:  $set\ (rev\ xs) = set\ xs$   
**by** (*induct xs*) *auto*

**lemma** *set-map* [*simp*]:  $set\ (map\ f\ xs) = f\cdot(set\ xs)$   
**by** (*induct xs*) *auto*

**lemma** *set-filter* [*simp*]:  $set\ (filter\ P\ xs) = \{x. x : set\ xs \wedge P\ x\}$   
**by** (*induct xs*) *auto*

**lemma** *set-upt* [*simp*]:  $\text{set}[i..<j] = \{i..<j\}$   
**by** (*induct j*) (*simp-all add: atLeastLessThanSuc*)

**lemma** *split-list*:  $x : \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs$   
**proof** (*induct xs*)  
 case *Nil* **thus** ?*case* **by** *simp*  
**next**  
 case *Cons* **thus** ?*case* **by** (*auto intro: Cons-eq-appendI*)  
**qed**

**lemma** *in-set-conv-decomp*:  $x \in \text{set } xs \longleftrightarrow (\exists ys\ zs. xs = ys @ x \# zs)$   
**by** (*auto elim: split-list*)

**lemma** *split-list-first*:  $x : \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys$   
**proof** (*induct xs*)  
 case *Nil* **thus** ?*case* **by** *simp*  
**next**  
 case (*Cons a xs*)  
 show ?*case*  
**proof** *cases*  
 assume  $x = a$  **thus** ?*case* **using** *Cons* **by** *fastsimp*  
**next**  
 assume  $x \neq a$  **thus** ?*case* **using** *Cons* **by** (*fastsimp intro!: Cons-eq-appendI*)  
**qed**  
**qed**

**lemma** *in-set-conv-decomp-first*:  
 $(x : \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys)$   
**by** (*auto dest!: split-list-first*)

**lemma** *split-list-last*:  $x : \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs$   
**proof** (*induct xs rule: rev-induct*)  
 case *Nil* **thus** ?*case* **by** *simp*  
**next**  
 case (*snoc a xs*)  
 show ?*case*  
**proof** *cases*  
 assume  $x = a$  **thus** ?*case* **using** *snoc* **by** *simp* (*metis ex-in-conv set-empty2*)  
**next**  
 assume  $x \neq a$  **thus** ?*case* **using** *snoc* **by** *fastsimp*  
**qed**  
**qed**

**lemma** *in-set-conv-decomp-last*:  
 $(x : \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs)$   
**by** (*auto dest!: split-list-last*)

**lemma** *split-list-prop*:  $\exists x \in \text{set } xs. P\ x \implies \exists ys\ x\ zs. xs = ys @ x \# zs \ \& \ P\ x$

```

proof (induct xs)
  case Nil thus ?case by simp
next
  case Cons thus ?case
    by(simp add:Bex-def)(metis append-Cons append.simps(1))
qed

```

```

lemma split-list-propE:
  assumes  $\exists x \in \text{set } xs. P x$ 
  obtains  $ys\ x\ zs$  where  $xs = ys @ x \# zs$  and  $P x$ 
using split-list-prop [OF assms] by blast

```

```

lemma split-list-first-prop:
   $\exists x \in \text{set } xs. P x \implies$ 
   $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P x \wedge (\forall y \in \text{set } ys. \neg P y)$ 
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  show ?case
  proof cases
    assume  $P x$ 
    thus ?thesis by simp
    (metis Un-upper1 contra-subsetD in-set-conv-decomp-first self-append-conv2
  set-append)
  next
    assume  $\neg P x$ 
    hence  $\exists x \in \text{set } xs. P x$  using Cons(2) by simp
    thus ?thesis using  $\langle \neg P x \rangle$  Cons(1) by (metis append-Cons set-ConsD)
  qed
qed

```

```

lemma split-list-first-propE:
  assumes  $\exists x \in \text{set } xs. P x$ 
  obtains  $ys\ x\ zs$  where  $xs = ys @ x \# zs$  and  $P x$  and  $\forall y \in \text{set } ys. \neg P y$ 
using split-list-first-prop [OF assms] by blast

```

```

lemma split-list-first-prop-iff:
   $(\exists x \in \text{set } xs. P x) \longleftrightarrow$ 
   $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P x \wedge (\forall y \in \text{set } ys. \neg P y))$ 
by (rule, erule split-list-first-prop) auto

```

```

lemma split-list-last-prop:
   $\exists x \in \text{set } xs. P x \implies$ 
   $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P x \wedge (\forall z \in \text{set } zs. \neg P z)$ 
proof(induct xs rule:rev-induct)
  case Nil thus ?case by simp
next
  case (snoc x xs)

```

```

show ?case
proof cases
  assume  $P\ x$  thus ?thesis by (metis emptyE set-empty)
next
  assume  $\neg P\ x$ 
  hence  $\exists x \in \text{set } xs. P\ x$  using snoc(2) by simp
  thus ?thesis using  $\langle \neg P\ x \rangle$  snoc(1) by fastsimp
qed
qed

```

**lemma** *split-list-last-propE*:  
 assumes  $\exists x \in \text{set } xs. P\ x$   
 obtains  $ys\ x\ zs$  where  $xs = ys @ x \# zs$  and  $P\ x$  and  $\forall z \in \text{set } zs. \neg P\ z$   
 using *split-list-last-prop* [OF *assms*] by blast

**lemma** *split-list-last-prop-iff*:  
 $(\exists x \in \text{set } xs. P\ x) \longleftrightarrow$   
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in \text{set } zs. \neg P\ z))$   
 by (metis *split-list-last-prop* [where  $P=P$ ] *in-set-conv-decomp*)

**lemma** *finite-list*:  $\text{finite } A \implies \exists xs. \text{set } xs = A$   
 by (erule *finite-induct*)  
 (auto simp add: *set.simps*(2) [symmetric] *simp del: set.simps*(2))

**lemma** *card-length*:  $\text{card } (\text{set } xs) \leq \text{length } xs$   
 by (induct  $xs$ ) (auto simp add: *card-insert-if*)

**lemma** *set-minus-filter-out*:  
 $\text{set } xs - \{y\} = \text{set } (\text{filter } (\lambda x. \neg (x = y))\ xs)$   
 by (induct  $xs$ ) auto

#### 43.1.8 filter

**lemma** *filter-append* [simp]:  $\text{filter } P\ (xs @ ys) = \text{filter } P\ xs @ \text{filter } P\ ys$   
 by (induct  $xs$ ) auto

**lemma** *rev-filter*:  $\text{rev } (\text{filter } P\ xs) = \text{filter } P\ (\text{rev } xs)$   
 by (induct  $xs$ ) *simp-all*

**lemma** *filter-filter* [simp]:  $\text{filter } P\ (\text{filter } Q\ xs) = \text{filter } (\lambda x. Q\ x \wedge P\ x)\ xs$   
 by (induct  $xs$ ) auto

**lemma** *length-filter-le* [simp]:  $\text{length } (\text{filter } P\ xs) \leq \text{length } xs$   
 by (induct  $xs$ ) (auto simp add: *le-SucI*)

**lemma** *sum-length-filter-compl*:  
 $\text{length } (\text{filter } P\ xs) + \text{length } (\text{filter } (\%x. \neg P\ x)\ xs) = \text{length } xs$   
 by (induct  $xs$ ) *simp-all*



**lemma** *filter-True* [simp]:  $\forall x \in \text{set } xs. P\ x ==> \text{filter } P\ xs = xs$   
**by** (induct xs) auto

**lemma** *filter-False* [simp]:  $\forall x \in \text{set } xs. \neg P\ x ==> \text{filter } P\ xs = []$   
**by** (induct xs) auto

**lemma** *filter-empty-conv*:  $(\text{filter } P\ xs = []) = (\forall x \in \text{set } xs. \neg P\ x)$   
**by** (induct xs) simp-all

**lemma** *filter-id-conv*:  $(\text{filter } P\ xs = xs) = (\forall x \in \text{set } xs. P\ x)$   
**apply** (induct xs)  
**apply** auto  
**apply** (cut-tac  $P=P$  and  $xs=xs$  in *length-filter-le*)  
**apply** simp  
**done**

**lemma** *filter-map*:  
 $\text{filter } P\ (\text{map } f\ xs) = \text{map } f\ (\text{filter } (P \circ f)\ xs)$   
**by** (induct xs) simp-all

**lemma** *length-filter-map*[simp]:  
 $\text{length } (\text{filter } P\ (\text{map } f\ xs)) = \text{length } (\text{filter } (P \circ f)\ xs)$   
**by** (simp add: *filter-map*)

**lemma** *filter-is-subset* [simp]:  $\text{set } (\text{filter } P\ xs) \leq \text{set } xs$   
**by** auto

**lemma** *length-filter-less*:  
 $\llbracket x : \text{set } xs; \sim P\ x \rrbracket \implies \text{length } (\text{filter } P\ xs) < \text{length } xs$   
**proof** (induct xs)  
**case** Nil **thus** ?case **by** simp  
**next**  
**case** (Cons x xs) **thus** ?case  
**apply** (auto split: split-if-asm)  
**using** *length-filter-le*[of P xs] **apply** arith  
**done**  
**qed**

**lemma** *length-filter-conv-card*:  
 $\text{length } (\text{filter } p\ xs) = \text{card}\{i. i < \text{length } xs \ \& \ p(xs!i)\}$   
**proof** (induct xs)  
**case** Nil **thus** ?case **by** simp  
**next**  
**case** (Cons x xs)  
**let** ?S =  $\{i. i < \text{length } xs \ \& \ p(xs!i)\}$   
**have** fin: finite ?S **by** (fast intro: bounded-nat-set-is-finite)  
**show** ?case **(is** ?l = card ?S')  
**proof** (cases)  
**assume** p x

```

hence eq: ?S' = insert 0 (Suc ' ?S)
  by(auto simp: image-def split:nat.split dest:gr0-implies-Suc)
have length (filter p (x # xs)) = Suc(card ?S)
  using Cons ⟨p x⟩ by simp
also have ... = Suc(card(Suc ' ?S)) using fin
  by (simp add: card-image inj-Suc)
also have ... = card ?S' using eq fin
  by (simp add: card-insert-if) (simp add: image-def)
finally show ?thesis .
next
  assume ¬ p x
  hence eq: ?S' = Suc ' ?S
    by(auto simp add: image-def split:nat.split elim:lessE)
  have length (filter p (x # xs)) = card ?S
    using Cons ⟨¬ p x⟩ by simp
  also have ... = card(Suc ' ?S) using fin
    by (simp add: card-image inj-Suc)
  also have ... = card ?S' using eq fin
    by (simp add: card-insert-if)
  finally show ?thesis .
qed
qed

lemma Cons-eq-filterD:
  x#xs = filter P ys ⟹
  ∃ us vs. ys = us @ x # vs ∧ (∀ u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs
  (is - ⟹ ∃ us vs. ?P ys us vs)
proof(induct ys)
  case Nil thus ?case by simp
next
  case (Cons y ys)
  show ?case (is ∃ x. ?Q x)
  proof cases
    assume Py: P y
    show ?thesis
    proof cases
      assume x = y
      with Py Cons.prem1 have ?Q [] by simp
      then show ?thesis ..
    next
      assume x ≠ y
      with Py Cons.prem1 show ?thesis by simp
    qed
  qed
next
  assume ¬ P y
  with Cons obtain us vs where ?P (y#ys) (y#us) vs by fastsimp
  then have ?Q (y#us) by simp
  then show ?thesis ..
qed

```

**qed**

**lemma** *filter-eq-ConsD*:

*filter P ys = x # xs  $\implies$*   
 $\exists us\ vs. ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P\ u) \wedge P\ x \wedge xs = \text{filter } P\ vs$   
**by**(*rule Cons-eq-filterD*) *simp*

**lemma** *filter-eq-Cons-iff*:

*(filter P ys = x # xs) =*  
 $(\exists us\ vs. ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P\ u) \wedge P\ x \wedge xs = \text{filter } P\ vs)$   
**by**(*auto dest:filter-eq-ConsD*)

**lemma** *Cons-eq-filter-iff*:

*(x # xs = filter P ys) =*  
 $(\exists us\ vs. ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P\ u) \wedge P\ x \wedge xs = \text{filter } P\ vs)$   
**by**(*auto dest:Cons-eq-filterD*)

**lemma** *filter-cong[fundef-cong, recdef-cong]*:

$xs = ys \implies (\bigwedge x. x \in \text{set } ys \implies P\ x = Q\ x) \implies \text{filter } P\ xs = \text{filter } Q\ ys$   
**apply** *simp*  
**apply**(*erule thin-rl*)  
**by** (*induct ys*) *simp-all*

#### 43.1.9 List partitioning

**primrec** *partition* ::  $('a \Rightarrow \text{bool}) \Rightarrow 'a\ \text{list} \Rightarrow 'a\ \text{list} \times 'a\ \text{list}$  **where**

*partition P [] = ([], [])*  
 $| \text{partition } P\ (x \# xs) =$   
 $(\text{let } (yes, no) = \text{partition } P\ xs$   
 $\text{in if } P\ x \text{ then } (x \# yes, no) \text{ else } (yes, x \# no))$

**lemma** *partition-filter1*:

$\text{fst } (\text{partition } P\ xs) = \text{filter } P\ xs$   
**by** (*induct xs*) (*auto simp add: Let-def split-def*)

**lemma** *partition-filter2*:

$\text{snd } (\text{partition } P\ xs) = \text{filter } (\text{Not } o\ P)\ xs$   
**by** (*induct xs*) (*auto simp add: Let-def split-def*)

**lemma** *partition-P*:

**assumes**  $\text{partition } P\ xs = (yes, no)$   
**shows**  $(\forall p \in \text{set } yes. P\ p) \wedge (\forall p \in \text{set } no. \neg P\ p)$   
**proof** –  
**from** *assms* **have**  $yes = \text{fst } (\text{partition } P\ xs)$  **and**  $no = \text{snd } (\text{partition } P\ xs)$   
**by** *simp-all*  
**then show** *?thesis* **by** (*simp-all add: partition-filter1 partition-filter2*)  
**qed**

**lemma** *partition-set*:

```

assumes partition  $P$   $xs = (yes, no)$ 
shows  $set\ yes \cup set\ no = set\ xs$ 
proof –
  from  $assms$  have  $yes = fst\ (partition\ P\ xs)$  and  $no = snd\ (partition\ P\ xs)$ 
  by  $simp$ -all
  then show  $?thesis$  by (auto  $simp$  add: partition-filter1 partition-filter2)
qed

```

```

lemma partition-filter-conv[ $simp$ ]:
   $partition\ f\ xs = (filter\ f\ xs, filter\ (Not\ o\ f)\ xs)$ 
unfolding partition-filter2[ $symmetric$ ]
unfolding partition-filter1[ $symmetric$ ] by  $simp$ 

```

```

declare partition.simps[ $simp\ del$ ]

```

#### 43.1.10 concat

```

lemma concat-append [ $simp$ ]:  $concat\ (xs\ @\ ys) = concat\ xs\ @\ concat\ ys$ 
by (induct  $xs$ ) auto

```

```

lemma concat-eq-Nil-conv [ $simp$ ]:  $(concat\ xss = []) = (\forall\ xs \in set\ xss. xs = [])$ 
by (induct  $xss$ ) auto

```

```

lemma Nil-eq-concat-conv [ $simp$ ]:  $([] = concat\ xss) = (\forall\ xs \in set\ xss. xs = [])$ 
by (induct  $xss$ ) auto

```

```

lemma set-concat [ $simp$ ]:  $set\ (concat\ xs) = (UN\ x:set\ xs. set\ x)$ 
by (induct  $xs$ ) auto

```

```

lemma concat-map-singleton[ $simp$ ]:  $concat(map\ (\%x. [f\ x])\ xs) = map\ f\ xs$ 
by (induct  $xs$ ) auto

```

```

lemma map-concat:  $map\ f\ (concat\ xs) = concat\ (map\ (map\ f)\ xs)$ 
by (induct  $xs$ ) auto

```

```

lemma filter-concat:  $filter\ p\ (concat\ xs) = concat\ (map\ (filter\ p)\ xs)$ 
by (induct  $xs$ ) auto

```

```

lemma rev-concat:  $rev\ (concat\ xs) = concat\ (map\ rev\ (rev\ xs))$ 
by (induct  $xs$ ) auto

```

#### 43.1.11 nth

```

lemma nth-Cons-0 [ $simp$ ,  $code$ ]:  $(x\ \# \ xs)!0 = x$ 
by auto

```

```

lemma nth-Cons-Suc [ $simp$ ,  $code$ ]:  $(x\ \# \ xs)!(Suc\ n) = xs!n$ 
by auto

```

```

declare nth.simps [ $simp\ del$ ]

```

**lemma** *nth-append*:

$(xs @ ys)!n = (\text{if } n < \text{length } xs \text{ then } xs!n \text{ else } ys!(n - \text{length } xs))$   
**apply** (*induct xs arbitrary: n, simp*)  
**apply** (*case-tac n, auto*)  
**done**

**lemma** *nth-append-length* [*simp*]:  $(xs @ x \# ys) ! \text{length } xs = x$   
**by** (*induct xs*) *auto*

**lemma** *nth-append-length-plus* [*simp*]:  $(xs @ ys) ! (\text{length } xs + n) = ys ! n$   
**by** (*induct xs*) *auto*

**lemma** *nth-map* [*simp*]:  $n < \text{length } xs \implies (\text{map } f \text{ } xs)!n = f(xs!n)$   
**apply** (*induct xs arbitrary: n, simp*)  
**apply** (*case-tac n, auto*)  
**done**

**lemma** *hd-conv-nth*:  $xs \neq [] \implies \text{hd } xs = xs!0$   
**by**(*cases xs*) *simp-all*

**lemma** *list-eq-iff-nth-eq*:

$(xs = ys) = (\text{length } xs = \text{length } ys \wedge (\text{ALL } i < \text{length } xs. xs!i = ys!i))$   
**apply**(*induct xs arbitrary: ys*)  
**apply** *force*  
**apply**(*case-tac ys*)  
**apply** *simp*  
**apply**(*simp add:nth-Cons split:nat.split*)**apply** *blast*  
**done**

**lemma** *set-conv-nth*:  $\text{set } xs = \{xs!i \mid i. i < \text{length } xs\}$   
**apply** (*induct xs, simp, simp*)  
**apply** *safe*  
**apply** (*metis nat-case-0 nth.simps zero-less-Suc*)  
**apply** (*metis less-Suc-eq-0-disj nth-Cons-Suc*)  
**apply** (*case-tac i, simp*)  
**apply** (*metis diff-Suc-Suc nat-case-Suc nth.simps zero-less-diff*)  
**done**

**lemma** *in-set-conv-nth*:  $(x \in \text{set } xs) = (\exists i < \text{length } xs. xs!i = x)$   
**by**(*auto simp:set-conv-nth*)

**lemma** *list-ball-nth*:  $[\mid n < \text{length } xs; !x : \text{set } xs. P \ x] \implies P(xs!n)$   
**by** (*auto simp add: set-conv-nth*)

**lemma** *nth-mem* [*simp*]:  $n < \text{length } xs \implies xs!n : \text{set } xs$   
**by** (*auto simp add: set-conv-nth*)

**lemma** *all-nth-imp-all-set*:  
 $[[ !i < \text{length } xs. P(xs!i); x : \text{set } xs]] ==> P x$   
**by** (*auto simp add: set-conv-nth*)

**lemma** *all-set-conv-all-nth*:  
 $(\forall x \in \text{set } xs. P x) = (\forall i. i < \text{length } xs \longrightarrow P (xs ! i))$   
**by** (*auto simp add: set-conv-nth*)

**lemma** *rev-nth*:  
 $n < \text{size } xs \implies \text{rev } xs ! n = xs ! (\text{length } xs - \text{Suc } n)$   
**proof** (*induct xs arbitrary: n*)  
 case Nil **thus** ?case **by** *simp*  
**next**  
 case (Cons x xs)  
 hence  $n < \text{Suc } (\text{length } xs)$  **by** *simp*  
 moreover  
 { **assume**  $n < \text{length } xs$   
   **with**  $n$  **obtain**  $n'$  **where**  $\text{length } xs - n = \text{Suc } n'$   
   **by** (*cases length xs - n, auto*)  
   moreover  
   **then have**  $\text{length } xs - \text{Suc } n = n'$  **by** *simp*  
   **ultimately**  
   **have**  $xs ! (\text{length } xs - \text{Suc } n) = (x \# xs) ! (\text{length } xs - n)$  **by** *simp*  
 }  
**ultimately**  
**show** ?case **by** (*clarsimp simp add: Cons nth-append*)  
**qed**

**lemma** *Skolem-list-nth*:  
 $(\text{ALL } i < k. \text{EX } x. P i x) = (\text{EX } xs. \text{size } xs = k \ \& \ (\text{ALL } i < k. P i (xs!i)))$   
 $(\text{is } - = (\text{EX } xs. ?P k xs))$   
**proof** (*induct k*)  
 case 0 **show** ?case **by** *simp*  
**next**  
 case (Suc k)  
**show** ?case **(is ?L = ?R is - = (EX xs. ?P' xs))**  
**proof**  
**assume** ?R **thus** ?L **using** *Suc* **by** *auto*  
**next**  
**assume** ?L  
**with** *Suc* **obtain**  $x$   $xs$  **where**  $?P k xs \ \& \ P k x$  **by** (*metis less-Suc-eq*)  
**hence**  $?P'(xs@[x])$  **by** (*simp add: nth-append less-Suc-eq*)  
**thus** ?R ..  
**qed**  
**qed**

#### 43.1.12 list-update

**lemma** *length-list-update* [*simp*]:  $\text{length}(xs[i:=x]) = \text{length } xs$

**by** (induct xs arbitrary: i) (auto split: nat.split)

**lemma** nth-list-update:

$i < \text{length } xs \implies (xs[i:=x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)$

**by** (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

**lemma** nth-list-update-eq [simp]:  $i < \text{length } xs \implies (xs[i:=x])!i = x$

**by** (simp add: nth-list-update)

**lemma** nth-list-update-neq [simp]:  $i \neq j \implies xs[i:=x]!j = xs!j$

**by** (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

**lemma** list-update-id[simp]:  $xs[i := xs!i] = xs$

**by** (induct xs arbitrary: i) (simp-all split:nat.splits)

**lemma** list-update-beyond[simp]:  $\text{length } xs \leq i \implies xs[i:=x] = xs$

**apply** (induct xs arbitrary: i)

**apply** simp

**apply** (case-tac i)

**apply** simp-all

**done**

**lemma** list-update-nonempty[simp]:  $xs[k:=x] = [] \longleftrightarrow xs=[]$

**by**(metis length-0-conv length-list-update)

**lemma** list-update-same-conv:

$i < \text{length } xs \implies (xs[i := x] = xs) = (xs!i = x)$

**by** (induct xs arbitrary: i) (auto split: nat.split)

**lemma** list-update-append1:

$i < \text{size } xs \implies (xs @ ys)[i:=x] = xs[i:=x] @ ys$

**apply** (induct xs arbitrary: i, simp)

**apply**(simp split:nat.split)

**done**

**lemma** list-update-append:

$(xs @ ys) [n := x] =$

$(\text{if } n < \text{length } xs \text{ then } xs[n := x] @ ys \text{ else } xs @ (ys [n - \text{length } xs := x]))$

**by** (induct xs arbitrary: n) (auto split:nat.splits)

**lemma** list-update-length [simp]:

$(xs @ x \# ys)[\text{length } xs := y] = (xs @ y \# ys)$

**by** (induct xs, auto)

**lemma** map-update:  $\text{map } f (xs[k := y]) = (\text{map } f xs)[k := f y]$

**by**(induct xs arbitrary: k)(auto split:nat.splits)

**lemma** rev-update:

$k < \text{length } xs \implies \text{rev } (xs[k := y]) = (\text{rev } xs)[\text{length } xs - k - 1 := y]$

**by** (*induct xs arbitrary: k*) (*auto simp: list-update-append split:nat.splits*)

**lemma** *update-zip*:

(*zip xs ys*)[*i:=xy*] = *zip (xs[i:=fst xy]) (ys[i:=snd xy])*

**by** (*induct ys arbitrary: i xy xs*) (*auto, case-tac xs, auto split: nat.split*)

**lemma** *set-update-subset-insert*: *set(xs[i:=x]) <= insert x (set xs)*

**by** (*induct xs arbitrary: i*) (*auto split: nat.split*)

**lemma** *set-update-subsetI*: [*set xs <= A; x:A*] ==> *set(xs[i := x]) <= A*

**by** (*blast dest!: set-update-subset-insert [THEN subsetD]*)

**lemma** *set-update-memI*: *n < length xs ==> x ∈ set (xs[n := x])*

**by** (*induct xs arbitrary: n*) (*auto split:nat.splits*)

**lemma** *list-update-overwrite*[*simp*]:

*xs [i := x, i := y] = xs [i := y]*

**apply** (*induct xs arbitrary: i*) **apply** *simp*

**apply** (*case-tac i, simp-all*)

**done**

**lemma** *list-update-swap*:

*i ≠ i' ==> xs [i := x, i' := x'] = xs [i' := x', i := x]*

**apply** (*induct xs arbitrary: i i'*)

**apply** *simp*

**apply** (*case-tac i, case-tac i'*)

**apply** *auto*

**apply** (*case-tac i'*)

**apply** *auto*

**done**

**lemma** *list-update-code* [*code*]:

[*i := y*] = []

(*x # xs*)[*0 := y*] = *y # xs*

(*x # xs*)[*Suc i := y*] = *x # xs[i := y]*

**by** *simp-all*

### 43.1.13 last and butlast

**lemma** *last-snoc* [*simp*]: *last (xs @ [x]) = x*

**by** (*induct xs*) *auto*

**lemma** *butlast-snoc* [*simp*]: *butlast (xs @ [x]) = xs*

**by** (*induct xs*) *auto*

**lemma** *last-ConsL*: *xs = [] ==> last(x#xs) = x*

**by**(*simp add:last.simps*)

**lemma** *last-ConsR*: *xs ≠ [] ==> last(x#xs) = last xs*



**by**(*simp add:last.simps*)

**lemma** *last-append*:  $\text{last}(xs \text{ @ } ys) = (\text{if } ys = [] \text{ then last } xs \text{ else last } ys)$   
**by** (*induct xs*) (*auto*)

**lemma** *last-appendL[simp]*:  $ys = [] \implies \text{last}(xs \text{ @ } ys) = \text{last } xs$   
**by**(*simp add:last-append*)

**lemma** *last-appendR[simp]*:  $ys \neq [] \implies \text{last}(xs \text{ @ } ys) = \text{last } ys$   
**by**(*simp add:last-append*)

**lemma** *hd-rev*:  $xs \neq [] \implies \text{hd}(\text{rev } xs) = \text{last } xs$   
**by**(*rule rev-exhaust[of xs]*) *simp-all*

**lemma** *last-rev*:  $xs \neq [] \implies \text{last}(\text{rev } xs) = \text{hd } xs$   
**by**(*cases xs*) *simp-all*

**lemma** *last-in-set[simp]*:  $as \neq [] \implies \text{last } as \in \text{set } as$   
**by** (*induct as*) *auto*

**lemma** *length-butlast [simp]*:  $\text{length } (\text{butlast } xs) = \text{length } xs - 1$   
**by** (*induct xs rule: rev-induct*) *auto*

**lemma** *butlast-append*:  
 $\text{butlast } (xs \text{ @ } ys) = (\text{if } ys = [] \text{ then butlast } xs \text{ else } xs \text{ @ butlast } ys)$   
**by** (*induct xs arbitrary: ys*) *auto*

**lemma** *append-butlast-last-id [simp]*:  
 $xs \neq [] \implies \text{butlast } xs \text{ @ } [\text{last } xs] = xs$   
**by** (*induct xs*) *auto*

**lemma** *in-set-butlastD*:  $x : \text{set } (\text{butlast } xs) \implies x : \text{set } xs$   
**by** (*induct xs*) (*auto split: split-if-asm*)

**lemma** *in-set-butlast-appendI*:  
 $x : \text{set } (\text{butlast } xs) \mid x : \text{set } (\text{butlast } ys) \implies x : \text{set } (\text{butlast } (xs \text{ @ } ys))$   
**by** (*auto dest: in-set-butlastD simp add: butlast-append*)

**lemma** *last-drop[simp]*:  $n < \text{length } xs \implies \text{last } (\text{drop } n \text{ } xs) = \text{last } xs$   
**apply** (*induct xs arbitrary: n*)  
**apply** *simp*  
**apply** (*auto split:nat.split*)  
**done**

**lemma** *last-conv-nth*:  $xs \neq [] \implies \text{last } xs = xs!(\text{length } xs - 1)$   
**by**(*induct xs*)(*auto simp:neq-Nil-conv*)

**lemma** *butlast-conv-take*:  $\text{butlast } xs = \text{take } (\text{length } xs - 1) \text{ } xs$   
**by** (*induct xs, simp, case-tac xs, simp-all*)

**lemma** *last-list-update*:  
 $xs \neq [] \implies \text{last}(xs[k:=x]) = (\text{if } k = \text{size } xs - 1 \text{ then } x \text{ else } \text{last } xs)$   
**by** (*auto simp: last-conv-nth*)

**lemma** *butlast-list-update*:  
 $\text{butlast}(xs[k:=x]) =$   
 $(\text{if } k = \text{size } xs - 1 \text{ then } \text{butlast } xs \text{ else } (\text{butlast } xs)[k:=x])$   
**apply**(*cases xs rule: rev-cases*)  
**apply** *simp*  
**apply**(*simp add: list-update-append split: nat.splits*)  
**done**

**lemma** *last-map*:  
 $xs \neq [] \implies \text{last } (\text{map } f \text{ } xs) = f \text{ } (\text{last } xs)$   
**by** (*cases xs rule: rev-cases simp-all*)

**lemma** *map-butlast*:  
 $\text{map } f \text{ } (\text{butlast } xs) = \text{butlast } (\text{map } f \text{ } xs)$   
**by** (*induct xs simp-all*)

#### 43.1.14 *take and drop*

**lemma** *take-0* [*simp*]:  $\text{take } 0 \text{ } xs = []$   
**by** (*induct xs auto*)

**lemma** *drop-0* [*simp*]:  $\text{drop } 0 \text{ } xs = xs$   
**by** (*induct xs auto*)

**lemma** *take-Suc-Cons* [*simp*]:  $\text{take } (\text{Suc } n) \text{ } (x \# xs) = x \# \text{take } n \text{ } xs$   
**by** *simp*

**lemma** *drop-Suc-Cons* [*simp*]:  $\text{drop } (\text{Suc } n) \text{ } (x \# xs) = \text{drop } n \text{ } xs$   
**by** *simp*

**declare** *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

**lemma** *take-1-Cons* [*simp*]:  $\text{take } 1 \text{ } (x \# xs) = [x]$   
**unfolding** *One-nat-def* **by** *simp*

**lemma** *drop-1-Cons* [*simp*]:  $\text{drop } 1 \text{ } (x \# xs) = xs$   
**unfolding** *One-nat-def* **by** *simp*

**lemma** *take-Suc*:  $xs \sim [] \implies \text{take } (\text{Suc } n) \text{ } xs = \text{hd } xs \# \text{take } n \text{ } (\text{tl } xs)$   
**by**(*clarsimp simp add: neq-Nil-conv*)

**lemma** *drop-Suc*:  $\text{drop } (\text{Suc } n) \text{ } xs = \text{drop } n \text{ } (\text{tl } xs)$   
**by**(*cases xs, simp-all*)

**lemma** *take-tl*:  $\text{take } n \text{ (tl } xs) = \text{tl (take (Suc } n) xs)$   
**by** (*induct xs arbitrary: n, simp-all*)

**lemma** *drop-tl*:  $\text{drop } n \text{ (tl } xs) = \text{tl (drop } n \text{ xs)}$   
**by**(*induct xs arbitrary: n, simp-all add:drop-Cons drop-Suc split:nat.split*)

**lemma** *tl-take*:  $\text{tl (take } n \text{ xs)} = \text{take (n - 1) (tl xs)}$   
**by** (*cases n, simp, cases xs, auto*)

**lemma** *tl-drop*:  $\text{tl (drop } n \text{ xs)} = \text{drop } n \text{ (tl xs)}$   
**by** (*simp only: drop-tl*)

**lemma** *nth-via-drop*:  $\text{drop } n \text{ xs} = y \# ys \implies xs!n = y$   
**apply** (*induct xs arbitrary: n, simp*)  
**apply**(*simp add:drop-Cons nth-Cons split:nat.splits*)  
**done**

**lemma** *take-Suc-conv-app-nth*:  
 $i < \text{length } xs \implies \text{take (Suc } i) \text{ xs} = \text{take } i \text{ xs} @ [xs!i]$   
**apply** (*induct xs arbitrary: i, simp*)  
**apply** (*case-tac i, auto*)  
**done**

**lemma** *drop-Suc-conv-tl*:  
 $i < \text{length } xs \implies (xs!i) \# (\text{drop (Suc } i) \text{ xs}) = \text{drop } i \text{ xs}$   
**apply** (*induct xs arbitrary: i, simp*)  
**apply** (*case-tac i, auto*)  
**done**

**lemma** *length-take* [*simp*]:  $\text{length (take } n \text{ xs)} = \min (\text{length } xs) \text{ } n$   
**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *length-drop* [*simp*]:  $\text{length (drop } n \text{ xs)} = (\text{length } xs - n)$   
**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *take-all* [*simp*]:  $\text{length } xs \leq n \implies \text{take } n \text{ xs} = xs$   
**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *drop-all* [*simp*]:  $\text{length } xs \leq n \implies \text{drop } n \text{ xs} = []$   
**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *take-append* [*simp*]:  
 $\text{take } n \text{ (xs @ ys)} = (\text{take } n \text{ xs} @ \text{take (n - length xs) ys})$   
**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

**lemma** *drop-append* [*simp*]:  
 $\text{drop } n \text{ (xs @ ys)} = \text{drop } n \text{ xs} @ \text{drop (n - length xs) ys}$   
**by** (*induct n arbitrary: xs*) (*auto, case-tac xs, auto*)

```

lemma take-take [simp]: take n (take m xs) = take (min n m) xs
apply (induct m arbitrary: xs n, auto)
apply (case-tac xs, auto)
apply (case-tac n, auto)
done

```

```

lemma drop-drop [simp]: drop n (drop m xs) = drop (n + m) xs
apply (induct m arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

```

lemma take-drop: take n (drop m xs) = drop m (take (n + m) xs)
apply (induct m arbitrary: xs n, auto)
apply (case-tac xs, auto)
done

```

```

lemma drop-take: drop n (take m xs) = take (m - n) (drop n xs)
apply (induct xs arbitrary: m n)
  apply simp
apply (simp add: take-Cons drop-Cons split:nat.split)
done

```

```

lemma append-take-drop-id [simp]: take n xs @ drop n xs = xs
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

```

lemma take-eq-Nil[simp]: (take n xs = []) = (n = 0  $\vee$  xs = [])
apply (induct xs arbitrary: n)
  apply simp
apply (simp add: take-Cons split:nat.split)
done

```

```

lemma drop-eq-Nil[simp]: (drop n xs = []) = (length xs <= n)
apply (induct xs arbitrary: n)
apply simp
apply (simp add: drop-Cons split:nat.split)
done

```

```

lemma take-map: take n (map f xs) = map f (take n xs)
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

```

lemma drop-map: drop n (map f xs) = map f (drop n xs)
apply (induct n arbitrary: xs, auto)
apply (case-tac xs, auto)
done

```

**lemma** *rev-take*:  $\text{rev } (\text{take } i \text{ } xs) = \text{drop } (\text{length } xs - i) (\text{rev } xs)$   
**apply** (*induct* *xs* *arbitrary*: *i*, *auto*)  
**apply** (*case-tac* *i*, *auto*)  
**done**

**lemma** *rev-drop*:  $\text{rev } (\text{drop } i \text{ } xs) = \text{take } (\text{length } xs - i) (\text{rev } xs)$   
**apply** (*induct* *xs* *arbitrary*: *i*, *auto*)  
**apply** (*case-tac* *i*, *auto*)  
**done**

**lemma** *nth-take* [*simp*]:  $i < n \implies (\text{take } n \text{ } xs)!i = xs!i$   
**apply** (*induct* *xs* *arbitrary*: *i* *n*, *auto*)  
**apply** (*case-tac* *n*, *blast*)  
**apply** (*case-tac* *i*, *auto*)  
**done**

**lemma** *nth-drop* [*simp*]:  
 $n + i \leq \text{length } xs \implies (\text{drop } n \text{ } xs)!i = xs!(n + i)$   
**apply** (*induct* *n* *arbitrary*: *xs* *i*, *auto*)  
**apply** (*case-tac* *xs*, *auto*)  
**done**

**lemma** *butlast-take*:  
 $n \leq \text{length } xs \implies \text{butlast } (\text{take } n \text{ } xs) = \text{take } (n - 1) \text{ } xs$   
**by** (*simp* *add*: *butlast-conv-take* *min-max.inf-absorb1* *min-max.inf-absorb2*)

**lemma** *butlast-drop*:  $\text{butlast } (\text{drop } n \text{ } xs) = \text{drop } n (\text{butlast } xs)$   
**by** (*simp* *add*: *butlast-conv-take* *drop-take* *add-ac*)

**lemma** *take-butlast*:  $n < \text{length } xs \implies \text{take } n (\text{butlast } xs) = \text{take } n \text{ } xs$   
**by** (*simp* *add*: *butlast-conv-take* *min-max.inf-absorb1*)

**lemma** *drop-butlast*:  $\text{drop } n (\text{butlast } xs) = \text{butlast } (\text{drop } n \text{ } xs)$   
**by** (*simp* *add*: *butlast-conv-take* *drop-take* *add-ac*)

**lemma** *hd-drop-conv-nth*:  $\llbracket xs \neq []; n < \text{length } xs \rrbracket \implies \text{hd}(\text{drop } n \text{ } xs) = xs!n$   
**by**(*simp* *add*: *hd-conv-nth*)

**lemma** *set-take-subset-set-take*:  
 $m \leq n \implies \text{set}(\text{take } m \text{ } xs) \subseteq \text{set}(\text{take } n \text{ } xs)$   
**by**(*induct* *xs* *arbitrary*: *m* *n*)(*auto* *simp*:*take-Cons* *split*:*nat.split*)

**lemma** *set-take-subset*:  $\text{set}(\text{take } n \text{ } xs) \subseteq \text{set } xs$   
**by**(*induct* *xs* *arbitrary*: *n*)(*auto* *simp*:*take-Cons* *split*:*nat.split*)

**lemma** *set-drop-subset*:  $\text{set}(\text{drop } n \text{ } xs) \subseteq \text{set } xs$   
**by**(*induct* *xs* *arbitrary*: *n*)(*auto* *simp*:*drop-Cons* *split*:*nat.split*)

**lemma** *set-drop-subset-set-drop*:

```

  m >= n ==> set(drop m xs) <= set(drop n xs)
apply(induct xs arbitrary: m n)
apply(auto simp:drop-Cons split:nat.split)
apply (metis set-drop-subset subset-iff)
done

```

```

lemma in-set-takeD:  $x : \text{set}(\text{take } n \text{ } xs) \implies x : \text{set } xs$ 
using set-take-subset by fast

```

```

lemma in-set-dropD:  $x : \text{set}(\text{drop } n \text{ } xs) \implies x : \text{set } xs$ 
using set-drop-subset by fast

```

```

lemma append-eq-conv-conj:
   $(xs @ ys = zs) = (xs = \text{take } (\text{length } xs) \text{ } zs \wedge ys = \text{drop } (\text{length } xs) \text{ } zs)$ 
apply (induct xs arbitrary: zs, simp, clarsimp)
apply (case-tac zs, auto)
done

```

```

lemma take-add:
   $i+j \leq \text{length}(xs) \implies \text{take } (i+j) \text{ } xs = \text{take } i \text{ } xs @ \text{take } j \text{ } (\text{drop } i \text{ } xs)$ 
apply (induct xs arbitrary: i, auto)
apply (case-tac i, simp-all)
done

```

```

lemma append-eq-append-conv-if:
   $(xs_1 @ xs_2 = ys_1 @ ys_2) =$ 
  (if  $\text{size } xs_1 \leq \text{size } ys_1$ 
    then  $xs_1 = \text{take } (\text{size } xs_1) \text{ } ys_1 \wedge xs_2 = \text{drop } (\text{size } xs_1) \text{ } ys_1 @ ys_2$ 
    else  $\text{take } (\text{size } ys_1) \text{ } xs_1 = ys_1 \wedge \text{drop } (\text{size } ys_1) \text{ } xs_1 @ xs_2 = ys_2$ )
apply(induct xs1 arbitrary: ys1)
apply simp
apply(case-tac ys1)
apply simp-all
done

```

```

lemma take-hd-drop:
   $n < \text{length } xs \implies \text{take } n \text{ } xs @ [\text{hd } (\text{drop } n \text{ } xs)] = \text{take } (\text{Suc } n) \text{ } xs$ 
apply(induct xs arbitrary: n)
apply simp
apply(simp add:drop-Cons split:nat.split)
done

```

```

lemma id-take-nth-drop:
   $i < \text{length } xs \implies xs = \text{take } i \text{ } xs @ xs[i] \# \text{drop } (\text{Suc } i) \text{ } xs$ 
proof –
  assume si:  $i < \text{length } xs$ 
  hence  $xs = \text{take } (\text{Suc } i) \text{ } xs @ \text{drop } (\text{Suc } i) \text{ } xs$  by auto
  moreover
  from si have  $\text{take } (\text{Suc } i) \text{ } xs = \text{take } i \text{ } xs @ [xs[i]]$ 

```

```

  apply (rule-tac take-Suc-conv-app-nth) by arith
  ultimately show ?thesis by auto
qed

```

```

lemma upd-conv-take-nth-drop:
   $i < \text{length } xs \implies xs[i:=a] = \text{take } i \text{ } xs @ a \# \text{drop } (\text{Suc } i) \text{ } xs$ 
proof -
  assume  $i < \text{length } xs$ 
  have  $xs[i:=a] = (\text{take } i \text{ } xs @ xs!i \# \text{drop } (\text{Suc } i) \text{ } xs)[i:=a]$ 
    by(rule arg-cong[OF id-take-nth-drop[OF i]])
  also have  $\dots = \text{take } i \text{ } xs @ a \# \text{drop } (\text{Suc } i) \text{ } xs$ 
    using  $i$  by (simp add: list-update-append)
  finally show ?thesis .
qed

```

```

lemma nth-drop':
   $i < \text{length } xs \implies xs ! i \# \text{drop } (\text{Suc } i) \text{ } xs = \text{drop } i \text{ } xs$ 
apply (induct i arbitrary: xs)
apply (simp add: neq-Nil-conv)
apply (erule exE)+
apply simp
apply (case-tac xs)
apply simp-all
done

```

#### 43.1.15 takeWhile and dropWhile

```

lemma length-takeWhile-le:  $\text{length } (\text{takeWhile } P \text{ } xs) \leq \text{length } xs$ 
  by (induct xs) auto

```

```

lemma takeWhile-dropWhile-id [simp]:  $\text{takeWhile } P \text{ } xs @ \text{dropWhile } P \text{ } xs = xs$ 
  by (induct xs) auto

```

```

lemma takeWhile-append1 [simp]:
   $[| x:\text{set } xs; \sim P(x)|] \implies \text{takeWhile } P \text{ } (xs @ ys) = \text{takeWhile } P \text{ } xs$ 
  by (induct xs) auto

```

```

lemma takeWhile-append2 [simp]:
   $(!!x. x : \text{set } xs \implies P x) \implies \text{takeWhile } P \text{ } (xs @ ys) = xs @ \text{takeWhile } P \text{ } ys$ 
  by (induct xs) auto

```

```

lemma takeWhile-tail:  $\neg P x \implies \text{takeWhile } P \text{ } (xs @ (x\#l)) = \text{takeWhile } P \text{ } xs$ 
  by (induct xs) auto

```

```

lemma takeWhile-nth:  $j < \text{length } (\text{takeWhile } P \text{ } xs) \implies \text{takeWhile } P \text{ } xs ! j = xs ! j$ 
! j
apply (subst (3) takeWhile-dropWhile-id[symmetric]) unfolding nth-append by
auto

```

**lemma** *dropWhile-nth*:  $j < \text{length } (\text{dropWhile } P \text{ } xs) \implies \text{dropWhile } P \text{ } xs ! j = xs ! (j + \text{length } (\text{takeWhile } P \text{ } xs))$

**apply** (*subst* ( $\exists$ ) *takeWhile-dropWhile-id*[*symmetric*]) **unfolding** *nth-append* **by** *auto*

**lemma** *length-dropWhile-le*:  $\text{length } (\text{dropWhile } P \text{ } xs) \leq \text{length } xs$   
**by** (*induct xs*) *auto*

**lemma** *dropWhile-append1* [*simp*]:  
 $[[x : \text{set } xs; \sim P(x)]] \implies \text{dropWhile } P \text{ } (xs @ ys) = (\text{dropWhile } P \text{ } xs) @ ys$   
**by** (*induct xs*) *auto*

**lemma** *dropWhile-append2* [*simp*]:  
 $(!!x. x : \text{set } xs \implies P(x)) \implies \text{dropWhile } P \text{ } (xs @ ys) = \text{dropWhile } P \text{ } ys$   
**by** (*induct xs*) *auto*

**lemma** *set-takeWhileD*:  $x : \text{set } (\text{takeWhile } P \text{ } xs) \implies x : \text{set } xs \wedge P \text{ } x$   
**by** (*induct xs*) (*auto split: split-if-asm*)

**lemma** *takeWhile-eq-all-conv*[*simp*]:  
 $(\text{takeWhile } P \text{ } xs = xs) = (\forall x \in \text{set } xs. P \text{ } x)$   
**by**(*induct xs, auto*)

**lemma** *dropWhile-eq-Nil-conv*[*simp*]:  
 $(\text{dropWhile } P \text{ } xs = []) = (\forall x \in \text{set } xs. P \text{ } x)$   
**by**(*induct xs, auto*)

**lemma** *dropWhile-eq-Cons-conv*:  
 $(\text{dropWhile } P \text{ } xs = y \# ys) = (xs = \text{takeWhile } P \text{ } xs @ y \# ys \ \& \ \neg P \text{ } y)$   
**by**(*induct xs, auto*)

**lemma** *distinct-takeWhile*[*simp*]:  $\text{distinct } xs \implies \text{distinct } (\text{takeWhile } P \text{ } xs)$   
**by** (*induct xs*) (*auto dest: set-takeWhileD*)

**lemma** *distinct-dropWhile*[*simp*]:  $\text{distinct } xs \implies \text{distinct } (\text{dropWhile } P \text{ } xs)$   
**by** (*induct xs*) *auto*

**lemma** *takeWhile-map*:  $\text{takeWhile } P \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{takeWhile } (P \circ f) \text{ } xs)$   
**by** (*induct xs*) *auto*

**lemma** *dropWhile-map*:  $\text{dropWhile } P \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{dropWhile } (P \circ f) \text{ } xs)$   
**by** (*induct xs*) *auto*

**lemma** *takeWhile-eq-take*:  $\text{takeWhile } P \text{ } xs = \text{take } (\text{length } (\text{takeWhile } P \text{ } xs)) \text{ } xs$   
**by** (*induct xs*) *auto*

**lemma** *dropWhile-eq-drop*:  $\text{dropWhile } P \text{ } xs = \text{drop } (\text{length } (\text{takeWhile } P \text{ } xs)) \text{ } xs$   
**by** (*induct xs*) *auto*



**lemma** *hd-dropWhile*:

*dropWhile P xs*  $\neq [] \implies \neg P$  (*hd* (*dropWhile P xs*))

**using** *assms* **by** (*induct xs*) *auto*

**lemma** *takeWhile-eq-filter*:

**assumes**  $\bigwedge x. x \in \text{set } (\text{dropWhile } P \text{ } xs) \implies \neg P \text{ } x$

**shows** *takeWhile P xs* = *filter P xs*

**proof** –

**have** *A*: *filter P xs* = *filter P (takeWhile P xs @ dropWhile P xs)*

**by** *simp*

**have** *B*: *filter P (dropWhile P xs)* = []

**unfolding** *filter-empty-conv* **using** *assms* **by** *blast*

**have** *filter P xs* = *takeWhile P xs*

**unfolding** *A* *filter-append B*

**by** (*auto simp add: filter-id-conv dest: set-takeWhileD*)

**thus** *?thesis* ..

**qed**

**lemma** *takeWhile-eq-take-P-nth*:

$\llbracket \bigwedge i. \llbracket i < n ; i < \text{length } xs \rrbracket \implies P (xs ! i) ; n < \text{length } xs \implies \neg P (xs ! n) \rrbracket \implies$

*takeWhile P xs* = *take n xs*

**proof** (*induct xs arbitrary: n*)

**case** (*Cons x xs*)

**thus** *?case*

**proof** (*cases n*)

**case** (*Suc n'*) **note** *this[simp]*

**have** *P x* **using** *Cons.prem1[of 0]* **by** *simp*

**moreover** **have** *takeWhile P xs* = *take n' xs*

**proof** (*rule Cons.hyps*)

**case** *goal1* **thus** *P (xs ! i)* **using** *Cons.prem1[of Suc i]* **by** *simp*

**next** **case** *goal2* **thus** *?case* **using** *Cons* **by** *auto*

**qed**

**ultimately** **show** *?thesis* **by** *simp*

**qed** *simp*

**qed** *simp*

**lemma** *nth-length-takeWhile*:

*length (takeWhile P xs)* < *length xs*  $\implies \neg P (xs ! \text{length } (\text{takeWhile } P \text{ } xs))$

**by** (*induct xs*) *auto*

**lemma** *length-takeWhile-less-P-nth*:

**assumes** *all*:  $\bigwedge i. i < j \implies P (xs ! i)$  **and**  $j \leq \text{length } xs$

**shows**  $j \leq \text{length } (\text{takeWhile } P \text{ } xs)$

**proof** (*rule classical*)

**assume**  $\neg ?thesis$

**hence** *length (takeWhile P xs)* < *length xs* **using** *assms* **by** *simp*

**thus** *?thesis* **using** *all*  $\langle \neg ?thesis \rangle$  *nth-length-takeWhile[of P xs]* **by** *auto*

**qed**

The following two lemmas could be generalized to an arbitrary property.

**lemma** *takeWhile-neq-rev*:  $\llbracket \text{distinct } xs; x \in \text{set } xs \rrbracket \implies$   
 $\text{takeWhile } (\lambda y. y \neq x) (\text{rev } xs) = \text{rev } (\text{tl } (\text{dropWhile } (\lambda y. y \neq x) xs))$   
**by** (*induct xs*) (*auto simp: takeWhile-tail* [**where**  $l = []$ ])

**lemma** *dropWhile-neq-rev*:  $\llbracket \text{distinct } xs; x \in \text{set } xs \rrbracket \implies$   
 $\text{dropWhile } (\lambda y. y \neq x) (\text{rev } xs) = x \# \text{rev } (\text{takeWhile } (\lambda y. y \neq x) xs)$   
**apply** (*induct xs*)  
**apply** *simp*  
**apply** *auto*  
**apply** (*subst dropWhile-append2*)  
**apply** *auto*  
**done**

**lemma** *takeWhile-not-last*:  
 $\llbracket xs \neq []; \text{distinct } xs \rrbracket \implies \text{takeWhile } (\lambda y. y \neq \text{last } xs) xs = \text{butlast } xs$   
**apply** (*induct xs*)  
**apply** *simp*  
**apply** (*case-tac xs*)  
**apply** (*auto*)  
**done**

**lemma** *takeWhile-cong* [*fundef-cong, recdef-cong*]:  
 $\llbracket l = k; \forall x. x : \text{set } l \implies P x = Q x \rrbracket$   
 $\implies \text{takeWhile } P l = \text{takeWhile } Q k$   
**by** (*induct k arbitrary: l*) (*simp-all*)

**lemma** *dropWhile-cong* [*fundef-cong, recdef-cong*]:  
 $\llbracket l = k; \forall x. x : \text{set } l \implies P x = Q x \rrbracket$   
 $\implies \text{dropWhile } P l = \text{dropWhile } Q k$   
**by** (*induct k arbitrary: l, simp-all*)

#### 43.1.16 *zip*

**lemma** *zip-Nil* [*simp*]:  $\text{zip } [] ys = []$   
**by** (*induct ys*) *auto*

**lemma** *zip-Cons-Cons* [*simp*]:  $\text{zip } (x \# xs) (y \# ys) = (x, y) \# \text{zip } xs ys$   
**by** *simp*

**declare** *zip-Cons* [*simp del*]

**lemma** [*code*]:  
 $\text{zip } [] ys = []$   
 $\text{zip } xs [] = []$   
 $\text{zip } (x \# xs) (y \# ys) = (x, y) \# \text{zip } xs ys$   
**by** (*fact zip-Nil zip.simps(1) zip-Cons-Cons*) +

**lemma** *zip-Cons1*:

$zip\ (x\#xs)\ ys = (case\ ys\ of\ [] \Rightarrow [] \mid y\#ys \Rightarrow (x,y)\#zip\ xs\ ys)$   
**by**(*auto split:list.split*)

**lemma** *length-zip [simp]*:  
 $length\ (zip\ xs\ ys) = \min\ (length\ xs)\ (length\ ys)$   
**by** (*induct xs ys rule:list-induct2'*) *auto*

**lemma** *zip-obtain-same-length*:  
**assumes**  $\bigwedge zs\ ws\ n.\ length\ zs = length\ ws \Longrightarrow n = \min\ (length\ xs)\ (length\ ys)$   
 $\Longrightarrow zs = take\ n\ xs \Longrightarrow ws = take\ n\ ys \Longrightarrow P\ (zip\ zs\ ws)$   
**shows**  $P\ (zip\ xs\ ys)$   
**proof** –  
**let**  $?n = \min\ (length\ xs)\ (length\ ys)$   
**have**  $P\ (zip\ (take\ ?n\ xs)\ (take\ ?n\ ys))$   
**by** (*rule assms*) *simp-all*  
**moreover** **have**  $zip\ xs\ ys = zip\ (take\ ?n\ xs)\ (take\ ?n\ ys)$   
**proof** (*induct xs arbitrary: ys*)  
**case** *Nil* **then show**  $?case$  **by** *simp*  
**next**  
**case** (*Cons x xs*) **then show**  $?case$  **by** (*cases ys*) *simp-all*  
**qed**  
**ultimately show**  $?thesis$  **by** *simp*  
**qed**

**lemma** *zip-append1*:  
 $zip\ (xs\ @\ ys)\ zs =$   
 $zip\ xs\ (take\ (length\ xs)\ zs)\ @\ zip\ ys\ (drop\ (length\ xs)\ zs)$   
**by** (*induct xs zs rule:list-induct2'*) *auto*

**lemma** *zip-append2*:  
 $zip\ xs\ (ys\ @\ zs) =$   
 $zip\ (take\ (length\ ys)\ xs)\ ys\ @\ zip\ (drop\ (length\ ys)\ xs)\ zs$   
**by** (*induct xs ys rule:list-induct2'*) *auto*

**lemma** *zip-append [simp]*:  
 $[[]\ length\ xs = length\ us;\ length\ ys = length\ vs\ []] ==>$   
 $zip\ (xs@ys)\ (us@vs) = zip\ xs\ us\ @\ zip\ ys\ vs$   
**by** (*simp add: zip-append1*)

**lemma** *zip-rev*:  
 $length\ xs = length\ ys ==> zip\ (rev\ xs)\ (rev\ ys) = rev\ (zip\ xs\ ys)$   
**by** (*induct rule:list-induct2, simp-all*)

**lemma** *zip-map-map*:  
 $zip\ (map\ f\ xs)\ (map\ g\ ys) = map\ (\lambda\ (x,\ y).\ (f\ x,\ g\ y))\ (zip\ xs\ ys)$   
**proof** (*induct xs arbitrary: ys*)  
**case** (*Cons x xs*) **note**  $Cons\ x\ xs = Cons.hyps$   
**show**  $?case$   
**proof** (*cases ys*)

```

    case (Cons y ys')
    show ?thesis unfolding Cons using Cons-x-xs by simp
  qed simp
qed simp

```

```

lemma zip-map1:
  zip (map f xs) ys = map (λ(x, y). (f x, y)) (zip xs ys)
using zip-map-map[of f xs λx. x ys] by simp

```

```

lemma zip-map2:
  zip xs (map f ys) = map (λ(x, y). (x, f y)) (zip xs ys)
using zip-map-map[of λx. x xs f ys] by simp

```

```

lemma map-zip-map:
  map f (zip (map g xs) ys) = map (λ(x,y). f(g x, y)) (zip xs ys)
unfolding zip-map1 by auto

```

```

lemma map-zip-map2:
  map f (zip xs (map g ys)) = map (λ(x,y). f(x, g y)) (zip xs ys)
unfolding zip-map2 by auto

```

Courtesy of Andreas Lochbihler:

```

lemma zip-same-conv-map: zip xs xs = map (λx. (x, x)) xs
by(induct xs) auto

```

```

lemma nth-zip [simp]:
  [| i < length xs; i < length ys |] ==> (zip xs ys)!i = (xs!i, ys!i)
apply (induct ys arbitrary: i xs, simp)
apply (case-tac xs)
apply (simp-all add: nth.simps split: nat.split)
done

```

```

lemma set-zip:
  set (zip xs ys) = {(xs!i, ys!i) | i. i < min (length xs) (length ys)}
by(simp add: set-conv-nth cong: rev-conj-cong)

```

```

lemma zip-same: ((a,b) ∈ set (zip xs xs)) = (a ∈ set xs ∧ a = b)
by(induct xs) auto

```

```

lemma zip-update:
  zip (xs[i:=x]) (ys[i:=y]) = (zip xs ys)[i:=(x,y)]
by(rule sym, simp add: update-zip)

```

```

lemma zip-replicate [simp]:
  zip (replicate i x) (replicate j y) = replicate (min i j) (x,y)
apply (induct i arbitrary: j, auto)
apply (case-tac j, auto)
done

```

**lemma** *take-zip*:

```

  take n (zip xs ys) = zip (take n xs) (take n ys)
apply (induct n arbitrary: xs ys)
apply simp
apply (case-tac xs, simp)
apply (case-tac ys, simp-all)
done

```

**lemma** *drop-zip*:

```

  drop n (zip xs ys) = zip (drop n xs) (drop n ys)
apply (induct n arbitrary: xs ys)
apply simp
apply (case-tac xs, simp)
apply (case-tac ys, simp-all)
done

```

**lemma** *zip-takeWhile-fst*:  $\text{zip } (\text{takeWhile } P \text{ } xs) \text{ } ys = \text{takeWhile } (P \circ \text{fst}) (\text{zip } xs \text{ } ys)$

```

proof (induct xs arbitrary: ys)
  case (Cons x xs) thus ?case by (cases ys) auto
qed simp

```

**lemma** *zip-takeWhile-snd*:  $\text{zip } xs \text{ } (\text{takeWhile } P \text{ } ys) = \text{takeWhile } (P \circ \text{snd}) (\text{zip } xs \text{ } ys)$

```

proof (induct xs arbitrary: ys)
  case (Cons x xs) thus ?case by (cases ys) auto
qed simp

```

**lemma** *set-zip-leftD*:

```

  (x,y) ∈ set (zip xs ys)  $\implies$  x ∈ set xs
by (induct xs ys rule: list-induct2') auto

```

**lemma** *set-zip-rightD*:

```

  (x,y) ∈ set (zip xs ys)  $\implies$  y ∈ set ys
by (induct xs ys rule: list-induct2') auto

```

**lemma** *in-set-zipE*:

```

  (x,y) : set (zip xs ys)  $\implies$  ( $\llbracket x : \text{set } xs; y : \text{set } ys \rrbracket \implies R$ )  $\implies$  R
by (blast dest: set-zip-leftD set-zip-rightD)

```

**lemma** *zip-map-fst-snd*:

```

  zip (map fst zs) (map snd zs) = zs
by (induct zs) simp-all

```

**lemma** *zip-eq-conv*:

```

  length xs = length ys  $\implies$  zip xs ys = zs  $\longleftrightarrow$  map fst zs = xs  $\wedge$  map snd zs = ys
by (auto simp add: zip-map-fst-snd)

```

**lemma** *distinct-zipI1*:

```

distinct xs  $\implies$  distinct (zip xs ys)
proof (induct xs arbitrary: ys)
  case (Cons x xs)
  show ?case
  proof (cases ys)
    case (Cons y ys')
    have (x, y)  $\notin$  set (zip xs ys')
    using Cons.prem by (auto simp: set-zip)
    thus ?thesis
    unfolding Cons zip-Cons-Cons distinct.simp
    using Cons.hyps Cons.prem by simp
  qed simp
qed simp

```

```

lemma distinct-zipI2:
  distinct xs  $\implies$  distinct (zip xs ys)
proof (induct xs arbitrary: ys)
  case (Cons x xs)
  show ?case
  proof (cases ys)
    case (Cons y ys')
    have (x, y)  $\notin$  set (zip xs ys')
    using Cons.prem by (auto simp: set-zip)
    thus ?thesis
    unfolding Cons zip-Cons-Cons distinct.simp
    using Cons.hyps Cons.prem by simp
  qed simp
qed simp

```

#### 43.1.17 list-all2

```

lemma list-all2-lengthD [intro?]:
  list-all2 P xs ys  $\implies$  length xs = length ys
by (simp add: list-all2-def)

```

```

lemma list-all2-Nil [iff, code]: list-all2 P [] ys = (ys = [])
by (simp add: list-all2-def)

```

```

lemma list-all2-Nil2 [iff, code]: list-all2 P xs [] = (xs = [])
by (simp add: list-all2-def)

```

```

lemma list-all2-Cons [iff, code]:
  list-all2 P (x # xs) (y # ys) = (P x y  $\wedge$  list-all2 P xs ys)
by (auto simp add: list-all2-def)

```

```

lemma list-all2-Cons1:
  list-all2 P (x # xs) ys = ( $\exists$  z zs. ys = z # zs  $\wedge$  P x z  $\wedge$  list-all2 P xs zs)
by (cases ys) auto

```

**lemma** *list-all2-Cons2*:

*list-all2 P xs (y # ys) = ( $\exists z$  zs. xs = z # zs  $\wedge$  P z y  $\wedge$  list-all2 P zs ys)*

**by** (cases xs) auto

**lemma** *list-all2-rev [iff]*:

*list-all2 P (rev xs) (rev ys) = list-all2 P xs ys*

**by** (simp add: list-all2-def zip-rev cong: conj-cong)

**lemma** *list-all2-rev1*:

*list-all2 P (rev xs) ys = list-all2 P xs (rev ys)*

**by** (subst list-all2-rev [symmetric]) simp

**lemma** *list-all2-append1*:

*list-all2 P (xs @ ys) zs =*

*(EX us vs. zs = us @ vs  $\wedge$  length us = length xs  $\wedge$  length vs = length ys  $\wedge$*

*list-all2 P xs us  $\wedge$  list-all2 P ys vs)*

**apply** (simp add: list-all2-def zip-append1)

**apply** (rule iffI)

**apply** (rule-tac x = take (length xs) zs **in** exI)

**apply** (rule-tac x = drop (length xs) zs **in** exI)

**apply** (force split: nat-diff-split simp add: min-def, clarify)

**apply** (simp add: ball-Un)

**done**

**lemma** *list-all2-append2*:

*list-all2 P xs (ys @ zs) =*

*(EX us vs. xs = us @ vs  $\wedge$  length us = length ys  $\wedge$  length vs = length zs  $\wedge$*

*list-all2 P us ys  $\wedge$  list-all2 P vs zs)*

**apply** (simp add: list-all2-def zip-append2)

**apply** (rule iffI)

**apply** (rule-tac x = take (length ys) xs **in** exI)

**apply** (rule-tac x = drop (length ys) xs **in** exI)

**apply** (force split: nat-diff-split simp add: min-def, clarify)

**apply** (simp add: ball-Un)

**done**

**lemma** *list-all2-append*:

*length xs = length ys  $\implies$*

*list-all2 P (xs@us) (ys@vs) = (list-all2 P xs ys  $\wedge$  list-all2 P us vs)*

**by** (induct rule:list-induct2, simp-all)

**lemma** *list-all2-appendI [intro?, trans]*:

$\llbracket \text{list-all2 } P \text{ a b; list-all2 } P \text{ c d} \rrbracket \implies \text{list-all2 } P \text{ (a@c) (b@d)}$

**by** (simp add: list-all2-append list-all2-lengthD)

**lemma** *list-all2-conv-all-nth*:

*list-all2 P xs ys =*

*(length xs = length ys  $\wedge$  ( $\forall i < \text{length xs. } P \text{ (xs!i) (ys!i)}$ ))*

**by** (force simp add: list-all2-def set-zip)

**lemma** *list-all2-trans*:

**assumes** *tr*:  $!!a\ b\ c. P1\ a\ b \implies P2\ b\ c \implies P3\ a\ c$

**shows**  $!!bs\ cs. list\_all2\ P1\ as\ bs \implies list\_all2\ P2\ bs\ cs \implies list\_all2\ P3\ as\ cs$   
 (**is**  $!!bs\ cs. PROP\ ?Q\ as\ bs\ cs$ )

**proof** (*induct as*)

**fix** *x xs bs* **assume** *I1*:  $!!bs\ cs. PROP\ ?Q\ xs\ bs\ cs$

**show**  $!!cs. PROP\ ?Q\ (x\ \#\ xs)\ bs\ cs$

**proof** (*induct bs*)

**fix** *y ys cs* **assume** *I2*:  $!!cs. PROP\ ?Q\ (x\ \#\ xs)\ ys\ cs$

**show**  $PROP\ ?Q\ (x\ \#\ xs)\ (y\ \#\ ys)\ cs$

**by** (*induct cs*) (*auto intro: tr I1 I2*)

**qed** *simp*

**qed** *simp*

**lemma** *list-all2-all-nthI* [*intro?*]:

$length\ a = length\ b \implies (\bigwedge n. n < length\ a \implies P\ (a!n)\ (b!n)) \implies list\_all2\ P\ a\ b$

**by** (*simp add: list-all2-conv-all-nth*)

**lemma** *list-all2I*:

$\forall x \in set\ (zip\ a\ b). split\ P\ x \implies length\ a = length\ b \implies list\_all2\ P\ a\ b$

**by** (*simp add: list-all2-def*)

**lemma** *list-all2-nthD*:

$\llbracket list\_all2\ P\ xs\ ys; p < size\ xs \rrbracket \implies P\ (xs!p)\ (ys!p)$

**by** (*simp add: list-all2-conv-all-nth*)

**lemma** *list-all2-nthD2*:

$\llbracket list\_all2\ P\ xs\ ys; p < size\ ys \rrbracket \implies P\ (xs!p)\ (ys!p)$

**by** (*frule list-all2-lengthD*) (*auto intro: list-all2-nthD*)

**lemma** *list-all2-map1*:

$list\_all2\ P\ (map\ f\ as)\ bs = list\_all2\ (\lambda x\ y. P\ (f\ x)\ y)\ as\ bs$

**by** (*simp add: list-all2-conv-all-nth*)

**lemma** *list-all2-map2*:

$list\_all2\ P\ as\ (map\ f\ bs) = list\_all2\ (\lambda x\ y. P\ x\ (f\ y))\ as\ bs$

**by** (*auto simp add: list-all2-conv-all-nth*)

**lemma** *list-all2-refl* [*intro?*]:

$(\bigwedge x. P\ x\ x) \implies list\_all2\ P\ xs\ xs$

**by** (*simp add: list-all2-conv-all-nth*)

**lemma** *list-all2-update-cong*:

$\llbracket i < size\ xs; list\_all2\ P\ xs\ ys; P\ x\ y \rrbracket \implies list\_all2\ P\ (xs[i:=x])\ (ys[i:=y])$

**by** (*simp add: list-all2-conv-all-nth nth-list-update*)

**lemma** *list-all2-update-cong2*:

$\llbracket list\_all2\ P\ xs\ ys; P\ x\ y; i < length\ ys \rrbracket \implies list\_all2\ P\ (xs[i:=x])\ (ys[i:=y])$



**by** (*simp add: list-all2-lengthD list-all2-update-cong*)

**lemma** *list-all2-takeI* [*simp,intro?*]:

$list-all2\ P\ xs\ ys \implies list-all2\ P\ (take\ n\ xs)\ (take\ n\ ys)$

**apply** (*induct xs arbitrary: n ys*)

**apply** *simp*

**apply** (*clarsimp simp add: list-all2-Cons1*)

**apply** (*case-tac n*)

**apply** *auto*

**done**

**lemma** *list-all2-dropI* [*simp,intro?*]:

$list-all2\ P\ as\ bs \implies list-all2\ P\ (drop\ n\ as)\ (drop\ n\ bs)$

**apply** (*induct as arbitrary: n bs, simp*)

**apply** (*clarsimp simp add: list-all2-Cons1*)

**apply** (*case-tac n, simp, simp*)

**done**

**lemma** *list-all2-mono* [*intro?*]:

$list-all2\ P\ xs\ ys \implies (\bigwedge xs\ ys. P\ xs\ ys \implies Q\ xs\ ys) \implies list-all2\ Q\ xs\ ys$

**apply** (*induct xs arbitrary: ys, simp*)

**apply** (*case-tac ys, auto*)

**done**

**lemma** *list-all2-eq*:

$xs = ys \iff list-all2\ (op =)\ xs\ ys$

**by** (*induct xs ys rule: list-induct2'*) *auto*

#### 43.1.18 foldl and foldr

**lemma** *foldl-append* [*simp*]:

$foldl\ f\ a\ (xs\ @\ ys) = foldl\ f\ (foldl\ f\ a\ xs)\ ys$

**by** (*induct xs arbitrary: a*) *auto*

**lemma** *foldr-append*[*simp*]:  $foldr\ f\ (xs\ @\ ys)\ a = foldr\ f\ xs\ (foldr\ f\ ys\ a)$

**by** (*induct xs*) *auto*

**lemma** *foldr-map*:  $foldr\ g\ (map\ f\ xs)\ a = foldr\ (g\ o\ f)\ xs\ a$

**by**(*induct xs*) *simp-all*

For efficient code generation: avoid intermediate list.

**lemma** *foldl-map*[*code-unfold*]:

$foldl\ g\ a\ (map\ f\ xs) = foldl\ (\%a\ x. g\ a\ (f\ x))\ a\ xs$

**by**(*induct xs arbitrary:a*) *simp-all*

**lemma** *foldl-apply*:

**assumes**  $\bigwedge x. x \in set\ xs \implies f\ x\ o\ h = h\ o\ g\ x$

**shows**  $foldl\ (\lambda s\ x. f\ x\ s)\ (h\ s)\ xs = h\ (foldl\ (\lambda s\ x. g\ x\ s)\ s\ xs)$

**by** (*rule sym, insert assms, induct xs arbitrary: s*) (*simp-all add: expand-fun-eq*)

**lemma** *foldl-cong* [*fundef-cong*, *recdef-cong*]:  

$$[| a = b; l = k; !!a\ x.\ x : \text{set } l \implies f\ a\ x = g\ a\ x |]$$

$$\implies \text{foldl } f\ a\ l = \text{foldl } g\ b\ k$$
**by** (*induct k arbitrary: a b l*) *simp-all*

**lemma** *foldr-cong* [*fundef-cong*, *recdef-cong*]:  

$$[| a = b; l = k; !!a\ x.\ x : \text{set } l \implies f\ x\ a = g\ x\ a |]$$

$$\implies \text{foldr } f\ l\ a = \text{foldr } g\ k\ b$$
**by** (*induct k arbitrary: a b l*) *simp-all*

**lemma** *foldl-fun-comm*:  
**assumes**  $\bigwedge x\ y\ s.\ f\ (f\ s\ x)\ y = f\ (f\ s\ y)\ x$   
**shows**  $f\ (\text{foldl } f\ s\ xs)\ x = \text{foldl } f\ (f\ s\ x)\ xs$   
**by** (*induct xs arbitrary: s*)  
*(simp-all add: assms)*

**lemma** (*in semigroup-add*) *foldl-assoc*:  
**shows**  $\text{foldl } op\ +\ (x+y)\ zs = x + (\text{foldl } op\ +\ y\ zs)$   
**by** (*induct zs arbitrary: y*) (*simp-all add:add-assoc*)

**lemma** (*in monoid-add*) *foldl-absorb0*:  
**shows**  $x + (\text{foldl } op\ +\ 0\ zs) = \text{foldl } op\ +\ x\ zs$   
**by** (*induct zs*) (*simp-all add:foldl-assoc*)

**lemma** *foldl-rev*:  
**assumes**  $\bigwedge x\ y\ s.\ f\ (f\ s\ x)\ y = f\ (f\ s\ y)\ x$   
**shows**  $\text{foldl } f\ s\ (\text{rev } xs) = \text{foldl } f\ s\ xs$   
**proof** (*induct xs arbitrary: s*)  
**case** *Nil* **then show** ?*case* **by** *simp*  
**next**  
**case** (*Cons x xs*) **with** *assms* **show** ?*case* **by** (*simp add: foldl-fun-comm*)  
**qed**

The “First Duality Theorem” in Bird & Wadler:

**lemma** *foldl-foldr1-lemma*:  

$$\text{foldl } op\ +\ a\ xs = a + \text{foldr } op\ +\ xs\ (0::'a::\text{monoid-add})$$
**by** (*induct xs arbitrary: a*) (*auto simp:add-assoc*)

**corollary** *foldl-foldr1*:  

$$\text{foldl } op\ +\ 0\ xs = \text{foldr } op\ +\ xs\ (0::'a::\text{monoid-add})$$
**by** (*simp add:foldl-foldr1-lemma*)

The “Third Duality Theorem” in Bird & Wadler:

**lemma** *foldr-foldl*:  $\text{foldr } f\ xs\ a = \text{foldl } (\%x\ y.\ f\ y\ x)\ a\ (\text{rev } xs)$   
**by** (*induct xs*) *auto*

**lemma** *foldl-foldr*:  $\text{foldl } f\ a\ xs = \text{foldr } (\%x\ y.\ f\ y\ x)\ (\text{rev } xs)\ a$   
**by** (*simp add: foldr-foldl [of %x y. f y x rev xs]*)

**lemma** (in *ab-semigroup-add*) *foldr-conv-foldl*:  $\text{foldr } op + xs \ a = \text{foldl } op + a \ xs$   
**by** (*induct xs*, *auto simp add: foldl-assoc add-commute*)

Note:  $n \leq \text{foldl } (op +) \ n \ ns$  looks simpler, but is more difficult to use because it requires an additional transitivity step.

**lemma** *start-le-sum*:  $(m::nat) \leq n \implies m \leq \text{foldl } (op +) \ n \ ns$   
**by** (*induct ns arbitrary: n*) *auto*

**lemma** *elem-le-sum*:  $(n::nat) : \text{set } ns \implies n \leq \text{foldl } (op +) \ 0 \ ns$   
**by** (*force intro: start-le-sum simp add: in-set-conv-decomp*)

**lemma** *sum-eq-0-conv* [iff]:  
 $(\text{foldl } (op +) \ (m::nat) \ ns = 0) = (m = 0 \wedge (\forall n \in \text{set } ns. n = 0))$   
**by** (*induct ns arbitrary: m*) *auto*

**lemma** *foldr-invariant*:  
 $\llbracket Q \ x ; \forall x \in \text{set } xs. P \ x ; \forall x \ y. P \ x \wedge Q \ y \longrightarrow Q \ (f \ x \ y) \rrbracket \implies Q \ (\text{foldr } f \ xs \ x)$   
**by** (*induct xs*, *simp-all*)

**lemma** *foldl-invariant*:  
 $\llbracket Q \ x ; \forall x \in \text{set } xs. P \ x ; \forall x \ y. P \ x \wedge Q \ y \longrightarrow Q \ (f \ y \ x) \rrbracket \implies Q \ (\text{foldl } f \ x \ xs)$   
**by** (*induct xs arbitrary: x*, *simp-all*)

**lemma** *foldl-weak-invariant*:  
**assumes**  $P \ s$   
**and**  $\bigwedge s \ x. x \in \text{set } xs \implies P \ s \implies P \ (f \ s \ x)$   
**shows**  $P \ (\text{foldl } f \ s \ xs)$   
**using** *assms* **by** (*induct xs arbitrary: s*) *simp-all*

*foldl* and *concat*

**lemma** *foldl-conv-concat*:  
 $\text{foldl } (op \ @) \ xs \ xss = xs \ @ \ \text{concat } xss$   
**proof** (*induct xss arbitrary: xs*)  
**case** *Nil* **show** ?case **by** *simp*  
**next**  
**interpret** *monoid-add op @* **proof** *qed simp-all*  
**case** *Cons* **then show** ?case **by** (*simp add: foldl-absorb0*)  
**qed**

**lemma** *concat-conv-foldl*:  $\text{concat } xss = \text{foldl } (op \ @) \ [] \ xss$   
**by** (*simp add: foldl-conv-concat*)

*fold* and *foldl*

**lemma** (in *fun-left-comm*) *fold-set-remdups*:  
 $\text{fold } f \ y \ (\text{set } xs) = \text{foldl } (\lambda y \ x. f \ x \ y) \ y \ (\text{remdups } xs)$   
**by** (*rule sym*, *induct xs arbitrary: y*) (*simp-all add: fold-fun-comm insert-absorb*)

**lemma** (in *fun-left-comm-idem*) *fold-set*:

$fold\ f\ y\ (set\ xs) = foldl\ (\lambda y\ x.\ f\ x\ y)\ y\ xs$   
**by** (*rule sym, induct xs arbitrary: y*) (*simp-all add: fold-fun-comm*)

**lemma** (*in ab-semigroup-idem-mult*) *fold1-set*:  
**assumes**  $xs \neq []$   
**shows**  $fold1\ times\ (set\ xs) = foldl\ times\ (hd\ xs)\ (tl\ xs)$   
**proof** –  
**interpret** *fun-left-comm-idem times* **by** (*fact fun-left-comm-idem*)  
**from** *assms* **obtain**  $y\ ys$  **where**  $xs: xs = y \# ys$   
**by** (*cases xs*) *auto*  
**show** *?thesis*  
**proof** (*cases set ys = {}*)  
**case** *True* **with**  $xs$  **show** *?thesis* **by** *simp*  
**next**  
**case** *False*  
**then have**  $fold1\ times\ (insert\ y\ (set\ ys)) = fold\ times\ y\ (set\ ys)$   
**by** (*simp only: finite-set fold1-eq-fold-idem*)  
**with**  $xs$  **show** *?thesis* **by** (*simp add: fold-set mult-commute*)  
**qed**  
**qed**

**lemma** (*in lattice*) *Inf-fin-set-fold* [*code-unfold*]:  
 $Inf-fin\ (set\ (x \# xs)) = foldl\ inf\ x\ xs$   
**proof** –  
**interpret** *ab-semigroup-idem-mult inf*  $:: 'a \Rightarrow 'a \Rightarrow 'a$   
**by** (*fact ab-semigroup-idem-mult-inf*)  
**show** *?thesis*  
**by** (*simp add: Inf-fin-def fold1-set del: set.simps*)  
**qed**

**lemma** (*in lattice*) *Sup-fin-set-fold* [*code-unfold*]:  
 $Sup-fin\ (set\ (x \# xs)) = foldl\ sup\ x\ xs$   
**proof** –  
**interpret** *ab-semigroup-idem-mult sup*  $:: 'a \Rightarrow 'a \Rightarrow 'a$   
**by** (*fact ab-semigroup-idem-mult-sup*)  
**show** *?thesis*  
**by** (*simp add: Sup-fin-def fold1-set del: set.simps*)  
**qed**

**lemma** (*in linorder*) *Min-fin-set-fold* [*code-unfold*]:  
 $Min\ (set\ (x \# xs)) = foldl\ min\ x\ xs$   
**proof** –  
**interpret** *ab-semigroup-idem-mult min*  $:: 'a \Rightarrow 'a \Rightarrow 'a$   
**by** (*fact ab-semigroup-idem-mult-min*)  
**show** *?thesis*  
**by** (*simp add: Min-def fold1-set del: set.simps*)  
**qed**

**lemma** (*in linorder*) *Max-fin-set-fold* [*code-unfold*]:

```

Max (set (x # xs)) = foldl max x xs
proof -
  interpret ab-semigroup-idem-mult max :: 'a ⇒ 'a ⇒ 'a
  by (fact ab-semigroup-idem-mult-max)
  show ?thesis
  by (simp add: Max-def fold1-set del: set.simps)
qed

```

```

lemma (in complete-lattice) Inf-set-fold [code-unfold]:
  Inf (set xs) = foldl inf top xs
proof -
  interpret fun-left-comm-idem inf :: 'a ⇒ 'a ⇒ 'a
  by (fact fun-left-comm-idem-inf)
  show ?thesis by (simp add: Inf-fold-inf fold-set inf-commute)
qed

```

```

lemma (in complete-lattice) Sup-set-fold [code-unfold]:
  Sup (set xs) = foldl sup bot xs
proof -
  interpret fun-left-comm-idem sup :: 'a ⇒ 'a ⇒ 'a
  by (fact fun-left-comm-idem-sup)
  show ?thesis by (simp add: Sup-fold-sup fold-set sup-commute)
qed

```

```

lemma (in complete-lattice) INFI-set-fold:
  INFI (set xs) f = foldl (λy x. inf (f x) y) top xs
unfolding INFI-def set-map [symmetric] Inf-set-fold foldl-map
  by (simp add: inf-commute)

```

```

lemma (in complete-lattice) SUPR-set-fold:
  SUPR (set xs) f = foldl (λy x. sup (f x) y) bot xs
unfolding SUPR-def set-map [symmetric] Sup-set-fold foldl-map
  by (simp add: sup-commute)

```

#### 43.1.19 List summation: *listsum* and $\sum$

```

lemma listsum-append [simp]: listsum (xs @ ys) = listsum xs + listsum ys
by (induct xs) (simp-all add: add-assoc)

```

```

lemma listsum-rev [simp]:
  fixes xs :: 'a::comm-monoid-add list
  shows listsum (rev xs) = listsum xs
by (induct xs) (simp-all add: add-ac)

```

```

lemma listsum-map-remove1:
  fixes f :: 'a ⇒ ('b::comm-monoid-add)
  shows x : set xs ⇒ listsum (map f xs) = f x + listsum (map f (remove1 x xs))
by (induct xs) (auto simp add: add-ac)

```

**lemma** *list-size-conv-listsum*:

*list-size*  $f$   $xs = \text{listsum } (\text{map } f \text{ } xs) + \text{size } xs$

**by** (*induct*  $xs$ ) *auto*

**lemma** *listsum-foldr*:  $\text{listsum } xs = \text{foldr } (op \ +) \ xs \ 0$

**by** (*induct*  $xs$ ) *auto*

**lemma** *length-concat*:  $\text{length } (\text{concat } xss) = \text{listsum } (\text{map } \text{length } xss)$

**by** (*induct*  $xss$ ) *simp-all*

**lemma** *listsum-map-filter*:

**fixes**  $f :: 'a \Rightarrow 'b :: \text{comm-monoid-add}$

**assumes**  $\bigwedge x. \llbracket x \in \text{set } xs ; \neg P \ x \rrbracket \Longrightarrow f \ x = 0$

**shows**  $\text{listsum } (\text{map } f \ (\text{filter } P \ xs)) = \text{listsum } (\text{map } f \ xs)$

**using** *assms* **by** (*induct*  $xs$ ) *auto*

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

**lemma** *listsum[code-unfold]*:  $\text{listsum } xs = \text{foldl } (op \ +) \ 0 \ xs$

**by** (*simp* *add:listsum-foldr foldl-foldr1*)

**lemma** *distinct-listsum-conv-Setsum*:

*distinct*  $xs \Longrightarrow \text{listsum } xs = \text{Setsum}(\text{set } xs)$

**by** (*induct*  $xs$ ) *simp-all*

**lemma** *listsum-eq-0-nat-iff-nat*[*simp*]:

$\text{listsum } ns = (0::\text{nat}) \longleftrightarrow (\forall \ n \in \text{set } ns. \ n = 0)$

**by** (*simp* *add:listsum*)

**lemma** *elem-le-listsum-nat*:  $k < \text{size } ns \Longrightarrow ns!k \leq \text{listsum}(ns::\text{nat list})$

**apply** (*induct*  $ns$  *arbitrary:k*)

**apply** *simp*

**apply** (*fastsimp* *simp* *add:nth-Cons split: nat.split*)

**done**

**lemma** *listsum-update-nat*:  $k < \text{size } ns \Longrightarrow$

$\text{listsum } (ns[k := (n::\text{nat})]) = \text{listsum } ns + n - ns!k$

**apply** (*induct*  $ns$  *arbitrary:k*)

**apply** (*auto* *split:nat.split*)

**apply** (*drule* *elem-le-listsum-nat*)

**apply** *arith*

**done**

Some syntactic sugar for summing a function over a list:

**syntax**

*-listsum* ::  $p\text{trn} \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \quad ((\mathcal{S}UM \ -<--. \ -) \ [0, \ 51, \ 10] \ 10)$

**syntax** (*xsymbols*)

*-listsum* ::  $p\text{trn} \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \quad ((\mathcal{S}\sum \ -<--. \ -) \ [0, \ 51, \ 10] \ 10)$

**syntax** (*HTML output*)

*-listsum* ::  $p\text{trn} \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \quad ((\mathcal{S}\sum \ -<--. \ -) \ [0, \ 51, \ 10] \ 10)$

**translations** — Beware of argument permutation!

$SUM\ x \leftarrow xs.\ b == CONST\ listsum\ (CONST\ map\ (\%x.\ b)\ xs)$   
 $\sum\ x \leftarrow xs.\ b == CONST\ listsum\ (CONST\ map\ (\%x.\ b)\ xs)$

**lemma** *listsum-triv*:  $(\sum\ x \leftarrow xs.\ r) = of\_nat\ (length\ xs) * r$   
**by** (*induct xs*) (*simp-all add: left-distrib*)

**lemma** *listsum-0* [*simp*]:  $(\sum\ x \leftarrow xs.\ 0) = 0$   
**by** (*induct xs*) (*simp-all add: left-distrib*)

For non-Abelian groups *xs* needs to be reversed on one side:

**lemma** *uminus-listsum-map*:  
**fixes**  $f :: 'a \Rightarrow 'b::ab\_group\_add$   
**shows** —  $listsum\ (map\ f\ xs) = (listsum\ (map\ (uminus\ o\ f)\ xs))$   
**by** (*induct xs*) *simp-all*

**lemma** *listsum-addf*:  
**fixes**  $f\ g :: 'a \Rightarrow 'b::comm\_monoid\_add$   
**shows**  $(\sum\ x \leftarrow xs.\ f\ x + g\ x) = listsum\ (map\ f\ xs) + listsum\ (map\ g\ xs)$   
**by** (*induct xs*) (*simp-all add: algebra-simps*)

**lemma** *listsum-subtractf*:  
**fixes**  $f\ g :: 'a \Rightarrow 'b::ab\_group\_add$   
**shows**  $(\sum\ x \leftarrow xs.\ f\ x - g\ x) = listsum\ (map\ f\ xs) - listsum\ (map\ g\ xs)$   
**by** (*induct xs*) *simp-all*

**lemma** *listsum-const-mult*:  
**fixes**  $f :: 'a \Rightarrow 'b::semiring\_0$   
**shows**  $(\sum\ x \leftarrow xs.\ c * f\ x) = c * (\sum\ x \leftarrow xs.\ f\ x)$   
**by** (*induct xs*, *simp-all add: algebra-simps*)

**lemma** *listsum-mult-const*:  
**fixes**  $f :: 'a \Rightarrow 'b::semiring\_0$   
**shows**  $(\sum\ x \leftarrow xs.\ f\ x * c) = (\sum\ x \leftarrow xs.\ f\ x) * c$   
**by** (*induct xs*, *simp-all add: algebra-simps*)

**lemma** *listsum-abs*:  
**fixes**  $xs :: 'a::ordered\_ab\_group\_add\_abs\ list$   
**shows**  $|listsum\ xs| \leq listsum\ (map\ abs\ xs)$   
**by** (*induct xs*, *simp*, *simp add: order-trans [OF abs-triangle-ineq]*)

**lemma** *listsum-mono*:  
**fixes**  $f\ g :: 'a \Rightarrow 'b::\{comm\_monoid\_add,\ ordered\_ab\_semigroup\_add\}$   
**shows**  $(\bigwedge x.\ x \in set\ xs \implies f\ x \leq g\ x) \implies (\sum\ x \leftarrow xs.\ f\ x) \leq (\sum\ x \leftarrow xs.\ g\ x)$   
**by** (*induct xs*, *simp*, *simp add: add-mono*)

**43.1.20** *upt*

**lemma** *upt-rec*[code]:  $[i..<j] = (\text{if } i < j \text{ then } i \# [Suc\ i..<j] \text{ else } [])$

— simp does not terminate!

**by** (*induct j*) *auto*

**lemmas** *upt-rec-number-of*[simp] = *upt-rec*[of *number-of m number-of n, standard*]

**lemma** *upt-conv-Nil* [simp]:  $j \leq i \implies [i..<j] = []$

**by** (*subst upt-rec*) *simp*

**lemma** *upt-eq-Nil-conv*[simp]:  $([i..<j] = []) = (j = 0 \vee j \leq i)$

**by**(*induct j*)*simp-all*

**lemma** *upt-eq-Cons-conv*:

$([i..<j] = x \# xs) = (i < j \ \& \ i = x \ \& \ [i+1..<j] = xs)$

**apply**(*induct j arbitrary: x xs*)

**apply** *simp*

**apply**(*clarsimp simp add: append-eq-Cons-conv*)

**apply** *arith*

**done**

**lemma** *upt-Suc-append*:  $i \leq j \implies [i..<(Suc\ j)] = [i..<j]@[j]$

— Only needed if *upt-Suc* is deleted from the simpset.

**by** *simp*

**lemma** *upt-conv-Cons*:  $i < j \implies [i..<j] = i \# [Suc\ i..<j]$

**by** (*simp add: upt-rec*)

**lemma** *upt-add-eq-append*:  $i \leq j \implies [i..<j+k] = [i..<j]@[j..<j+k]$

— LOOPS as a simprule, since  $j \leq j$ .

**by** (*induct k*) *auto*

**lemma** *length-upt* [simp]:  $\text{length } [i..<j] = j - i$

**by** (*induct j*) (*auto simp add: Suc-diff-le*)

**lemma** *nth-upt* [simp]:  $i + k < j \implies [i..<j] ! k = i + k$

**apply** (*induct j*)

**apply** (*auto simp add: less-Suc-eq nth-append split: nat-diff-split*)

**done**

**lemma** *hd-upt*[simp]:  $i < j \implies \text{hd}[i..<j] = i$

**by**(*simp add:upt-conv-Cons*)

**lemma** *last-upt*[simp]:  $i < j \implies \text{last}[i..<j] = j - 1$

**apply**(*cases j*)

**apply** *simp*

**by**(*simp add:upt-Suc-append*)



```

lemma take-upt [simp]:  $i+m \leq n \implies \text{take } m [i..<n] = [i..<i+m]$ 
apply (induct m arbitrary: i, simp)
apply (subst upt-rec)
apply (rule sym)
apply (subst upt-rec)
apply (simp del: upt.simps)
done

```

```

lemma drop-upt[simp]:  $\text{drop } m [i..<j] = [i+m..<j]$ 
apply (induct j)
apply auto
done

```

```

lemma map-Suc-upt:  $\text{map } \text{Suc } [m..<n] = [\text{Suc } m..<\text{Suc } n]$ 
by (induct n) auto

```

```

lemma nth-map-upt:  $i < n-m \implies (\text{map } f [m..<n]) ! i = f(m+i)$ 
apply (induct n m arbitrary: i rule: diff-induct)
prefer 3 apply (subst map-Suc-upt[symmetric])
apply (auto simp add: less-diff-conv nth-upt)
done

```

```

lemma nth-take-lemma:
   $k \leq \text{length } xs \implies k \leq \text{length } ys \implies$ 
   $(!!i. i < k \longrightarrow xs!i = ys!i) \implies \text{take } k xs = \text{take } k ys$ 
apply (atomize, induct k arbitrary: xs ys)
apply (simp-all add: less-Suc-eq-0-disj all-conj-distrib, clarify)

```

Both lists must be non-empty

```

apply (case-tac xs, simp)
apply (case-tac ys, clarify)
apply (simp (no-asm-use))
apply clarify

```

prenexing's needed, not miniscoping

```

apply (simp (no-asm-use) add: all-simps [symmetric] del: all-simps)
apply blast
done

```

```

lemma nth-equalityI:
   $[| \text{length } xs = \text{length } ys; \text{ALL } i < \text{length } xs. xs!i = ys!i |] \implies xs = ys$ 
apply (frule nth-take-lemma [OF le-refl eq-imp-le])
apply (simp-all add: take-all)
done

```

```

lemma map-nth:
   $\text{map } (\lambda i. xs ! i) [0..<\text{length } xs] = xs$ 
by (rule nth-equalityI, auto)

```

**lemma** *list-all2-antisym*:

```

  [ (∧ x y. [P x y; Q y x] ==> x = y); list-all2 P xs ys; list-all2 Q ys xs ]
  ==> xs = ys
  apply (simp add: list-all2-conv-all-nth)
  apply (rule nth-equalityI, blast, simp)
  done

```

**lemma** *take-equalityI*:  $(\forall i. \text{take } i \text{ xs} = \text{take } i \text{ ys}) ==> \text{xs} = \text{ys}$

— The famous take-lemma.

```

  apply (drule-tac x = max (length xs) (length ys) in spec)
  apply (simp add: le-max-iff-disj take-all)
  done

```

**lemma** *take-Cons'*:

```

  take n (x # xs) = (if n = 0 then [] else x # take (n - 1) xs)
  by (cases n) simp-all

```

**lemma** *drop-Cons'*:

```

  drop n (x # xs) = (if n = 0 then x # xs else drop (n - 1) xs)
  by (cases n) simp-all

```

**lemma** *nth-Cons'*:  $(x \# xs)!n = (\text{if } n = 0 \text{ then } x \text{ else } xs!(n - 1))$

```

  by (cases n) simp-all

```

**lemmas** *take-Cons-number-of* = *take-Cons'*[of number-of v, standard]

**lemmas** *drop-Cons-number-of* = *drop-Cons'*[of number-of v, standard]

**lemmas** *nth-Cons-number-of* = *nth-Cons'*[of - - number-of v, standard]

**declare** *take-Cons-number-of* [simp]

*drop-Cons-number-of* [simp]

*nth-Cons-number-of* [simp]

### 43.1.21 upto: interval-list on int

**function** *upto* ::  $\text{int} \Rightarrow \text{int} \Rightarrow \text{int list}$  ( $(1[-..-])$ ) **where**

*upto*  $i \ j = (\text{if } i \leq j \text{ then } i \# [i+1..j] \text{ else } [])$

**by** *auto*

**termination**

**by**(*relation measure*( $\%(i::\text{int}, j). \text{nat}(j - i + 1)$ )) *auto*

**declare** *upto.simps*[code, simp del]

**lemmas** *upto-rec-number-of*[simp] =

*upto.simps*[of number-of m number-of n, standard]

**lemma** *upto-empty*[simp]:  $j < i \implies [i..j] = []$

**by**(*simp add: upto.simps*)

```

lemma set-upto[simp]:  $\text{set}[i..j] = \{i..j\}$ 
apply(induct  $i\ j$  rule:upto.induct)
apply(simp add: upto.simps simp-from-to)
done

```

#### 43.1.22 *distinct and remdups*

```

lemma distinct-append [simp]:
 $\text{distinct } (xs @ ys) = (\text{distinct } xs \wedge \text{distinct } ys \wedge \text{set } xs \cap \text{set } ys = \{\})$ 
by (induct xs) auto

```

```

lemma distinct-rev[simp]:  $\text{distinct}(\text{rev } xs) = \text{distinct } xs$ 
by(induct xs) auto

```

```

lemma set-remdups [simp]:  $\text{set } (\text{remdups } xs) = \text{set } xs$ 
by (induct xs) (auto simp add: insert-absorb)

```

```

lemma distinct-remdups [iff]:  $\text{distinct } (\text{remdups } xs)$ 
by (induct xs) auto

```

```

lemma distinct-remdups-id:  $\text{distinct } xs ==> \text{remdups } xs = xs$ 
by (induct xs, auto)

```

```

lemma remdups-id-iff-distinct [simp]:  $\text{remdups } xs = xs \longleftrightarrow \text{distinct } xs$ 
by (metis distinct-remdups distinct-remdups-id)

```

```

lemma finite-distinct-list:  $\text{finite } A \implies \exists x. \text{set } xs = A \ \& \ \text{distinct } xs$ 
by (metis distinct-remdups finite-list set-remdups)

```

```

lemma remdups-eq-nil-iff [simp]:  $(\text{remdups } x = []) = (x = [])$ 
by (induct x, auto)

```

```

lemma remdups-eq-nil-right-iff [simp]:  $([] = \text{remdups } x) = (x = [])$ 
by (induct x, auto)

```

```

lemma length-remdups-leq[iff]:  $\text{length}(\text{remdups } xs) \leq \text{length } xs$ 
by (induct xs) auto

```

```

lemma length-remdups-eq[iff]:
 $(\text{length } (\text{remdups } xs) = \text{length } xs) = (\text{remdups } xs = xs)$ 
apply(induct xs)
apply auto
apply(subgoal-tac length (remdups xs) <= length xs)
apply arith
apply(rule length-remdups-leq)
done

```

```

lemma remdups-filter:  $\text{remdups}(\text{filter } P \ xs) = \text{filter } P \ (\text{remdups } xs)$ 

```

```

apply(induct xs)
apply auto
done

```

```

lemma distinct-map:
  distinct(map f xs) = (distinct xs & inj-on f (set xs))
by (induct xs) auto

```

```

lemma distinct-filter [simp]: distinct xs ==> distinct (filter P xs)
by (induct xs) auto

```

```

lemma distinct-upt[simp]: distinct[i..<j]
by (induct j) auto

```

```

lemma distinct-upto[simp]: distinct[i..j]
apply(induct i j rule:upto.induct)
apply(subst upto.simps)
apply(simp)
done

```

```

lemma distinct-take[simp]: distinct xs ==> distinct (take i xs)
apply(induct xs arbitrary: i)
  apply simp
apply (case-tac i)
  apply simp-all
apply(blast dest:in-set-takeD)
done

```

```

lemma distinct-drop[simp]: distinct xs ==> distinct (drop i xs)
apply(induct xs arbitrary: i)
  apply simp
apply (case-tac i)
  apply simp-all
done

```

```

lemma distinct-list-update:
assumes d: distinct xs and a: a ∉ set xs - {xs!i}
shows distinct (xs[i:=a])
proof (cases i < length xs)
  case True
    with a have a ∉ set (take i xs @ xs ! i # drop (Suc i) xs) - {xs!i}
    apply (drule-tac id-take-nth-drop) by simp
    with d True show ?thesis
    apply (simp add: upd-conv-take-nth-drop)
    apply (drule subst [OF id-take-nth-drop]) apply assumption
    apply simp apply (cases a = xs!i) apply simp by blast
  next
    case False with d show ?thesis by auto

```

qed

**lemma** *distinct-concat*:

**assumes** *distinct xs*

**and**  $\bigwedge ys. ys \in \text{set } xs \implies \text{distinct } ys$

**and**  $\bigwedge ys\ zs. \llbracket ys \in \text{set } xs ; zs \in \text{set } xs ; ys \neq zs \rrbracket \implies \text{set } ys \cap \text{set } zs = \{\}$

**shows** *distinct (concat xs)*

**using** *assms* **by** (*induct xs*) *auto*

It is best to avoid this indexed version of *distinct*, but sometimes it is useful.

**lemma** *distinct-conv-nth*:

*distinct xs = ( $\forall i < \text{size } xs. \forall j < \text{size } xs. i \neq j \longrightarrow xs!i \neq xs!j$ )*

**apply** (*induct xs, simp, simp*)

**apply** (*rule iffI, clarsimp*)

**apply** (*case-tac i*)

**apply** (*case-tac j, simp*)

**apply** (*simp add: set-conv-nth*)

**apply** (*case-tac j*)

**apply** (*clarsimp simp add: set-conv-nth, simp*)

**apply** (*rule conjI*)

**apply** (*clarsimp simp add: set-conv-nth*)

**apply** (*erule-tac x = 0 in allE, simp*)

**apply** (*erule-tac x = Suc i in allE, simp, clarsimp*)

**apply** (*erule-tac x = Suc i in allE, simp*)

**apply** (*erule-tac x = Suc j in allE, simp*)

**done**

**lemma** *nth-eq-iff-index-eq*:

$\llbracket \text{distinct } xs ; i < \text{length } xs ; j < \text{length } xs \rrbracket \implies (xs!i = xs!j) = (i = j)$

**by**(*auto simp: distinct-conv-nth*)

**lemma** *distinct-card*: *distinct xs ==> card (set xs) = size xs*

**by** (*induct xs*) *auto*

**lemma** *card-distinct*: *card (set xs) = size xs ==> distinct xs*

**proof** (*induct xs*)

**case** *Nil* **thus** ?*case* **by** *simp*

**next**

**case** (*Cons x xs*)

**show** ?*case*

**proof** (*cases x ∈ set xs*)

**case** *False* **with** *Cons* **show** ?*thesis* **by** *simp*

**next**

**case** *True* **with** *Cons.prem*s

**have** *card (set xs) = Suc (length xs)*

**by** (*simp add: card-insert-if split: split-if-asm*)

**moreover** **have** *card (set xs) ≤ length xs* **by** (*rule card-length*)

ultimately have *False* by *simp*  
 thus ?thesis ..  
 qed  
 qed

**lemma** *not-distinct-decomp*:  $\sim \text{distinct } ws \implies \exists x\ y\ z\ y. ws = xs @ [y] @ ys @ [y] @ zs$   
**apply** (induct  $n == \text{length } ws$  arbitrary:ws) **apply** *simp*  
**apply** (case-tac ws) **apply** *simp*  
**apply** (simp split:split-if-asm)  
**apply** (metis Cons-eq-appendI eq-Nil-appendI split-list)  
**done**

**lemma** *length-remdups-concat*:  
 $\text{length}(\text{remdups}(\text{concat } xss)) = \text{card}(\bigcup xs \in \text{set } xss. \text{set } xs)$   
**by** (simp add: set-concat distinct-card[symmetric])

**lemma** *length-remdups-card-conv*:  $\text{length}(\text{remdups } xs) = \text{card}(\text{set } xs)$   
**proof** –  
 have  $xs: \text{concat}[xs] = xs$  **by** *simp*  
 from *length-remdups-concat* [of [xs]] **show** ?thesis **unfolding** xs **by** *simp*  
**qed**

**lemma** *remdups-remdups*:  
 $\text{remdups}(\text{remdups } xs) = \text{remdups } xs$   
**by** (induct xs) *simp-all*

**lemma** *distinct-butlast*:  
 assumes  $xs \neq []$  and *distinct* xs  
 shows *distinct* (butlast xs)  
**proof** –  
 from  $\langle xs \neq [] \rangle$  **obtain** ys y **where**  $xs = ys @ [y]$  **by** (cases xs rule: rev-cases)  
 auto  
 with  $\langle \text{distinct } xs \rangle$  **show** ?thesis **by** *simp*  
**qed**

### 43.1.23 insert

**lemma** *in-set-insert* [simp]:  
 $x \in \text{set } xs \implies \text{List.insert } x\ xs = xs$   
**by** (simp add: List.insert-def)

**lemma** *not-in-set-insert* [simp]:  
 $x \notin \text{set } xs \implies \text{List.insert } x\ xs = x \# xs$   
**by** (simp add: List.insert-def)

**lemma** *insert-Nil* [simp]:  
 $\text{List.insert } x\ [] = [x]$   
**by** *simp*

**lemma** *set-insert* [simp]:

*set (List.insert x xs) = insert x (set xs)*

**by** (auto simp add: List.insert-def)

**lemma** *distinct-insert* [simp]:

*distinct xs  $\implies$  distinct (List.insert x xs)*

**by** (simp add: List.insert-def)

**lemma** *insert-remdups*:

*List.insert x (remdups xs) = remdups (List.insert x xs)*

**by** (simp add: List.insert-def)

**lemma** *distinct-induct* [consumes 1, case-names Nil insert]:

**assumes** *distinct xs*

**assumes** *P []*

**assumes** *insert:  $\bigwedge x xs. \text{distinct } xs \implies x \notin \text{set } xs$*

*$\implies P xs \implies P (\text{List.insert } x xs)$*

**shows** *P xs*

**using** *distinct xs* **proof** (induct xs)

**case** Nil **from** *P []* **show** ?case .

**next**

**case** (Cons x xs)

**then have** *distinct xs* **and** *x  $\notin$  set xs* **and** *P xs* **by** *simp-all*

**with insert** **have** *P (List.insert x xs)* .

**with** *x  $\notin$  set xs* **show** ?case **by** *simp*

**qed**

#### 43.1.24 *remove1*

**lemma** *remove1-append*:

*remove1 x (xs @ ys) =*

*(if x  $\in$  set xs then remove1 x xs @ ys else xs @ remove1 x ys)*

**by** (induct xs) auto

**lemma** *remove1-commute*: *remove1 x (remove1 y zs) = remove1 y (remove1 x zs)*

**by** (induct zs) auto

**lemma** *in-set-remove1*[simp]:

*a  $\neq$  b  $\implies$  a : set(remove1 b xs) = (a : set xs)*

**apply** (induct xs)

**apply** auto

**done**

**lemma** *set-remove1-subset*: *set(remove1 x xs)  $\leq$  set xs*

**apply**(induct xs)

**apply** *simp*

**apply** *simp*

**apply** *blast*

**done**

```

lemma set-remove1-eq [simp]: distinct xs ==> set(remove1 x xs) = set xs - {x}
apply (induct xs)
  apply simp
apply simp
apply blast
done

```

```

lemma length-remove1:
  length(remove1 x xs) = (if x : set xs then length xs - 1 else length xs)
apply (induct xs)
  apply (auto dest!:length-pos-if-in-set)
done

```

```

lemma remove1-filter-not [simp]:
   $\neg P\ x \implies \text{remove1 } x\ (\text{filter } P\ xs) = \text{filter } P\ xs$ 
by (induct xs) auto

```

```

lemma notin-set-remove1 [simp]:  $x \notin \text{set } xs \implies x \notin \text{set}(\text{remove1 } y\ xs)$ 
apply (insert set-remove1-subset)
apply fast
done

```

```

lemma distinct-remove1 [simp]: distinct xs ==> distinct(remove1 x xs)
by (induct xs) simp-all

```

```

lemma remove1-remdups:
  distinct xs ==> remove1 x (remdups xs) = remdups (remove1 x xs)
by (induct xs) simp-all

```

```

lemma remove1-idem:
  assumes  $x \notin \text{set } xs$ 
  shows remove1 x xs = xs
  using assms by (induct xs) simp-all

```

#### 43.1.25 *removeAll*

```

lemma removeAll-filter-not-eq:
  removeAll x = filter ( $\lambda y. x \neq y$ )
proof
  fix xs
  show removeAll x xs = filter ( $\lambda y. x \neq y$ ) xs
  by (induct xs) auto
qed

```

```

lemma removeAll-append [simp]:
  removeAll x (xs @ ys) = removeAll x xs @ removeAll x ys
by (induct xs) auto

```



**lemma** *set-removeAll*[simp]:  $\text{set}(\text{removeAll } x \text{ } xs) = \text{set } xs - \{x\}$   
**by** (induct xs) auto

**lemma** *removeAll-id*[simp]:  $x \notin \text{set } xs \implies \text{removeAll } x \text{ } xs = xs$   
**by** (induct xs) auto

**lemma** *removeAll-filter-not*[simp]:  
 $\neg P \ x \implies \text{removeAll } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$   
**by**(induct xs) auto

**lemma** *distinct-removeAll*:  
 $\text{distinct } xs \implies \text{distinct } (\text{removeAll } x \text{ } xs)$   
**by** (simp add: removeAll-filter-not-eq)

**lemma** *distinct-remove1-removeAll*:  
 $\text{distinct } xs \implies \text{remove1 } x \text{ } xs = \text{removeAll } x \text{ } xs$   
**by** (induct xs) simp-all

**lemma** *map-removeAll-inj-on*:  $\text{inj-on } f \ (\text{insert } x \ (\text{set } xs)) \implies$   
 $\text{map } f \ (\text{removeAll } x \text{ } xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$   
**by** (induct xs) (simp-all add: inj-on-def)

**lemma** *map-removeAll-inj*:  $\text{inj } f \implies$   
 $\text{map } f \ (\text{removeAll } x \text{ } xs) = \text{removeAll } (f \ x) \ (\text{map } f \ xs)$   
**by**(metis map-removeAll-inj-on subset-inj-on subset-UNIV)

### 43.1.26 replicate

**lemma** *length-replicate* [simp]:  $\text{length } (\text{replicate } n \ x) = n$   
**by** (induct n) auto

**lemma** *Ex-list-of-length*:  $\exists xs. \text{length } xs = n$   
**by** (rule exI[of - replicate n undefined]) simp

**lemma** *map-replicate* [simp]:  $\text{map } f \ (\text{replicate } n \ x) = \text{replicate } n \ (f \ x)$   
**by** (induct n) auto

**lemma** *map-replicate-const*:  
 $\text{map } (\lambda x. k) \text{ lst} = \text{replicate } (\text{length } \text{lst}) \ k$   
**by** (induct lst) auto

**lemma** *replicate-app-Cons-same*:  
 $(\text{replicate } n \ x) @ (x \# xs) = x \# \text{replicate } n \ x @ xs$   
**by** (induct n) auto

**lemma** *rev-replicate* [simp]:  $\text{rev } (\text{replicate } n \ x) = \text{replicate } n \ x$   
**apply** (induct n, simp)

```

apply (simp add: replicate-app-Cons-same)
done

```

```

lemma replicate-add: replicate (n + m) x = replicate n x @ replicate m x
by (induct n) auto

```

Courtesy of Matthias Daum:

```

lemma append-replicate-commute:
  replicate n x @ replicate k x = replicate k x @ replicate n x
apply (simp add: replicate-add [THEN sym])
apply (simp add: add-commute)
done

```

Courtesy of Andreas Lochbihler:

```

lemma filter-replicate:
  filter P (replicate n x) = (if P x then replicate n x else [])
by(induct n) auto

```

```

lemma hd-replicate [simp]: n ≠ 0 ==> hd (replicate n x) = x
by (induct n) auto

```

```

lemma tl-replicate [simp]: n ≠ 0 ==> tl (replicate n x) = replicate (n - 1) x
by (induct n) auto

```

```

lemma last-replicate [simp]: n ≠ 0 ==> last (replicate n x) = x
by (atomize (full), induct n) auto

```

```

lemma nth-replicate[simp]: i < n ==> (replicate n x)!i = x
apply (induct n arbitrary: i, simp)
apply (simp add: nth-Cons split: nat.split)
done

```

Courtesy of Matthias Daum (2 lemmas):

```

lemma take-replicate[simp]: take i (replicate k x) = replicate (min i k) x
apply (case-tac k ≤ i)
  apply (simp add: min-def)
apply (drule not-leE)
apply (simp add: min-def)
apply (subgoal-tac replicate k x = replicate i x @ replicate (k - i) x)
  apply simp
apply (simp add: replicate-add [symmetric])
done

```

```

lemma drop-replicate[simp]: drop i (replicate k x) = replicate (k-i) x
apply (induct k arbitrary: i)
  apply simp
apply clarsimp
apply (case-tac i)
  apply simp

```

**apply** *clarsimp*  
**done**

**lemma** *set-replicate-Suc*:  $\text{set } (\text{replicate } (\text{Suc } n) \ x) = \{x\}$   
**by** (*induct n*) *auto*

**lemma** *set-replicate [simp]*:  $n \neq 0 \implies \text{set } (\text{replicate } n \ x) = \{x\}$   
**by** (*fast dest!*; *not0-implies-Suc intro!*; *set-replicate-Suc*)

**lemma** *set-replicate-conv-if*:  $\text{set } (\text{replicate } n \ x) = (\text{if } n = 0 \text{ then } \{\} \text{ else } \{x\})$   
**by** *auto*

**lemma** *in-set-replicateD*:  $x : \text{set } (\text{replicate } n \ y) \implies x = y$   
**by** (*simp add: set-replicate-conv-if split: split-if-asm*)

**lemma** *replicate-append-same*:  
 $\text{replicate } i \ x \ @ \ [x] = x \ \# \ \text{replicate } i \ x$   
**by** (*induct i*) *simp-all*

**lemma** *map-replicate-trivial*:  
 $\text{map } (\lambda i. \ x) \ [0..<i] = \text{replicate } i \ x$   
**by** (*induct i*) (*simp-all add: replicate-append-same*)

**lemma** *concat-replicate-trivial[simp]*:  
 $\text{concat } (\text{replicate } i \ []) = []$   
**by** (*induct i*) (*auto simp add: map-replicate-const*)

**lemma** *replicate-empty[simp]*:  $(\text{replicate } n \ x = []) \longleftrightarrow n=0$   
**by** (*induct n*) *auto*

**lemma** *empty-replicate[simp]*:  $([] = \text{replicate } n \ x) \longleftrightarrow n=0$   
**by** (*induct n*) *auto*

**lemma** *replicate-eq-replicate[simp]*:  
 $(\text{replicate } m \ x = \text{replicate } n \ y) \longleftrightarrow (m=n \ \& \ (m \neq 0 \longrightarrow x=y))$   
**apply** (*induct m arbitrary: n*)  
**apply** *simp*  
**apply** (*induct-tac n*)  
**apply** *auto*  
**done**

#### 43.1.27 *rotate1* and *rotate*

**lemma** *rotate-simps[simp]*:  $\text{rotate1 } [] = [] \ \wedge \ \text{rotate1 } (x \ \# \ xs) = xs \ @ \ [x]$   
**by** (*simp add: rotate1-def*)

**lemma** *rotate0[simp]*:  $\text{rotate } 0 = \text{id}$   
**by** (*simp add: rotate-def*)

**lemma** *rotate-Suc*[simp]:  $\text{rotate } (\text{Suc } n) \text{ } xs = \text{rotate1 } (\text{rotate } n \text{ } xs)$   
**by**(simp add:rotate-def)

**lemma** *rotate-add*:  
 $\text{rotate } (m+n) = \text{rotate } m \circ \text{rotate } n$   
**by**(simp add:rotate-def funpow-add)

**lemma** *rotate-rotate*:  $\text{rotate } m \text{ } (\text{rotate } n \text{ } xs) = \text{rotate } (m+n) \text{ } xs$   
**by**(simp add:rotate-add)

**lemma** *rotate1-rotate-swap*:  $\text{rotate1 } (\text{rotate } n \text{ } xs) = \text{rotate } n \text{ } (\text{rotate1 } xs)$   
**by**(simp add:rotate-def funpow-swap1)

**lemma** *rotate1-length01*[simp]:  $\text{length } xs \leq 1 \implies \text{rotate1 } xs = xs$   
**by**(cases xs) simp-all

**lemma** *rotate-length01*[simp]:  $\text{length } xs \leq 1 \implies \text{rotate } n \text{ } xs = xs$   
**apply**(induct n)  
**apply** simp  
**apply** (simp add:rotate-def)  
**done**

**lemma** *rotate1-hd-tl*:  $xs \neq [] \implies \text{rotate1 } xs = \text{tl } xs @ [\text{hd } xs]$   
**by**(simp add:rotate1-def split:list.split)

**lemma** *rotate-drop-take*:  
 $\text{rotate } n \text{ } xs = \text{drop } (n \bmod \text{length } xs) \text{ } xs @ \text{take } (n \bmod \text{length } xs) \text{ } xs$   
**apply**(induct n)  
**apply** simp  
**apply**(simp add:rotate-def)  
**apply**(cases xs = [])  
**apply** (simp)  
**apply**(case-tac n mod length xs = 0)  
**apply**(simp add:mod-Suc)  
**apply**(simp add: rotate1-hd-tl drop-Suc take-Suc)  
**apply**(simp add:mod-Suc rotate1-hd-tl drop-Suc[symmetric] drop-tl[symmetric]  
take-hd-drop linorder-not-le)  
**done**

**lemma** *rotate-conv-mod*:  $\text{rotate } n \text{ } xs = \text{rotate } (n \bmod \text{length } xs) \text{ } xs$   
**by**(simp add:rotate-drop-take)

**lemma** *rotate-id*[simp]:  $n \bmod \text{length } xs = 0 \implies \text{rotate } n \text{ } xs = xs$   
**by**(simp add:rotate-drop-take)

**lemma** *length-rotate1*[simp]:  $\text{length}(\text{rotate1 } xs) = \text{length } xs$   
**by**(simp add:rotate1-def split:list.split)

**lemma** *length-rotate*[simp]:  $\text{length}(\text{rotate } n \text{ } xs) = \text{length } xs$   
**by** (*induct* *n* *arbitrary*: *xs*) (*simp-all* *add*:*rotate-def*)

**lemma** *distinct1-rotate*[simp]:  $\text{distinct}(\text{rotate1 } xs) = \text{distinct } xs$   
**by**(*simp* *add*:*rotate1-def* *split*:*list.split*) *blast*

**lemma** *distinct-rotate*[simp]:  $\text{distinct}(\text{rotate } n \text{ } xs) = \text{distinct } xs$   
**by** (*induct* *n*) (*simp-all* *add*:*rotate-def*)

**lemma** *rotate-map*:  $\text{rotate } n \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{rotate } n \text{ } xs)$   
**by**(*simp* *add*:*rotate-drop-take* *take-map* *drop-map*)

**lemma** *set-rotate1*[simp]:  $\text{set}(\text{rotate1 } xs) = \text{set } xs$   
**by**(*simp* *add*:*rotate1-def* *split*:*list.split*)

**lemma** *set-rotate*[simp]:  $\text{set}(\text{rotate } n \text{ } xs) = \text{set } xs$   
**by** (*induct* *n*) (*simp-all* *add*:*rotate-def*)

**lemma** *rotate1-is-Nil-conv*[simp]:  $(\text{rotate1 } xs = []) = (xs = [])$   
**by**(*simp* *add*:*rotate1-def* *split*:*list.split*)

**lemma** *rotate-is-Nil-conv*[simp]:  $(\text{rotate } n \text{ } xs = []) = (xs = [])$   
**by** (*induct* *n*) (*simp-all* *add*:*rotate-def*)

**lemma** *rotate-rev*:  
 $\text{rotate } n \text{ } (\text{rev } xs) = \text{rev}(\text{rotate } (\text{length } xs - (n \bmod \text{length } xs)) \text{ } xs)$   
**apply**(*simp* *add*:*rotate-drop-take* *rev-drop* *rev-take*)  
**apply**(*cases*  $\text{length } xs = 0$ )  
**apply** *simp*  
**apply**(*cases*  $n \bmod \text{length } xs = 0$ )  
**apply** *simp*  
**apply**(*simp* *add*:*rotate-drop-take* *rev-drop* *rev-take*)  
**done**

**lemma** *hd-rotate-conv-nth*:  $xs \neq [] \implies \text{hd}(\text{rotate } n \text{ } xs) = xs!(n \bmod \text{length } xs)$   
**apply**(*simp* *add*:*rotate-drop-take* *hd-append* *hd-drop-conv-nth* *hd-conv-nth*)  
**apply**(*subgoal-tac*  $\text{length } xs \neq 0$ )  
**prefer** 2 **apply** *simp*  
**using** *mod-less-divisor*[*of*  $\text{length } xs \text{ } n$ ] **by** *arith*

#### 43.1.28 *sublist* — a generalization of *nth* to sets

**lemma** *sublist-empty* [simp]:  $\text{sublist } xs \text{ } \{\} = []$   
**by** (*auto* *simp* *add*: *sublist-def*)

**lemma** *sublist-nil* [simp]:  $\text{sublist } [] \text{ } A = []$   
**by** (*auto* *simp* *add*: *sublist-def*)

**lemma** *length-sublist*:

$\text{length}(\text{sublist } xs \ I) = \text{card}\{i. i < \text{length } xs \wedge i : I\}$   
**by**(*simp add: sublist-def length-filter-conv-card cong:conj-cong*)

**lemma** *sublist-shift-lemma-Suc*:  
 $\text{map fst } (\text{filter } (\%p. P(\text{Suc}(\text{snd } p))) (\text{zip } xs \ is)) =$   
 $\text{map fst } (\text{filter } (\%p. P(\text{snd } p)) (\text{zip } xs \ (\text{map } \text{Suc } is)))$   
**apply**(*induct xs arbitrary: is*)  
**apply** *simp*  
**apply** (*case-tac is*)  
**apply** *simp*  
**apply** *simp*  
**done**

**lemma** *sublist-shift-lemma*:  
 $\text{map fst } [p < - \text{zip } xs \ [i..<i + \text{length } xs] . \text{snd } p : A] =$   
 $\text{map fst } [p < - \text{zip } xs \ [0..<\text{length } xs] . \text{snd } p + i : A]$   
**by** (*induct xs rule: rev-induct*) (*simp-all add: add-commute*)

**lemma** *sublist-append*:  
 $\text{sublist } (l \ @ \ l') \ A = \text{sublist } l \ A \ @ \ \text{sublist } l' \ \{j. j + \text{length } l : A\}$   
**apply** (*unfold sublist-def*)  
**apply** (*induct l' rule: rev-induct, simp*)  
**apply** (*simp add: upt-add-eq-append[of 0] zip-append sublist-shift-lemma*)  
**apply** (*simp add: add-commute*)  
**done**

**lemma** *sublist-Cons*:  
 $\text{sublist } (x \ \# \ l) \ A = (\text{if } 0:A \text{ then } [x] \text{ else } []) \ @ \ \text{sublist } l \ \{j. \text{Suc } j : A\}$   
**apply** (*induct l rule: rev-induct*)  
**apply** (*simp add: sublist-def*)  
**apply** (*simp del: append-Cons add: append-Cons[symmetric] sublist-append*)  
**done**

**lemma** *set-sublist*:  $\text{set}(\text{sublist } xs \ I) = \{xs!i \mid i. i < \text{size } xs \wedge i \in I\}$   
**apply**(*induct xs arbitrary: I*)  
**apply**(*auto simp: sublist-Cons nth-Cons split:nat.split dest!: gr0-implies-Suc*)  
**done**

**lemma** *set-sublist-subset*:  $\text{set}(\text{sublist } xs \ I) \subseteq \text{set } xs$   
**by**(*auto simp add: set-sublist*)

**lemma** *notin-set-sublistI*[*simp*]:  $x \notin \text{set } xs \implies x \notin \text{set}(\text{sublist } xs \ I)$   
**by**(*auto simp add: set-sublist*)

**lemma** *in-set-sublistD*:  $x \in \text{set}(\text{sublist } xs \ I) \implies x \in \text{set } xs$   
**by**(*auto simp add: set-sublist*)

**lemma** *sublist-singleton* [*simp*]:  $\text{sublist } [x] \ A = (\text{if } 0 : A \text{ then } [x] \text{ else } [])$   
**by** (*simp add: sublist-Cons*)

```

lemma distinct-sublistI[simp]: distinct xs  $\implies$  distinct(sublist xs I)
apply(induct xs arbitrary: I)
  apply simp
apply(auto simp add:sublist-Cons)
done

```

```

lemma sublist-upt-eq-take [simp]: sublist l {..n} = take n l
apply (induct l rule: rev-induct, simp)
apply (simp split: nat-diff-split add: sublist-append)
done

```

```

lemma filter-in-sublist:
  distinct xs  $\implies$  filter (%x. x  $\in$  set(sublist xs s)) xs = sublist xs s
proof (induct xs arbitrary: s)
  case Nil thus ?case by simp
next
  case (Cons a xs)
  moreover hence !x. x: set xs  $\longrightarrow$  x  $\neq$  a by auto
  ultimately show ?case by(simp add: sublist-Cons cong:filter-cong)
qed

```

#### 43.1.29 splice

```

lemma splice-Nil2 [simp, code]:
  splice xs [] = xs
by (cases xs) simp-all

```

```

lemma splice-Cons-Cons [simp, code]:
  splice (x#xs) (y#ys) = x # y # splice xs ys
by simp

```

```

declare splice.simps(2) [simp del, code del]

```

```

lemma length-splice[simp]: length(splice xs ys) = length xs + length ys
apply(induct xs arbitrary: ys) apply simp
apply(case-tac ys)
  apply auto
done

```

#### 43.1.30 Transpose

```

function transpose where
  transpose [] = [] |
  transpose ([_] # xss) = transpose xss |
  transpose ((x#xs) # xss) =
    (x # [h. (h#t)  $\leftarrow$  xss]) # transpose (xs # [t. (h#t)  $\leftarrow$  xss])
by pat-completeness auto

```

**lemma** *transpose-aux-filter-head*:

*concat* (*map* (*list-case* [] ( $\lambda h t. [h]$ )) *xss*) =  
*map* ( $\lambda xs. hd\ xs$ ) [*ys*←*xss* . *ys* ≠ []]  
**by** (*induct* *xss*) (*auto split: list.split*)

**lemma** *transpose-aux-filter-tail*:

*concat* (*map* (*list-case* [] ( $\lambda h t. [t]$ )) *xss*) =  
*map* ( $\lambda xs. tl\ xs$ ) [*ys*←*xss* . *ys* ≠ []]  
**by** (*induct* *xss*) (*auto split: list.split*)

**lemma** *transpose-aux-max*:

*max* (*Suc* (*length* *xs*)) (*foldr* ( $\lambda xs. max\ (length\ xs)$ ) *xss* 0) =  
*Suc* (*max* (*length* *xs*) (*foldr* ( $\lambda x. max\ (length\ x - Suc\ 0)$ ) [*ys*←*xss* . *ys* ≠ []] 0))  
(is *max* - ?*foldB* = *Suc* (*max* - ?*foldA*))

**proof** (*cases* [*ys*←*xss* . *ys* ≠ []] = [])

**case** *True*

**hence** *foldr* ( $\lambda xs. max\ (length\ xs)$ ) *xss* 0 = 0

**proof** (*induct* *xss*)

**case** (*Cons* *x xs*)

**moreover** **hence** *x* = [] **by** (*cases* *x*) *auto*

**ultimately show** ?*case* **by** *auto*

**qed** *simp*

**thus** ?*thesis* **using** *True* **by** *simp*

**next**

**case** *False*

**have** *foldA*: ?*foldA* = *foldr* ( $\lambda x. max\ (length\ x)$ ) [*ys*←*xss* . *ys* ≠ []] 0 - 1

**by** (*induct* *xss*) *auto*

**have** *foldB*: ?*foldB* = *foldr* ( $\lambda x. max\ (length\ x)$ ) [*ys*←*xss* . *ys* ≠ []] 0

**by** (*induct* *xss*) *auto*

**have** 0 < ?*foldB*

**proof** -

**from** *False*

**obtain** *z zs* **where** *zs*: [*ys*←*xss* . *ys* ≠ []] = *z*#*zs* **by** (*auto simp: neg-Nil-conv*)

**hence** *z* ∈ *set* ([*ys*←*xss* . *ys* ≠ []]) **by** *auto*

**hence** *z* ≠ [] **by** *auto*

**thus** ?*thesis*

**unfolding** *foldB* *zs*

**by** (*auto simp: max-def intro: less-le-trans*)

**qed**

**thus** ?*thesis*

**unfolding** *foldA foldB max-Suc-Suc[symmetric]*

**by** *simp*

**qed**

**termination** *transpose*

**by** (*relation measure* ( $\lambda xs. foldr\ (\lambda xs. max\ (length\ xs))\ xs\ 0 + length\ xs$ ))



(*auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max less-Suc-eq-le*)

**lemma** *transpose-empty*:  $(\text{transpose } xs = []) \longleftrightarrow (\forall x \in \text{set } xs. x = [])$   
**by** (*induct rule: transpose.induct simp-all*)

**lemma** *length-transpose*:  
**fixes**  $xs :: 'a \text{ list list}$   
**shows**  $\text{length } (\text{transpose } xs) = \text{foldr } (\lambda xs. \max (\text{length } xs)) \text{ } xs \ 0$   
**by** (*induct rule: transpose.induct*)  
(*auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max max-Suc-Suc[symmetric] simp del: max-Suc-Suc*)

**lemma** *nth-transpose*:  
**fixes**  $xs :: 'a \text{ list list}$   
**assumes**  $i < \text{length } (\text{transpose } xs)$   
**shows**  $\text{transpose } xs ! i = \text{map } (\lambda xs. xs ! i) [ys \leftarrow xs. i < \text{length } ys]$   
**using** *assms proof* (*induct arbitrary: i rule: transpose.induct*)  
**case** ( $\exists x \ xs \ xss$ )  
**def**  $XS == (x \# xs) \# xss$   
**hence** [*simp*]:  $XS \neq []$  **by** *auto*  
**thus** ?*case*  
**proof** (*cases i*)  
**case** 0  
**thus** ?*thesis* **by** (*simp add: transpose-aux-filter-head hd-conv-nth*)  
**next**  
**case** (*Suc j*)  
**have** \*:  $\bigwedge xss. xs \# \text{map } \text{tl } xss = \text{map } \text{tl } ((x \# xs) \# xss)$  **by** *simp*  
**have** \*\*:  $\bigwedge xss. (x \# xs) \# \text{filter } (\lambda ys. ys \neq []) \ xss = \text{filter } (\lambda ys. ys \neq []) ((x \# xs) \# xss)$  **by** *simp*  
**{** **fix**  $x$  **have**  $\text{Suc } j < \text{length } x \longleftrightarrow x \neq [] \wedge j < \text{length } x - \text{Suc } 0$   
**by** (*cases x simp-all*)  
**}** **note** \*\*\* = *this*  
  
**have** *j-less*:  $j < \text{length } (\text{transpose } (xs \# \text{concat } (\text{map } (\text{list-case } [] (\lambda h \ t. [t]))) \ xss)))$   
**using**  $\exists$ .*prems* **by** (*simp add: transpose-aux-filter-tail length-transpose Suc*)  
  
**show** ?*thesis*  
**unfolding** *transpose.simps*  $\langle i = \text{Suc } j \rangle$  *nth-Cons-Suc*  $\exists$ .*hyps*[*OF j-less*]  
**apply** (*auto simp: transpose-aux-filter-tail filter-map comp-def length-transpose*  
\* \*\* \*\*\* *XS-def[symmetric]*)  
**apply** (*rule-tac y=x in list.exhaust*)  
**by** *auto*  
**qed**  
**qed** *simp-all*

**lemma** *transpose-map-map*:  
 $\text{transpose } (\text{map } (\text{map } f) \ xs) = \text{map } (\text{map } f) (\text{transpose } xs)$

```

proof (rule nth-equalityI, safe)
  have [simp]: length (transpose (map (map f) xs)) = length (transpose xs)
    by (simp add: length-transpose foldr-map comp-def)
  show length (transpose (map (map f) xs)) = length (map (map f) (transpose
xs)) by simp

  fix i assume i < length (transpose (map (map f) xs))
  thus transpose (map (map f) xs) ! i = map (map f) (transpose xs) ! i
    by (simp add: nth-transpose filter-map comp-def)
qed

```

#### 43.1.31 (In)finiteness

```

lemma finite-maxlen:
  finite (M::'a list set) ==> EX n. ALL s:M. size s < n
proof (induct rule: finite.induct)
  case emptyI show ?case by simp
next
  case (insertI M xs)
  then obtain n where  $\forall s \in M. \text{length } s < n$  by blast
  hence ALL s:insert xs M. size s < max n (size xs) + 1 by auto
  thus ?case ..
qed

```

```

lemma finite-lists-length-eq:
assumes finite A
shows finite {xs. set xs  $\subseteq$  A  $\wedge$  length xs = n} (is finite (?S n))
proof (induct n)
  case 0 show ?case by simp
next
  case (Suc n)
  have ?S (Suc n) = ( $\bigcup x \in A. (\lambda xs. x \# xs)$  ‘ ?S n)
    by (auto simp: length-Suc-conv)
  then show ?case using (finite A)
    by (auto intro: finite-imageI Suc)
qed

```

```

lemma finite-lists-length-le:
assumes finite A shows finite {xs. set xs  $\subseteq$  A  $\wedge$  length xs  $\leq$  n}
(is finite ?S)
proof –
  have ?S = ( $\bigcup n \in \{0..n\}. \{xs. \text{set } xs \subseteq A \wedge \text{length } xs = n\}$ ) by auto
  thus ?thesis by (auto intro: finite-lists-length-eq[OF (finite A)])
qed

```

```

lemma infinite-UNIV-listI:  $\sim$  finite (UNIV::'a list set)
apply (rule notI)
apply (drule finite-maxlen)
apply (metis UNIV-I length-replicate less-not-refl)

```

done

## 43.2 Sorting

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

**context** *linorder*  
**begin**

**lemma** *length-insert[simp]* : *length (insort-key f x xs) = Suc (length xs)*  
**by** (*induct xs, auto*)

**lemma** *insort-left-comm*:  
*insort x (insort y xs) = insort y (insort x xs)*  
**by** (*induct xs auto*)

**lemma** *fun-left-comm-insort*:  
*fun-left-comm insort*  
**proof**  
**qed** (*fact insort-left-comm*)

**lemma** *sort-key-simps [simp]*:  
*sort-key f [] = []*  
*sort-key f (x#xs) = insort-key f x (sort-key f xs)*  
**by** (*simp-all add: sort-key-def*)

**lemma** *sort-foldl-insort*:  
*sort xs = foldl ( $\lambda y x. \text{insort } x \ y$ ) [] xs*  
**by** (*simp add: sort-key-def foldr-foldl foldl-rev insort-left-comm*)

**lemma** *length-sort[simp]*: *length (sort-key f xs) = length xs*  
**by** (*induct xs, auto*)

**lemma** *sorted-Cons*: *sorted (x#xs) = (sorted xs & (ALL y:set xs. x <= y))*  
**apply**(*induct xs arbitrary: x*) **apply** *simp*  
**by** *simp (blast intro: order-trans)*

**lemma** *sorted-append*:  
*sorted (xs@ys) = (sorted xs & sorted ys & ( $\forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y$ ))*  
**by** (*induct xs (auto simp add:sorted-Cons)*)

**lemma** *sorted-nth-mono*:  
*sorted xs  $\implies i \leq j \implies j < \text{length } xs \implies xs!i \leq xs!j$*   
**by** (*induct xs arbitrary: i j (auto simp:nth-Cons' sorted-Cons)*)

**lemma** *sorted-rev-nth-mono*:

*sorted* (rev *xs*)  $\implies i \leq j \implies j < \text{length } xs \implies xs!j \leq xs!i$   
**using** *sorted-nth-mono*[ of rev *xs* length *xs* - *j* - 1 length *xs* - *i* - 1 ]  
     rev-nth[of length *xs* - *i* - 1 *xs*] rev-nth[of length *xs* - *j* - 1 *xs*]  
**by** *auto*

**lemma** *sorted-nth-monoI*:

$(\bigwedge i j. \llbracket i \leq j ; j < \text{length } xs \rrbracket \implies xs ! i \leq xs ! j) \implies \text{sorted } xs$   
**proof** (induct *xs*)  
   **case** (Cons *x xs*)  
   **have** *sorted xs*  
   **proof** (rule Cons.hyps)  
     **fix** *i j* **assume**  $i \leq j$  **and**  $j < \text{length } xs$   
     **with** Cons.prem[s of Suc *i* Suc *j*]  
     **show**  $xs ! i \leq xs ! j$  **by** *auto*  
   **qed**  
**moreover**  
 {  
   **fix** *y* **assume**  $y \in \text{set } xs$   
   **then obtain** *j* **where**  $j < \text{length } xs$  **and**  $xs ! j = y$   
     **unfolding** in-set-conv-nth **by** blast  
   **with** Cons.prem[s of 0 Suc *j*]  
   **have**  $x \leq y$   
   **by** *auto*  
 }  
**ultimately**  
**show** ?case  
   **unfolding** sorted-Cons **by** *auto*  
**qed** *simp*

**lemma** *sorted-equals-nth-mono*:

*sorted xs* =  $(\forall j < \text{length } xs. \forall i \leq j. xs ! i \leq xs ! j)$   
**by** (auto intro: sorted-nth-monoI sorted-nth-mono)

**lemma** *set-insort*: *set*(insort-key *f x xs*) = *insert x* (*set xs*)  
**by** (induct *xs*) *auto*

**lemma** *set-sort[simp]*: *set*(sort-key *f xs*) = *set xs*  
**by** (induct *xs*) (simp-all add:set-insort)

**lemma** *distinct-insort*: *distinct* (insort-key *f x xs*) =  $(x \notin \text{set } xs \wedge \text{distinct } xs)$   
**by**(induct *xs*)(auto simp:set-insort)

**lemma** *distinct-sort[simp]*: *distinct* (sort-key *f xs*) = *distinct xs*  
**by**(induct *xs*)(simp-all add:distinct-insort set-sort)

**lemma** *sorted-insort-key*: *sorted* (map *f* (insort-key *f x xs*)) = *sorted* (map *f xs*)  
**by**(induct *xs*)(auto simp:sorted-Cons set-insort)

**lemma** *sorted-insort*: *sorted (insort x xs) = sorted xs*  
**using** *sorted-insort-key*[**where**  $f = \lambda x. x$ ] **by** *simp*

**theorem** *sorted-sort-key*[*simp*]: *sorted (map f (sort-key f xs))*  
**by**(*induct xs*)(*auto simp:sorted-insort-key*)

**theorem** *sorted-sort*[*simp*]: *sorted (sort xs)*  
**by**(*induct xs*)(*auto simp:sorted-insort*)

**lemma** *sorted-butlast*:  
**assumes**  $xs \neq []$  **and** *sorted xs*  
**shows** *sorted (butlast xs)*  
**proof** –  
**from**  $\langle xs \neq [] \rangle$  **obtain**  $ys\ y$  **where**  $xs = ys @ [y]$  **by** (*cases xs rule: rev-cases*)  
*auto*  
**with**  $\langle sorted\ xs \rangle$  **show** *?thesis* **by** (*simp add: sorted-append*)  
**qed**

**lemma** *insort-not-Nil* [*simp*]:  
*insort-key f a xs  $\neq []$*   
**by** (*induct xs*) *simp-all*

**lemma** *insort-is-Cons*:  $\forall x \in set\ xs. f\ a \leq f\ x \implies insort-key\ f\ a\ xs = a \# xs$   
**by** (*cases xs*) *auto*

**lemma** *sorted-remove1*: *sorted xs  $\implies sorted (remove1 a xs)$*   
**by**(*induct xs*)(*auto simp add: sorted-Cons*)

**lemma** *insort-key-remove1*:  $\llbracket a \in set\ xs; sorted\ (map\ f\ xs) ; inj-on\ f\ (set\ xs) \rrbracket$   
 $\implies insort-key\ f\ a\ (remove1\ a\ xs) = xs$   
**proof** (*induct xs*)  
**case** (*Cons x xs*)  
**thus** *?case*  
**proof** (*cases x = a*)  
**case** *False*  
**hence**  $f\ x \neq f\ a$  **using** *Cons.prem*s **by** *auto*  
**hence**  $f\ x < f\ a$  **using** *Cons.prem*s **by** (*auto simp: sorted-Cons*)  
**thus** *?thesis* **using** *Cons* **by** (*auto simp: sorted-Cons insort-is-Cons*)  
**qed** (*auto simp: sorted-Cons insort-is-Cons*)  
**qed** *simp*

**lemma** *insort-remove1*:  $\llbracket a \in set\ xs; sorted\ xs \rrbracket \implies insort\ a\ (remove1\ a\ xs) = xs$   
**using** *insort-key-remove1*[**where**  $f = \lambda x. x$ ] **by** *simp*

**lemma** *sorted-remdups*[*simp*]:  
*sorted l  $\implies sorted (remdups l)$*   
**by** (*induct l*) (*auto simp: sorted-Cons*)

**lemma** *sorted-distinct-set-unique*:

```

assumes sorted xs distinct xs sorted ys distinct ys set xs = set ys
shows xs = ys
proof –
  from assms have 1: length xs = length ys by (auto dest!: distinct-card)
  from assms show ?thesis
  proof(induct rule:list-induct2[OF 1])
    case 1 show ?case by simp
  next
    case 2 thus ?case by (simp add:sorted-Cons)
      (metis Diff-insert-absorb antisym insertE insert-iff)
  qed
qed

```

```

lemma map-sorted-distinct-set-unique:
  assumes inj-on f (set xs ∪ set ys)
  assumes sorted (map f xs) distinct (map f xs)
    sorted (map f ys) distinct (map f ys)
  assumes set xs = set ys
  shows xs = ys
proof –
  from assms have map f xs = map f ys
    by (simp add: sorted-distinct-set-unique)
  moreover with (inj-on f (set xs ∪ set ys)) show xs = ys
    by (blast intro: map-inj-on)
qed

```

```

lemma finite-sorted-distinct-unique:
shows finite A ⟹ EX! xs. set xs = A & sorted xs & distinct xs
apply(drule finite-distinct-list)
apply clarify
apply(rule-tac a=sort xs in ex1I)
apply (auto simp: sorted-distinct-set-unique)
done

```

```

lemma sorted-take:
  sorted xs ⟹ sorted (take n xs)
proof (induct xs arbitrary: n rule: sorted.induct)
  case 1 show ?case by simp
next
  case 2 show ?case by (cases n) simp-all
next
  case (3 x y xs)
  then have x ≤ y by simp
  show ?case proof (cases n)
    case 0 then show ?thesis by simp
  next
  case (Suc m)
  with 3 have sorted (take m (y # xs)) by simp
  with Suc ⟨x ≤ y⟩ show ?thesis by (cases m) simp-all

```

qed  
qed

**lemma** *sorted-drop*:  
 $sorted\ xs \implies sorted\ (drop\ n\ xs)$   
**proof** (*induct xs arbitrary: n rule: sorted.induct*)  
 case 1 **show** ?case **by** *simp*  
**next**  
 case 2 **show** ?case **by** (*cases n*) *simp-all*  
**next**  
 case 3 **then show** ?case **by** (*cases n*) *simp-all*  
**qed**

**lemma** *sorted-dropWhile*:  $sorted\ xs \implies sorted\ (dropWhile\ P\ xs)$   
**unfolding** *dropWhile-eq-drop* **by** (*rule sorted-drop*)

**lemma** *sorted-takeWhile*:  $sorted\ xs \implies sorted\ (takeWhile\ P\ xs)$   
**apply** (*subst takeWhile-eq-take*) **by** (*rule sorted-take*)

**lemma** *sorted-filter*:  
 $sorted\ (map\ f\ xs) \implies sorted\ (map\ f\ (filter\ P\ xs))$   
**by** (*induct xs*) (*simp-all add: sorted-Cons*)

**lemma** *foldr-max-sorted*:  
**assumes** *sorted (rev xs)*  
**shows**  $foldr\ max\ xs\ y = (if\ xs = []\ then\ y\ else\ max\ (xs\ !\ 0)\ y)$   
**using** *assms* **proof** (*induct xs*)  
 case (*Cons x xs*)  
**moreover** **hence** *sorted (rev xs)* **using** *sorted-append* **by** *auto*  
**ultimately show** ?case  
**by** (*cases xs, auto simp add: sorted-append max-def*)  
**qed** *simp*

**lemma** *filter-equals-takeWhile-sorted-rev*:  
**assumes** *sorted: sorted (rev (map f xs))*  
**shows**  $[x \leftarrow xs.\ t < f\ x] = takeWhile\ (\lambda\ x.\ t < f\ x)\ xs$   
 (*is filter ?P xs = ?tW*)  
**proof** (*rule takeWhile-eq-filter[symmetric]*)  
**let** ?dW = *dropWhile ?P xs*  
**fix** x **assume**  $x \in set\ ?dW$   
**then obtain** i **where**  $i < length\ ?dW$  **and**  $nth\ i:\ x = ?dW\ !\ i$   
**unfolding** *in-set-conv-nth* **by** *auto*  
**hence**  $length\ ?tW + i < length\ (?tW\ @\ ?dW)$   
**unfolding** *length-append* **by** *simp*  
**hence**  $i':\ length\ (map\ f\ ?tW) + i < length\ (map\ f\ xs)$  **by** *simp*  
**have**  $(map\ f\ ?tW\ @\ map\ f\ ?dW)\ !\ (length\ (map\ f\ ?tW) + i) \leq$   
 $(map\ f\ ?tW\ @\ map\ f\ ?dW)\ !\ (length\ (map\ f\ ?tW) + 0)$   
**using** *sorted-rev-nth-mono[OF sorted - i', of length ?tW]*  
**unfolding** *map-append[symmetric]* **by** *simp*

hence  $f x \leq f (?dW ! 0)$   
 unfolding *nth-append-length-plus nth-i*  
 using *i preorder-class.le-less-trans [OF le0 i]* by *simp*  
 also have  $\dots \leq t$   
 using *hd-dropWhile [of ?P xs] le0 [THEN preorder-class.le-less-trans, OF i]*  
 using *hd-conv-nth [of ?dW]* by *simp*  
 finally show  $\neg t < f x$  by *simp*  
 qed

**lemma** *set-insort-insert*:  
 $set (insort-insert x xs) = insert x (set xs)$   
 by (*auto simp add: insort-insert-def set-insort*)

**lemma** *distinct-insort-insert*:  
 assumes *distinct xs*  
 shows *distinct (insort-insert x xs)*  
 using *assms* by (*induct xs*) (*auto simp add: insort-insert-def set-insort*)

**lemma** *sorted-insort-insert*:  
 assumes *sorted xs*  
 shows *sorted (insort-insert x xs)*  
 using *assms* by (*simp add: insort-insert-def sorted-insort*)

**lemma** *filter-insort-key-triv*:  
 $\neg P x \implies filter P (insort-key f x xs) = filter P xs$   
 by (*induct xs*) *simp-all*

**lemma** *filter-insort-key*:  
 $sorted (map f xs) \implies P x \implies filter P (insort-key f x xs) = insort-key f x (filter P xs)$   
 using *assms* by (*induct xs*)  
 (*auto simp add: sorted-Cons, subst insort-is-Cons, auto*)

**lemma** *filter-sort-key*:  
 $filter P (sort-key f xs) = sort-key f (filter P xs)$   
 by (*induct xs*) (*simp-all add: filter-insort-key-triv filter-insort-key*)

**lemma** *sorted-same [simp]*:  
 $sorted [x \leftarrow xs. x = f xs]$   
**proof** (*induct xs arbitrary: f*)  
 case *Nil* then show *?case* by *simp*  
**next**  
 case (*Cons x xs*)  
 then have  $sorted [y \leftarrow xs. y = (\lambda xs. x) xs]$ .  
 moreover from *Cons* have  $sorted [y \leftarrow xs. y = (f \circ Cons x) xs]$ .  
 ultimately show *?case* by (*simp-all add: sorted-Cons*)  
 qed

**lemma** *remove1-insort [simp]*:



```

  remove1 x (insort x xs) = xs
  by (induct xs) simp-all

```

```

end

```

```

lemma sorted-upt[simp]: sorted[i..<j]
by (induct j) (simp-all add:sorted-append)

```

```

lemma sorted-upto[simp]: sorted[i..j]
apply(induct i j rule:upto.induct)
apply(subst upto.simps)
apply(simp add:sorted-Cons)
done

```

#### 43.2.1 transpose on sorted lists

```

lemma sorted-transpose[simp]:
  shows sorted (rev (map length (transpose xs)))
  by (auto simp: sorted-equals-nth-mono rev-nth nth-transpose
    length-filter-conv-card intro: card-mono)

```

```

lemma transpose-max-length:
  foldr (λxs. max (length xs)) (transpose xs) 0 = length [x ← xs. x ≠ []]
  (is ?L = ?R)
proof (cases transpose xs = [])
case False
  have ?L = foldr max (map length (transpose xs)) 0
    by (simp add: foldr-map comp-def)
  also have ... = length (transpose xs ! 0)
    using False sorted-transpose by (simp add: foldr-max-sorted)
  finally show ?thesis
    using False by (simp add: nth-transpose)
next
case True
  hence [x ← xs. x ≠ []] = []
    by (auto intro!: filter-False simp: transpose-empty)
  thus ?thesis by (simp add: transpose-empty True)
qed

```

```

lemma length-transpose-sorted:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows length (transpose xs) = (if xs = [] then 0 else length (xs ! 0))
proof (cases xs = [])
case False
  thus ?thesis
    using foldr-max-sorted[OF sorted] False
    unfolding length-transpose foldr-map comp-def
    by simp

```

qed *simp*

lemma *nth-nth-transpose-sorted*[*simp*]:

fixes *xs* :: 'a list list  
 assumes *sorted*: *sorted* (rev (map length *xs*))  
 and *i*: *i* < length (transpose *xs*)  
 and *j*: *j* < length [ys ← *xs*. *i* < length *ys*]  
 shows *transpose xs* ! *i* ! *j* = *xs* ! *j* ! *i*  
 using *j* *filter-equals-takeWhile-sorted-rev*[*OF sorted*, of *i*]  
   *nth-transpose*[*OF i*] *nth-map*[*OF j*]  
 by (*simp add: takeWhile-nth*)

lemma *transpose-column-length*:

fixes *xs* :: 'a list list  
 assumes *sorted*: *sorted* (rev (map length *xs*)) and *i* < length *xs*  
 shows length (filter (λys. *i* < length *ys*) (transpose *xs*)) = length (*xs* ! *i*)

proof –

have *xs* ≠ [] using ⟨*i* < length *xs*⟩ by auto  
 note *filter-equals-takeWhile-sorted-rev*[*OF sorted*, *simp*]  
 { fix *j* assume *j* ≤ *i*  
   note *sorted-rev-nth-mono*[*OF sorted*, of *j i*, *simplified*, *OF this* ⟨*i* < length *xs*⟩]  
 } note *sortedE* = *this*[*consumes 1*]

have {*j*. *j* < length (transpose *xs*) ∧ *i* < length (transpose *xs* ! *j*)}  
 = {..*length* (*xs* ! *i*)}

proof safe

fix *j*  
 assume *j* < length (transpose *xs*) and *i* < length (transpose *xs* ! *j*)  
 with *this*(2) *nth-transpose*[*OF this*(1)]  
 have *i* < length (takeWhile (λys. *j* < length *ys*) *xs*) by *simp*  
 from *nth-mem*[*OF this*] *takeWhile-nth*[*OF this*]  
 show *j* < length (*xs* ! *i*) by (auto dest: *set-takeWhileD*)

next

fix *j* assume *j* < length (*xs* ! *i*)  
 thus *j* < length (transpose *xs*)  
   using *foldr-max-sorted*[*OF sorted*] ⟨*xs* ≠ []⟩ *sortedE*[*OF le0*]  
   by (auto *simp: length-transpose comp-def foldr-map*)

have *Suc i* ≤ length (takeWhile (λys. *j* < length *ys*) *xs*)  
   using ⟨*i* < length *xs*⟩ ⟨*j* < length (*xs* ! *i*)⟩ *less-Suc-eq-le*  
   by (auto intro!: *length-takeWhile-less-P-nth* dest!: *sortedE*)  
 with *nth-transpose*[*OF* ⟨*j* < length (transpose *xs*)⟩]  
 show *i* < length (transpose *xs* ! *j*) by *simp*

qed

thus ?*thesis* by (*simp add: length-filter-conv-card*)

qed

lemma *transpose-column*:

fixes *xs* :: 'a list list

```

assumes sorted: sorted (rev (map length xs)) and  $i < \text{length } xs$ 
shows map ( $\lambda ys. ys ! i$ ) (filter ( $\lambda ys. i < \text{length } ys$ ) (transpose xs))
  = xs ! i (is ?R = -)
proof (rule nth-equalityI, safe)
  show length: length ?R = length (xs ! i)
    using transpose-column-length[OF assms] by simp

  fix j assume j:  $j < \text{length } ?R$ 
  note * = less-le-trans[OF this, unfolded length-map, OF length-filter-le]
  from j have j-less:  $j < \text{length } (xs ! i)$  using length by simp
  have i-less-tW:  $\text{Suc } i \leq \text{length } (\text{takeWhile } (\lambda ys. \text{Suc } j \leq \text{length } ys) xs)$ 
  proof (rule length-takeWhile-less-P-nth)
    show  $\text{Suc } i \leq \text{length } xs$  using  $\langle i < \text{length } xs \rangle$  by simp
    fix k assume k < Suc i
    hence  $k \leq i$  by auto
    with sorted-rev-nth-mono[OF sorted this]  $\langle i < \text{length } xs \rangle$ 
    have length (xs ! i)  $\leq$  length (xs ! k) by simp
    thus  $\text{Suc } j \leq \text{length } (xs ! k)$  using j-less by simp
  qed
  have i-less-filter:  $i < \text{length } [ys \leftarrow xs \mid j < \text{length } ys]$ 
    unfolding filter-equals-takeWhile-sorted-rev[OF sorted, of j]
    using i-less-tW by (simp-all add: Suc-le-eq)
  from j show ?R ! j = xs ! i ! j
    unfolding filter-equals-takeWhile-sorted-rev[OF sorted-transpose, of i]
    by (simp add: takeWhile-nth nth-nth-transpose-sorted[OF sorted * i-less-filter])
  qed

lemma transpose-transpose:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows transpose (transpose xs) = takeWhile ( $\lambda x. x \neq []$ ) xs (is ?L = ?R)
proof -
  have len: length ?L = length ?R
    unfolding length-transpose transpose-max-length
    using filter-equals-takeWhile-sorted-rev[OF sorted, of 0]
    by simp

  { fix i assume  $i < \text{length } ?R$ 
    with less-le-trans[OF - length-takeWhile-le[of - xs]]
    have  $i < \text{length } xs$  by simp
  } note * = this
  show ?thesis
    by (rule nth-equalityI)
    (simp-all add: len nth-transpose transpose-column[OF sorted] * takeWhile-nth)
  qed

theorem transpose-rectangle:
  assumes xs = []  $\implies n = 0$ 
  assumes rect:  $\bigwedge i. i < \text{length } xs \implies \text{length } (xs ! i) = n$ 

```

```

shows transpose xs = map (λ i. map (λ j. xs ! j ! i) [0..is ?trans = ?map)
proof (rule nth-equalityI)
  have sorted (rev (map length xs))
    by (auto simp: rev-nth rect intro!: sorted-nth-monoI)
  from foldr-max-sorted[OF this] assms
  show len: length ?trans = length ?map
    by (simp-all add: length-transpose foldr-map comp-def)
  moreover
  { fix i assume i < n hence [ys←xs . i < length ys] = xs
    using rect by (auto simp: in-set-conv-nth intro!: filter-True) }
  ultimately show ∀ i < length ?trans. ?trans ! i = ?map ! i
    by (auto simp: nth-transpose intro: nth-equalityI)
qed

```

### 43.2.2 sorted-list-of-set

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

```

context linorder
begin

```

```

definition sorted-list-of-set :: 'a set ⇒ 'a list where
  sorted-list-of-set = Finite-Set.fold insort []

```

```

lemma sorted-list-of-set-empty [simp]:
  sorted-list-of-set {} = []
by (simp add: sorted-list-of-set-def)

```

```

lemma sorted-list-of-set-insert [simp]:
  assumes finite A
  shows sorted-list-of-set (insert x A) = insort x (sorted-list-of-set (A - {x}))
proof -
  interpret fun-left-comm insort by (fact fun-left-comm-insort)
  with assms show ?thesis by (simp add: sorted-list-of-set-def fold-insert-remove)
qed

```

```

lemma sorted-list-of-set [simp]:
  finite A ⇒ set (sorted-list-of-set A) = A ∧ sorted (sorted-list-of-set A)
    ∧ distinct (sorted-list-of-set A)
  by (induct A rule: finite-induct) (simp-all add: set-insort sorted-insort distinct-insort)

```

```

lemma sorted-list-of-set-sort-remdups:
  sorted-list-of-set (set xs) = sort (remdups xs)
proof -
  interpret fun-left-comm insort by (fact fun-left-comm-insort)
  show ?thesis by (simp add: sort-foldl-insort sorted-list-of-set-def fold-set-remdups)
qed

```

```

lemma sorted-list-of-set-remove:
  assumes finite A
  shows sorted-list-of-set (A - {x}) = remove1 x (sorted-list-of-set A)
proof (cases x ∈ A)
  case False with assms have x ∉ set (sorted-list-of-set A) by simp
  with False show ?thesis by (simp add: remove1-idem)
next
  case True then obtain B where A = insert x B by (rule Set.set-insert)
  with assms show ?thesis by simp
qed

end

```

```

lemma sorted-list-of-set-range [simp]:
  sorted-list-of-set {m..n} = [m..n]
  by (rule sorted-distinct-set-unique) simp-all

```

### 43.2.3 lists: the list-forming operator over sets

```

inductive-set
  lists :: 'a set => 'a list set
  for A :: 'a set
where
  Nil [intro!]: []: lists A
  | Cons [intro!,no-atp]: [| a: A; l: lists A|] ==> a#l : lists A

```

```

inductive-cases listsE [elim!,no-atp]: x#l : lists A
inductive-cases listspE [elim!,no-atp]: listsp A (x # l)

```

```

lemma listsp-mono [mono]: A ≤ B ==> listsp A ≤ listsp B
by (rule predicate1I, erule listsp.induct, (blast dest: predicate1D)+)

```

```

lemmas lists-mono = listsp-mono [to-set pred-subset-eq]

```

```

lemma listsp-infI:
  assumes l: listsp A l shows listsp B l ==> listsp (inf A B) l using l
by induct blast+

```

```

lemmas lists-IntI = listsp-infI [to-set]

```

```

lemma listsp-inf-eq [simp]: listsp (inf A B) = inf (listsp A) (listsp B)
proof (rule mono-inf [where f=listsp, THEN order-antisym])
  show mono listsp by (simp add: mono-def listsp-mono)
  show inf (listsp A) (listsp B) ≤ listsp (inf A B) by (blast intro!: listsp-infI predicate1I)
qed

```

```

lemmas listsp-conj-eq [simp] = listsp-inf-eq [simplified inf-fun-eq inf-bool-eq]

```

**lemmas** *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set pred-equals-eq*]

**lemma** *append-in-listsp-conv* [*iff*]:  
 $(\text{listsp } A \ (xs \ @ \ ys)) = (\text{listsp } A \ xs \wedge \text{listsp } A \ ys)$   
**by** (*induct xs*) *auto*

**lemmas** *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

**lemma** *in-listsp-conv-set*:  $(\text{listsp } A \ xs) = (\forall x \in \text{set } xs. A \ x)$   
— eliminate *listsp* in favour of *set*  
**by** (*induct xs*) *auto*

**lemmas** *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

**lemma** *in-listspD* [*dest!,no-atp*]:  $\text{listsp } A \ xs \implies \forall x \in \text{set } xs. A \ x$   
**by** (*rule in-listsp-conv-set* [*THEN iffD1*])

**lemmas** *in-listsD* [*dest!,no-atp*] = *in-listspD* [*to-set*]

**lemma** *in-listspI* [*intro!,no-atp*]:  $\forall x \in \text{set } xs. A \ x \implies \text{listsp } A \ xs$   
**by** (*rule in-listsp-conv-set* [*THEN iffD2*])

**lemmas** *in-listsI* [*intro!,no-atp*] = *in-listspI* [*to-set*]

**lemma** *lists-UNIV* [*simp*]:  $\text{lists } UNIV = UNIV$   
**by** *auto*

#### 43.2.4 Inductive definition for membership

**inductive** *ListMem* :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**

*elem*:  $\text{ListMem } x \ (x \ \# \ xs)$   
| *insert*:  $\text{ListMem } x \ xs \implies \text{ListMem } x \ (y \ \# \ xs)$

**lemma** *ListMem-iff*:  $(\text{ListMem } x \ xs) = (x \in \text{set } xs)$

**apply** (*rule iffI*)

**apply** (*induct set: ListMem*)

**apply** *auto*

**apply** (*induct xs*)

**apply** (*auto intro: ListMem.intros*)

**done**

#### 43.2.5 Lists as Cartesian products

*set-Cons* *A* *Xs*: the set of lists with head drawn from *A* and tail drawn from *Xs*.

**definition**

*set-Cons* :: 'a set  $\Rightarrow$  'a list set  $\Rightarrow$  'a list set **where**

[code del]:  $\text{set-Cons } A \text{ } XS = \{z. \exists x \text{ } xs. z = x \# xs \wedge x \in A \wedge xs \in XS\}$

**lemma** *set-Cons-sing-Nil* [simp]:  $\text{set-Cons } A \ \{\} = (\%x. [x]) 'A$   
**by** (*auto simp add: set-Cons-def*)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

**primrec**

*listset* :: 'a set list  $\Rightarrow$  'a list set **where**  
*listset*  $\{\}$  =  $\{\}$   
| *listset* ( $A \# As$ ) =  $\text{set-Cons } A \ (\text{listset } As)$

### 43.3 Relations on Lists

#### 43.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

**primrec** — The lexicographic ordering for lists of the specified length

*lexn* :: ('a  $\times$  'a) set  $\Rightarrow$  nat  $\Rightarrow$  ('a list  $\times$  'a list) set **where**  
*lexn*  $r \ 0 = \{\}$   
| *lexn*  $r \ (\text{Suc } n) = (\text{prod-fun } (\%(x, xs). x \# xs) \ (\%(x, xs). x \# xs)) ' (r < *lex* > \text{lexn } r \ n)) \ \text{Int}$   
 $\{(xs, ys). \text{length } xs = \text{Suc } n \wedge \text{length } ys = \text{Suc } n\}$

**definition**

*lex* :: ('a  $\times$  'a) set  $\Rightarrow$  ('a list  $\times$  'a list) set **where**  
[code del]:  $\text{lex } r = (\bigcup n. \text{lexn } r \ n)$  — Holds only between lists of the same length

**definition**

*lenlex* :: ('a  $\times$  'a) set  $\Rightarrow$  ('a list  $\times$  'a list) set **where**  
[code del]:  $\text{lenlex } r = \text{inv-image } (\text{less-than } < *lex* > \text{lex } r) \ (\lambda xs. (\text{length } xs, xs))$   
— Compares lists by their length and then lexicographically

**lemma** *wf-lexn*:  $\text{wf } r \implies \text{wf } (\text{lexn } r \ n)$

**apply** (*induct n, simp, simp*)

**apply** (*rule wf-subset*)

**prefer 2 apply** (*rule Int-lower1*)

**apply** (*rule wf-prod-fun-image*)

**prefer 2 apply** (*rule inj-onI, auto*)

**done**

**lemma** *lexn-length*:

$(xs, ys) : \text{lexn } r \ n \implies \text{length } xs = n \wedge \text{length } ys = n$

**by** (*induct n arbitrary: xs ys*) *auto*

**lemma** *wf-lex* [intro!]:  $\text{wf } r \implies \text{wf } (\text{lex } r)$

**apply** (*unfold lex-def*)

**apply** (*rule wf-UN*)

```

apply (blast intro: wf-lexn, clarify)
apply (rename-tac m n)
apply (subgoal-tac m  $\neq$  n)
  prefer 2 apply blast
apply (blast dest: lexn-length not-sym)
done

```

**lemma** *lexn-conv*:

```

lexn r n =
  {(xs,ys). length xs = n  $\wedge$  length ys = n  $\wedge$ 
    ( $\exists$  xys x y xs' ys'. xs = xys @ x # xs'  $\wedge$  ys = xys @ y # ys'  $\wedge$  (x, y):r)}
apply (induct n, simp)
apply (simp add: image-Collect lex-prod-def, safe, blast)
  apply (rule-tac x = ab # xys in exI, simp)
apply (case-tac xys, simp-all, blast)
done

```

**lemma** *lex-conv*:

```

lex r =
  {(xs,ys). length xs = length ys  $\wedge$ 
    ( $\exists$  xys x y xs' ys'. xs = xys @ x # xs'  $\wedge$  ys = xys @ y # ys'  $\wedge$  (x, y):r)}
by (force simp add: lex-def lexn-conv)

```

**lemma** *wf-lenlex* [intro!]: wf r  $\implies$  wf (lenlex r)  
**by** (unfold lenlex-def) blast

**lemma** *lenlex-conv*:

```

lenlex r = {(xs,ys). length xs < length ys |
  length xs = length ys  $\wedge$  (xs, ys) : lex r}
by (simp add: lenlex-def Id-on-def lex-prod-def inv-image-def)

```

**lemma** *Nil-notin-lex* [iff]: ( $\square$ , ys)  $\notin$  lex r  
**by** (simp add: lex-conv)

**lemma** *Nil2-notin-lex* [iff]: (xs,  $\square$ )  $\notin$  lex r  
**by** (simp add: lex-conv)

**lemma** *Cons-in-lex* [simp]:

```

((x # xs, y # ys) : lex r) =
  ((x, y) : r  $\wedge$  length xs = length ys | x = y  $\wedge$  (xs, ys) : lex r)
apply (simp add: lex-conv)
apply (rule iffI)
  prefer 2 apply (blast intro: Cons-eq-appendI, clarify)
apply (case-tac xys, simp, simp)
apply blast
done

```



### 43.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. ”a”  $\leq$  ”ab”  $\leq$  ”b”. This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

**definition**

$lexord :: ('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$  **where**  
 $[code\ del]: lexord\ r = \{(x,y). \exists a\ v. y = x @ a \# v \vee$   
 $(\exists u\ a\ b\ v\ w. (a,b) \in r \wedge x = u @ (a \# v) \wedge y = u @ (b \# w))\}$

**lemma**  $lexord\text{-}Nil\text{-}left[simp]: (\[],y) \in lexord\ r = (\exists a\ x. y = a \# x)$   
**by** ( $unfold\ lexord\text{-}def, induct\text{-}tac\ y, auto$ )

**lemma**  $lexord\text{-}Nil\text{-}right[simp]: (x,\[]) \notin lexord\ r$   
**by** ( $unfold\ lexord\text{-}def, induct\text{-}tac\ x, auto$ )

**lemma**  $lexord\text{-}cons\text{-}cons[simp]:$   
 $((a \# x, b \# y) \in lexord\ r) = ((a,b) \in r \mid (a = b \ \& \ (x,y) \in lexord\ r))$   
**apply** ( $unfold\ lexord\text{-}def, safe, simp\text{-}all$ )  
**apply** ( $case\text{-}tac\ u, simp, simp$ )  
**apply** ( $case\text{-}tac\ u, simp, clarsimp, blast, blast, clarsimp$ )  
**apply** ( $erule\text{-}tac\ x=b \# u\ in\ allE$ )  
**by**  $force$

**lemmas**  $lexord\text{-}simps = lexord\text{-}Nil\text{-}left\ lexord\text{-}Nil\text{-}right\ lexord\text{-}cons\text{-}cons$

**lemma**  $lexord\text{-}append\text{-}rightI: \exists b\ z. y = b \# z \implies (x, x @ y) \in lexord\ r$   
**by** ( $induct\text{-}tac\ x, auto$ )

**lemma**  $lexord\text{-}append\text{-}left\text{-}rightI:$   
 $(a,b) \in r \implies (u @ a \# x, u @ b \# y) \in lexord\ r$   
**by** ( $induct\text{-}tac\ u, auto$ )

**lemma**  $lexord\text{-}append\text{-}leftI: (u,v) \in lexord\ r \implies (x @ u, x @ v) \in lexord\ r$   
**by** ( $induct\ x, auto$ )

**lemma**  $lexord\text{-}append\text{-}leftD:$   
 $\llbracket (x @ u, x @ v) \in lexord\ r; (! a. (a,a) \notin r) \rrbracket \implies (u,v) \in lexord\ r$   
**by** ( $erule\ rev\text{-}mp, induct\text{-}tac\ x, auto$ )

**lemma**  $lexord\text{-}take\text{-}index\text{-}conv:$   
 $((x,y) : lexord\ r) =$   
 $((length\ x < length\ y \wedge take\ (length\ x)\ y = x) \vee$   
 $(\exists i. i < \min(length\ x)(length\ y) \ \& \ take\ i\ x = take\ i\ y \ \& \ (x!i,y!i) \in r))$   
**apply** ( $unfold\ lexord\text{-}def\ Let\text{-}def, clarsimp$ )  
**apply** ( $rule\text{-}tac\ f = (\% a\ b. a \vee b)\ in\ arg\text{-}cong2$ )  
**apply**  $auto$   
**apply** ( $rule\text{-}tac\ x=hd\ (drop\ (length\ x)\ y)\ in\ exI$ )  
**apply** ( $rule\text{-}tac\ x=tl\ (drop\ (length\ x)\ y)\ in\ exI$ )  
**apply** ( $erule\ subst, simp\ add: min\text{-}def$ )

```

apply (rule-tac x=length u in exI, simp)
apply (rule-tac x=take i x in exI)
apply (rule-tac x=x ! i in exI)
apply (rule-tac x=y ! i in exI, safe)
apply (rule-tac x=drop (Suc i) x in exI)
apply (drule sym, simp add: drop-Suc-conv-tl)
apply (rule-tac x=drop (Suc i) y in exI)
by (simp add: drop-Suc-conv-tl)

```

— lexord is extension of partial ordering List.lex

```

lemma lexord-lex: (x,y) ∈ lex r = ((x,y) ∈ lexord r ∧ length x = length y)
apply (rule-tac x = y in spec)
apply (induct-tac x, clarsimp)
by (clarify, case-tac x, simp, force)

```

```

lemma lexord-irreflexive: (! x. (x,x) ∉ r) ⇒ (y,y) ∉ lexord r
by (induct y, auto)

```

**lemma** lexord-trans:

```

  [| (x, y) ∈ lexord r; (y, z) ∈ lexord r; trans r |] ⇒ (x, z) ∈ lexord r
apply (erule rev-mp)+
apply (rule-tac x = x in spec)
apply (rule-tac x = z in spec)
apply (induct-tac y, simp, clarify)
apply (case-tac xa, erule ssubst)
apply (erule allE, erule allE) — avoid simp recursion
apply (case-tac x, simp, simp)
apply (case-tac x, erule allE, erule allE, simp)
apply (erule-tac x = listb in allE)
apply (erule-tac x = lista in allE, simp)
apply (unfold trans-def)
by blast

```

```

lemma lexord-transI: trans r ⇒ trans (lexord r)
by (rule transI, drule lexord-trans, blast)

```

```

lemma lexord-linear: (! a b. (a,b) ∈ r | a = b | (b,a) ∈ r) ⇒ (x,y) : lexord r | x
= y | (y,x) : lexord r
apply (rule-tac x = y in spec)
apply (induct-tac x, rule allI)
apply (case-tac x, simp, simp)
apply (rule allI, case-tac x, simp, simp)
by blast

```

### 43.4 Lexicographic combination of measure functions

These are useful for termination proofs

**definition**

```

measures fs = inv-image (lex less-than) (%a. map (%f. f a) fs)

```

**lemma** *wf-measures*[*recdef-wf*, *simp*]: *wf* (*measures fs*)  
**unfolding** *measures-def*  
**by** *blast*

**lemma** *in-measures*[*simp*]:  
 $(x, y) \in \text{measures } [] = \text{False}$   
 $(x, y) \in \text{measures } (f \# fs)$   
 $= (f x < f y \vee (f x = f y \wedge (x, y) \in \text{measures } fs))$   
**unfolding** *measures-def*  
**by** *auto*

**lemma** *measures-less*:  $f x < f y \implies (x, y) \in \text{measures } (f \# fs)$   
**by** *simp*

**lemma** *measures-lesseq*:  $f x \leq f y \implies (x, y) \in \text{measures } fs \implies (x, y) \in \text{measures } (f \# fs)$   
**by** *auto*

#### 43.4.1 Lifting a Relation on List Elements to the Lists

**inductive-set**  
 $\text{listrel} :: ('a * 'a)\text{set} \Rightarrow ('a \text{ list} * 'a \text{ list})\text{set}$   
**for**  $r :: ('a * 'a)\text{set}$   
**where**  
 $\text{Nil}: ([], []) \in \text{listrel } r$   
 $| \text{Cons}: [(x, y) \in r; (xs, ys) \in \text{listrel } r] \implies (x \# xs, y \# ys) \in \text{listrel } r$

**inductive-cases** *listrel-Nil1* [*elim!*]:  $([], xs) \in \text{listrel } r$   
**inductive-cases** *listrel-Nil2* [*elim!*]:  $(xs, []) \in \text{listrel } r$   
**inductive-cases** *listrel-Cons1* [*elim!*]:  $(y \# ys, xs) \in \text{listrel } r$   
**inductive-cases** *listrel-Cons2* [*elim!*]:  $(xs, y \# ys) \in \text{listrel } r$

**lemma** *listrel-mono*:  $r \subseteq s \implies \text{listrel } r \subseteq \text{listrel } s$   
**apply** *clarify*  
**apply** (*erule listrel.induct*)  
**apply** (*blast intro: listrel.intros*)  
**done**

**lemma** *listrel-subset*:  $r \subseteq A \times A \implies \text{listrel } r \subseteq \text{lists } A \times \text{lists } A$   
**apply** *clarify*  
**apply** (*erule listrel.induct, auto*)  
**done**

**lemma** *listrel-refl-on*:  $\text{refl-on } A \ r \implies \text{refl-on } (\text{lists } A) \ (\text{listrel } r)$   
**apply** (*simp add: refl-on-def listrel-subset Ball-def*)  
**apply** (*rule allI*)  
**apply** (*induct-tac x*)

**apply** (*auto intro: listrel.intros*)  
**done**

**lemma** *listrel-sym*:  $\text{sym } r \implies \text{sym } (\text{listrel } r)$   
**apply** (*auto simp add: sym-def*)  
**apply** (*erule listrel.induct*)  
**apply** (*blast intro: listrel.intros*)  
**done**

**lemma** *listrel-trans*:  $\text{trans } r \implies \text{trans } (\text{listrel } r)$   
**apply** (*simp add: trans-def*)  
**apply** (*intro allI*)  
**apply** (*rule impI*)  
**apply** (*erule listrel.induct*)  
**apply** (*blast intro: listrel.intros*)  
**done**

**theorem** *equiv-listrel*:  $\text{equiv } A \ r \implies \text{equiv } (\text{lists } A) \ (\text{listrel } r)$   
**by** (*simp add: equiv-def listrel-refl-on listrel-sym listrel-trans*)

**lemma** *listrel-Nil* [*simp*]:  $\text{listrel } r \text{ “ } \{\} = \{\}$   
**by** (*blast intro: listrel.intros*)

**lemma** *listrel-Cons*:  
 $\text{listrel } r \text{ “ } \{x \# xs\} = \text{set-Cons } (r \text{ “ } \{x\}) \ (\text{listrel } r \text{ “ } \{xs\})$   
**by** (*auto simp add: set-Cons-def intro: listrel.intros*)

### 43.5 Size function

**lemma** [*measure-function*]:  $\text{is-measure } f \implies \text{is-measure } (\text{list-size } f)$   
**by** (*rule is-measure-trivial*)

**lemma** [*measure-function*]:  $\text{is-measure } f \implies \text{is-measure } (\text{option-size } f)$   
**by** (*rule is-measure-trivial*)

**lemma** *list-size-estimation*[*termination-simp*]:  
 $x \in \text{set } xs \implies y < f \ x \implies y < \text{list-size } f \ xs$   
**by** (*induct xs*) *auto*

**lemma** *list-size-estimation'*[*termination-simp*]:  
 $x \in \text{set } xs \implies y \leq f \ x \implies y \leq \text{list-size } f \ xs$   
**by** (*induct xs*) *auto*

**lemma** *list-size-map*[*simp*]:  $\text{list-size } f \ (\text{map } g \ xs) = \text{list-size } (f \circ g) \ xs$   
**by** (*induct xs*) *auto*

**lemma** *list-size-pointwise*[*termination-simp*]:  
 $(\bigwedge x. x \in \text{set } xs \implies f \ x < g \ x) \implies \text{list-size } f \ xs \leq \text{list-size } g \ xs$   
**by** (*induct xs*) *force*+

## 43.6 Transfer

### definition

*embed-list* :: *nat list*  $\Rightarrow$  *int list*

### where

*embed-list* *l* = *map int l*

### definition

*nat-list* :: *int list*  $\Rightarrow$  *bool*

### where

*nat-list* *l* = *nat-set (set l)*

### definition

*return-list* :: *int list*  $\Rightarrow$  *nat list*

### where

*return-list* *l* = *map nat l*

**lemma** *transfer-nat-int-list-return-embed*: *nat-list l*  $\longrightarrow$

*embed-list (return-list l)* = *l*

**unfolding** *embed-list-def return-list-def nat-list-def nat-set-def*

**apply** (*induct l*)

**apply** *auto*

**done**

**lemma** *transfer-nat-int-list-functions*:

*l* @ *m* = *return-list (embed-list l @ embed-list m)*

[] = *return-list* []

**unfolding** *return-list-def embed-list-def*

**apply** *auto*

**apply** (*induct l, auto*)

**apply** (*induct m, auto*)

**done**

## 43.7 Code generator

### 43.7.1 Setup

**use** *Tools/list-code.ML*

**code-type** *list*

(*SML* - *list*)

(*OCaml* - *list*)

(*Haskell* ![(*-*)])

(*Scala* *List*[(*-*)])

**code-const** *Nil*

(*SML* [])

(*OCaml* [])

(*Haskell* [])

(*Scala* *Nil*)

```

code-instance list :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq list ⇒ 'a list ⇒ bool
  (Haskell infixl 4 ==)

code-reserved SML
  list

code-reserved OCaml
  list

types-code
  list (- list)
attach (term-of) ⟨⟨
  fun term-of-list f T = HOLogic.mk-list T o map f;
  ⟩⟩
attach (test) ⟨⟨
  fun gen-list' aG aT i j = frequency
    [(i, fn () =>
      let
        val (x, t) = aG j;
        val (xs, ts) = gen-list' aG aT (i-1) j
        in (x :: xs, fn () => HOLogic.cons-const aT $ t () $ ts ()) end),
      (1, fn () => ([], fn () => HOLogic.nil-const aT))] ()
  and gen-list aG aT i = gen-list' aG aT i i;
  ⟩⟩

consts-code Cons ((- ::/ -))

setup ⟨⟨
  let
    fun list-codegen thy defs dep thynname b t gr =
      let
        val ts = HOLogic.dest-list t;
        val (-, gr') = Codegen.invoke-tycodegen thy defs dep thynname false
          (fastype-of t) gr;
        val (ps, gr'') = fold-map
          (Codegen.invoke-codegen thy defs dep thynname false) ts gr'
        in SOME (Pretty.list [] ps, gr'') end handle TERM - => NONE;
      in
        fold (List-Code.add-literal-list) [SML, OCaml, Haskell, Scala]
        #> Codegen.add-codegen list-codegen list-codegen
      end
  ⟩⟩

```

**43.7.2 Generation of efficient code**

**definition**  $member :: 'a\ list \Rightarrow 'a \Rightarrow bool$  **where**  
 $mem\text{-}iff\ [code\text{-}post]: member\ xs\ x \longleftrightarrow x \in set\ xs$

**primrec**

$null :: 'a\ list \Rightarrow bool$

**where**

$null\ [] = True$

$| null\ (x\#\!xs) = False$

**primrec**

$list\text{-}inter :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$

**where**

$list\text{-}inter\ []\ bs = []$

$| list\text{-}inter\ (a\#\!as)\ bs =$

$(if\ a \in set\ bs\ then\ a\ \#\ list\text{-}inter\ as\ bs\ else\ list\text{-}inter\ as\ bs)$

**primrec**

$list\text{-}all :: ('a \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow bool)$

**where**

$list\text{-}all\ P\ [] = True$

$| list\text{-}all\ P\ (x\#\!xs) = (P\ x \wedge list\text{-}all\ P\ xs)$

**primrec**

$list\text{-}ex :: ('a \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow bool)$

**where**

$list\text{-}ex\ P\ [] = False$

$| list\text{-}ex\ P\ (x\#\!xs) = (P\ x \vee list\text{-}ex\ P\ xs)$

**primrec**

$filtermap :: ('a \Rightarrow 'b\ option) \Rightarrow 'a\ list \Rightarrow 'b\ list$

**where**

$filtermap\ f\ [] = []$

$| filtermap\ f\ (x\#\!xs) =$

$(case\ f\ x\ of\ None \Rightarrow filtermap\ f\ xs$

$| Some\ y \Rightarrow y\ \#\ filtermap\ f\ xs)$

**primrec**

$map\text{-}filter :: ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list$

**where**

$map\text{-}filter\ f\ P\ [] = []$

$| map\text{-}filter\ f\ P\ (x\#\!xs) =$

$(if\ P\ x\ then\ f\ x\ \#\ map\text{-}filter\ f\ P\ xs\ else\ map\text{-}filter\ f\ P\ xs)$

**primrec**

$length\text{-}unique :: 'a\ list \Rightarrow nat$

**where**

$length\text{-}unique\ [] = 0$

$| length\text{-}unique\ (x\#\!xs) =$

(if  $x \in \text{set } xs$  then  $\text{length-unique } xs$  else  $\text{Suc } (\text{length-unique } xs)$ )

**primrec**

$\text{concat-map} :: ('a \Rightarrow 'b \text{ list}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list}$

**where**

$\text{concat-map } f \ [] = []$

$| \text{concat-map } f (x \# xs) = f x @ \text{concat-map } f xs$

Only use *member* for generating executable code. Otherwise use  $x \in \text{set } xs$  instead — it is much easier to reason about. The same is true for *list-all* and *list-ex*: write  $\forall x \in \text{set } xs$  and  $\exists x \in \text{set } xs$  instead because the HOL quantifiers are already known to the automatic provers. In fact, the declarations in the code subsection make sure that  $\in$ ,  $\forall x \in \text{set } xs$  and  $\exists x \in \text{set } xs$  are implemented efficiently.

Efficient emptiness check is implemented by *null*.

The functions *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

**lemma** *rev-foldl-cons* [code]:

$\text{rev } xs = \text{foldl } (\lambda xs x. x \# xs) [] xs$

**proof** (induct xs)

**case** *Nil* **then show** ?case **by** *simp*

**next**

**case** *Cons*

{

**fix**  $x \ xs \ ys$

**have**  $\text{foldl } (\lambda xs x. x \# xs) \ ys \ xs @ [x]$

$= \text{foldl } (\lambda xs x. x \# xs) \ (ys @ [x]) \ xs$

**by** (induct xs arbitrary: ys) *auto*

}

**note**  $aux = this$

**show** ?case **by** (induct xs) (auto simp add: Cons aux)

**qed**

**lemmas** *in-set-code* [code-unfold] = *mem-iff* [symmetric]

**lemma** *member-simps* [simp, code]:

$\text{member } (x \# xs) \ y \longleftrightarrow x = y \vee \text{member } xs \ y$

$\text{member } [] \ y \longleftrightarrow \text{False}$

**by** (auto simp add: *mem-iff*)

**lemma** *member-set*:

$\text{member} = \text{set}$

**by** (simp add: *expand-fun-eq mem-iff mem-def*)

**abbreviation** (input) *mem* ::  $'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$  (**infixl** *mem* 55) **where**

$x \ \text{mem} \ xs \equiv \text{member } xs \ x$  — for backward compatibility

**lemma** *empty-null*:



$xs = [] \longleftrightarrow \text{null } xs$   
**by** (*cases xs simp-all*)

**lemma** [*code-unfold*]:  
 $\text{eq-class.eq } xs [] \longleftrightarrow \text{null } xs$   
**by** (*simp add: eq empty-null*)

**lemmas** *null-empty* [*code-post*] =  
*empty-null* [*symmetric*]

**lemma** *list-inter-conv*:  
 $\text{set } (\text{list-inter } xs \ ys) = \text{set } xs \cap \text{set } ys$   
**by** (*induct xs auto*)

**lemma** *list-all-iff* [*code-post*]:  
 $\text{list-all } P \ xs \longleftrightarrow (\forall x \in \text{set } xs. P \ x)$   
**by** (*induct xs auto*)

**lemmas** *list-ball-code* [*code-unfold*] = *list-all-iff* [*symmetric*]

**lemma** *list-all-append* [*simp*]:  
 $\text{list-all } P \ (xs \ @ \ ys) \longleftrightarrow (\text{list-all } P \ xs \wedge \text{list-all } P \ ys)$   
**by** (*induct xs auto*)

**lemma** *list-all-rev* [*simp*]:  
 $\text{list-all } P \ (\text{rev } xs) \longleftrightarrow \text{list-all } P \ xs$   
**by** (*simp add: list-all-iff*)

**lemma** *list-all-length*:  
 $\text{list-all } P \ xs \longleftrightarrow (\forall n < \text{length } xs. P \ (xs \ ! \ n))$   
**unfolding** *list-all-iff* **by** (*auto intro: all-nth-imp-all-set*)

**lemma** *list-ex-iff* [*code-post*]:  
 $\text{list-ex } P \ xs \longleftrightarrow (\exists x \in \text{set } xs. P \ x)$   
**by** (*induct xs simp-all*)

**lemmas** *list-bex-code* [*code-unfold*] =  
*list-ex-iff* [*symmetric*]

**lemma** *list-ex-length*:  
 $\text{list-ex } P \ xs \longleftrightarrow (\exists n < \text{length } xs. P \ (xs \ ! \ n))$   
**unfolding** *list-ex-iff* *set-conv-nth* **by** *auto*

**lemma** *filtermap-conv*:  
 $\text{filtermap } f \ xs = \text{map } (\lambda x. \text{the } (f \ x)) \ (\text{filter } (\lambda x. f \ x \neq \text{None}) \ xs)$   
**by** (*induct xs*) (*simp-all split: option.split*)

**lemma** *map-filter-conv* [*simp*]:  
 $\text{map-filter } f \ P \ xs = \text{map } f \ (\text{filter } P \ xs)$

**by** (*induct xs*) *auto*

**lemma** *length-remdups-length-unique* [*code-unfold*]:  
 $\text{length } (\text{remdups } xs) = \text{length-unique } xs$   
**by** (*induct xs*) *simp-all*

**lemma** *concat-map-code* [*code-unfold*]:  
 $\text{concat}(\text{map } f \text{ } xs) = \text{concat-map } f \text{ } xs$   
**by** (*induct xs*) *simp-all*

**declare** *INFI-def* [*code-unfold*]  
**declare** *SUPR-def* [*code-unfold*]

**declare** *set-map* [*symmetric, code-unfold*]

**hide-const** (**open**) *length-unique*

Code for bounded quantification and summation over nats.

**lemma** *atMost-upto* [*code-unfold*]:  
 $\{..n\} = \text{set } [0..<\text{Suc } n]$   
**by** *auto*

**lemma** *atLeast-upt* [*code-unfold*]:  
 $\{..<n\} = \text{set } [0..<n]$   
**by** *auto*

**lemma** *greaterThanLessThan-upt* [*code-unfold*]:  
 $\{n<..  
**by** *auto*$

**lemmas** *atLeastLessThan-upt* [*code-unfold*] = *set-upt* [*symmetric*]

**lemma** *greaterThanAtMost-upt* [*code-unfold*]:  
 $\{n<..  
**by** *auto*$

**lemma** *atLeastAtMost-upt* [*code-unfold*]:  
 $\{n..  
**by** *auto*$

**lemma** *all-nat-less-eq* [*code-unfold*]:  
 $(\forall m<n::\text{nat}. P \text{ } m) \longleftrightarrow (\forall m \in \{0..  
**by** *auto*$

**lemma** *ex-nat-less-eq* [*code-unfold*]:  
 $(\exists m<n::\text{nat}. P \text{ } m) \longleftrightarrow (\exists m \in \{0..  
**by** *auto*$

**lemma** *all-nat-less* [*code-unfold*]:

$(\forall m \leq n :: \text{nat}. P\ m) \longleftrightarrow (\forall m \in \{0..n\}. P\ m)$   
**by** *auto*

**lemma** *ex-nat-less* [code-unfold]:  
 $(\exists m \leq n :: \text{nat}. P\ m) \longleftrightarrow (\exists m \in \{0..n\}. P\ m)$   
**by** *auto*

**lemma** *setsum-set-distinct-conv-listsum*:  
 $\text{distinct } xs \implies \text{setsum } f\ (\text{set } xs) = \text{listsum } (\text{map } f\ xs)$   
**by** (*induct xs*) *simp-all*

**lemma** *setsum-set-upt-conv-listsum* [code-unfold]:  
 $\text{setsum } f\ (\text{set } [m..<n]) = \text{listsum } (\text{map } f\ [m..<n])$   
**by** (*rule setsum-set-distinct-conv-listsum*) *simp*

General equivalence between *listsum* and *setsum*

**lemma** *listsum-setsum-nth*:  
 $\text{listsum } xs = (\sum i = 0 ..< \text{length } xs. xs\ !\ i)$   
**using** *setsum-set-upt-conv-listsum*[*of op ! xs 0 length xs*]  
**by** (*simp add: map-nth*)

Code for summation over ints.

**lemma** *greaterThanLessThan-upto* [code-unfold]:  
 $\{i < ..< j :: \text{int}\} = \text{set } [i+1..j - 1]$   
**by** *auto*

**lemma** *atLeastLessThan-upto* [code-unfold]:  
 $\{i ..< j :: \text{int}\} = \text{set } [i..j - 1]$   
**by** *auto*

**lemma** *greaterThanAtMost-upto* [code-unfold]:  
 $\{i < ..j :: \text{int}\} = \text{set } [i+1..j]$   
**by** *auto*

**lemmas** *atLeastAtMost-upto* [code-unfold] = *set-upto*[*symmetric*]

**lemma** *setsum-set-upto-conv-listsum* [code-unfold]:  
 $\text{setsum } f\ (\text{set } [i..j :: \text{int}]) = \text{listsum } (\text{map } f\ [i..j])$   
**by** (*rule setsum-set-distinct-conv-listsum*) *simp*

Optimized code for  $\forall i \in \{a..b :: \text{int}\}$  and  $\forall n :: \{a..<b :: \text{nat}\}$  and similiarly for  $\exists$ .

**function** *all-from-to-nat* ::  $(\text{nat} \Rightarrow \text{bool}) \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$  **where**  
*all-from-to-nat*  $P\ i\ j =$   
 (*if*  $i < j$  *then* *if*  $P\ i$  *then* *all-from-to-nat*  $P\ (i+1)\ j$  *else* *False*  
   *else* *True*)  
**by** *auto*  
**termination**  
**by** (*relation measure*( $\%(P,i,j). j - i$ )) *auto*

**declare** *all-from-to-nat.simps*[*simp del*]

**lemma** *all-from-to-nat-iff-ball*:

*all-from-to-nat P i j = (ALL n : {i ..< j}. P n)*

**proof**(*induct P i j rule:all-from-to-nat.induct*)

**case** (*1 P i j*)

**let** *?yes = i < j & P i*

**show** *?case*

**proof** (*cases*)

**assume** *?yes*

**hence** *all-from-to-nat P i j = (P i & all-from-to-nat P (i+1) j)*

**by**(*simp add: all-from-to-nat.simps*)

**also have** ... = (*P i & (ALL n : {i+1 ..< j}. P n)*) **using** (*?yes*) *1* **by** *simp*

**also have** ... = (*ALL n : {i ..< j}. P n*) (**is** *?L = ?R*)

**proof**

**assume** *L: ?L*

**show** *?R*

**proof** *clarify*

**fix** *n* **assume** *n: n : {i..<j}*

**show** *P n*

**proof** *cases*

**assume** *n = i* **thus** *P n* **using** *L* **by** *simp*

**next**

**assume** *n ~ i*

**hence** *i+1 <= n* **using** *n* **by** *auto*

**thus** *P n* **using** *L n* **by** *simp*

**qed**

**qed**

**next**

**assume** *R: ?R* **thus** *?L* **using** (*?yes*) *1* **by** *auto*

**qed**

**finally show** *?thesis* .

**next**

**assume** *~?yes* **thus** *?thesis* **by**(*auto simp add: all-from-to-nat.simps*)

**qed**

**qed**

**lemma** *list-all-iff-all-from-to-nat*[*code-unfold*]:

*list-all P [i..<j] = all-from-to-nat P i j*

**by**(*simp add: all-from-to-nat-iff-ball list-all-iff*)

**lemma** *list-ex-iff-not-all-from-to-not-nat*[*code-unfold*]:

*list-ex P [i..<j] = (~all-from-to-nat (%x. ~P x) i j)*

**by**(*simp add: all-from-to-nat-iff-ball list-ex-iff*)

**function** *all-from-to-int* :: (*int*  $\Rightarrow$  *bool*)  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\Rightarrow$  *bool* **where**

```

all-from-to-int P i j =
  (if i <= j then if P i then all-from-to-int P (i+1) j else False
   else True)
by auto
termination
by (relation measure(?(P,i,j). nat(j - i + 1))) auto

declare all-from-to-int.simps[simp del]

lemma all-from-to-int-iff-ball:
  all-from-to-int P i j = (ALL n : {i .. j}. P n)
proof(induct P i j rule:all-from-to-int.induct)
  case (1 P i j)
  let ?yes = i <= j & P i
  show ?case
  proof (cases)
    assume ?yes
    hence all-from-to-int P i j = (P i & all-from-to-int P (i+1) j)
    by(simp add: all-from-to-int.simps)
    also have ... = (P i & (ALL n : {i+1 .. j}. P n)) using ⟨?yes⟩ 1 by simp
    also have ... = (ALL n : {i .. j}. P n) (is ?L = ?R)
    proof
      assume L: ?L
      show ?R
      proof clarify
        fix n assume n: n : {i..j}
        show P n
        proof cases
          assume n = i thus P n using L by simp
        next
          assume n ~ = i
          hence i+1 <= n using n by auto
          thus P n using L n by simp
        qed
      qed
    next
      assume R: ?R thus ?L using ⟨?yes⟩ 1 by auto
    qed
  finally show ?thesis .
next
  assume ~?yes thus ?thesis by(auto simp add: all-from-to-int.simps)
qed
qed

lemma list-all-iff-all-from-to-int[code-unfold]:
  list-all P [i..j] = all-from-to-int P i j
by(simp add: all-from-to-int-iff-ball list-all-iff)

lemma list-ex-iff-not-all-from-to-not-int[code-unfold]:

```

```

    list-ex P [i..j] = ( $\sim$  all-from-to-int ( $\%x.$   $\sim P$  x) i j)
  by(simp add: all-from-to-int-iff-ball list-ex-iff)

end

```

## 44 Random: A HOL random engine

```

theory Random
imports Code-Numeral List
begin

```

```

notation fcomp (infixl o> 60)
notation scomp (infixl o→ 60)

```

### 44.1 Auxiliary functions

```

fun log :: code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral where
  log b i = (if b  $\leq$  1  $\vee$  i < b then 1 else 1 + log b (i div b))

```

```

definition inc-shift :: code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral where
  inc-shift v k = (if v = k then 1 else k + 1)

```

```

definition minus-shift :: code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral
where
  minus-shift r k l = (if k < l then r + k - l else k - l)

```

### 44.2 Random seeds

```

types seed = code-numeral  $\times$  code-numeral

```

```

primrec next :: seed  $\Rightarrow$  code-numeral  $\times$  seed where
  next (v, w) = (let
    k = v div 53668;
    v' = minus-shift 2147483563 ((v mod 53668) * 40014) (k * 12211);
    l = w div 52774;
    w' = minus-shift 2147483399 ((w mod 52774) * 40692) (l * 3791);
    z = minus-shift 2147483562 v' (w' + 1) + 1
  in (z, (v', w')))

```

```

definition split-seed :: seed  $\Rightarrow$  seed  $\times$  seed where
  split-seed s = (let
    (v, w) = s;
    (v', w') = snd (next s);
    v'' = inc-shift 2147483562 v;
    w'' = inc-shift 2147483398 w
  in ((v'', w'), (v', w')))

```

### 44.3 Base selectors

**fun** *iterate* :: *code-numeral*  $\Rightarrow$  (*'b*  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\times$  *'a*)  $\Rightarrow$  *'b*  $\Rightarrow$  *'a*  $\Rightarrow$  *'b*  $\times$  *'a* **where**  
*iterate* *k* *f* *x* = (if *k* = 0 then *Pair* *x* else *f* *x*  $\circ\rightarrow$  *iterate* (*k* - 1) *f*)

**definition** *range* :: *code-numeral*  $\Rightarrow$  *seed*  $\Rightarrow$  *code-numeral*  $\times$  *seed* **where**  
*range* *k* = *iterate* (*log* 2147483561 *k*)  
 ( $\lambda l.$  *next*  $\circ\rightarrow$  ( $\lambda v.$  *Pair* (*v* + *l* \* 2147483561))) 1  
 $\circ\rightarrow$  ( $\lambda v.$  *Pair* (*v* mod *k*))

**lemma** *range*:  
*k* > 0  $\implies$  *fst* (*range* *k* *s*) < *k*  
**by** (*simp* *add*: *range-def scomp-apply split-def del*: *log.simps iterate.simps*)

**definition** *select* :: *'a* *list*  $\Rightarrow$  *seed*  $\Rightarrow$  *'a*  $\times$  *seed* **where**  
*select* *xs* = *range* (*Code-Numeral.of-nat* (*length* *xs*))  
 $\circ\rightarrow$  ( $\lambda k.$  *Pair* (*nth* *xs* (*Code-Numeral.nat-of* *k*)))

**lemma** *select*:  
*assumes* *xs*  $\neq$  []  
*shows* *fst* (*select* *xs* *s*)  $\in$  *set* *xs*

**proof** –  
**from** *assms* **have** *Code-Numeral.of-nat* (*length* *xs*) > 0 **by** *simp*  
**with** *range* **have**  
*fst* (*range* (*Code-Numeral.of-nat* (*length* *xs*)) *s*) < *Code-Numeral.of-nat* (*length* *xs*) **by** *best*  
**then have**  
*Code-Numeral.nat-of* (*fst* (*range* (*Code-Numeral.of-nat* (*length* *xs*)) *s*)) < *length* *xs* **by** *simp*  
**then show** ?thesis  
**by** (*simp* *add*: *scomp-apply split-beta select-def*)  
**qed**

**primrec** *pick* :: (*code-numeral*  $\times$  *'a*) *list*  $\Rightarrow$  *code-numeral*  $\Rightarrow$  *'a* **where**  
*pick* (*x* # *xs*) *i* = (if *i* < *fst* *x* then *snd* *x* else *pick* *xs* (*i* - *fst* *x*))

**lemma** *pick-member*:  
*i* < *listsum* (*map* *fst* *xs*)  $\implies$  *pick* *xs* *i*  $\in$  *set* (*map* *snd* *xs*)  
**by** (*induct* *xs* *arbitrary*: *i*) *simp-all*

**lemma** *pick-drop-zero*:  
*pick* (*filter* ( $\lambda(k, -). k > 0$ ) *xs*) = *pick* *xs*  
**by** (*induct* *xs*) (*auto* *simp* *add*: *expand-fun-eq*)

**lemma** *pick-same*:  
*l* < *length* *xs*  $\implies$  *Random.pick* (*map* (*Pair* 1) *xs*) (*Code-Numeral.of-nat* *l*) =  
*nth* *xs* *l*  
**proof** (*induct* *xs* *arbitrary*: *l*)  
**case** *Nil* **then show** ?case **by** *simp*  
**next**

**case** (*Cons* *x xs*) **then show** ?*case* **by** (*cases* *l*) *simp-all*  
**qed**

**definition** *select-weight* :: (*code-numeral* × 'a) *list* ⇒ *seed* ⇒ 'a × *seed* **where**  
*select-weight xs = range (listsum (map fst xs))*  
*o → (λk. Pair (pick xs k))*

**lemma** *select-weight-member*:  
**assumes**  $0 < \text{listsum } (\text{map } \text{fst } xs)$   
**shows**  $\text{fst } (\text{select-weight } xs \ s) \in \text{set } (\text{map } \text{snd } xs)$   
**proof** –  
**from** *range assms*  
**have**  $\text{fst } (\text{range } (\text{listsum } (\text{map } \text{fst } xs)) \ s) < \text{listsum } (\text{map } \text{fst } xs)$  .  
**with** *pick-member*  
**have**  $\text{pick } xs \ (\text{fst } (\text{range } (\text{listsum } (\text{map } \text{fst } xs)) \ s)) \in \text{set } (\text{map } \text{snd } xs)$  .  
**then show** ?*thesis* **by** (*simp add: select-weight-def scomp-def split-def*)  
**qed**

**lemma** *select-weight-cons-zero*:  
*select-weight ((0, x) # xs) = select-weight xs*  
**by** (*simp add: select-weight-def*)

**lemma** *select-weight-drop-zero*:  
*select-weight (filter (λ(k, -). k > 0) xs) = select-weight xs*  
**proof** –  
**have**  $\text{listsum } (\text{map } \text{fst } [(k, -) \leftarrow xs \ . \ 0 < k]) = \text{listsum } (\text{map } \text{fst } xs)$   
**by** (*induct xs*) *auto*  
**then show** ?*thesis* **by** (*simp only: select-weight-def pick-drop-zero*)  
**qed**

**lemma** *select-weight-select*:  
**assumes**  $xs \neq []$   
**shows**  $\text{select-weight } (\text{map } (\text{Pair } 1) \ xs) = \text{select } xs$   
**proof** –  
**have** *less*:  $\bigwedge s. \text{fst } (\text{range } (\text{Code-Numeral.of-nat } (\text{length } xs)) \ s) < \text{Code-Numeral.of-nat } (\text{length } xs)$   
**using** *assms* **by** (*intro range*) *simp*  
**moreover** **have**  $\text{listsum } (\text{map } \text{fst } (\text{map } (\text{Pair } 1) \ xs)) = \text{Code-Numeral.of-nat } (\text{length } xs)$   
**by** (*induct xs*) *simp-all*  
**ultimately show** ?*thesis*  
**by** (*auto simp add: select-weight-def select-def scomp-def split-def expand-fun-eq pick-same [symmetric]*)  
**qed**

#### 44.4 ML interface

**code-reflect** *Random-Engine*  
**functions** *range select select-weight*



```

ML ⟨⟨
  structure Random-Engine =
  struct

    open Random-Engine;

    type seed = int * int;

    local

      val seed = Unsynchronized.ref
        (let
          val now = Time.toMilliseconds (Time.now ());
          val (q, s1) = IntInf.divMod (now, 2147483562);
          val s2 = q mod 2147483398;
          in (s1 + 1, s2 + 1) end);

      in

        fun next-seed () =
          let
            val (seed1, seed') = @{code split-seed} (! seed)
            val - = seed := seed'
          in
            seed1
          end

        fun run f =
          let
            val (x, seed') = f (! seed);
            val - = seed := seed'
          in x end;

        end;

        end;

        ⟩⟩

    hide-type (open) seed
    hide-const (open) inc-shift minus-shift log next split-seed
      iterate range select pick select-weight
    hide-fact (open) range-def

    no-notation fcomp (infixl o> 60)
    no-notation scomp (infixl o→ 60)

  end

```

## 45 String: Character and string types

```

theory String
imports List
uses
  (Tools/string-syntax.ML)
  (Tools/string-code.ML)
begin

```

### 45.1 Characters

```

datatype nibble =
  Nibble0 | Nibble1 | Nibble2 | Nibble3 | Nibble4 | Nibble5 | Nibble6 | Nibble7
  | Nibble8 | Nibble9 | NibbleA | NibbleB | NibbleC | NibbleD | NibbleE | NibbleF

```

**lemma** *UNIV-nibble*:

$UNIV = \{Nibble0, Nibble1, Nibble2, Nibble3, Nibble4, Nibble5, Nibble6, Nibble7, Nibble8, Nibble9, NibbleA, NibbleB, NibbleC, NibbleD, NibbleE, NibbleF\}$  (is -  
= ?A)

**proof** (rule *UNIV-eq-I*)

**fix**  $x$  **show**  $x \in ?A$  **by** (cases  $x$ ) *simp-all*  
**qed**

**instance** *nibble* :: *finite*

**by** *default* (*simp add: UNIV-nibble*)

**datatype** *char* = *Char nibble nibble*

— Note: canonical order of character encoding coincides with standard term ordering

**lemma** *UNIV-char*:

$UNIV = \text{image } (\text{split } \text{Char}) (UNIV \times UNIV)$

**proof** (rule *UNIV-eq-I*)

**fix**  $x$  **show**  $x \in \text{image } (\text{split } \text{Char}) (UNIV \times UNIV)$  **by** (cases  $x$ ) *auto*  
**qed**

**instance** *char* :: *finite*

**by** *default* (*simp add: UNIV-char*)

**lemma** *size-char* [*code*, *simp*]:

$\text{size } (c::\text{char}) = 0$  **by** (cases  $c$ ) *simp*

**lemma** *char-size* [*code*, *simp*]:

$\text{char-size } (c::\text{char}) = 0$  **by** (cases  $c$ ) *simp*

**primrec** *nibble-pair-of-char* ::  $\text{char} \Rightarrow \text{nibble} \times \text{nibble}$  **where**

$\text{nibble-pair-of-char } (\text{Char } n \ m) = (n, m)$

```

setup ⟨⟨
  let
    val nibbles = map-range (Thm.ctrm-of @{theory} o HOLogic.mk-nibble) 16;
    val thms = map-product
      (fn n => fn m => Drule.instantiate' [] [SOME n, SOME m] @{thm nibble-pair-of-char.simps})
      nibbles nibbles;
  in
    PureThy.note-thmss Thm.definitionK [((Binding.name nibble-pair-of-char.simps,
      []), [(thms, [])])]
    #-> (fn [(-, thms)] => fold-rev Code.add-eqn thms)
  end
  ⟩⟩

```

```

lemma char-case-nibble-pair [code, code-unfold]:
  char-case f = split f o nibble-pair-of-char
  by (simp add: expand-fun-eq split: char.split)

```

```

lemma char-rec-nibble-pair [code, code-unfold]:
  char-rec f = split f o nibble-pair-of-char
  unfolding char-case-nibble-pair [symmetric]
  by (simp add: expand-fun-eq split: char.split)

```

```

syntax
  -Char :: xstr => char (CHR -)

```

## 45.2 Strings

```

types string = char list

```

```

syntax
  -String :: xstr => string (-)

```

```

use Tools/string-syntax.ML
setup String-Syntax.setup

```

```

definition chars :: string where

```

```

  chars = [Char Nibble0 Nibble0, Char Nibble0 Nibble1, Char Nibble0 Nibble2,
    Char Nibble0 Nibble3, Char Nibble0 Nibble4, Char Nibble0 Nibble5,
    Char Nibble0 Nibble6, Char Nibble0 Nibble7, Char Nibble0 Nibble8,
    Char Nibble0 Nibble9, Char Nibble0 NibbleA, Char Nibble0 NibbleB,
    Char Nibble0 NibbleC, Char Nibble0 NibbleD, Char Nibble0 NibbleE,
    Char Nibble0 NibbleF, Char Nibble1 Nibble0, Char Nibble1 Nibble1,
    Char Nibble1 Nibble2, Char Nibble1 Nibble3, Char Nibble1 Nibble4,
    Char Nibble1 Nibble5, Char Nibble1 Nibble6, Char Nibble1 Nibble7,
    Char Nibble1 Nibble8, Char Nibble1 Nibble9, Char Nibble1 NibbleA,
    Char Nibble1 NibbleB, Char Nibble1 NibbleC, Char Nibble1 NibbleD,
    Char Nibble1 NibbleE, Char Nibble1 NibbleF, CHR " ", CHR "!",
    Char Nibble2 Nibble2, CHR "#", CHR "$", CHR "%", CHR "&",

```

Char Nibble2 Nibble7, CHR "(" , CHR ")" , CHR "\*" , CHR "+" , CHR ";" ,  
 CHR "-" , CHR ":" , CHR "/" , CHR "0" , CHR "1" , CHR "2" , CHR "3" ,  
 CHR "4" , CHR "5" , CHR "6" , CHR "7" , CHR "8" , CHR "9" , CHR ":" ,  
 CHR ";" , CHR "<" , CHR "=" , CHR ">" , CHR "?" , CHR "@" , CHR "A" ,  
 CHR "B" , CHR "C" , CHR "D" , CHR "E" , CHR "F" , CHR "G" , CHR "H" ,  
 CHR "I" , CHR "J" , CHR "K" , CHR "L" , CHR "M" , CHR "N" , CHR "O" ,  
 CHR "P" , CHR "Q" , CHR "R" , CHR "S" , CHR "T" , CHR "U" , CHR "V" ,  
 CHR "W" , CHR "X" , CHR "Y" , CHR "Z" , CHR "[" , Char Nibble5 NibbleC ,  
 CHR "]" , CHR "^" , CHR "\_" , Char Nibble6 Nibble0 , CHR "a" , CHR "b" ,  
 CHR "c" , CHR "d" , CHR "e" , CHR "f" , CHR "g" , CHR "h" , CHR "i" ,  
 CHR "j" , CHR "k" , CHR "l" , CHR "m" , CHR "n" , CHR "o" , CHR "p" ,  
 CHR "q" , CHR "r" , CHR "s" , CHR "t" , CHR "u" , CHR "v" , CHR "w" ,  
 CHR "x" , CHR "y" , CHR "z" , CHR "{" , CHR "|" , CHR "}" , CHR "~" ,  
 Char Nibble7 NibbleF , Char Nibble8 Nibble0 , Char Nibble8 Nibble1 ,  
 Char Nibble8 Nibble2 , Char Nibble8 Nibble3 , Char Nibble8 Nibble4 ,  
 Char Nibble8 Nibble5 , Char Nibble8 Nibble6 , Char Nibble8 Nibble7 ,  
 Char Nibble8 Nibble8 , Char Nibble8 Nibble9 , Char Nibble8 NibbleA ,  
 Char Nibble8 NibbleB , Char Nibble8 NibbleC , Char Nibble8 NibbleD ,  
 Char Nibble8 NibbleE , Char Nibble8 NibbleF , Char Nibble9 Nibble0 ,  
 Char Nibble9 Nibble1 , Char Nibble9 Nibble2 , Char Nibble9 Nibble3 ,  
 Char Nibble9 Nibble4 , Char Nibble9 Nibble5 , Char Nibble9 Nibble6 ,  
 Char Nibble9 Nibble7 , Char Nibble9 Nibble8 , Char Nibble9 Nibble9 ,  
 Char Nibble9 NibbleA , Char Nibble9 NibbleB , Char Nibble9 NibbleC ,  
 Char Nibble9 NibbleD , Char Nibble9 NibbleE , Char Nibble9 NibbleF ,  
 Char NibbleA Nibble0 , Char NibbleA Nibble1 , Char NibbleA Nibble2 ,  
 Char NibbleA Nibble3 , Char NibbleA Nibble4 , Char NibbleA Nibble5 ,  
 Char NibbleA Nibble6 , Char NibbleA Nibble7 , Char NibbleA Nibble8 ,  
 Char NibbleA Nibble9 , Char NibbleA NibbleA , Char NibbleA NibbleB ,  
 Char NibbleA NibbleC , Char NibbleA NibbleD , Char NibbleA NibbleE ,  
 Char NibbleA NibbleF , Char NibbleB Nibble0 , Char NibbleB Nibble1 ,  
 Char NibbleB Nibble2 , Char NibbleB Nibble3 , Char NibbleB Nibble4 ,  
 Char NibbleB Nibble5 , Char NibbleB Nibble6 , Char NibbleB Nibble7 ,  
 Char NibbleB Nibble8 , Char NibbleB Nibble9 , Char NibbleB NibbleA ,  
 Char NibbleB NibbleB , Char NibbleB NibbleC , Char NibbleB NibbleD ,  
 Char NibbleB NibbleE , Char NibbleB NibbleF , Char NibbleC Nibble0 ,  
 Char NibbleC Nibble1 , Char NibbleC Nibble2 , Char NibbleC Nibble3 ,  
 Char NibbleC Nibble4 , Char NibbleC Nibble5 , Char NibbleC Nibble6 ,  
 Char NibbleC Nibble7 , Char NibbleC Nibble8 , Char NibbleC Nibble9 ,  
 Char NibbleC NibbleA , Char NibbleC NibbleB , Char NibbleC NibbleC ,  
 Char NibbleC NibbleD , Char NibbleC NibbleE , Char NibbleC NibbleF ,  
 Char NibbleD Nibble0 , Char NibbleD Nibble1 , Char NibbleD Nibble2 ,  
 Char NibbleD Nibble3 , Char NibbleD Nibble4 , Char NibbleD Nibble5 ,  
 Char NibbleD Nibble6 , Char NibbleD Nibble7 , Char NibbleD Nibble8 ,  
 Char NibbleD Nibble9 , Char NibbleD NibbleA , Char NibbleD NibbleB ,  
 Char NibbleD NibbleC , Char NibbleD NibbleD , Char NibbleD NibbleE ,  
 Char NibbleD NibbleF , Char NibbleE Nibble0 , Char NibbleE Nibble1 ,  
 Char NibbleE Nibble2 , Char NibbleE Nibble3 , Char NibbleE Nibble4 ,  
 Char NibbleE Nibble5 , Char NibbleE Nibble6 , Char NibbleE Nibble7 ,  
 Char NibbleE Nibble8 , Char NibbleE Nibble9 , Char NibbleE NibbleA ,

```

Char NibbleE NibbleB, Char NibbleE NibbleC, Char NibbleE NibbleD,
Char NibbleE NibbleE, Char NibbleE NibbleF, Char NibbleF Nibble0,
Char NibbleF Nibble1, Char NibbleF Nibble2, Char NibbleF Nibble3,
Char NibbleF Nibble4, Char NibbleF Nibble5, Char NibbleF Nibble6,
Char NibbleF Nibble7, Char NibbleF Nibble8, Char NibbleF Nibble9,
Char NibbleF NibbleA, Char NibbleF NibbleB, Char NibbleF NibbleC,
Char NibbleF NibbleD, Char NibbleF NibbleE, Char NibbleF NibbleF]

```

### 45.3 Strings as dedicated datatype

**datatype** *literal* = *STR string*

**declare** *literal.cases* [code del] *literal.recs* [code del]

**lemma** [code]: *size (s::literal) = 0*  
**by** (*cases s*) *simp-all*

**lemma** [code]: *literal-size (s::literal) = 0*  
**by** (*cases s*) *simp-all*

### 45.4 Code generator

**use** *Tools/string-code.ML*

**code-reserved** *SML string*

**code-reserved** *OCaml string*

**code-reserved** *Scala string*

**code-type** *literal*  
*(SML string)*  
*(OCaml string)*  
*(Haskell String)*  
*(Scala String)*

**setup** ⟨⟨  
*fold String-Code.add-literal-string [SML, OCaml, Haskell, Scala]*  
 ⟩⟩

**code-instance** *literal* :: *eq*  
*(Haskell −)*

**code-const** *eq-class.eq* :: *literal* ⇒ *literal* ⇒ *bool*  
*(SML !((- : string) = -))*  
*(OCaml !((- : string) = -))*  
*(Haskell infixl 4 ==)*  
*(Scala infixl 5 ==)*

**types-code**

*char (string)*

**attach** (*term-of*) ⟨⟨

```

val term-of-char = HOLogic.mk-char o ord;
>>
attach (test) <<
fun gen-char i =
  let val j = random-range (ord a) (Int.min (ord a + i, ord z))
  in (chr j, fn () => HOLogic.mk-char j) end;
>>

setup <<
let

fun char-codegen thy defs dep thynome b t gr =
  let
    val i = HOLogic.dest-char t;
    val (-, gr') = Codegen.invoke-tycodegen thy defs dep thynome false
      (fastype-of t) gr;
  in SOME (Codegen.str (ML-Syntax.print-string (chr i)), gr')
  end handle TERM - => NONE;

in Codegen.add-codegen char-codegen char-codegen end
>>

hide-type (open) literal

end

```

## 46 Typerep: Reflecting Pure types into HOL

```

theory Typerep
imports Plain String
begin

datatype typerep = Typerep String.literal typerep list

class typerep =
  fixes typerep :: 'a itself => typerep
begin

definition typerep-of :: 'a => typerep where
  [simp]: typerep-of x = typerep TYPE('a)

end

syntax
  -TYPEREP :: type => logic ((1TYPEREP/(1'(-))))

parse-translation <<
let
  fun typerep-tr (*-TYPEREP*) [ty] =

```

```

    Syntax.const @{const-syntax typerep} $
      (Syntax.const @{syntax-const -constrain} $ Syntax.const @{const-syntax
TYPE} $
      (Syntax.const @{type-syntax itself} $ ty))
    | typerep-tr (*-TYPEREP*) ts = raise TERM (typerep-tr, ts);
in [(@{syntax-const -TYPEREP}, typerep-tr)] end
>>

```

**typed-print-translation** <<

```

let
  fun typerep-tr' show-sorts (*typerep*)
    (Type (@{type-name fun}, [Type (@{type-name itself}, [T]), -]))
    (Const (@{const-syntax TYPE}, -) :: ts) =
    Term.list-comb
    (Syntax.const @{syntax-const -TYPEREP} $ Syntax.term-of-typ show-sorts
T, ts)
    | typerep-tr' - T ts = raise Match;
in [(@{const-syntax typerep}, typerep-tr')] end
>>

```

**setup** <<

```

let
  fun add-typerep tyco thy =
    let
      val sorts = replicate (Sign.arity-number thy tyco) @{sort typerep};
      val vs = Name.names Name.context 'a sorts;
      val ty = Type (tyco, map TFree vs);
      val lhs = Const (@{const-name typerep}, Term.itselfT ty --> @{typ typerep})
        $ Free (T, Term.itselfT ty);
      val rhs = @{term Typerep} $ HOLogic.mk-literal tyco
        $ HOLogic.mk-list @{typ typerep} (map (HOLogic.mk-typerep o TFree) vs);
      val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
    in
      thy
      |> Theory-Target.instantiation ([tyco], vs, @{sort typerep})
      |> '(fn lthy => Syntax.check-term lthy eq)
      |-> (fn eq => Specification.definition (NONE, (Attrib.empty-binding, eq)))
      |> snd
      |> Class.prove-instantiation-exit (K (Class.intro-classes-tac []))
    end;

  fun ensure-typerep tyco thy = if not (can (Sorts.mg-domain (Sign.classes-of thy)
tyco) @{sort typerep})
    andalso can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort type}
    then add-typerep tyco thy else thy;

```

in

```

add-typerrep @{type-name fun}
#> Typedef.interpretation ensure-typerrep
#> Code.datatype-interpretation (ensure-typerrep o fst)
#> Code.abstype-interpretation (ensure-typerrep o fst)

end
>>

lemma [code]:
  eq-class.eq (Typerrep tyco1 tys1) (Typerrep tyco2 tys2)  $\longleftrightarrow$  eq-class.eq tyco1 tyco2
   $\wedge$  list-all2 eq-class.eq tys1 tys2
  by (auto simp add: equals-eq [symmetric] list-all2-eq [symmetric])

code-type typerrep
  (Eval Term.typ)

code-const Typerrep
  (Eval Term.Type / (-, -))

code-reserved Eval Term

hide-const (open) typerrep Typerrep

end

```

## 47 Code-Evaluation: Term evaluation using the generic code generator

```

theory Code-Evaluation
imports Plain Typerrep Code-Numeral
begin

```

### 47.1 Term representation

#### 47.1.1 Terms and class *term-of*

```
datatype term = dummy-term
```

```
definition Const :: String.literal  $\Rightarrow$  typerrep  $\Rightarrow$  term where
  Const - - = dummy-term
```

```
definition App :: term  $\Rightarrow$  term  $\Rightarrow$  term where
  App - - = dummy-term
```

```
code-datatype Const App
```

```
class term-of = typerrep +
  fixes term-of :: 'a  $\Rightarrow$  term
```



**lemma** *term-of-anything*:  $\text{term-of } x \equiv t$   
**by** (*rule eq-reflection*) (*cases term-of x, cases t, simp*)

**definition** *valapp* ::  $('a \Rightarrow 'b) \times (\text{unit} \Rightarrow \text{term})$   
 $\Rightarrow 'a \times (\text{unit} \Rightarrow \text{term}) \Rightarrow 'b \times (\text{unit} \Rightarrow \text{term})$  **where**  
*valapp* *f* *x* = (*fst* *f* (*fst* *x*),  $\lambda u. \text{App } (\text{snd } f \ ()) (\text{snd } x \ ())$ )

**lemma** *valapp-code* [*code, code-unfold*]:  
*valapp* (*f*, *tf*) (*x*, *tx*) = (*f* *x*,  $\lambda u. \text{App } (\text{tf } ()) (\text{tx } ())$ )  
**by** (*simp only: valapp-def fst-conv snd-conv*)

#### 47.1.2 *term-of* instances

**instantiation** *fun* :: (*typerep, typerep*) *term-of*  
**begin**

**definition**  
 $\text{term-of } (f :: 'a \Rightarrow 'b) = \text{Const } (\text{STR } \text{"dummy-pattern"}) (\text{Typerep.Typerep } (\text{STR } \text{"fun"}))$   
 $[\text{Typerep.typerep } \text{TYPE('a)}, \text{Typerep.typerep } \text{TYPE('b)}]]$

**instance** ..

**end**

**setup**  $\ll$   
**let**  
*fun* *add-term-of* *tyco* *raw-vs* *thy* =  
**let**  
 $\text{val } vs = \text{map } (\text{fn } (v, -) \Rightarrow (v, @\{\text{sort } \text{typerep}\})) \text{ raw-vs};$   
 $\text{val } ty = \text{Type } (\text{tyco}, \text{map } \text{TFree } vs);$   
 $\text{val } lhs = \text{Const } (@\{\text{const-name } \text{term-of}\}, ty \longrightarrow @\{\text{typ } \text{term}\})$   
 $\quad \$ \text{Free } (x, ty);$   
 $\text{val } rhs = @\{\text{term } \text{undefined} :: \text{term}\};$   
 $\text{val } eq = \text{HOLogic.mk-Trueprop } (\text{HOLogic.mk-eq } (lhs, rhs));$   
 $\text{fun } \text{triv-name-of } t = (\text{fst } o \text{ dest-Free } o \text{ fst } o \text{ strip-comb } o \text{ fst}$   
 $\quad o \text{ HOLogic.dest-eq } o \text{ HOLogic.dest-Trueprop}) t \wedge \text{-triv};$   
**in**  
 $\text{thy}$   
 $|> \text{Theory-Target.instantiation } ([\text{tyco}], vs, @\{\text{sort } \text{term-of}\})$   
 $|> '(\text{fn } lthy \Rightarrow \text{Syntax.check-term } lthy \text{ eq})$   
 $|> (\text{fn } eq \Rightarrow \text{Specification.definition } (\text{NONE}, ((\text{Binding.name } (\text{triv-name-of}$   
 $\text{eq}), []), \text{eq})))$   
 $|> \text{snd}$   
 $|> \text{Class.prove-instantiation-exit } (K (\text{Class.intro-classes-tac } []))$   
**end**;  
*fun* *ensure-term-of* (*tyco*, (*raw-vs*, -)) *thy* =  
**let**

```

    val need-inst = not (can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{\sort
term-of})
    andalso can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{\sort typerep};
    in if need-inst then add-term-of tyco raw-vs thy else thy end;
in
  Code.datatype-interpretation ensure-term-of
  #> Code.abstype-interpretation ensure-term-of
end
>>

setup <<
let
  fun mk-term-of-eq thy ty vs tyco (c, tys) =
    let
      val t = list-comb (Const (c, tys ----> ty),
        map Free (Name.names Name.context a tys));
      val (arg, rhs) =
        pairself (Thm.cterm-of thy o map-types Logic.unvarifyT-global o Logic.varify-global)
          (t, (map-aterms (fn t as Free (v, ty) => HOLogic.mk-term-of ty t | t =>
t) o HOLogic.reflect-term) t)
      val cty = Thm.ctyp-of thy ty;
    in
      @{\thm term-of-anything}
      |> Drule.instantiate' [SOME cty] [SOME arg, SOME rhs]
      |> Thm.varifyT-global
    end;
  fun add-term-of-code tyco raw-vs raw-cs thy =
    let
      val algebra = Sign.classes-of thy;
      val vs = map (fn (v, sort) =>
        (v, curry (Sorts.inter-sort algebra) @{\sort typerep} sort)) raw-vs;
      val ty = Type (tyco, map TFree vs);
      val cs = (map o apsnd o map o map-atyps)
        (fn TFree (v, -) => TFree (v, (the o AList.lookup (op =) vs) v)) raw-cs;
      val const = AxCClass.param-of-inst thy (@{\const-name term-of}, tyco);
      val eqs = map (mk-term-of-eq thy ty vs tyco) cs;
    in
      thy
      |> Code.del-eqns const
      |> fold Code.add-eqn eqs
    end;
  fun ensure-term-of-code (tyco, (raw-vs, cs)) thy =
    let
      val has-inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{\sort
term-of};
      in if has-inst then add-term-of-code tyco raw-vs cs thy else thy end;
    in
      Code.datatype-interpretation ensure-term-of-code
    end
end

```

```

>>

setup <<
  let
    fun mk-term-of-eq thy ty vs tyco abs ty-rep proj =
      let
        val arg = Var ((x, 0), ty);
        val rhs = Abs (y, @{typ term}, HOLogic.reflect-term (Const (abs, ty-rep
--> ty) $ Bound 0)) $
          (HOLogic.mk-term-of ty-rep (Const (proj, ty --> ty-rep) $ arg))
          |> Thm.ctrm-of thy;
        val cty = Thm.ctyp-of thy ty;
      in
        @{thm term-of-anything}
        |> Drule.instantiate' [SOME cty] [SOME (Thm.ctrm-of thy arg), SOME rhs]
        |> Thm.varifyT-global
      end;
    fun add-term-of-code tyco raw-vs abs raw-ty-rep proj thy =
      let
        val algebra = Sign.classes-of thy;
        val vs = map (fn (v, sort) =>
          (v, curry (Sorts.inter-sort algebra) @{sort typerep} sort)) raw-vs;
        val ty = Type (tyco, map TFree vs);
        val ty-rep = map-atyps
          (fn TFree (v, -) => TFree (v, (the o AList.lookup (op =) vs) v)) raw-ty-rep;
        val const = AxClass.param-of-inst thy (@{const-name term-of}, tyco);
        val eq = mk-term-of-eq thy ty vs tyco abs ty-rep proj;
      in
        thy
        |> Code.del-eqns const
        |> Code.add-eqn eq
      end;
    fun ensure-term-of-code (tyco, (raw-vs, ((abs, ty), (proj, -)))) thy =
      let
        val has-inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort
term-of};
        in if has-inst then add-term-of-code tyco raw-vs abs ty proj thy else thy end;
      in
        Code.abstype-interpretation ensure-term-of-code
      end
    >>

```

### 47.1.3 Code generator setup

```

lemmas [code del] = term.recs term.cases term.size
lemma [code, code del]: eq-class.eq (t1::term) t2  $\longleftrightarrow$  eq-class.eq t1 t2 ..

lemma [code, code del]: (term-of :: typerep  $\Rightarrow$  term) = term-of ..
lemma [code, code del]: (term-of :: term  $\Rightarrow$  term) = term-of ..

```

```

lemma [code, code del]: (term-of :: String.literal  $\Rightarrow$  term) = term-of ..
lemma [code, code del]:
  (Code-Evaluation.term-of :: 'a::{type, term-of} Predicate.pred  $\Rightarrow$  Code-Evaluation.term)
= Code-Evaluation.term-of ..
lemma [code, code del]:
  (Code-Evaluation.term-of :: 'a::{type, term-of} Predicate.seq  $\Rightarrow$  Code-Evaluation.term)
= Code-Evaluation.term-of ..

```

```

lemma term-of-char [unfolded typerep-fun-def typerep-char-def typerep-nibble-def,
code]: Code-Evaluation.term-of c =
  (let (n, m) = nibble-pair-of-char c
   in Code-Evaluation.App (Code-Evaluation.App (Code-Evaluation.Const (STR
"String.char.Char")) (TYPEREP(nibble  $\Rightarrow$  nibble  $\Rightarrow$  char)))
  (Code-Evaluation.term-of n)) (Code-Evaluation.term-of m))
by (subst term-of-anything) rule

```

```

code-type term
(Eval Term.term)

```

```

code-const Const and App
(Eval Term.Const/ ((-), (-)) and Term.$/ ((-), (-)))

```

```

code-const term-of :: String.literal  $\Rightarrow$  term
(Eval HOLogic.mk'-literal)

```

```

code-reserved Eval HOLogic

```

#### 47.1.4 Syntax

```

definition termify :: 'a  $\Rightarrow$  term where
[code del]: termify x = dummy-term

```

```

abbreviation valtermify :: 'a  $\Rightarrow$  'a  $\times$  (unit  $\Rightarrow$  term) where
valtermify x  $\equiv$  (x,  $\lambda u.$  termify x)

```

```

setup <<
let
  fun map-default f xs =
    let val ys = map f xs
    in if exists is-some ys
      then SOME (map2 the-default xs ys)
      else NONE
    end;
  fun subst-termify-app (Const (@{const-name termify}, T), [t]) =
    if not (Term.has-abs t)
    then if fold-aterms (fn Const - => I | - => K false) t true
      then SOME (HOLogic.reflect-term t)
      else error Cannot termify expression containing variables
    else error Cannot termify expression containing abstraction

```

```

| subst-termify-app (t, ts) = case map-default subst-termify ts
  of SOME ts' => SOME (list-comb (t, ts'))
  | NONE => NONE
and subst-termify (Abs (v, T, t)) = (case subst-termify t
  of SOME t' => SOME (Abs (v, T, t'))
  | NONE => NONE)
| subst-termify t = subst-termify-app (strip-comb t)
fun check-termify ts ctxt = map-default subst-termify ts
  |> Option.map (rpair ctxt)
in
  Context.theory-map (Syntax.add-term-check 0 termify check-termify)
end;
>>

```

```

locale term-syntax
begin

```

```

notation App (infixl <.> 70)
and valapp (infixl {.} 70)

```

```

end

```

```

interpretation term-syntax .

```

```

no-notation App (infixl <.> 70)
and valapp (infixl {.} 70)

```

## 47.2 Numeric types

**definition** *term-of-num* :: 'a::{semiring-div} ⇒ 'a::{semiring-div} ⇒ *term* **where**  
*term-of-num two* = (λ-. *dummy-term*)

**lemma** (**in** *term-syntax*) *term-of-num-code* [code]:  
*term-of-num two k* = (if *k* = 0 then *termify Int.Pl*  
 else (if *k mod two* = 0  
 then *termify Int.Bit0* <.> *term-of-num two (k div two)*  
 else *termify Int.Bit1* <.> *term-of-num two (k div two)*))  
**by** (auto simp add: *term-of-anything Const-def App-def term-of-num-def Let-def*)

**lemma** (**in** *term-syntax*) *term-of-nat-code* [code]:  
*term-of (n::nat)* = *termify (number-of :: int ⇒ nat)* <.> *term-of-num (2::nat)*  
*n*  
**by** (*simp only: term-of-anything*)

**lemma** (**in** *term-syntax*) *term-of-int-code* [code]:  
*term-of (k::int)* = (if *k* = 0 then *termify (0 :: int)*  
 else if *k* > 0 then *termify (number-of :: int ⇒ int)* <.> *term-of-num (2::int) k*  
 else *termify (uminus :: int ⇒ int)* <.> (*termify (number-of :: int ⇒ int)*  
 <.> *term-of-num (2::int) (- k)*))

**by** (*simp only: term-of-everything*)

**lemma** (**in** *term-syntax*) *term-of-code-numeral-code* [*code*]:

*term-of* (*k*::*code-numeral*) = *termify* (*number-of* :: *int*  $\Rightarrow$  *code-numeral*) *<·>*

*term-of-num* (*2*::*code-numeral*) *k*

**by** (*simp only: term-of-everything*)

### 47.3 Obfuscate

**print-translation**  $\langle\langle$

*let*

*val term* = *Const* (*<TERM>*, *dummyT*);

*fun tr1'* [*-*, *-*] = *term*;

*fun tr2'* [] = *term*;

*in*

[(*@{const-syntax Const}*, *tr1'*),

(*@{const-syntax App}*, *tr1'*),

(*@{const-syntax dummy-term}*, *tr2'*)]

*end*

$\rangle\rangle$

**hide-const** *dummy-term App valapp*

**hide-const** (**open**) *Const termify valtermify term-of term-of-num*

### 47.4 Tracing of generated and evaluated code

**definition** *tracing* :: *String.literal*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*

**where**

[*code del*]: *tracing s x* = *x*

**ML**  $\langle\langle$

*structure Code-Evaluation* =

*struct*

*fun tracing s x* = (*Output.tracing s*; *x*)

*end*

$\rangle\rangle$

**code-const** *tracing* :: *String.literal*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a*

(*Eval Code'-Evaluation.tracing*)

**hide-const** (**open**) *tracing*

**code-reserved** *Eval Code-Evaluation*

### 47.5 Evaluation setup

**ML**  $\langle\langle$

*signature EVAL* =

*sig*

```

    val eval-ref: (unit -> term) option Unsynchronized.ref
    val eval-term: theory -> term -> term
end;

structure Eval : EVAL =
struct

    val eval-ref = Unsynchronized.ref (NONE : (unit -> term) option);

    fun eval-term thy t =
        Code-Eval.eval NONE (Eval.eval-ref, eval-ref) I thy (HOLogic.mk-term-of (fastype-of
        t) t) [];

end;
>>

setup <<
    Value.add-evaluator (code, Eval.eval-term o ProofContext.theory-of)
>>

end

```

## 48 Quickcheck: A simple counterexample generator

```

theory Quickcheck
imports Random Code-Evaluation
uses (Tools/quickcheck-generators.ML)
begin

```

```

notation fcomp (infixl o> 60)
notation scomp (infixl o→ 60)

```

### 48.1 The random class

```

class random = typerep +
    fixes random :: code-numeral ⇒ Random.seed ⇒ ('a × (unit ⇒ term)) × Random.seed

```

### 48.2 Fundamental and numeric types

```

instantiation bool :: random
begin

```

```

definition
    random i = Random.range 2 o→
        (λk. Pair (if k = 0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify
        True))

```

**instance ..**

**end**

**instantiation** *itself* :: (*typerep*) *random*  
**begin**

**definition** *random-itself* :: *code-numeral*  $\Rightarrow$  *Random.seed*  $\Rightarrow$  (*'a itself*  $\times$  (*unit*  $\Rightarrow$  *term*))  $\times$  *Random.seed* **where**  
*random-itself* - = *Pair* (*Code-Evaluation.valtermify* *TYPE('a)*)

**instance ..**

**end**

**instantiation** *char* :: *random*  
**begin**

**definition**  
*random* - = *Random.select chars*  $o \rightarrow$  ( $\lambda c.$  *Pair* (*c*,  $\lambda u.$  *Code-Evaluation.term-of* *c*))

**instance ..**

**end**

**instantiation** *String.literal* :: *random*  
**begin**

**definition**  
*random* - = *Pair* (*STR ""*,  $\lambda u.$  *Code-Evaluation.term-of* (*STR ""*))

**instance ..**

**end**

**instantiation** *nat* :: *random*  
**begin**

**definition** *random-nat* :: *code-numeral*  $\Rightarrow$  *Random.seed*  $\Rightarrow$  (*nat*  $\times$  (*unit*  $\Rightarrow$  *Code-Evaluation.term*))  $\times$  *Random.seed* **where**  
*random-nat* *i* = *Random.range* (*i* + 1)  $o \rightarrow$  ( $\lambda k.$  *Pair* (  
*let* *n* = *Code-Numeral.nat-of* *k*  
in (*n*,  $\lambda -.$  *Code-Evaluation.term-of* *n*)))

**instance ..**

**end**



**instantiation** *int* :: *random*  
**begin**

**definition**  
*random i* = *Random.range* ( $2 * i + 1$ ) *o*→ ( $\lambda k.$  *Pair* (  
*let j* = (*if*  $k \geq i$  *then* *Code-Numeral.int-of* ( $k - i$ ) *else*  $- \text{Code-Numeral.int-of}$   
 $(i - k)$ )  
*in* (*j*,  $\lambda-. \text{Code-Evaluation.term-of } j$ )))

**instance** ..

**end**

### 48.3 Complex generators

Towards  $'a \Rightarrow 'b$

**axiomatization** *random-fun-aux* :: *typerep*  $\Rightarrow$  *typerep*  $\Rightarrow$  ( $'a \Rightarrow 'a \Rightarrow \text{bool}$ )  $\Rightarrow$  ( $'a \Rightarrow \text{term}$ )  
 $\Rightarrow$  (*Random.seed*  $\Rightarrow$  ( $'b \times (\text{unit} \Rightarrow \text{term})$ )  $\times$  *Random.seed*)  $\Rightarrow$  (*Random.seed*  $\Rightarrow$   
*Random.seed*  $\times$  *Random.seed*)  
 $\Rightarrow$  *Random.seed*  $\Rightarrow$  ( $(('a \Rightarrow 'b) \times (\text{unit} \Rightarrow \text{term})) \times \text{Random.seed}$

**definition** *random-fun-lift* :: (*Random.seed*  $\Rightarrow$  ( $'b \times (\text{unit} \Rightarrow \text{term})$ )  $\times$  *Random.seed*)  
 $\Rightarrow$  *Random.seed*  $\Rightarrow$  ( $(('a::\text{term-of} \Rightarrow 'b::\text{typerep}) \times (\text{unit} \Rightarrow \text{term})) \times \text{Random.seed}$   
**where**  
*random-fun-lift f* = *random-fun-aux* *TYPEREP*('a) *TYPEREP*('b) (*op* =) *Code-Evaluation.term-of*  
*f* *Random.split-seed*

**instantiation** *fun* :: ( $\{\text{eq}, \text{term-of}\}$ , *random*) *random*  
**begin**

**definition** *random-fun* :: *code-numeral*  $\Rightarrow$  *Random.seed*  $\Rightarrow$  ( $(('a \Rightarrow 'b) \times (\text{unit} \Rightarrow \text{term})) \times \text{Random.seed}$  **where**  
*random i* = *random-fun-lift* (*random i*)

**instance** ..

**end**

Towards type copies and datatypes

**definition** *collapse* :: ( $'a \Rightarrow ('a \Rightarrow 'b \times 'a) \times 'a$ )  $\Rightarrow$   $'a \Rightarrow 'b \times 'a$  **where**  
*collapse f* = (*f* *o*→ *id*)

**definition** *beyond* :: *code-numeral*  $\Rightarrow$  *code-numeral*  $\Rightarrow$  *code-numeral* **where**  
*beyond k l* = (*if*  $l > k$  *then* *l* *else* 0)

**lemma** *beyond-zero*:

```
beyond k 0 = 0
by (simp add: beyond-def)
```

```
lemma random-aux-rec:
  fixes random-aux :: code-numeral  $\Rightarrow$  'a
  assumes random-aux 0 = rhs 0
    and  $\bigwedge k. \text{random-aux } (\text{Suc-code-numeral } k) = \text{rhs } (\text{Suc-code-numeral } k)$ 
  shows random-aux k = rhs k
  using assms by (rule code-numeral.induct)
```

```
use Tools/quickcheck-generators.ML
setup Quickcheck-Generators.setup
```

#### 48.4 Code setup

**code-const** *random-fun-aux* (*Quickcheck Quickcheck'-Generators.random'-fun*)  
 — With enough criminal energy this can be abused to derive *False*; for this reason we use a distinguished target *Quickcheck* not spoiling the regular trusted code generation

**code-reserved** *Quickcheck Quickcheck-Generators*

```
no-notation fcomp (infixl o> 60)
no-notation scomp (infixl o $\rightarrow$  60)
```

#### 48.5 The Random-Predicate Monad

```
fun iter' ::
  'a itself  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral * code-numeral
 $\Rightarrow$  ('a::random) Predicate.pred
where
  iter' T nrandom sz seed = (if nrandom = 0 then bot-class.bot else
    let ((x, -), seed') = random sz seed
    in Predicate.Seq (%u. Predicate.Insert x (iter' T (nrandom - 1) sz seed')))
```

**definition** *iter* :: code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral \* code-numeral  
 $\Rightarrow$  ('a::random) Predicate.pred

**where**  
*iter* nrandom sz seed = *iter'* (TYPE('a)) nrandom sz seed

```
lemma [code]:
  iter nrandom sz seed = (if nrandom = 0 then bot-class.bot else
    let ((x, -), seed') = random sz seed
    in Predicate.Seq (%u. Predicate.Insert x (iter (nrandom - 1) sz seed'))
unfolding iter-def iter'.simps[of - nrandom] ..
```

**types** 'a randompred = Random.seed  $\Rightarrow$  ('a Predicate.pred  $\times$  Random.seed)

**definition** *empty* :: 'a randompred  
**where** *empty* = Pair (bot-class.bot)

**definition** *single* :: 'a => 'a randompred  
**where** *single* x = Pair (Predicate.single x)

**definition** *bind* :: 'a randompred => ('a => 'b randompred) => 'b randompred  
**where**  
*bind* R f = (λs. let  
(P, s') = R s;  
(s1, s2) = Random.split-seed s'  
in (Predicate.bind P (%a. fst (f a s1)), s2))

**definition** *union* :: 'a randompred => 'a randompred => 'a randompred  
**where**  
*union* R1 R2 = (λs. let  
(P1, s') = R1 s; (P2, s'') = R2 s'  
in (semilattice-sup-class.sup P1 P2, s''))

**definition** *if-randompred* :: bool => unit randompred  
**where**  
*if-randompred* b = (if b then *single* () else empty)

**definition** *iterate-upto* :: (code-numeral => 'a) => code-numeral => code-numeral  
=> 'a randompred  
**where**  
*iterate-upto* f n m = Pair (Code-Numeral.iterate-upto f n m)

**definition** *not-randompred* :: unit randompred => unit randompred  
**where**  
*not-randompred* P = (λs. let  
(P', s') = P s  
in if Predicate.eval P' () then (Orderings.bot, s') else (Predicate.single (), s'))

**definition** *Random* :: (Random.seed => ('a × (unit => term)) × Random.seed) =>  
'a randompred  
**where** *Random* g = scomp g (Pair o (Predicate.single o fst))

**definition** *map* :: ('a => 'b) => ('a randompred => 'b randompred)  
**where** *map* f P = bind P (*single* o f)

**hide-fact** (open) *iter'.simps iter-def empty-def single-def bind-def union-def if-randompred-def  
iterate-upto-def not-randompred-def Random-def map-def*

**hide-type** (open) *randompred*

**hide-const** (open) *random collapse beyond random-fun-aux random-fun-lift  
iter' iter empty single bind union if-randompred iterate-upto not-randompred Ran-  
dom map*

**end**

## 49 Lazy-Sequence: Lazy sequences

```

theory Lazy-Sequence
imports List Code-Numeral
begin

datatype 'a lazy-sequence = Empty | Insert 'a 'a lazy-sequence

definition Lazy-Sequence :: (unit => ('a * 'a lazy-sequence) option) => 'a lazy-sequence
where
  Lazy-Sequence f = (case f () of None => Empty | Some (x, xq) => Insert x xq)

code-datatype Lazy-Sequence

primrec yield :: 'a lazy-sequence => ('a * 'a lazy-sequence) option
where
  yield Empty = None
| yield (Insert x xq) = Some (x, xq)

lemma [simp]: yield xq = Some (x, xq') ==> size xq' < size xq
by (cases xq) auto

lemma yield-Seq [code]:
  yield (Lazy-Sequence f) = f ()
unfolding Lazy-Sequence-def by (cases f ()) auto

lemma Seq-yield:
  Lazy-Sequence (%u. yield f) = f
unfolding Lazy-Sequence-def by (cases f) auto

lemma lazy-sequence-size-code [code]:
  lazy-sequence-size s xq = (case yield xq of None => 0 | Some (x, xq') => s x +
    lazy-sequence-size s xq' + 1)
by (cases xq) auto

lemma size-code [code]:
  size xq = (case yield xq of None => 0 | Some (x, xq') => size xq' + 1)
by (cases xq) auto

lemma [code]: eq-class.eq xq yq = (case (yield xq, yield yq) of
  (None, None) => True | (Some (x, xq'), Some (y, yq')) => (HOL.eq x y) ∧
  (eq-class.eq xq yq) | - => False)
apply (cases xq) apply (cases yq) apply (auto simp add: eq-class.eq-equals)
apply (cases yq) apply (auto simp add: eq-class.eq-equals) done

lemma seq-case [code]:
  lazy-sequence-case f g xq = (case (yield xq) of None => f | Some (x, xq') => g
    x xq')
by (cases xq) auto

```

**lemma** *[code]*: *lazy-sequence-rec*  $f\ g\ xq = (\text{case } (\text{yield } xq) \text{ of } \text{None} \Rightarrow f \mid \text{Some } (x, xq') \Rightarrow g\ x\ xq' (\text{lazy-sequence-rec } f\ g\ xq'))$   
**by** (*cases*  $xq$ ) *auto*

**definition** *empty* :: *'a lazy-sequence*  
**where**  
*[code]*: *empty* = *Lazy-Sequence* (%*u.* *None*)

**definition** *single* :: *'a => 'a lazy-sequence*  
**where**  
*[code]*: *single*  $x = \text{Lazy-Sequence } (\%u. \text{Some } (x, \text{empty}))$

**primrec** *append* :: *'a lazy-sequence => 'a lazy-sequence => 'a lazy-sequence*  
**where**  
*append* *Empty*  $yq = yq$   
 $\mid \text{append } (\text{Insert } x\ xq)\ yq = \text{Insert } x\ (\text{append } xq\ yq)$

**lemma** *[code]*:  
*append*  $xq\ yq = \text{Lazy-Sequence } (\%u. \text{case } \text{yield } xq \text{ of } \text{None} \Rightarrow \text{yield } yq \mid \text{Some } (x, xq') \Rightarrow \text{Some } (x, \text{append } xq'\ yq))$   
**unfolding** *Lazy-Sequence-def*  
**apply** (*cases*  $xq$ )  
**apply** *auto*  
**apply** (*cases*  $yq$ )  
**apply** *auto*  
**done**

**primrec** *flat* :: *'a lazy-sequence lazy-sequence => 'a lazy-sequence*  
**where**  
*flat* *Empty* = *Empty*  
 $\mid \text{flat } (\text{Insert } xq\ xqq) = \text{append } xq\ (\text{flat } xqq)$

**lemma** *[code]*:  
*flat*  $xqq = \text{Lazy-Sequence } (\%u. \text{case } \text{yield } xqq \text{ of } \text{None} \Rightarrow \text{None} \mid \text{Some } (xq, xqq') \Rightarrow \text{yield } (\text{append } xq\ (\text{flat } xqq')))$   
**apply** (*cases*  $xqq$ )  
**apply** (*auto simp add: Seq-yield*)  
**unfolding** *Lazy-Sequence-def*  
**by** *auto*

**primrec** *map* :: (*'a => 'b*) => *'a lazy-sequence => 'b lazy-sequence*  
**where**  
*map*  $f\ \text{Empty} = \text{Empty}$   
 $\mid \text{map } f\ (\text{Insert } x\ xq) = \text{Insert } (f\ x)\ (\text{map } f\ xq)$

**lemma** *[code]*:

```

    map f xq = Lazy-Sequence (%u. Option.map (%(x, xq'). (f x, map f xq')) (yield
xq))
  apply (cases xq)
  apply (auto simp add: Seq-yield)
  unfolding Lazy-Sequence-def
  apply auto
  done

```

**definition** *bind* :: 'a lazy-sequence => ('a => 'b lazy-sequence) => 'b lazy-sequence  
**where**  
 [code]: *bind* xq f = flat (map f xq)

**definition** *if-seq* :: bool => unit lazy-sequence  
**where**  
*if-seq* b = (if b then single () else empty)

**function** *iterate-upto* :: (code-numeral => 'a) => code-numeral => code-numeral  
=> 'a Lazy-Sequence.lazy-sequence  
**where**  
*iterate-upto* f n m = Lazy-Sequence.Lazy-Sequence (%u. if n > m then None else  
Some (f n, *iterate-upto* f (n + 1) m))  
**by** pat-completeness auto

**termination by** (relation measure (%(f, n, m). Code-Numeral.nat-of (m + 1 -  
n))) auto

**definition** *not-seq* :: unit lazy-sequence => unit lazy-sequence  
**where**  
*not-seq* xq = (case yield xq of None => single () | Some ((), xq) => empty)

## 49.1 Code setup

**fun** *anamorph* :: ('a => ('b × 'a) option) => code-numeral => 'a => 'b list × 'a  
**where**  
*anamorph* f k x = (if k = 0 then ([], x)  
else case f x of None => ([], x) | Some (v, y) =>  
let (vs, z) = *anamorph* f (k - 1) y  
in (v # vs, z))

**definition** *yieldn* :: code-numeral => 'a lazy-sequence => 'a list × 'a lazy-sequence  
**where**  
*yieldn* = *anamorph* yield

**code-reflect** *Lazy-Sequence*  
**datatypes** *lazy-sequence* = *Lazy-Sequence*  
**functions** *map* *yield* *yieldn*

## 49.2 With Hit Bound Value

assuming in negative context

**types** *'a hit-bound-lazy-sequence* = *'a option lazy-sequence*

**definition** *hit-bound* :: *'a hit-bound-lazy-sequence*

**where**

[code]: *hit-bound* = *Lazy-Sequence* (%u. *Some (None, empty)*)

**definition** *hb-single* :: *'a => 'a hit-bound-lazy-sequence*

**where**

[code]: *hb-single* *x* = *Lazy-Sequence* (%u. *Some (Some x, empty)*)

**primrec** *hb-flat* :: *'a hit-bound-lazy-sequence hit-bound-lazy-sequence => 'a hit-bound-lazy-sequence*

**where**

*hb-flat Empty* = *Empty*  
| *hb-flat (Insert xq xqq)* = *append* (*case xq of None => hit-bound* | *Some xq => xq*) (*hb-flat xqq*)

**lemma** [code]:

*hb-flat xqq* = *Lazy-Sequence* (%u. *case yield xqq of*  
*None => None*  
*| Some (xq, xqq') => yield (append (case xq of None => hit-bound* | *Some xq*  
*=> xq) (hb-flat xqq'))*)

**apply** (*cases xqq*)

**apply** (*auto simp add: Seq-yield*)

**unfolding** *Lazy-Sequence-def*

**by** *auto*

**primrec** *hb-map* :: *('a => 'b) => 'a hit-bound-lazy-sequence => 'b hit-bound-lazy-sequence*

**where**

*hb-map f Empty* = *Empty*  
| *hb-map f (Insert x xq)* = *Insert (Option.map f x) (hb-map f xq)*

**lemma** [code]:

*hb-map f xq* = *Lazy-Sequence* (%u. *Option.map* (%(*x, xq'*). (*Option.map f x*,  
*hb-map f xq'*)) (*yield xq*))

**apply** (*cases xq*)

**apply** (*auto simp add: Seq-yield*)

**unfolding** *Lazy-Sequence-def*

**apply** *auto*

**done**

**definition** *hb-bind* :: *'a hit-bound-lazy-sequence => ('a => 'b hit-bound-lazy-sequence)*  
*=> 'b hit-bound-lazy-sequence*

**where**

[code]: *hb-bind* *xq f* = *hb-flat (hb-map f xq)*

**definition** *hb-if-seq* :: *bool* => *unit hit-bound-lazy-sequence*

**where**

*hb-if-seq* *b* = (if *b* then *hb-single* () else *empty*)

**definition** *hb-not-seq* :: *unit hit-bound-lazy-sequence* => *unit lazy-sequence*

**where**

*hb-not-seq* *xq* = (case *yield* *xq* of *None* => *single* () | *Some* (*x*, *xq*) => *empty*)

**hide-type** (open) *lazy-sequence*

**hide-const** (open) *Empty Insert Lazy-Sequence yield empty single append flat map*

*bind if-seq iterate-upto not-seq*

**hide-fact** *yield.simps empty-def single-def append.simps flat.simps map.simps bind-def*

*iterate-upto.simps if-seq-def not-seq-def*

**end**

## 50 DSequence: Depth-Limited Sequences with failure element

**theory** *DSequence*

**imports** *Lazy-Sequence Code-Numeral*

**begin**

**types** '*a* *dseq* = *code-numeral* => *bool* => '*a* *Lazy-Sequence.lazy-sequence option*

**definition** *empty* :: '*a* *dseq*

**where**

*empty* = (%*i* *pol*. *Some Lazy-Sequence.empty*)

**definition** *single* :: '*a* => '*a* *dseq*

**where**

*single* *x* = (%*i* *pol*. *Some (Lazy-Sequence.single x)*)

**fun** *eval* :: '*a* *dseq* => *code-numeral* => *bool* => '*a* *Lazy-Sequence.lazy-sequence option*

**where**

*eval* *f* *i* *pol* = *f* *i* *pol*

**definition** *yield* :: '*a* *dseq* => *code-numeral* => *bool* => ('*a* \* '*a* *dseq*) *option*

**where**

*yield* *dseq* *i* *pol* = (case *eval* *dseq* *i* *pol* of  
*None* => *None*  
| *Some* *s* => *Option.map* (*apsnd* (%*r* *i* *pol*. *Some* *r*)) (*Lazy-Sequence.yield* *s*))

**fun** *map-seq* :: ('*a* => '*b* *dseq*) => '*a* *Lazy-Sequence.lazy-sequence* => '*b* *dseq*

**where**

*map-seq* *f* *xq* *i* *pol* = (case *Lazy-Sequence.yield* *xq* of



```

    None => Some Lazy-Sequence.empty
  | Some (x, xq') => (case eval (f x) i pol of
    None => None
  | Some yq => (case map-seq f xq' i pol of
    None => None
  | Some zq => Some (Lazy-Sequence.append yq zq))))

```

**definition** *bind* :: 'a dseq => ('a => 'b dseq) => 'b dseq  
**where**

```

    bind x f = (%i pol.
      if i = 0 then
        (if pol then Some Lazy-Sequence.empty else None)
      else
        (case x (i - 1) pol of
          None => None
        | Some xq => map-seq f xq i pol))

```

**definition** *union* :: 'a dseq => 'a dseq => 'a dseq  
**where**

```

    union x y = (%i pol. case (x i pol, y i pol) of
      (Some xq, Some yq) => Some (Lazy-Sequence.append xq yq)
    | - => None)

```

**definition** *if-seq* :: bool => unit dseq  
**where**

```

    if-seq b = (if b then single () else empty)

```

**definition** *not-seq* :: unit dseq => unit dseq  
**where**

```

    not-seq x = (%i pol. case x i (¬pol) of
      None => Some Lazy-Sequence.empty
    | Some xq => (case Lazy-Sequence.yield xq of
      None => Some (Lazy-Sequence.single ())
    | Some - => Some (Lazy-Sequence.empty)))

```

**definition** *map* :: ('a => 'b) => 'a dseq => 'b dseq  
**where**

```

    map f g = (%i pol. case g i pol of
      None => None
    | Some xq => Some (Lazy-Sequence.map f xq))

```

**ML** <<

*signature* DSEQUENCE =

*sig*

```

    type 'a dseq = int -> bool -> 'a Lazy-Sequence.lazy-sequence option
    val yield : 'a dseq -> int -> bool -> ('a * 'a dseq) option
    val yieldn : int -> 'a dseq -> int -> bool -> 'a list * 'a dseq
    val map : ('a -> 'b) -> 'a dseq -> 'b dseq

```

*end*;

```

structure DSequence : DSEQUENCE =
struct

type 'a dseq = int -> bool -> 'a Lazy-Sequence.lazy-sequence option

val yield = @{code yield}
fun yieldn n s d pol = case s d pol of
  NONE => ([], fn d => fn pol => NONE)
  | SOME s' => let val (xs, s'') = Lazy-Sequence.yieldn n s in (xs, fn d => fn pol
=> SOME s'') end

val map = @{code map}

end;
>>

code-reserved Eval DSequence

hide-const (open) empty single eval map-seq bind union if-seq not-seq map
hide-fact (open) empty-def single-def eval.simps map-seq.simps bind-def union-def
if-seq-def not-seq-def map-def

end

theory Random-Sequence
imports Quickcheck DSequence
begin

types 'a random-dseq = code-numeral => code-numeral => Random.seed => ('a
DSequence.dseq × Random.seed)

definition empty :: 'a random-dseq
where
  empty = (%nrandom size. Pair (DSequence.empty))

definition single :: 'a => 'a random-dseq
where
  single x = (%nrandom size. Pair (DSequence.single x))

definition bind :: 'a random-dseq => ('a => 'b random-dseq) => 'b random-dseq
where
  bind R f = (λnrandom size s. let
    (P, s') = R nrandom size s;
    (s1, s2) = Random.split-seed s'
  in (DSequence.bind P (%a. fst (f a nrandom size s1)), s2))

```

**definition** *union* :: 'a random-dseq => 'a random-dseq => 'a random-dseq

**where**

*union* *R1 R2* = ( $\lambda n$ random size *s*. let  
     (*S1*, *s'*) = *R1* nrandom size *s*; (*S2*, *s''*) = *R2* nrandom size *s'*  
   in (*DSequence.union* *S1 S2*, *s''*))

**definition** *if-random-dseq* :: bool => unit random-dseq

**where**

*if-random-dseq* *b* = (if *b* then single () else empty)

**definition** *not-random-dseq* :: unit random-dseq => unit random-dseq

**where**

*not-random-dseq* *R* = ( $\lambda n$ random size *s*. let  
     (*S*, *s'*) = *R* nrandom size *s*  
   in (*DSequence.not-seq* *S*, *s'*))

**fun** *Random* :: (code-numeral => *Random.seed* => (('a × (unit => term)) × *Random.seed*)) => 'a random-dseq

**where**

*Random* *g* nrandom = (%size. if nrandom <= 0 then (Pair *DSequence.empty*)  
 else  
     (scomp (*g* size) (%r. scomp (*Random* *g* (nrandom - 1) size) (%rs. Pair  
       (*DSequence.union* (*DSequence.single* (fst *r*) rs))))))

**definition** *map* :: ('a => 'b) => 'a random-dseq => 'b random-dseq

**where**

*map* *f* *P* = bind *P* (single o *f*)

**hide-const** (**open**) empty single bind union if-random-dseq not-random-dseq *Random* map

**hide-type** *DSequence.dseq* random-dseq

**hide-fact** (**open**) empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def  
*Random.simps* map-def

**end**

## 51 New-DSequence: Depth-Limited Sequences with failure element

**theory** *New-DSequence*

**imports** *Random-Sequence*

**begin**

### 51.1 Positive Depth-Limited Sequence

**types**  $'a \text{ pos-dseq} = \text{code-numeral} \Rightarrow 'a \text{ Lazy-Sequence.lazy-sequence}$

**definition**  $\text{pos-empty} :: 'a \text{ pos-dseq}$

**where**

$\text{pos-empty} = (\%i. \text{Lazy-Sequence.empty})$

**definition**  $\text{pos-single} :: 'a \Rightarrow 'a \text{ pos-dseq}$

**where**

$\text{pos-single } x = (\%i. \text{Lazy-Sequence.single } x)$

**definition**  $\text{pos-bind} :: 'a \text{ pos-dseq} \Rightarrow ('a \Rightarrow 'b \text{ pos-dseq}) \Rightarrow 'b \text{ pos-dseq}$

**where**

$\text{pos-bind } x f = (\%i. \text{if } i = 0 \text{ then } \text{Lazy-Sequence.empty} \\ \text{else } \text{Lazy-Sequence.bind } (x (i - 1)) (\%a. f a i))$

**definition**  $\text{pos-union} :: 'a \text{ pos-dseq} \Rightarrow 'a \text{ pos-dseq} \Rightarrow 'a \text{ pos-dseq}$

**where**

$\text{pos-union } xq yq = (\%i. \text{Lazy-Sequence.append } (xq i) (yq i))$

**definition**  $\text{pos-if-seq} :: \text{bool} \Rightarrow \text{unit pos-dseq}$

**where**

$\text{pos-if-seq } b = (\text{if } b \text{ then } \text{pos-single } () \text{ else } \text{pos-empty})$

**definition**  $\text{pos-iterate-upto} :: (\text{code-numeral} \Rightarrow 'a) \Rightarrow \text{code-numeral} \Rightarrow \text{code-numeral} \Rightarrow 'a \text{ pos-dseq}$

**where**

$\text{pos-iterate-upto } f n m = (\%i. \text{Lazy-Sequence.iterate-upto } f n m)$

**definition**  $\text{pos-map} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ pos-dseq} \Rightarrow 'b \text{ pos-dseq}$

**where**

$\text{pos-map } f xq = (\%i. \text{Lazy-Sequence.map } f (xq i))$

### 51.2 Negative Depth-Limited Sequence

**types**  $'a \text{ neg-dseq} = \text{code-numeral} \Rightarrow 'a \text{ Lazy-Sequence.hit-bound-lazy-sequence}$

**definition**  $\text{neg-empty} :: 'a \text{ neg-dseq}$

**where**

$\text{neg-empty} = (\%i. \text{Lazy-Sequence.empty})$

**definition**  $\text{neg-single} :: 'a \Rightarrow 'a \text{ neg-dseq}$

**where**

$\text{neg-single } x = (\%i. \text{Lazy-Sequence.hb-single } x)$

**definition**  $\text{neg-bind} :: 'a \text{ neg-dseq} \Rightarrow ('a \Rightarrow 'b \text{ neg-dseq}) \Rightarrow 'b \text{ neg-dseq}$

**where**

```
neg-bind x f = (%i.
  if i = 0 then
    Lazy-Sequence.hit-bound
  else
    hb-bind (x (i - 1)) (%a. f a i))
```

**definition** *neg-union* :: 'a neg-dseq => 'a neg-dseq => 'a neg-dseq

**where**

```
neg-union x y = (%i. Lazy-Sequence.append (x i) (y i))
```

**definition** *neg-if-seq* :: bool => unit neg-dseq

**where**

```
neg-if-seq b = (if b then neg-single () else neg-empty)
```

**definition** *neg-iterate-upto*

**where**

```
neg-iterate-upto f n m = (%i. Lazy-Sequence.iterate-upto (%i. Some (f i)) n m)
```

**definition** *neg-map* :: ('a => 'b) => 'a neg-dseq => 'b neg-dseq

**where**

```
neg-map f xq = (%i. Lazy-Sequence.hb-map f (xq i))
```

### 51.3 Negation

**definition** *pos-not-seq* :: unit neg-dseq => unit pos-dseq

**where**

```
pos-not-seq xq = (%i. Lazy-Sequence.hb-not-seq (xq i))
```

**definition** *neg-not-seq* :: unit pos-dseq => unit neg-dseq

**where**

```
neg-not-seq x = (%i. case Lazy-Sequence.yield (x i) of
  None => Lazy-Sequence.hb-single ()
| Some ((), xq) => Lazy-Sequence.empty)
```

**hide-type** (open) *pos-dseq neg-dseq*

**hide-const** (open)

```
pos-empty pos-single pos-bind pos-union pos-if-seq pos-iterate-upto pos-not-seq
pos-map
```

```
neg-empty neg-single neg-bind neg-union neg-if-seq neg-iterate-upto neg-not-seq
neg-map
```

**hide-fact** (open)

```
pos-empty-def pos-single-def pos-bind-def pos-union-def pos-if-seq-def pos-iterate-upto-def
pos-not-seq-def pos-map-def
```

```
neg-empty-def neg-single-def neg-bind-def neg-union-def neg-if-seq-def neg-iterate-upto-def
neg-not-seq-def neg-map-def
```

**end**

```

theory New-Random-Sequence
imports Quickcheck New-DSequence
begin

```

```

types 'a pos-random-dseq = code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  Random.seed  $\Rightarrow$  'a
New-DSequence.pos-dseq
types 'a neg-random-dseq = code-numeral  $\Rightarrow$  code-numeral  $\Rightarrow$  Random.seed  $\Rightarrow$  'a
New-DSequence.neg-dseq

```

```

definition pos-empty :: 'a pos-random-dseq
where
  pos-empty = (%nrandom size seed. New-DSequence.pos-empty)

```

```

definition pos-single :: 'a  $\Rightarrow$  'a pos-random-dseq
where
  pos-single x = (%nrandom size seed. New-DSequence.pos-single x)

```

```

definition pos-bind :: 'a pos-random-dseq  $\Rightarrow$  ('a  $\Rightarrow$  'b pos-random-dseq)  $\Rightarrow$  'b
pos-random-dseq
where
  pos-bind R f = ( $\lambda$ nrandom size seed. New-DSequence.pos-bind (R nrandom size
seed) (%a. f a nrandom size seed))

```

```

definition pos-union :: 'a pos-random-dseq  $\Rightarrow$  'a pos-random-dseq  $\Rightarrow$  'a pos-random-dseq
where
  pos-union R1 R2 = ( $\lambda$ nrandom size seed. New-DSequence.pos-union (R1 nrandom
size seed) (R2 nrandom size seed))

```

```

definition pos-if-random-dseq :: bool  $\Rightarrow$  unit pos-random-dseq
where
  pos-if-random-dseq b = (if b then pos-single () else pos-empty)

```

```

definition pos-iterate-upto :: (code-numeral  $\Rightarrow$  'a)  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral
 $\Rightarrow$  'a pos-random-dseq
where
  pos-iterate-upto f n m = ( $\lambda$ nrandom size seed. New-DSequence.pos-iterate-upto f
n m)

```

```

definition pos-not-random-dseq :: unit neg-random-dseq  $\Rightarrow$  unit pos-random-dseq
where
  pos-not-random-dseq R = ( $\lambda$ nrandom size seed. New-DSequence.pos-not-seq (R
nrandom size seed))

```

```

fun iter :: (code-numeral * code-numeral  $\Rightarrow$  ('a * (unit  $\Rightarrow$  term)) * code-numeral
* code-numeral)  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral * code-numeral  $\Rightarrow$  'a Lazy-Sequence.lazy-sequence

```

**where**

*iter random nrandom seed =*  
*(if nrandom = 0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence*  
*(%u. let ((x, -), seed') = random seed in Some (x, iter random (nrandom - 1)*  
*seed'))))*

**definition** *Random* :: (code-numeral  $\Rightarrow$  *Random.seed*  $\Rightarrow$  (('a  $\times$  (unit  $\Rightarrow$  term))  
 $\times$  *Random.seed*))  $\Rightarrow$  'a pos-random-dseq

**where**

*Random g = (%nrandom size seed depth. iter (g size) nrandom seed)*

**definition** *pos-map* :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a pos-random-dseq  $\Rightarrow$  'b pos-random-dseq

**where**

*pos-map f P = pos-bind P (pos-single o f)*

**definition** *neg-empty* :: 'a neg-random-dseq

**where**

*neg-empty = (%nrandom size seed. New-DSequence.neg-empty)*

**definition** *neg-single* :: 'a  $\Rightarrow$  'a neg-random-dseq

**where**

*neg-single x = (%nrandom size seed. New-DSequence.neg-single x)*

**definition** *neg-bind* :: 'a neg-random-dseq  $\Rightarrow$  ('a  $\Rightarrow$  'b neg-random-dseq)  $\Rightarrow$  'b  
 neg-random-dseq

**where**

*neg-bind R f = ( $\lambda$ nrandom size seed. New-DSequence.neg-bind (R nrandom size  
 seed) (%a. f a nrandom size seed))*

**definition** *neg-union* :: 'a neg-random-dseq  $\Rightarrow$  'a neg-random-dseq  $\Rightarrow$  'a neg-random-dseq

**where**

*neg-union R1 R2 = ( $\lambda$ nrandom size seed. New-DSequence.neg-union (R1 nran-  
 dom size seed) (R2 nrandom size seed))*

**definition** *neg-if-random-dseq* :: bool  $\Rightarrow$  unit neg-random-dseq

**where**

*neg-if-random-dseq b = (if b then neg-single () else neg-empty)*

**definition** *neg-iterate-upto* :: (code-numeral  $\Rightarrow$  'a)  $\Rightarrow$  code-numeral  $\Rightarrow$  code-numeral  
 $\Rightarrow$  'a neg-random-dseq

**where**

*neg-iterate-upto f n m = ( $\lambda$ nrandom size seed. New-DSequence.neg-iterate-upto f  
 n m)*

**definition** *neg-not-random-dseq* :: unit pos-random-dseq  $\Rightarrow$  unit neg-random-dseq

**where**

*neg-not-random-dseq R = ( $\lambda$ nrandom size seed. New-DSequence.neg-not-seq (R  
 nrandom size seed))*

**definition**  $neg\_map :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ neg-random-dseq} \Rightarrow 'b \text{ neg-random-dseq}$

**where**

$neg\_map\ f\ P = neg\_bind\ P\ (neg\_single\ o\ f)$

**hide-const** (open)

$pos\_empty\ pos\_single\ pos\_bind\ pos\_union\ pos\_if\_random\_dseq\ pos\_iterate\_upto\ pos\_not\_random\_dseq$   
 $iter\ Random\ pos\_map$

$neg\_empty\ neg\_single\ neg\_bind\ neg\_union\ neg\_if\_random\_dseq\ neg\_iterate\_upto\ neg\_not\_random\_dseq$   
 $neg\_map$

**hide-type**  $New\_DSequence.pos\_dseq\ New\_DSequence.neg\_dseq\ pos\_random\_dseq\ neg\_random\_dseq$

**end**

## 52 Predicate-Compile: A compiler for predicates defined by introduction rules

**theory** *Predicate-Compile*

**imports** *New-Random-Sequence*

**uses**

*Tools/Predicate-Compile/predicate-compile-aux.ML*  
*Tools/Predicate-Compile/predicate-compile-compilations.ML*  
*Tools/Predicate-Compile/predicate-compile-core.ML*  
*Tools/Predicate-Compile/predicate-compile-data.ML*  
*Tools/Predicate-Compile/predicate-compile-fun.ML*  
*Tools/Predicate-Compile/predicate-compile-pred.ML*  
*Tools/Predicate-Compile/predicate-compile-specialisation.ML*  
*Tools/Predicate-Compile/predicate-compile.ML*

**begin**

**setup** *Predicate-Compile.setup*

**end**

## 53 Map: Maps

**theory** *Map*

**imports** *List*

**begin**

**types**  $('a, 'b)\ map = 'a \Rightarrow 'b\ option$  (**infixr**  $\sim\Rightarrow 0$ )

**translations**  $(type)\ 'a\ \sim\Rightarrow\ 'b\ \leq\ (type)\ 'a\ \Rightarrow\ 'b\ option$

**type-notation** (*xsymbols*)



*map* (**infixr**  $\rightarrow 0$ )

#### abbreviation

*empty* :: 'a  $\leadsto$  'b **where**  
*empty* == %x. None

#### definition

*map-comp* :: ('b  $\leadsto$  'c)  $\Rightarrow$  ('a  $\leadsto$  'b)  $\Rightarrow$  ('a  $\leadsto$  'c) (**infixl** o'-m 55)  
**where**  
*f* o-m *g* = ( $\lambda k$ . case *g* *k* of None  $\Rightarrow$  None | Some *v*  $\Rightarrow$  *f* *v*)

#### notation (*xsymbols*)

*map-comp* (**infixl**  $\circ_m$  55)

#### definition

*map-add* :: ('a  $\leadsto$  'b)  $\Rightarrow$  ('a  $\leadsto$  'b)  $\Rightarrow$  ('a  $\leadsto$  'b) (**infixl** ++ 100)  
**where**  
*m1* ++ *m2* = ( $\lambda x$ . case *m2* *x* of None  $\Rightarrow$  *m1* *x* | Some *y*  $\Rightarrow$  Some *y*)

#### definition

*restrict-map* :: ('a  $\leadsto$  'b)  $\Rightarrow$  'a set  $\Rightarrow$  ('a  $\leadsto$  'b) (**infixl** |' 110) **where**  
*m* | 'A = ( $\lambda x$ . if *x* : A then *m* *x* else None)

#### notation (*latex output*)

*restrict-map* (-|-. [111,110] 110)

#### definition

*dom* :: ('a  $\leadsto$  'b)  $\Rightarrow$  'a set **where**  
*dom* *m* = {*a*. *m* *a*  $\leadsto$  None}

#### definition

*ran* :: ('a  $\leadsto$  'b)  $\Rightarrow$  'b set **where**  
*ran* *m* = {*b*. EX *a*. *m* *a* = Some *b*}

#### definition

*map-le* :: ('a  $\leadsto$  'b)  $\Rightarrow$  ('a  $\leadsto$  'b)  $\Rightarrow$  bool (**infix**  $\subseteq_m$  50) **where**  
(*m*<sub>1</sub>  $\subseteq_m$  *m*<sub>2</sub>) = ( $\forall a \in \text{dom } m_1. m_1 a = m_2 a$ )

#### nonterminals

*maplets* *maplet*

#### syntax

-*maplet* :: ['a, 'a]  $\Rightarrow$  *maplet* (- /|->/ -)  
-*maplets* :: ['a, 'a]  $\Rightarrow$  *maplet* (- /|[->]/ -)  
:: *maplet*  $\Rightarrow$  *maplets* (-)  
-*Maplets* :: [*maplet*, *maplets*]  $\Rightarrow$  *maplets* (-, / -)  
-*MapUpd* :: ['a  $\leadsto$  'b, *maplets*]  $\Rightarrow$  'a  $\leadsto$  'b (-/'(-) [900,0]900)  
-*Map* :: *maplets*  $\Rightarrow$  'a  $\leadsto$  'b ((1[-]))

**syntax** (*xsymbols*)

$\text{-maplet} :: ['a, 'a] \Rightarrow \text{maplet} \quad (- \text{ /}\mapsto \text{ / } -)$   
 $\text{-maplets} :: ['a, 'a] \Rightarrow \text{maplet} \quad (- \text{ /}\lceil\mapsto\rceil \text{ / } -)$

**translations**

$\text{-MapUpd } m \text{ (-Maplets } xy \text{ } ms) == \text{-MapUpd } (-\text{MapUpd } m \text{ } xy) \text{ } ms$   
 $\text{-MapUpd } m \text{ (-maplet } x \text{ } y) == m(x := \text{CONST Some } y)$   
 $\text{-Map } ms == \text{-MapUpd } (\text{CONST empty}) \text{ } ms$   
 $\text{-Map } (-\text{Maplets } ms1 \text{ } ms2) <= \text{-MapUpd } (-\text{Map } ms1) \text{ } ms2$   
 $\text{-Maplets } ms1 \text{ (-Maplets } ms2 \text{ } ms3) <= \text{-Maplets } (-\text{Maplets } ms1 \text{ } ms2) \text{ } ms3$

**primrec**

$\text{map-of} :: ('a \times 'b) \text{ list} \Rightarrow 'a \rightarrow 'b \text{ where}$   
 $\text{map-of } [] = \text{empty}$   
 $| \text{map-of } (p \# ps) = (\text{map-of } ps)(\text{fst } p \mapsto \text{snd } p)$

**definition**

$\text{map-upds} :: ('a \rightarrow 'b) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ list} \Rightarrow 'a \rightarrow 'b \text{ where}$   
 $\text{map-upds } m \text{ } xs \text{ } ys = m ++ \text{map-of } (\text{rev } (\text{zip } xs \text{ } ys))$

**translations**

$\text{-MapUpd } m \text{ (-maplets } x \text{ } y) == \text{CONST map-upds } m \text{ } x \text{ } y$

**lemma** *map-of-Cons-code* [*code*]:

$\text{map-of } [] \text{ } k = \text{None}$   
 $\text{map-of } ((l, v) \# ps) \text{ } k = (\text{if } l = k \text{ then Some } v \text{ else map-of } ps \text{ } k)$   
**by** *simp-all*

### 53.1 empty

**lemma** *empty-upd-none* [*simp*]:  $\text{empty}(x := \text{None}) = \text{empty}$

**by** (*rule ext*) *simp*

### 53.2 map-upd

**lemma** *map-upd-triv*:  $t \text{ } k = \text{Some } x \Rightarrow t(k|-\>x) = t$

**by** (*rule ext*) *simp*

**lemma** *map-upd-nonempty* [*simp*]:  $t(k|-\>x) \sim= \text{empty}$

**proof**

**assume**  $t(k \mapsto x) = \text{empty}$   
**then have**  $(t(k \mapsto x)) \text{ } k = \text{None}$  **by** *simp*  
**then show** *False* **by** *simp*

**qed**

**lemma** *map-upd-eqD1*:

**assumes**  $m(a \mapsto x) = n(a \mapsto y)$   
**shows**  $x = y$

**proof** –

**from** *prems* **have**  $(m(a \mapsto x)) \text{ } a = (n(a \mapsto y)) \text{ } a$  **by** *simp*

**then show** *?thesis* **by** *simp*  
**qed**

**lemma** *map-upd-Some-unfold*:  
 $((m(a|->b))\ x = \text{Some } y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = \text{Some } y)$   
**by** *auto*

**lemma** *image-map-upd [simp]*:  $x \notin A \implies m(x \mapsto y) \text{ ` } A = m \text{ ` } A$   
**by** *auto*

**lemma** *finite-range-updI*:  $\text{finite } (\text{range } f) \implies \text{finite } (\text{range } (f(a|->b)))$   
**unfolding** *image-def*  
**apply** (*simp (no-asm-use) add:full-SetCompr-eq*)  
**apply** (*rule finite-subset*)  
**prefer** 2 **apply** *assumption*  
**apply** (*auto*)  
**done**

### 53.3 map-of

**lemma** *map-of-eq-None-iff*:  
 $(\text{map-of } xys\ x = \text{None}) = (x \notin \text{fst ` } (\text{set } xys))$   
**by** (*induct xys*) *simp-all*

**lemma** *map-of-is-SomeD*:  $\text{map-of } xys\ x = \text{Some } y \implies (x,y) \in \text{set } xys$   
**apply** (*induct xys*)  
**apply** *simp*  
**apply** (*clarsimp split: if-splits*)  
**done**

**lemma** *map-of-eq-Some-iff [simp]*:  
 $\text{distinct}(\text{map fst } xys) \implies (\text{map-of } xys\ x = \text{Some } y) = ((x,y) \in \text{set } xys)$   
**apply** (*induct xys*)  
**apply** *simp*  
**apply** (*auto simp: map-of-eq-None-iff [symmetric]*)  
**done**

**lemma** *Some-eq-map-of-iff [simp]*:  
 $\text{distinct}(\text{map fst } xys) \implies (\text{Some } y = \text{map-of } xys\ x) = ((x,y) \in \text{set } xys)$   
**by** (*auto simp del:map-of-eq-Some-iff simp add: map-of-eq-Some-iff [symmetric]*)

**lemma** *map-of-is-SomeI [simp]*:  $\llbracket \text{distinct}(\text{map fst } xys); (x,y) \in \text{set } xys \rrbracket$   
 $\implies \text{map-of } xys\ x = \text{Some } y$   
**apply** (*induct xys*)  
**apply** *simp*  
**apply** *force*  
**done**

**lemma** *map-of-zip-is-None [simp]*:

$length\ xs = length\ ys \implies (map-of\ (zip\ xs\ ys)\ x = None) = (x \notin set\ xs)$   
**by** (induct rule: list-induct2) simp-all

**lemma** map-of-zip-is-Some:

**assumes**  $length\ xs = length\ ys$

**shows**  $x \in set\ xs \iff (\exists y. map-of\ (zip\ xs\ ys)\ x = Some\ y)$

**using** *assms* **by** (induct rule: list-induct2) simp-all

**lemma** map-of-zip-upd:

**fixes**  $x :: 'a$  **and**  $xs :: 'a\ list$  **and**  $ys\ zs :: 'b\ list$

**assumes**  $length\ ys = length\ xs$

**and**  $length\ zs = length\ xs$

**and**  $x \notin set\ xs$

**and**  $map-of\ (zip\ xs\ ys)(x \mapsto y) = map-of\ (zip\ xs\ zs)(x \mapsto z)$

**shows**  $map-of\ (zip\ xs\ ys) = map-of\ (zip\ xs\ zs)$

**proof**

**fix**  $x' :: 'a$

**show**  $map-of\ (zip\ xs\ ys)\ x' = map-of\ (zip\ xs\ zs)\ x'$

**proof** (cases  $x = x'$ )

**case** *True*

**from** *assms* *True* map-of-zip-is-None [of  $xs\ ys\ x'$ ]

**have**  $map-of\ (zip\ xs\ ys)\ x' = None$  **by** *simp*

**moreover from** *assms* *True* map-of-zip-is-None [of  $xs\ zs\ x'$ ]

**have**  $map-of\ (zip\ xs\ zs)\ x' = None$  **by** *simp*

**ultimately show** ?thesis **by** *simp*

**next**

**case** *False* **from** *assms*

**have**  $(map-of\ (zip\ xs\ ys)(x \mapsto y))\ x' = (map-of\ (zip\ xs\ zs)(x \mapsto z))\ x'$  **by**

*auto*

**with** *False* **show** ?thesis **by** *simp*

**qed**

**qed**

**lemma** map-of-zip-inject:

**assumes**  $length\ ys = length\ xs$

**and**  $length\ zs = length\ xs$

**and** *dist: distinct xs*

**and**  $map-of: map-of\ (zip\ xs\ ys) = map-of\ (zip\ xs\ zs)$

**shows**  $ys = zs$

**using** *assms*(1) *assms*(2)[*symmetric*] **using** *dist* map-of **proof** (induct  $ys\ xs\ zs$  rule: list-induct3)

**case** *Nil* **show** ?case **by** *simp*

**next**

**case** (*Cons*  $y\ ys\ x\ xs\ z\ zs$ )

**from**  $\langle map-of\ (zip\ (x\#\!xs)\ (y\#\!ys)) = map-of\ (zip\ (x\#\!xs)\ (z\#\!zs)) \rangle$

**have**  $map-of: map-of\ (zip\ xs\ ys)(x \mapsto y) = map-of\ (zip\ xs\ zs)(x \mapsto z)$  **by** *simp*

**from** *Cons* **have**  $length\ ys = length\ xs$  **and**  $length\ zs = length\ xs$

**and**  $x \notin set\ xs$  **by** *simp-all*

**then have**  $map-of\ (zip\ xs\ ys) = map-of\ (zip\ xs\ zs)$  **using** map-of **by** (rule

```

map-of-zip-upd)
  with Cons.hyps ⟨distinct (x # xs)⟩ have ys = zs by simp
  moreover from map-of have y = z by (rule map-upd-eqD1)
  ultimately show ?case by simp
qed

```

```

lemma map-of-zip-map:
  map-of (zip xs (map f xs)) = (λx. if x ∈ set xs then Some (f x) else None)
  by (induct xs) (simp-all add: expand-fun-eq)

```

```

lemma finite-range-map-of: finite (range (map-of xys))
  apply (induct xys)
  apply (simp-all add: image-constant)
  apply (rule finite-subset)
  prefer 2 apply assumption
  apply auto
  done

```

```

lemma map-of-SomeD: map-of xs k = Some y ⟹ (k, y) ∈ set xs
  by (induct xs) (simp, atomize (full), auto)

```

```

lemma map-of-mapk-SomeI:
  inj f ⟹ map-of t k = Some x ⟹
  map-of (map (split (%k. Pair (f k))) t) (f k) = Some x
  by (induct t) (auto simp add: inj-eq)

```

```

lemma weak-map-of-SomeI: (k, x) : set l ⟹ ∃ x. map-of l k = Some x
  by (induct l) auto

```

```

lemma map-of-filter-in:
  map-of xs k = Some z ⟹ P k z ⟹ map-of (filter (split P) xs) k = Some z
  by (induct xs) auto

```

```

lemma map-of-map:
  map-of (map (λ(k, v). (k, f v)) xs) = Option.map f ∘ map-of xs
  by (induct xs) (auto simp add: expand-fun-eq)

```

```

lemma dom-option-map:
  dom (λk. Option.map (f k) (m k)) = dom m
  by (simp add: dom-def)

```

### 53.4 Option.map related

```

lemma option-map-o-empty [simp]: Option.map f o empty = empty
  by (rule ext) simp

```

```

lemma option-map-o-map-upd [simp]:
  Option.map f o m(a|→b) = (Option.map f o m)(a|→f b)
  by (rule ext) simp

```

**53.5** *map-comp related***lemma** *map-comp-empty* [simp]:

$$m \circ_m \text{empty} = \text{empty}$$

$$\text{empty} \circ_m m = \text{empty}$$

**by** (auto simp add: map-comp-def intro: ext split: option.splits)**lemma** *map-comp-simps* [simp]:

$$m2\ k = \text{None} \implies (m1 \circ_m m2)\ k = \text{None}$$

$$m2\ k = \text{Some } k' \implies (m1 \circ_m m2)\ k = m1\ k'$$

**by** (auto simp add: map-comp-def)**lemma** *map-comp-Some-iff*:

$$((m1 \circ_m m2)\ k = \text{Some } v) = (\exists k'. m2\ k = \text{Some } k' \wedge m1\ k' = \text{Some } v)$$

**by** (auto simp add: map-comp-def split: option.splits)**lemma** *map-comp-None-iff*:

$$((m1 \circ_m m2)\ k = \text{None}) = (m2\ k = \text{None} \vee (\exists k'. m2\ k = \text{Some } k' \wedge m1\ k' = \text{None}))$$

**by** (auto simp add: map-comp-def split: option.splits)**53.6** *++***lemma** *map-add-empty*[simp]:  $m ++ \text{empty} = m$ **by**(simp add: map-add-def)**lemma** *empty-map-add*[simp]:  $\text{empty} ++ m = m$ **by** (rule ext) (simp add: map-add-def split: option.split)**lemma** *map-add-assoc*[simp]:  $m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3$ **by** (rule ext) (simp add: map-add-def split: option.split)**lemma** *map-add-Some-iff*:

$$((m ++ n)\ k = \text{Some } x) = (n\ k = \text{Some } x \mid n\ k = \text{None} \ \& \ m\ k = \text{Some } x)$$

**by** (simp add: map-add-def split: option.split)**lemma** *map-add-SomeD* [dest!]:

$$(m ++ n)\ k = \text{Some } x \implies n\ k = \text{Some } x \vee n\ k = \text{None} \wedge m\ k = \text{Some } x$$

**by** (rule map-add-Some-iff [THEN iffD1])**lemma** *map-add-find-right* [simp]:  $!!xx. n\ k = \text{Some } xx \implies (m ++ n)\ k = \text{Some } xx$ **by** (subst map-add-Some-iff) fast**lemma** *map-add-None* [iff]:  $((m ++ n)\ k = \text{None}) = (n\ k = \text{None} \ \& \ m\ k = \text{None})$ **by** (simp add: map-add-def split: option.split)**lemma** *map-add-upd*[simp]:  $f ++ g(x|>y) = (f ++ g)(x|>y)$ **by** (rule ext) (simp add: map-add-def)

**lemma** *map-add-upds*[simp]:  $m1 \ ++ \ (m2(xs[\mapsto]ys)) = (m1 \ ++ \ m2)(xs[\mapsto]ys)$   
**by** (*simp add: map-upds-def*)

**lemma** *map-add-upd-left*:  $m \notin \text{dom } e2 \implies e1(m \mapsto u1) \ ++ \ e2 = (e1 \ ++ \ e2)(m \mapsto u1)$   
**by** (*rule ext*) (*auto simp: map-add-def dom-def split: option.split*)

**lemma** *map-of-append*[simp]:  $\text{map-of } (xs \ @ \ ys) = \text{map-of } ys \ ++ \ \text{map-of } xs$   
**unfolding** *map-add-def*  
**apply** (*induct xs*)  
**apply** *simp*  
**apply** (*rule ext*)  
**apply** (*simp split add: option.split*)  
**done**

**lemma** *finite-range-map-of-map-add*:  
 $\text{finite } (\text{range } f) \implies \text{finite } (\text{range } (f \ ++ \ \text{map-of } l))$   
**apply** (*induct l*)  
**apply** (*auto simp del: fun-upd-apply*)  
**apply** (*erule finite-range-updI*)  
**done**

**lemma** *inj-on-map-add-dom* [iff]:  
 $\text{inj-on } (m \ ++ \ m') \ (\text{dom } m') = \text{inj-on } m' \ (\text{dom } m')$   
**by** (*fastsimp simp: map-add-def dom-def inj-on-def split: option.splits*)

**lemma** *map-upds-fold-map-upd*:  
 $m(ks[\mapsto]vs) = \text{foldl } (\lambda m \ (k, v). m(k \mapsto v)) \ m \ (\text{zip } ks \ vs)$   
**unfolding** *map-upds-def* **proof** (*rule sym, rule zip-obtain-same-length*)  
**fix** *ks* :: 'a list **and** *vs* :: 'b list  
**assume**  $\text{length } ks = \text{length } vs$   
**then show**  $\text{foldl } (\lambda m \ (k, v). m(k \mapsto v)) \ m \ (\text{zip } ks \ vs) = m \ ++ \ \text{map-of } (\text{rev } (\text{zip } ks \ vs))$   
**by**(*induct arbitrary: m rule: list-induct2*) *simp-all*  
**qed**

**lemma** *map-add-map-of-foldr*:  
 $m \ ++ \ \text{map-of } ps = \text{foldr } (\lambda(k, v) \ m. m(k \mapsto v)) \ ps \ m$   
**by** (*induct ps*) (*auto simp add: expand-fun-eq map-add-def*)

### 53.7 restrict-map

**lemma** *restrict-map-to-empty* [simp]:  $m|'\{\} = \text{empty}$   
**by** (*simp add: restrict-map-def*)

**lemma** *restrict-map-insert*:  $f|'(\text{insert } a \ A) = (f|'A)(a := f \ a)$   
**by** (*auto simp add: restrict-map-def intro: ext*)

**lemma** *restrict-map-empty* [simp]:  $\text{empty} \upharpoonright D = \text{empty}$   
**by** (simp add: restrict-map-def)

**lemma** *restrict-in* [simp]:  $x \in A \implies (m \upharpoonright A) x = m x$   
**by** (simp add: restrict-map-def)

**lemma** *restrict-out* [simp]:  $x \notin A \implies (m \upharpoonright A) x = \text{None}$   
**by** (simp add: restrict-map-def)

**lemma** *ran-restrictD*:  $y \in \text{ran } (m \upharpoonright A) \implies \exists x \in A. m x = \text{Some } y$   
**by** (auto simp: restrict-map-def ran-def split: split-if-asm)

**lemma** *dom-restrict* [simp]:  $\text{dom } (m \upharpoonright A) = \text{dom } m \cap A$   
**by** (auto simp: restrict-map-def dom-def split: split-if-asm)

**lemma** *restrict-upd-same* [simp]:  $m(x \mapsto y) \upharpoonright (-\{x\}) = m \upharpoonright (-\{x\})$   
**by** (rule ext) (auto simp: restrict-map-def)

**lemma** *restrict-restrict* [simp]:  $m \upharpoonright A \upharpoonright B = m \upharpoonright (A \cap B)$   
**by** (rule ext) (auto simp: restrict-map-def)

**lemma** *restrict-fun-upd* [simp]:  
 $m(x := y) \upharpoonright D = (\text{if } x \in D \text{ then } (m \upharpoonright (D - \{x\}))(x := y) \text{ else } m \upharpoonright D)$   
**by** (simp add: restrict-map-def expand-fun-eq)

**lemma** *fun-upd-None-restrict* [simp]:  
 $(m \upharpoonright D)(x := \text{None}) = (\text{if } x : D \text{ then } m \upharpoonright (D - \{x\}) \text{ else } m \upharpoonright D)$   
**by** (simp add: restrict-map-def expand-fun-eq)

**lemma** *fun-upd-restrict*:  $(m \upharpoonright D)(x := y) = (m \upharpoonright (D - \{x\}))(x := y)$   
**by** (simp add: restrict-map-def expand-fun-eq)

**lemma** *fun-upd-restrict-conv* [simp]:  
 $x \in D \implies (m \upharpoonright D)(x := y) = (m \upharpoonright (D - \{x\}))(x := y)$   
**by** (simp add: restrict-map-def expand-fun-eq)

**lemma** *map-of-map-restrict*:  
 $\text{map-of } (\text{map } (\lambda k. (k, f k)) ks) = (\text{Some } \circ f) \upharpoonright \text{set } ks$   
**by** (induct ks) (simp-all add: expand-fun-eq restrict-map-insert)

**lemma** *restrict-complement-singleton-eq*:  
 $f \upharpoonright (-\{x\}) = f(x := \text{None})$   
**by** (simp add: restrict-map-def expand-fun-eq)

### 53.8 map-upds

**lemma** *map-upds-Nil1* [simp]:  $m(\square \parallel ->] bs) = m$   
**by** (simp add: map-upds-def)



**lemma** *map-upds-Nil2* [*simp*]:  $m(as \ [->] []) = m$   
**by** (*simp add: map-upds-def*)

**lemma** *map-upds-Cons* [*simp*]:  $m(a \# as \ [->] b \# bs) = (m(a \ [->] b))(as \ [->] bs)$   
**by** (*simp add: map-upds-def*)

**lemma** *map-upds-append1* [*simp*]:  $\bigwedge ys \ m. \ size \ xs < size \ ys \implies$   
 $m(xs @ [x] \ [\mapsto] \ ys) = m(xs \ [\mapsto] \ ys)(x \mapsto ys!size \ xs)$   
**apply** (*induct xs*)  
**apply** (*clarsimp simp add: neq-Nil-conv*)  
**apply** (*case-tac ys*)  
**apply** *simp*  
**apply** *simp*  
**done**

**lemma** *map-upds-list-update2-drop* [*simp*]:  
 $\llbracket size \ xs \leq i; i < size \ ys \rrbracket$   
 $\implies m(xs \ [\mapsto] \ ys[i:=y]) = m(xs \ [\mapsto] \ ys)$   
**apply** (*induct xs arbitrary: m ys i*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *simp*  
**apply** (*simp split: nat.split*)  
**done**

**lemma** *map-upd-upds-conv-if*:  
 $(f(x \ [->] y))(xs \ [->] \ ys) =$   
 $(if \ x : set \ (take \ (length \ ys) \ xs) \ then \ f(xs \ [->] \ ys)$   
 $else \ (f(xs \ [->] \ ys))(x \ [->] y))$   
**apply** (*induct xs arbitrary: x y ys f*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** (*auto split: split-if simp: fun-upd-twist*)  
**done**

**lemma** *map-upds-twist* [*simp*]:  
 $a \sim : set \ as \implies m(a \ [->] b)(as \ [->] bs) = m(as \ [->] bs)(a \ [->] b)$   
**using** *set-take-subset* **by** (*fastsimp simp add: map-upd-upds-conv-if*)

**lemma** *map-upds-apply-nontin* [*simp*]:  
 $x \sim : set \ xs \implies (f(xs \ [->] \ ys)) \ x = f \ x$   
**apply** (*induct xs arbitrary: ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** (*auto simp: map-upd-upds-conv-if*)  
**done**

**lemma** *fun-upds-append-drop* [*simp*]:  
 $size \ xs = size \ ys \implies m(xs @ zs \ [\mapsto] \ ys) = m(xs \ [\mapsto] \ ys)$

```

apply (induct xs arbitrary: m ys)
  apply simp
apply (case-tac ys)
  apply simp-all
done

```

```

lemma fun-upds-append2-drop [simp]:
   $\text{size } xs = \text{size } ys \implies m(xs[\mapsto]ys @ zs) = m(xs[\mapsto]ys)$ 
apply (induct xs arbitrary: m ys)
  apply simp
apply (case-tac ys)
  apply simp-all
done

```

```

lemma restrict-map-upds [simp]:
   $\llbracket \text{length } xs = \text{length } ys; \text{set } xs \subseteq D \rrbracket$ 
   $\implies m(xs[\mapsto]ys) \upharpoonright D = (m \upharpoonright (D - \text{set } xs))(xs[\mapsto]ys)$ 
apply (induct xs arbitrary: m ys)
  apply simp
apply (case-tac ys)
  apply simp
apply (simp add: Diff-insert [symmetric] insert-absorb)
apply (simp add: map-upd-upds-conv-if)
done

```

### 53.9 *dom*

```

lemma dom-eq-empty-conv [simp]:  $\text{dom } f = \{\} \longleftrightarrow f = \text{empty}$ 
by(auto intro!: ext simp: dom-def)

```

```

lemma domI:  $m \ a = \text{Some } b \implies a : \text{dom } m$ 
by(simp add: dom-def)

```

```

lemma domD:  $a : \text{dom } m \implies \exists b. m \ a = \text{Some } b$ 
by (cases m a) (auto simp add: dom-def)

```

```

lemma domIff [iff, simp del]:  $(a : \text{dom } m) = (m \ a \sim \text{None})$ 
by(simp add: dom-def)

```

```

lemma dom-empty [simp]:  $\text{dom } \text{empty} = \{\}$ 
by(simp add: dom-def)

```

```

lemma dom-fun-upd [simp]:
   $\text{dom}(f(x := y)) = (\text{if } y = \text{None} \text{ then } \text{dom } f - \{x\} \text{ else } \text{insert } x (\text{dom } f))$ 
by(auto simp add: dom-def)

```

```

lemma dom-if:

```

$\text{dom } (\lambda x. \text{ if } P \ x \text{ then } f \ x \text{ else } g \ x) = \text{dom } f \cap \{x. P \ x\} \cup \text{dom } g \cap \{x. \neg P \ x\}$   
**by** (*auto split: if-splits*)

**lemma** *dom-map-of-conv-image-fst*:  
 $\text{dom } (\text{map-of } xys) = \text{fst } \text{' set } xys$   
**by** (*induct xys*) (*auto simp add: dom-if*)

**lemma** *dom-map-of-zip* [*simp*]:  $[\text{length } xs = \text{length } ys; \text{distinct } xs] \implies$   
 $\text{dom}(\text{map-of}(\text{zip } xs \ ys)) = \text{set } xs$   
**by** (*induct rule: list-induct2*) *simp-all*

**lemma** *finite-dom-map-of*:  $\text{finite } (\text{dom } (\text{map-of } l))$   
**by** (*induct l*) (*auto simp add: dom-def insert-Collect [symmetric]*)

**lemma** *dom-map-upds* [*simp*]:  
 $\text{dom}(m(xs[->]ys)) = \text{set}(\text{take } (\text{length } ys) \ xs) \cup \text{dom } m$   
**apply** (*induct xs arbitrary: m ys*)  
**apply** *simp*  
**apply** (*case-tac ys*)  
**apply** *auto*  
**done**

**lemma** *dom-map-add* [*simp*]:  $\text{dom}(m++n) = \text{dom } n \cup \text{dom } m$   
**by**(*auto simp: dom-def*)

**lemma** *dom-override-on* [*simp*]:  
 $\text{dom}(\text{override-on } f \ g \ A) =$   
 $(\text{dom } f - \{a. a : A - \text{dom } g\}) \cup \{a. a : A \text{ Int } \text{dom } g\}$   
**by**(*auto simp: dom-def override-on-def*)

**lemma** *map-add-comm*:  $\text{dom } m1 \cap \text{dom } m2 = \{\} \implies m1++m2 = m2++m1$   
**by** (*rule ext*) (*force simp: map-add-def dom-def split: option.split*)

**lemma** *map-add-dom-app-simps*:  
 $[\text{m} \in \text{dom } l2] \implies (l1++l2) \ m = l2 \ m$   
 $[\text{m} \notin \text{dom } l1] \implies (l1++l2) \ m = l2 \ m$   
 $[\text{m} \notin \text{dom } l2] \implies (l1++l2) \ m = l1 \ m$   
**by** (*auto simp add: map-add-def split: option.split-asm*)

**lemma** *dom-const* [*simp*]:  
 $\text{dom } (\lambda x. \text{Some } (f \ x)) = \text{UNIV}$   
**by** *auto*

**lemma** *finite-map-freshness*:  
 $\text{finite } (\text{dom } (f :: 'a \rightarrow 'b)) \implies \neg \text{finite } (\text{UNIV} :: 'a \text{ set}) \implies$   
 $\exists x. f \ x = \text{None}$   
**by**(*bestsimp dest: ex-new-if-finite*)

**lemma** *dom-minus*:

$f\ x = \text{None} \implies \text{dom } f - \text{insert } x\ A = \text{dom } f - A$

**unfolding** *dom-def* **by** *simp*

**lemma** *insert-dom*:

$f\ x = \text{Some } y \implies \text{insert } x\ (\text{dom } f) = \text{dom } f$

**unfolding** *dom-def* **by** *auto*

**lemma** *map-of-map-keys*:

$\text{set } xs = \text{dom } m \implies \text{map-of } (\text{map } (\lambda k. (k, \text{the } (m\ k))))\ xs = m$

**by** (*rule ext*) (*auto simp add: map-of-map-restrict restrict-map-def*)

### 53.10 *ran*

**lemma** *ranI*:  $m\ a = \text{Some } b \implies b : \text{ran } m$

**by** (*auto simp: ran-def*)

**lemma** *ran-empty* [*simp*]:  $\text{ran } \text{empty} = \{\}$

**by** (*auto simp: ran-def*)

**lemma** *ran-map-upd* [*simp*]:  $m\ a = \text{None} \implies \text{ran}(m(a|->b)) = \text{insert } b\ (\text{ran } m)$

**unfolding** *ran-def*

**apply** *auto*

**apply** (*subgoal-tac aa ~ = a*)

**apply** *auto*

**done**

**lemma** *ran-distinct*:

**assumes** *dist: distinct* ( $\text{map } \text{fst } al$ )

**shows**  $\text{ran } (\text{map-of } al) = \text{snd } \text{'set } al$

**using** *assms* **proof** (*induct al*)

**case** *Nil* **then show** *?case* **by** *simp*

**next**

**case** ( $\text{Cons } kv\ al$ )

**then have**  $\text{ran } (\text{map-of } al) = \text{snd } \text{'set } al$  **by** *simp*

**moreover from** *Cons.prem*s **have**  $\text{map-of } al\ (\text{fst } kv) = \text{None}$

**by** (*simp add: map-of-eq-None-iff*)

**ultimately show** *?case* **by** (*simp only: map-of.simps ran-map-upd*) *simp*

**qed**

### 53.11 *map-le*

**lemma** *map-le-empty* [*simp*]:  $\text{empty} \subseteq_m g$

**by** (*simp add: map-le-def*)

**lemma** *upd-None-map-le* [*simp*]:  $f(x := \text{None}) \subseteq_m f$

**by** (*force simp add: map-le-def*)

**lemma** *map-le-upd[simp]*:  $f \subseteq_m g \implies f(a := b) \subseteq_m g(a := b)$   
**by** (*fastsimp simp add: map-le-def*)

**lemma** *map-le-imp-upd-le [simp]*:  $m1 \subseteq_m m2 \implies m1(x := \text{None}) \subseteq_m m2(x \mapsto y)$   
**by** (*force simp add: map-le-def*)

**lemma** *map-le-upds [simp]*:  
 $f \subseteq_m g \implies f(as \llbracket - \triangleright \rrbracket bs) \subseteq_m g(as \llbracket - \triangleright \rrbracket bs)$   
**apply** (*induct as arbitrary: f g bs*)  
**apply** *simp*  
**apply** (*case-tac bs*)  
**apply** *auto*  
**done**

**lemma** *map-le-implies-dom-le*:  $(f \subseteq_m g) \implies (\text{dom } f \subseteq \text{dom } g)$   
**by** (*fastsimp simp add: map-le-def dom-def*)

**lemma** *map-le-refl [simp]*:  $f \subseteq_m f$   
**by** (*simp add: map-le-def*)

**lemma** *map-le-trans[trans]*:  $\llbracket m1 \subseteq_m m2; m2 \subseteq_m m3 \rrbracket \implies m1 \subseteq_m m3$   
**by** (*auto simp add: map-le-def dom-def*)

**lemma** *map-le-antisym*:  $\llbracket f \subseteq_m g; g \subseteq_m f \rrbracket \implies f = g$   
**unfolding** *map-le-def*  
**apply** (*rule ext*)  
**apply** (*case-tac x \in dom f, simp*)  
**apply** (*case-tac x \in dom g, simp, fastsimp*)  
**done**

**lemma** *map-le-map-add [simp]*:  $f \subseteq_m (g ++ f)$   
**by** (*fastsimp simp add: map-le-def*)

**lemma** *map-le-iff-map-add-commute*:  $(f \subseteq_m f ++ g) = (f ++ g = g ++ f)$   
**by** (*fastsimp simp: map-add-def map-le-def expand-fun-eq split: option.splits*)

**lemma** *map-add-le-mapE*:  $f ++ g \subseteq_m h \implies g \subseteq_m h$   
**by** (*fastsimp simp add: map-le-def map-add-def dom-def*)

**lemma** *map-add-le-mapI*:  $\llbracket f \subseteq_m h; g \subseteq_m h; f \subseteq_m f ++ g \rrbracket \implies f ++ g \subseteq_m h$   
**by** (*clarsimp simp add: map-le-def map-add-def dom-def split: option.splits*)

**lemma** *dom-eq-singleton-conv*:  $\text{dom } f = \{x\} \longleftrightarrow (\exists v. f = [x \mapsto v])$   
**proof** (*rule iffI*)  
**assume**  $\exists v. f = [x \mapsto v]$   
**thus**  $\text{dom } f = \{x\}$  **by** (*auto split: split-if-asm*)  
**next**  
**assume**  $\text{dom } f = \{x\}$

then obtain  $v$  where  $f\ x = \text{Some } v$  by *auto*  
 hence  $[x \mapsto v] \subseteq_m f$  by (*auto simp add: map-le-def*)  
 moreover have  $f \subseteq_m [x \mapsto v]$  using  $\langle \text{dom } f = \{x\} \rangle \langle f\ x = \text{Some } v \rangle$   
 by (*auto simp add: map-le-def*)  
 ultimately have  $f = [x \mapsto v]$  by- (*rule map-le-antisym*)  
 thus  $\exists v. f = [x \mapsto v]$  by *blast*  
 qed

### 53.12 Various

**lemma** *set-map-of-compr*:  
 assumes *distinct*: *distinct* (*map fst xs*)  
 shows *set xs* =  $\{(k, v). \text{map-of } xs\ k = \text{Some } v\}$   
 using *assms* **proof** (*induct xs*)  
 case *Nil* then show ?*case* by *simp*  
 next  
 case (*Cons x xs*)  
 obtain  $k\ v$  where  $x = (k, v)$  by (*cases x*) *blast*  
 with *Cons.prem*s have  $k \notin \text{dom} (\text{map-of } xs)$   
 by (*simp add: dom-map-of-conv-image-fst*)  
 then have \*: *insert*  $(k, v) \{(k, v). \text{map-of } xs\ k = \text{Some } v\} =$   
 $\{(k', v'). (\text{map-of } xs(k \mapsto v))\ k' = \text{Some } v'\}$   
 by (*auto split: if-splits*)  
 from *Cons* have *set xs* =  $\{(k, v). \text{map-of } xs\ k = \text{Some } v\}$  by *simp*  
 with \*  $\langle x = (k, v) \rangle$  show ?*case* by *simp*  
 qed

**lemma** *map-of-inject-set*:  
 assumes *distinct*: *distinct* (*map fst xs*) *distinct* (*map fst ys*)  
 shows *map-of xs* = *map-of ys*  $\longleftrightarrow$  *set xs* = *set ys* (**is** ?*lhs*  $\longleftrightarrow$  ?*rhs*)  
**proof**  
 assume ?*lhs*  
 moreover from  $\langle \text{distinct } (\text{map fst } xs) \rangle$  have *set xs* =  $\{(k, v). \text{map-of } xs\ k = \text{Some } v\}$   
 by (*rule set-map-of-compr*)  
 moreover from  $\langle \text{distinct } (\text{map fst } ys) \rangle$  have *set ys* =  $\{(k, v). \text{map-of } ys\ k = \text{Some } v\}$   
 by (*rule set-map-of-compr*)  
 ultimately show ?*rhs* by *simp*  
 next  
 assume ?*rhs* show ?*lhs* **proof**  
 fix  $k$   
 show *map-of xs*  $k = \text{map-of } ys\ k$  **proof** (*cases map-of xs k*)  
 case *None*  
 moreover with  $\langle ?\text{rhs} \rangle$  have *map-of ys*  $k = \text{None}$   
 by (*simp add: map-of-eq-None-iff*)  
 ultimately show ?*thesis* by *simp*  
 next  
 case (*Some v*)

```

    moreover with distinct  $\langle ?rhs \rangle$  have map-of  $ys\ k = Some\ v$ 
      by simp
    ultimately show ?thesis by simp
  qed
qed
qed
end

```

## 54 Quotient: Definition of Quotient Types

```

theory Quotient
imports Plain Sledgehammer
uses
  (~~/src/HOL/Tools/Quotient/quotient-info.ML)
  (~~/src/HOL/Tools/Quotient/quotient-typ.ML)
  (~~/src/HOL/Tools/Quotient/quotient-def.ML)
  (~~/src/HOL/Tools/Quotient/quotient-term.ML)
  (~~/src/HOL/Tools/Quotient/quotient-tacs.ML)
begin

```

Basic definition for equivalence relations that are represented by predicates.

**definition**

$$equivp\ E \equiv \forall x\ y. E\ x\ y = (E\ x = E\ y)$$

**definition**

$$reflp\ E \equiv \forall x. E\ x\ x$$

**definition**

$$symp\ E \equiv \forall x\ y. E\ x\ y \longrightarrow E\ y\ x$$

**definition**

$$transp\ E \equiv \forall x\ y\ z. E\ x\ y \wedge E\ y\ z \longrightarrow E\ x\ z$$

**lemma** *equivp-reflp-symp-transp*:

**shows**  $equivp\ E = (reflp\ E \wedge symp\ E \wedge transp\ E)$

**unfolding** *equivp-def* *reflp-def* *symp-def* *transp-def* *expand-fun-eq*

**by** *blast*

**lemma** *equivp-reflp*:

**shows**  $equivp\ E \implies E\ x\ x$

**by** (*simp only: equivp-reflp-symp-transp reflp-def*)

**lemma** *equivp-symp*:

**shows**  $equivp\ E \implies E\ x\ y \implies E\ y\ x$

**by** (*metis equivp-reflp-symp-transp symp-def*)

**lemma** *equivp-transp*:

**shows**  $\text{equivp } E \implies E \ x \ y \implies E \ y \ z \implies E \ x \ z$   
**by** (*metis equivp-reflp-symp-transp transp-def*)

**lemma** *equivpI*:  
**assumes** *reflp R symp R transp R*  
**shows** *equivp R*  
**using** *assms* **by** (*simp add: equivp-reflp-symp-transp*)

**lemma** *identity-equivp*:  
**shows** *equivp (op =)*  
**unfolding** *equivp-def*  
**by** *auto*

Partial equivalences: not yet used anywhere

**definition**  
 $\text{part-equivp } E \equiv (\exists x. E \ x \ x) \wedge (\forall x \ y. E \ x \ y = (E \ x \ x \wedge E \ y \ y \wedge (E \ x = E \ y)))$

**lemma** *equivp-implies-part-equivp*:  
**assumes** *a: equivp E*  
**shows** *part-equivp E*  
**using** *a*  
**unfolding** *equivp-def part-equivp-def*  
**by** *auto*

Composition of Relations

**abbreviation**  
 $\text{rel-conj} \text{ (infixr } \text{OOO } 75)$   
**where**  
 $r1 \text{ OOO } r2 \equiv r1 \text{ OO } r2 \text{ OO } r1$

**lemma** *eq-comp-r*:  
**shows**  $((\text{op } =) \text{ OOO } R) = R$   
**by** (*auto simp add: expand-fun-eq*)

## 54.1 Respects predicate

**definition**  
 $\text{Respects}$   
**where**  
 $\text{Respects } R \ x \equiv R \ x \ x$

**lemma** *in-respects*:  
**shows**  $(x \in \text{Respects } R) = R \ x \ x$   
**unfolding** *mem-def Respects-def*  
**by** *simp*

## 54.2 Function map and function relation

**definition**



*fun-map* (**infixr**  $--->$  55)

**where**

[*simp*]: *fun-map* *f g h x* = *g* (*h* (*f x*))

**definition**

*fun-rel* (**infixr**  $===>$  55)

**where**

[*simp*]: *fun-rel* *E1 E2 f g* =  $(\forall x y. E1\ x\ y \longrightarrow E2\ (f\ x)\ (g\ y))$

**lemma** *fun-relI* [*intro*]:

**assumes**  $\bigwedge a\ b. P\ a\ b \implies Q\ (x\ a)\ (y\ b)$

**shows**  $(P\ ===>\ Q)\ x\ y$

**using** *assms* **by** (*simp add: fun-rel-def*)

**lemma** *fun-map-id*:

**shows**  $(id\ --->\ id) = id$

**by** (*simp add: expand-fun-eq id-def*)

**lemma** *fun-rel-eq*:

**shows**  $((op\ =)\ ===>\ (op\ =)) = (op\ =)$

**by** (*simp add: expand-fun-eq*)

**lemma** *fun-rel-id*:

**assumes**  $a: \bigwedge x\ y. R1\ x\ y \implies R2\ (f\ x)\ (g\ y)$

**shows**  $(R1\ ===>\ R2)\ f\ g$

**using** *a* **by** *simp*

**lemma** *fun-rel-id-asm*:

**assumes**  $a: \bigwedge x\ y. R1\ x\ y \implies (A \longrightarrow R2\ (f\ x)\ (g\ y))$

**shows**  $A \longrightarrow (R1\ ===>\ R2)\ f\ g$

**using** *a* **by** *auto*

### 54.3 Quotient Predicate

**definition**

*Quotient* *E Abs Rep*  $\equiv$

$(\forall a. Abs\ (Rep\ a) = a) \wedge (\forall a. E\ (Rep\ a)\ (Rep\ a)) \wedge$

$(\forall r\ s. E\ r\ s = (E\ r\ r \wedge E\ s\ s \wedge (Abs\ r = Abs\ s)))$

**lemma** *Quotient-abs-rep*:

**assumes**  $a: Quotient\ E\ Abs\ Rep$

**shows**  $Abs\ (Rep\ a) = a$

**using** *a*

**unfolding** *Quotient-def*

**by** *simp*

**lemma** *Quotient-rep-refl*:

**assumes**  $a: Quotient\ E\ Abs\ Rep$

**shows**  $E\ (Rep\ a)\ (Rep\ a)$

using *a*  
 unfolding *Quotient-def*  
 by *blast*

lemma *Quotient-rel*:  
 assumes *a*: *Quotient E Abs Rep*  
 shows  $E\ r\ s = (E\ r\ r \wedge E\ s\ s \wedge (Abs\ r = Abs\ s))$   
 using *a*  
 unfolding *Quotient-def*  
 by *blast*

lemma *Quotient-rel-rep*:  
 assumes *a*: *Quotient R Abs Rep*  
 shows  $R\ (Rep\ a)\ (Rep\ b) = (a = b)$   
 using *a*  
 unfolding *Quotient-def*  
 by *metis*

lemma *Quotient-rep-abs*:  
 assumes *a*: *Quotient R Abs Rep*  
 shows  $R\ r\ r \implies R\ (Rep\ (Abs\ r))\ r$   
 using *a* unfolding *Quotient-def*  
 by *blast*

lemma *Quotient-rel-abs*:  
 assumes *a*: *Quotient E Abs Rep*  
 shows  $E\ r\ s \implies Abs\ r = Abs\ s$   
 using *a* unfolding *Quotient-def*  
 by *blast*

lemma *Quotient-symp*:  
 assumes *a*: *Quotient E Abs Rep*  
 shows *symp E*  
 using *a* unfolding *Quotient-def symp-def*  
 by *metis*

lemma *Quotient-transp*:  
 assumes *a*: *Quotient E Abs Rep*  
 shows *transp E*  
 using *a* unfolding *Quotient-def transp-def*  
 by *metis*

lemma *identity-quotient*:  
 shows *Quotient (op =) id id*  
 unfolding *Quotient-def id-def*  
 by *blast*

lemma *fun-quotient*:  
 assumes *q1*: *Quotient R1 abs1 rep1*

```

and    q2: Quotient R2 abs2 rep2
shows Quotient (R1 ===> R2) (rep1 ----> abs2) (abs1 ----> rep2)
proof -
  have  $\forall a. (rep1 ----> abs2) ((abs1 ----> rep2) a) = a$ 
    using q1 q2
    unfolding Quotient-def
    unfolding expand-fun-eq
    by simp
  moreover
  have  $\forall a. (R1 ===> R2) ((abs1 ----> rep2) a) ((abs1 ----> rep2) a)$ 
    using q1 q2
    unfolding Quotient-def
    by (simp (no-asm)) (metis)
  moreover
  have  $\forall r s. (R1 ===> R2) r s = ((R1 ===> R2) r r \wedge (R1 ===> R2) s s$ 
 $\wedge$ 
     $(rep1 ----> abs2) r = (rep1 ----> abs2) s)$ 
    unfolding expand-fun-eq
    apply(auto)
    using q1 q2 unfolding Quotient-def
    apply(metis)
    using q1 q2 unfolding Quotient-def
    apply(metis)
    using q1 q2 unfolding Quotient-def
    apply(metis)
    using q1 q2 unfolding Quotient-def
    apply(metis)
    done
  ultimately
  show Quotient (R1 ===> R2) (rep1 ----> abs2) (abs1 ----> rep2)
    unfolding Quotient-def by blast
qed

```

**lemma** *abs-o-rep*:

```

assumes a: Quotient R Abs Rep
shows Abs o Rep = id
unfolding expand-fun-eq
by (simp add: Quotient-abs-rep[OF a])

```

**lemma** *equals-rsp*:

```

assumes q: Quotient R Abs Rep
and    a: R xa xb R ya yb
shows R xa ya = R xb yb
using a Quotient-symp[OF q] Quotient-transp[OF q]
unfolding symp-def transp-def
by blast

```

**lemma** *lambda-prs*:

```

assumes q1: Quotient R1 Abs1 Rep1

```

```

and    q2: Quotient R2 Abs2 Rep2
shows (Rep1 ---> Abs2) (λx. Rep2 (f (Abs1 x))) = (λx. f x)
unfolding expand-fun-eq
using Quotient-abs-rep[OF q1] Quotient-abs-rep[OF q2]
by simp

```

```

lemma lambda-prs1:
  assumes q1: Quotient R1 Abs1 Rep1
  and    q2: Quotient R2 Abs2 Rep2
  shows (Rep1 ---> Abs2) (λx. (Abs1 ---> Rep2) f x) = (λx. f x)
  unfolding expand-fun-eq
  using Quotient-abs-rep[OF q1] Quotient-abs-rep[OF q2]
  by simp

```

```

lemma rep-abs-rsp:
  assumes q: Quotient R Abs Rep
  and    a: R x1 x2
  shows R x1 (Rep (Abs x2))
  using a Quotient-rel[OF q] Quotient-abs-rep[OF q] Quotient-rep-reflp[OF q]
  by metis

```

```

lemma rep-abs-rsp-left:
  assumes q: Quotient R Abs Rep
  and    a: R x1 x2
  shows R (Rep (Abs x1)) x2
  using a Quotient-rel[OF q] Quotient-abs-rep[OF q] Quotient-rep-reflp[OF q]
  by metis

```

In the following theorem R1 can be instantiated with anything, but we know some of the types of the Rep and Abs functions; so by solving Quotient assumptions we can get a unique R1 that will be provable; which is why we need to use *apply-rsp* and not the primed version

```

lemma apply-rsp:
  fixes f g::'a ⇒ 'c
  assumes q: Quotient R1 Abs1 Rep1
  and    a: (R1 ==> R2) f g R1 x y
  shows R2 (f x) (g y)
  using a by simp

```

```

lemma apply-rsp':
  assumes a: (R1 ==> R2) f g R1 x y
  shows R2 (f x) (g y)
  using a by simp

```

#### 54.4 lemmas for regularisation of ball and bex

```

lemma ball-reg-eqv:
  fixes P :: 'a ⇒ bool
  assumes a: equivp R

```

shows  $Ball\ (Respects\ R)\ P = (All\ P)$   
 using  $a$   
 unfolding  $equivp\text{-}def$   
 by  $(auto\ simp\ add:\ in\text{-}respects)$

lemma  $bex\text{-}reg\text{-}eqv$ :  
 fixes  $P :: 'a \Rightarrow bool$   
 assumes  $a: equivp\ R$   
 shows  $Bex\ (Respects\ R)\ P = (Ex\ P)$   
 using  $a$   
 unfolding  $equivp\text{-}def$   
 by  $(auto\ simp\ add:\ in\text{-}respects)$

lemma  $ball\text{-}reg\text{-}right$ :  
 assumes  $a: \bigwedge x. R\ x \Longrightarrow P\ x \longrightarrow Q\ x$   
 shows  $All\ P \longrightarrow Ball\ R\ Q$   
 using  $a$  by  $(metis\ COMBC\text{-}def\ Collect\text{-}def\ Collect\text{-}mem\text{-}eq)$

lemma  $bex\text{-}reg\text{-}left$ :  
 assumes  $a: \bigwedge x. R\ x \Longrightarrow Q\ x \longrightarrow P\ x$   
 shows  $Bex\ R\ Q \longrightarrow Ex\ P$   
 using  $a$  by  $(metis\ COMBC\text{-}def\ Collect\text{-}def\ Collect\text{-}mem\text{-}eq)$

lemma  $ball\text{-}reg\text{-}left$ :  
 assumes  $a: equivp\ R$   
 shows  $(\bigwedge x. (Q\ x \longrightarrow P\ x)) \Longrightarrow Ball\ (Respects\ R)\ Q \longrightarrow All\ P$   
 using  $a$  by  $(metis\ equivp\text{-}reflp\ in\text{-}respects)$

lemma  $bex\text{-}reg\text{-}right$ :  
 assumes  $a: equivp\ R$   
 shows  $(\bigwedge x. (Q\ x \longrightarrow P\ x)) \Longrightarrow Ex\ Q \longrightarrow Bex\ (Respects\ R)\ P$   
 using  $a$  by  $(metis\ equivp\text{-}reflp\ in\text{-}respects)$

lemma  $ball\text{-}reg\text{-}eqv\text{-}range$ :  
 fixes  $P :: 'a \Rightarrow bool$   
 and  $x :: 'a$   
 assumes  $a: equivp\ R2$   
 shows  $(Ball\ (Respects\ (R1 ==> R2))\ (\lambda f. P\ (f\ x))) = All\ (\lambda f. P\ (f\ x))$   
 apply  $(rule\ iffI)$   
 apply  $(rule\ allI)$   
 apply  $(drule\ tac\ x=\lambda y. f\ x\ in\ bspec)$   
 apply  $(simp\ add:\ in\text{-}respects)$   
 apply  $(rule\ impI)$   
 using  $a\ equivp\text{-}reflp\text{-}symp\text{-}transp[of\ R2]$   
 apply  $(simp\ add:\ reflp\text{-}def)$   
 apply  $(simp)$   
 apply  $(simp)$   
 done

```

lemma bex-reg-equiv-range:
  assumes a: equivp R2
  shows (Bex (Respects (R1  $\implies$  R2)) ( $\lambda f. P (f x)$ ) = Ex ( $\lambda f. P (f x)$ ))
  apply(auto)
  apply(rule-tac x= $\lambda y. f x$  in bexI)
  apply(simp)
  apply(simp add: Respects-def in-respects)
  apply(rule impI)
  using a equivp-reflp-symp-transp[of R2]
  apply(simp add: reflp-def)
  done

```

```

lemma all-reg:
  assumes a:  $\!x :: 'a. (P x \longrightarrow Q x)$ 
  and b: All P
  shows All Q
  using a b by (metis)

```

```

lemma ex-reg:
  assumes a:  $\!x :: 'a. (P x \longrightarrow Q x)$ 
  and b: Ex P
  shows Ex Q
  using a b by metis

```

```

lemma ball-reg:
  assumes a:  $\!x :: 'a. (R x \longrightarrow P x \longrightarrow Q x)$ 
  and b: Ball R P
  shows Ball R Q
  using a b by (metis COMBC-def Collect-def Collect-mem-eq)

```

```

lemma bex-reg:
  assumes a:  $\!x :: 'a. (R x \longrightarrow P x \longrightarrow Q x)$ 
  and b: Bex R P
  shows Bex R Q
  using a b by (metis COMBC-def Collect-def Collect-mem-eq)

```

```

lemma ball-all-comm:
  assumes  $\bigwedge y. (\forall x \in P. A x y) \longrightarrow (\forall x. B x y)$ 
  shows  $(\forall x \in P. \forall y. A x y) \longrightarrow (\forall x. \forall y. B x y)$ 
  using assms by auto

```

```

lemma bex-ex-comm:
  assumes  $(\exists y. \exists x. A x y) \longrightarrow (\exists y. \exists x \in P. B x y)$ 
  shows  $(\exists x. \exists y. A x y) \longrightarrow (\exists x \in P. \exists y. B x y)$ 
  using assms by auto

```

### 54.5 Bounded abstraction

**definition**

$Babs :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

**where**

$x \in p \Longrightarrow Babs\ p\ m\ x = m\ x$

**lemma** *babs-rsp*:

**assumes**  $q$ : *Quotient*  $R1$   $Abs1$   $Rep1$

**and**  $a$ :  $(R1 \Longrightarrow R2) f\ g$

**shows**  $(R1 \Longrightarrow R2) (Babs\ (Respects\ R1)\ f) (Babs\ (Respects\ R1)\ g)$

**apply** (*auto simp add: Babs-def in-respects*)

**apply** (*subgoal-tac*  $x \in Respects\ R1 \wedge y \in Respects\ R1$ )

**using**  $a$  **apply** (*simp add: Babs-def*)

**apply** (*simp add: in-respects*)

**using** *Quotient-rel*[*OF*  $q$ ]

**by** *metis*

**lemma** *babs-prs*:

**assumes**  $q1$ : *Quotient*  $R1$   $Abs1$   $Rep1$

**and**  $q2$ : *Quotient*  $R2$   $Abs2$   $Rep2$

**shows**  $((Rep1 \dashrightarrow Abs2) (Babs\ (Respects\ R1) ((Abs1 \dashrightarrow Rep2)\ f))) = f$

**apply** (*rule ext*)

**apply** (*simp*)

**apply** (*subgoal-tac*  $Rep1\ x \in Respects\ R1$ )

**apply** (*simp add: Babs-def Quotient-abs-rep*[*OF*  $q1$ ] *Quotient-abs-rep*[*OF*  $q2$ ])

**apply** (*simp add: in-respects Quotient-rel-rep*[*OF*  $q1$ ])

**done**

**lemma** *babs-simp*:

**assumes**  $q$ : *Quotient*  $R1$   $Abs$   $Rep$

**shows**  $((R1 \Longrightarrow R2) (Babs\ (Respects\ R1)\ f) (Babs\ (Respects\ R1)\ g)) = ((R1 \Longrightarrow R2)\ f\ g)$

**apply** (*rule iffI*)

**apply** (*simp-all only: babs-rsp*[*OF*  $q$ ])

**apply** (*auto simp add: Babs-def*)

**apply** (*subgoal-tac*  $x \in Respects\ R1 \wedge y \in Respects\ R1$ )

**apply** (*metis Babs-def*)

**apply** (*simp add: in-respects*)

**using** *Quotient-rel*[*OF*  $q$ ]

**by** *metis*

**lemma** *babs-reg-eqv*:

**shows** *equivp*  $R \Longrightarrow Babs\ (Respects\ R)\ P = P$

**by** (*simp add: expand-fun-eq Babs-def in-respects equivp-reflp*)

**lemma** *ball-rsp*:

**assumes**  $a$ :  $(R == => (op =)) f g$   
**shows**  $Ball (Respects R) f = Ball (Respects R) g$   
**using**  $a$  **by** (*simp add: Ball-def in-respects*)

**lemma** *bex-rsp*:

**assumes**  $a$ :  $(R == => (op =)) f g$   
**shows**  $(Bex (Respects R) f = Bex (Respects R) g)$   
**using**  $a$  **by** (*simp add: Bex-def in-respects*)

**lemma** *bex1-rsp*:

**assumes**  $a$ :  $(R == => (op =)) f g$   
**shows**  $Ex1 (\lambda x. x \in Respects R \wedge f x) = Ex1 (\lambda x. x \in Respects R \wedge g x)$   
**using**  $a$   
**by** (*simp add: Ex1-def in-respects*) *auto*

**lemma** *all-prs*:

**assumes**  $a$ : *Quotient*  $R$  *absf repf*  
**shows**  $Ball (Respects R) ((absf ---> id) f) = All f$   
**using**  $a$  **unfolding** *Quotient-def Ball-def in-respects fun-map-def id-apply*  
**by** *metis*

**lemma** *ex-prs*:

**assumes**  $a$ : *Quotient*  $R$  *absf repf*  
**shows**  $Bex (Respects R) ((absf ---> id) f) = Ex f$   
**using**  $a$  **unfolding** *Quotient-def Bex-def in-respects fun-map-def id-apply*  
**by** *metis*

## 54.6 *Bex1-rel* quantifier

**definition**

$Bex1-rel :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

**where**

$Bex1-rel R P \longleftrightarrow (\exists x \in Respects R. P x) \wedge (\forall x \in Respects R. \forall y \in Respects R. ((P x \wedge P y) \longrightarrow (R x y)))$

**lemma** *bex1-rel-aux*:

$\llbracket \forall xa ya. R xa ya \longrightarrow x xa = y ya; Bex1-rel R x \rrbracket \Longrightarrow Bex1-rel R y$   
**unfolding** *Bex1-rel-def*  
**apply** (*erule conjE*)  
**apply** (*erule bexE*)  
**apply** *rule*  
**apply** (*rule-tac x=xa in bexI*)  
**apply** *metis*  
**apply** *metis*  
**apply** *rule*  
**apply** (*erule-tac x=xaa in ballE*)  
**prefer** 2



```

apply (metis)
apply (erule-tac x=ya in ballE)
prefer 2
apply (metis)
apply (metis in-respects)
done

```

**lemma** *bex1-rel-aux2*:

```

 $\llbracket \forall xa ya. R xa ya \longrightarrow x xa = y ya; Bex1-rel R y \rrbracket \Longrightarrow Bex1-rel R x$ 
unfolding Bex1-rel-def
apply (erule conjE)+
apply (erule bexE)
apply rule
apply (rule-tac x=xa in bexI)
apply metis
apply metis
apply rule+
apply (erule-tac x=xaa in ballE)
prefer 2
apply (metis)
apply (erule-tac x=ya in ballE)
prefer 2
apply (metis)
apply (metis in-respects)
done

```

**lemma** *bex1-rel-rsp*:

```

assumes a: Quotient R absf repf
shows ((R ==> op =) ==> op =) (Bex1-rel R) (Bex1-rel R)
apply simp
apply clarify
apply rule
apply (simp-all add: bex1-rel-aux bex1-rel-aux2)
apply (erule bex1-rel-aux2)
apply assumption
done

```

**lemma** *ex1-prs*:

```

assumes a: Quotient R absf repf
shows ((absf ---> id) ---> id) (Bex1-rel R) f = Ex1 f
apply simp
apply (subst Bex1-rel-def)
apply (subst Bex-def)
apply (subst Ex1-def)
apply simp
apply rule
apply (erule conjE)+
apply (erule-tac exE)

```

```

apply (erule conjE)
apply (subgoal-tac  $\forall y. R\ y\ y \longrightarrow f\ (absf\ y) \longrightarrow R\ x\ y$ )
apply (rule-tac  $x=absf\ x$  in exI)
apply (simp)
apply rule+
using a unfolding Quotient-def
apply metis
apply rule+
apply (erule-tac  $x=x$  in ballE)
apply (erule-tac  $x=y$  in ballE)
apply simp
apply (simp add: in-respects)
apply (simp add: in-respects)
apply (erule-tac exE)
apply rule
apply (rule-tac  $x=repf\ x$  in exI)
apply (simp only: in-respects)
apply rule
apply (metis Quotient-rel-rep[OF a])
using a unfolding Quotient-def apply (simp)
apply rule+
using a unfolding Quotient-def in-respects
apply metis
done

```

```

lemma bex1-bexeq-reg: ( $\exists !x \in Respects\ R. P\ x$ )  $\longrightarrow$  (Bex1-rel  $R\ (\lambda x. P\ x)$ )
apply (simp add: Ex1-def Bex1-rel-def in-respects)
apply clarify
apply auto
apply (rule bexI)
apply assumption
apply (simp add: in-respects)
apply (simp add: in-respects)
apply auto
done

```

## 54.7 Various respects and preserve lemmas

```

lemma quot-rel-rsp:
  assumes a: Quotient  $R\ Abs\ Rep$ 
  shows ( $R\ ==> R\ ==> op =$ )  $R\ R$ 
  apply (rule fun-rel-id)+
  apply (rule equals-rsp[OF a])
  apply (assumption)+
  done

```

```

lemma o-prs:
  assumes q1: Quotient  $R1\ Abs1\ Rep1$ 
  and q2: Quotient  $R2\ Abs2\ Rep2$ 

```

```

and    q3: Quotient R3 Abs3 Rep3
shows ((Abs2 ----> Rep3) ----> (Abs1 ----> Rep2) ----> (Rep1 ---->
Abs3)) op ◦ = op ◦
and    (id ----> (Abs1 ----> id) ----> Rep1 ----> id) op ◦ = op ◦
using Quotient-abs-rep[OF q1] Quotient-abs-rep[OF q2] Quotient-abs-rep[OF q3]
unfolding o-def expand-fun-eq by simp-all

```

**lemma** *o-rsp*:

```

((R2 ===> R3) ===> (R1 ===> R2) ===> (R1 ===> R3)) op ◦ op ◦
(op ===> (R1 ===> op =) ===> R1 ===> op =) op ◦ op ◦
unfolding fun-rel-def o-def expand-fun-eq by auto

```

**lemma** *cond-prs*:

```

assumes a: Quotient R absf repf
shows absf (if a then repf b else repf c) = (if a then b else c)
using a unfolding Quotient-def by auto

```

**lemma** *if-prs*:

```

assumes q: Quotient R Abs Rep
shows (id ----> Rep ----> Rep ----> Abs) If = If
using Quotient-abs-rep[OF q]
by (auto simp add: expand-fun-eq)

```

**lemma** *if-rsp*:

```

assumes q: Quotient R Abs Rep
shows (op ===> R ===> R ===> R) If If
by auto

```

**lemma** *let-prs*:

```

assumes q1: Quotient R1 Abs1 Rep1
and    q2: Quotient R2 Abs2 Rep2
shows (Rep2 ----> (Abs2 ----> Rep1) ----> Abs1) Let = Let
using Quotient-abs-rep[OF q1] Quotient-abs-rep[OF q2]
by (auto simp add: expand-fun-eq)

```

**lemma** *let-rsp*:

```

shows (R1 ===> (R1 ===> R2) ===> R2) Let Let
by auto

```

**locale** *quot-type* =

```

fixes R :: 'a ⇒ 'a ⇒ bool
and    Abs :: ('a ⇒ bool) ⇒ 'b
and    Rep :: 'b ⇒ ('a ⇒ bool)
assumes equivp: equivp R
and    rep-prop: ⋀y. ∃x. Rep y = R x
and    rep-inverse: ⋀x. Abs (Rep x) = x
and    abs-inverse: ⋀x. (Rep (Abs (R x))) = (R x)
and    rep-inject: ⋀x y. (Rep x = Rep y) = (x = y)
begin

```

**definition**

$abs::'a \Rightarrow 'b$

**where**

$abs\ x \equiv Abs\ (R\ x)$

**definition**

$rep::'b \Rightarrow 'a$

**where**

$rep\ a = Eps\ (Rep\ a)$

**lemma** *homeier-lem9*:

**shows**  $R\ (Eps\ (R\ x)) = R\ x$

**proof** –

**have**  $a: R\ x\ x$  **using** *equivp* **by** (*simp add: equivp-reflp-symp-transp reflp-def*)

**then have**  $R\ x\ (Eps\ (R\ x))$  **by** (*rule someI*)

**then show**  $R\ (Eps\ (R\ x)) = R\ x$

**using** *equivp unfolding equivp-def* **by** *simp*

**qed**

**theorem** *homeier-thm10*:

**shows**  $abs\ (rep\ a) = a$

**unfolding** *abs-def rep-def*

**proof** –

**from** *rep-prop*

**obtain**  $x$  **where**  $eq: Rep\ a = R\ x$  **by** *auto*

**have**  $Abs\ (R\ (Eps\ (Rep\ a))) = Abs\ (R\ (Eps\ (R\ x)))$  **using** *eq* **by** *simp*

**also have**  $\dots = Abs\ (R\ x)$  **using** *homeier-lem9* **by** *simp*

**also have**  $\dots = Abs\ (Rep\ a)$  **using** *eq* **by** *simp*

**also have**  $\dots = a$  **using** *rep-inverse* **by** *simp*

**finally**

**show**  $Abs\ (R\ (Eps\ (Rep\ a))) = a$  **by** *simp*

**qed**

**lemma** *homeier-lem7*:

**shows**  $(R\ x = R\ y) = (Abs\ (R\ x) = Abs\ (R\ y))$  (**is** *?LHS = ?RHS*)

**proof** –

**have** *?RHS*  $= (Rep\ (Abs\ (R\ x)) = Rep\ (Abs\ (R\ y)))$  **by** (*simp add: rep-inject*)

**also have**  $\dots = ?LHS$  **by** (*simp add: abs-inverse*)

**finally show** *?LHS = ?RHS* **by** *simp*

**qed**

**theorem** *homeier-thm11*:

**shows**  $R\ r\ r' = (abs\ r = abs\ r')$

**unfolding** *abs-def*

**by** (*simp only: equivp[simplified equivp-def] homeier-lem7*)

**lemma** *rep-refl*:

**shows**  $R\ (rep\ a)\ (rep\ a)$

```

unfolding rep-def
by (simp add: equivp[simplified equivp-def])

lemma rep-abs-rsp:
  shows  $R\ f\ (rep\ (abs\ g)) = R\ f\ g$ 
  and  $R\ (rep\ (abs\ g))\ f = R\ g\ f$ 
  by (simp-all add: homeier-thm10 homeier-thm11)

lemma Quotient:
  shows Quotient  $R\ abs\ rep$ 
  unfolding Quotient-def
  apply (simp add: homeier-thm10)
  apply (simp add: rep-refl)
  apply (subst homeier-thm11[symmetric])
  apply (simp add: equivp[simplified equivp-def])
  done

end

```

## 54.8 ML setup

Auxiliary data for the quotient package

```
use ~/src/HOL/Tools/Quotient/quotient-info.ML
```

```
declare [[map fun = (fun-map, fun-rel)]]
```

```

lemmas [quot-thm] = fun-quotient
lemmas [quot-respect] = quot-rel-rsp if-rsp o-rsp let-rsp
lemmas [quot-preserve] = if-prs o-prs let-prs
lemmas [quot-equiv] = identity-equivp

```

Lemmas about simplifying id's.

```

lemmas [id-simps] =
  id-def[symmetric]
  fun-map-id
  id-apply
  id-o
  o-id
  eq-comp-r

```

Translation functions for the lifting process.

```
use ~/src/HOL/Tools/Quotient/quotient-term.ML
```

Definitions of the quotient types.

```
use ~/src/HOL/Tools/Quotient/quotient-typ.ML
```

Definitions for quotient constants.

**use** `~~/src/HOL/Tools/Quotient/quotient-def.ML`

An auxiliary constant for recording some information about the lifted theorem in a tactic.

**definition**

$Quot-True\ (x :: 'a) \equiv True$

**lemma**

**shows**  $QT-all: Quot-True\ (All\ P) \implies Quot-True\ P$   
**and**  $QT-ex: Quot-True\ (Ex\ P) \implies Quot-True\ P$   
**and**  $QT-ex1: Quot-True\ (Ex1\ P) \implies Quot-True\ P$   
**and**  $QT-lam: Quot-True\ (\lambda x. P\ x) \implies (\bigwedge x. Quot-True\ (P\ x))$   
**and**  $QT-ext: (\bigwedge x. Quot-True\ (a\ x) \implies f\ x = g\ x) \implies (Quot-True\ a \implies f = g)$   
**by**  $(simp-all\ add: Quot-True-def\ ext)$

**lemma**  $QT-imp: Quot-True\ a \equiv Quot-True\ b$

**by**  $(simp\ add: Quot-True-def)$

Tactics for proving the lifted theorems

**use** `~~/src/HOL/Tools/Quotient/quotient-tacs.ML`

## 54.9 Methods / Interface

**method-setup** *lifting* =

$\ll\ \text{Attrib.thms} \gg (\text{fn thms} \Rightarrow \text{fn ctxt} \Rightarrow \text{SIMPLE-METHOD}\ (\text{HEADGOAL}\ (\text{Quotient-Tacs.lift-tac}\ \text{ctxt}\ \text{thms}))) \gg$   
 $\ll\ \text{lifts theorems to quotient types} \gg$

**method-setup** *lifting-setup* =

$\ll\ \text{Attrib.thm} \gg (\text{fn thms} \Rightarrow \text{fn ctxt} \Rightarrow \text{SIMPLE-METHOD}\ (\text{HEADGOAL}\ (\text{Quotient-Tacs.procedure-tac}\ \text{ctxt}\ \text{thms}))) \gg$   
 $\ll\ \text{sets up the three goals for the quotient lifting procedure} \gg$

**method-setup** *regularize* =

$\ll\ \text{Scan.succeed}\ (\text{fn ctxt} \Rightarrow \text{SIMPLE-METHOD}\ (\text{HEADGOAL}\ (\text{Quotient-Tacs.regularize-tac}\ \text{ctxt}))) \gg$   
 $\ll\ \text{proves the regularization goals from the quotient lifting procedure} \gg$

**method-setup** *injection* =

$\ll\ \text{Scan.succeed}\ (\text{fn ctxt} \Rightarrow \text{SIMPLE-METHOD}\ (\text{HEADGOAL}\ (\text{Quotient-Tacs.all-injection-tac}\ \text{ctxt}))) \gg$   
 $\ll\ \text{proves the rep/abs injection goals from the quotient lifting procedure} \gg$

**method-setup** *cleaning* =

$\ll\ \text{Scan.succeed}\ (\text{fn ctxt} \Rightarrow \text{SIMPLE-METHOD}\ (\text{HEADGOAL}\ (\text{Quotient-Tacs.clean-tac}\ \text{ctxt}))) \gg$   
 $\ll\ \text{proves the cleaning goals from the quotient lifting procedure} \gg$

**no-notation**  
*rel-conj* (**infixr** *OOO* 75) and  
*fun-map* (**infixr** *---* 55) and  
*fun-rel* (**infixr** *==* 55)

```

theory Refute
imports Hilbert-Choice List
uses
  Tools/refute.ML
  Tools/refute-isar.ML
begin

```

```
(* ----- *)
(* REFUTE                                     *)
(*                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given *)
(* HOL formula.                             *)
(* ----- *)

(* ----- *)
(* NOTE                                     *)
(*                                           *)
(* I strongly recommend that you install a stand-alone SAT solver if you *)
(* want to use 'refute'. For details see 'HOL/Tools/sat_solver.ML'. If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME' *)
(* in 'etc/settings'.                     *)
(* ----- *)

(* ----- *)
(* USAGE                                     *)
(*                                           *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples. The supported *)
(* parameters are explained below.      *)
(* ----- *)

(* ----- *)
```

```

(* CURRENT LIMITATIONS *)
(*
(* 'refute' currently accepts formulas of higher-order predicate logic (with
(* equality), including free/bound/schematic variables, lambda abstractions,
(* sets and set membership, "arbitrary", "The", "Eps", records and
(* inductively defined sets. Constants are unfolded automatically, and sort
(* axioms are added as well. Other, user-asserted axioms however are
(* ignored. Inductive datatypes and recursive functions are supported, but
(* may lead to spurious countermodels.
(*
(* The (space) complexity of the algorithm is non-elementary.
(*
(* Schematic type variables are not supported.
(* ----- *)

(* ----- *)
(* PARAMETERS *)
(*
(* The following global parameters are currently supported (and required,
(* except for "expect"):
(*
(* Name          Type      Description
(*
(* "minsize"      int       Only search for models with size at least
(*                          'minsize'.
(*
(* "maxsize"      int       If >0, only search for models with size at most
(*                          'maxsize'.
(*
(* "maxvars"      int       If >0, use at most 'maxvars' boolean variables
(*                          when transforming the term into a propositional
(*                          formula.
(*
(* "maxtime"      int       If >0, terminate after at most 'maxtime' seconds.
(*                          This value is ignored under some ML compilers.
(*
(* "satsolver"    string    Name of the SAT solver to be used.
(*
(* "no_assms"     bool      If "true", assumptions in structured proofs are
(*                          not considered.
(*
(* "expect"       string    Expected result ("genuine", "potential", "none", or
(*                          "unknown").
(*
(* See 'HOL/SAT.thy' for default values.
(*
(* The size of particular types can be specified in the form type=size
(* (where 'type' is a string, and 'size' is an int). Examples:
(* "'a'=1
(* "List.list=2
(* ----- *)

(* ----- *)
(* FILES
(*

```



```

(* HOL/Tools/prop_logic.ML      Propositional logic      *)
(* HOL/Tools/sat_solver.ML      SAT solvers              *)
(* HOL/Tools/refute.ML          Translation HOL -> propositional logic and *)
(*                               Boolean assignment -> HOL model          *)
(* HOL/Tools/refute_isar.ML      Adds 'refute'/'refute_params' to Isabelle's *)
(*                               syntax                                      *)
(* HOL/Refute.thy                This file: loads the ML files, basic setup, *)
(*                               documentation                             *)
(* HOL/SAT.thy                  Sets default parameters          *)
(* HOL/ex/RefuteExamples.thy     Examples                      *)
(* ----- *)

```

end

## 56 SAT: Reconstructing external resolution proofs for propositional logic

```

theory SAT
imports Refute
uses
  Tools/cnf-funcs.ML
  Tools/sat-funcs.ML
begin

```

Late package setup: default values for refute, see also theory *Refute*.

**refute-params**

```

[itself=1,
 minsize=1,
 maxsize=8,
 maxvars=10000,
 maxtime=60,
 satsolver=auto,
 no-assms=false]

```

**ML**  $\ll$  *structure* *sat* = *SATFunc*(*cnf*)  $\gg$

**method-setup** *sat* =  $\ll$  *Scan.succeed* (*SIMPLE-METHOD'* *o* *sat.sat-tac*)  $\gg$   
*SAT solver*

**method-setup** *satx* =  $\ll$  *Scan.succeed* (*SIMPLE-METHOD'* *o* *sat.satx-tac*)  $\gg$   
*SAT solver (with definitional CNF)*

end

## 57 Nitpick: Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

```

theory Nitpick
imports Map Quotient SAT
uses (Tools/Nitpick/kodkod.ML)
      (Tools/Nitpick/kodkod-sat.ML)
      (Tools/Nitpick/nitpick-util.ML)
      (Tools/Nitpick/nitpick-hol.ML)
      (Tools/Nitpick/nitpick-preproc.ML)
      (Tools/Nitpick/nitpick-mono.ML)
      (Tools/Nitpick/nitpick-scope.ML)
      (Tools/Nitpick/nitpick-peephole.ML)
      (Tools/Nitpick/nitpick-rep.ML)
      (Tools/Nitpick/nitpick-nut.ML)
      (Tools/Nitpick/nitpick-kodkod.ML)
      (Tools/Nitpick/nitpick-model.ML)
      (Tools/Nitpick/nitpick.ML)
      (Tools/Nitpick/nitpick-isar.ML)
      (Tools/Nitpick/nitpick-tests.ML)
      (Tools/Nitpick/minipick.ML)
begin

typedecl bisim-iterator

axiomatization unknown :: 'a
  and is-unknown :: 'a  $\Rightarrow$  bool
  and undefined-fast-The :: 'a
  and undefined-fast-Eps :: 'a
  and bisim :: bisim-iterator  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  bool
  and bisim-iterator-max :: bisim-iterator
  and Quot :: 'a  $\Rightarrow$  'b
  and safe-The :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a
  and safe-Eps :: ('a  $\Rightarrow$  bool)  $\Rightarrow$  'a

datatype ('a, 'b) fin-fun = FinFun ('a  $\Rightarrow$  'b)
datatype ('a, 'b) fun-box = FunBox ('a  $\Rightarrow$  'b)
datatype ('a, 'b) pair-box = PairBox 'a 'b

typedecl unsigned-bit
typedecl signed-bit

datatype 'a word = Word ('a set)

Alternative definitions.

lemma If-def [nitpick-def, no-atp]:
  (if P then Q else R)  $\equiv$  (P  $\longrightarrow$  Q)  $\wedge$  ( $\neg$  P  $\longrightarrow$  R)
by (rule eq-reflection) (rule if-bool-eq-conj)

```

**lemma** *Ex1-def* [*nitpick-def*, *no-atp*]:

*Ex1*  $P \equiv \exists x. P = \{x\}$

**apply** (*rule eq-reflection*)

**apply** (*simp add: Ex1-def expand-set-eq*)

**apply** (*rule iffI*)

**apply** (*erule exE*)

**apply** (*erule conjE*)

**apply** (*rule-tac*  $x = x$  **in** *exI*)

**apply** (*rule allI*)

**apply** (*rename-tac* *y*)

**apply** (*erule-tac*  $x = y$  **in** *allE*)

**by** (*auto simp: mem-def*)

**lemma** *split-def* [*nitpick-def*]: *split*  $f = (\lambda p. f (fst\ p) (snd\ p))$

**by** (*simp add: prod-case-unfold split-def*)

**lemma** *rtrancl-def* [*nitpick-def*, *no-atp*]:  $r^* \equiv (r^+)^=$

**by** *simp*

**lemma** *rtranclp-def* [*nitpick-def*, *no-atp*]:

*rtranclp*  $r\ a\ b \equiv (a = b \vee \text{trancplp}\ r\ a\ b)$

**by** (*rule eq-reflection*) (*auto dest: rtranclpD*)

**lemma** *trancplp-def* [*nitpick-def*, *no-atp*]:

*trancplp*  $r\ a\ b \equiv \text{trancI}\ (\text{split}\ r)\ (a, b)$

**by** (*simp add: trancI-def Collect-def mem-def*)

**definition** *refl'* ::  $('a \times 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$  **where**

*refl'*  $r \equiv \forall x. (x, x) \in r$

**definition** *wf'* ::  $('a \times 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$  **where**

*wf'*  $r \equiv \text{acyclic}\ r \wedge (\text{finite}\ r \vee \text{unknown})$

**axiomatization** *wf-wfrec* ::  $('a \times 'a \Rightarrow \text{bool}) \Rightarrow (( 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

**definition** *wf-wfrec'* ::  $('a \times 'a \Rightarrow \text{bool}) \Rightarrow (( 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$  **where**

[*nitpick-simp*]: *wf-wfrec'*  $R\ F\ x = F\ (\text{Recdef.cut}\ (\text{wf-wfrec}\ R\ F)\ R\ x)\ x$

**definition** *wfrec'* ::  $('a \times 'a \Rightarrow \text{bool}) \Rightarrow (( 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$  **where**

*wfrec'*  $R\ F\ x \equiv \text{if}\ \text{wf}\ R\ \text{then}\ \text{wf-wfrec}'\ R\ F\ x$

*else THE y. wfrec-rel R (%f x. F (Recdef.cut f R x) x) x y*

**definition** *card'* ::  $('a \Rightarrow \text{bool}) \Rightarrow \text{nat}$  **where**

*card'*  $A \equiv \text{if}\ \text{finite}\ A\ \text{then}\ \text{length}\ (\text{safe-Eps}\ (\lambda xs. \text{set}\ xs = A \wedge \text{distinct}\ xs))\ \text{else}\ 0$

**definition** *setsum'* ::  $('a \Rightarrow 'b::\text{comm-monoid-add}) \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow 'b$  **where**

*setsum'*  $f\ A \equiv \text{if}\ \text{finite}\ A\ \text{then}\ \text{listsum}\ (\text{map}\ f\ (\text{safe-Eps}\ (\lambda xs. \text{set}\ xs = A \wedge \text{distinct}$

$xs)))$  else 0

**inductive** *fold-graph'* :: ('a  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  'b  $\Rightarrow$  bool **where**  
*fold-graph'* f z { } z |  
 $\llbracket x \in A; \text{fold-graph}' f z (A - \{x\}) y \rrbracket \Longrightarrow \text{fold-graph}' f z A (f x y)$

The following lemmas are not strictly necessary but they help the *special\_level* optimization.

**lemma** *The-psimp* [*nitpick-psimp*, *no-atp*]:  
 $P = \{x\} \Longrightarrow \text{The } P = x$   
**by** (*subgoal-tac* { $x$ } = ( $\lambda y. y = x$ )) (*auto simp: mem-def*)

**lemma** *Eps-psimp* [*nitpick-psimp*, *no-atp*]:  
 $\llbracket P x; \neg P y; \text{Eps } P = y \rrbracket \Longrightarrow \text{Eps } P = x$   
**apply** (*case-tac* P (*Eps* P))  
**apply** *auto*  
**apply** (*erule contrapos-np*)  
**by** (*rule someI*)

**lemma** *unit-case-def* [*nitpick-def*, *no-atp*]:  
 $\text{unit-case } x u \equiv x$   
**apply** (*subgoal-tac* u = ())  
**apply** (*simp only: unit.cases*)  
**by** *simp*

**declare** *unit.cases* [*nitpick-simp del*]

**lemma** *nat-case-def* [*nitpick-def*, *no-atp*]:  
 $\text{nat-case } x f n \equiv \text{if } n = 0 \text{ then } x \text{ else } f (n - 1)$   
**apply** (*rule eq-reflection*)  
**by** (*case-tac* n) *auto*

**declare** *nat.cases* [*nitpick-simp del*]

**lemma** *list-size-simp* [*nitpick-simp*, *no-atp*]:  
 $\text{list-size } f xs = (\text{if } xs = [] \text{ then } 0$   
 $\quad \text{else } \text{Suc } (f (\text{hd } xs) + \text{list-size } f (\text{tl } xs)))$   
 $\text{size } xs = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Suc } (\text{size } (\text{tl } xs)))$   
**by** (*case-tac* xs) *auto*

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

**function** *nat-gcd* :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat **where**  
[*simp del*]:  $\text{nat-gcd } x y = (\text{if } y = 0 \text{ then } x \text{ else } \text{nat-gcd } y (x \bmod y))$   
**by** *auto*  
**termination**  
**apply** (*relation measure* ( $\lambda(x, y). x + y + (\text{if } y > x \text{ then } 1 \text{ else } 0)$ ))  
**apply** *auto*  
**apply** (*metis mod-less-divisor xt1(9)*)

**by** (*metis mod-mod-trivial mod-self nat-neq-iff xt1 (10)*)

**definition** *nat-lcm* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat* **where**  
*nat-lcm* *x y* = *x* \* *y* div (*nat-gcd* *x y*)

**definition** *int-gcd* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int* **where**  
*int-gcd* *x y* = *int* (*nat-gcd* (*nat* (*abs* *x*)) (*nat* (*abs* *y*)))

**definition** *int-lcm* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int* **where**  
*int-lcm* *x y* = *int* (*nat-lcm* (*nat* (*abs* *x*)) (*nat* (*abs* *y*)))

**definition** *Frac* :: *int*  $\times$  *int*  $\Rightarrow$  *bool* **where**  
*Frac*  $\equiv \lambda(a, b). b > 0 \wedge \text{int-gcd } a \ b = 1$

**axiomatization** *Abs-Frac* :: *int*  $\times$  *int*  $\Rightarrow$  '*a*  
**and** *Rep-Frac* :: '*a*  $\Rightarrow$  *int*  $\times$  *int*

**definition** *zero-frac* :: '*a* **where**  
*zero-frac*  $\equiv \text{Abs-Frac } (0, 1)$

**definition** *one-frac* :: '*a* **where**  
*one-frac*  $\equiv \text{Abs-Frac } (1, 1)$

**definition** *num* :: '*a*  $\Rightarrow$  *int* **where**  
*num*  $\equiv \text{fst } o \ \text{Rep-Frac}$

**definition** *denom* :: '*a*  $\Rightarrow$  *int* **where**  
*denom*  $\equiv \text{snd } o \ \text{Rep-Frac}$

**function** *norm-frac* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int*  $\times$  *int* **where**  
 $[\text{simp del}]: \text{norm-frac } a \ b = (\text{if } b < 0 \text{ then } \text{norm-frac } (-a) \ (-b) \\ \text{else if } a = 0 \vee b = 0 \text{ then } (0, 1) \\ \text{else let } c = \text{int-gcd } a \ b \text{ in } (a \ \text{div } c, b \ \text{div } c))$

**by** *pat-completeness auto*

**termination by** (*relation measure* ( $\lambda(-, b). \text{if } b < 0 \text{ then } 1 \text{ else } 0$ )) *auto*

**definition** *frac* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  '*a* **where**  
*frac* *a b*  $\equiv \text{Abs-Frac } (\text{norm-frac } a \ b)$

**definition** *plus-frac* :: '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a* **where**  
 $[\text{nitpick-simp}]:$   
*plus-frac* *q r* = (let *d* = *int-lcm* (*denom* *q*) (*denom* *r*) in  
*frac* (*num* *q* \* (*d* div *denom* *q*) + *num* *r* \* (*d* div *denom* *r*)) *d*)

**definition** *times-frac* :: '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a* **where**  
 $[\text{nitpick-simp}]:$   
*times-frac* *q r* = *frac* (*num* *q* \* *num* *r*) (*denom* *q* \* *denom* *r*)

**definition** *uminus-frac* :: '*a*  $\Rightarrow$  '*a* **where**

$uminus\text{-}frac\ q \equiv Abs\text{-}Frac\ (-\ num\ q,\ denom\ q)$

**definition**  $number\text{-}of\text{-}frac :: int \Rightarrow 'a$  **where**  
 $number\text{-}of\text{-}frac\ n \equiv Abs\text{-}Frac\ (n,\ 1)$

**definition**  $inverse\text{-}frac :: 'a \Rightarrow 'a$  **where**  
 $inverse\text{-}frac\ q \equiv frac\ (denom\ q)\ (num\ q)$

**definition**  $less\text{-}eq\text{-}frac :: 'a \Rightarrow 'a \Rightarrow bool$  **where**  
 $[nitpick\text{-}simp]:$   
 $less\text{-}eq\text{-}frac\ q\ r \longleftrightarrow num\ (plus\text{-}frac\ q\ (uminus\text{-}frac\ r)) \leq 0$

**definition**  $of\text{-}frac :: 'a \Rightarrow 'b::\{inverse,ring-1\}$  **where**  
 $of\text{-}frac\ q \equiv of\text{-}int\ (num\ q)\ /\ of\text{-}int\ (denom\ q)$

**use** *Tools/Nitpick/kodkod.ML*  
**use** *Tools/Nitpick/kodkod-sat.ML*  
**use** *Tools/Nitpick/nitpick-util.ML*  
**use** *Tools/Nitpick/nitpick-hol.ML*  
**use** *Tools/Nitpick/nitpick-mono.ML*  
**use** *Tools/Nitpick/nitpick-preproc.ML*  
**use** *Tools/Nitpick/nitpick-scope.ML*  
**use** *Tools/Nitpick/nitpick-peephole.ML*  
**use** *Tools/Nitpick/nitpick-rep.ML*  
**use** *Tools/Nitpick/nitpick-nut.ML*  
**use** *Tools/Nitpick/nitpick-kodkod.ML*  
**use** *Tools/Nitpick/nitpick-model.ML*  
**use** *Tools/Nitpick/nitpick.ML*  
**use** *Tools/Nitpick/nitpick-isar.ML*  
**use** *Tools/Nitpick/nitpick-tests.ML*  
**use** *Tools/Nitpick/minipick.ML*

**setup**  $\ll Nitpick\text{-}Isar.setup \gg$

**hide-const (open)** *unknown is-unknown undefined-fast-The undefined-fast-Eps bisim*

*bisim-iterator-max Quot safe-The safe-Eps FinFun FunBox PairBox Word refl'*  
*wf' wf-wfrec wf-wfrec' wfrec' card' setsum' fold-graph' nat-gcd nat-lcm*  
*int-gcd int-lcm Frac Abs-Frac Rep-Frac zero-frac one-frac num denom*  
*norm-frac frac plus-frac times-frac uminus-frac number-of-frac inverse-frac*  
*less-eq-frac of-frac*

**hide-type (open)** *bisim-iterator fin-fun fun-box pair-box unsigned-bit signed-bit word*

**hide-fact (open)** *If-def Ex1-def split-def rtrancl-def rtranclp-def tranclp-def*  
*refl'-def wf'-def wf-wfrec'-def wfrec'-def card'-def setsum'-def*  
*fold-graph'-def The-psimp Eps-psimp unit-case-def nat-case-def*  
*list-size-simp nat-gcd-def nat-lcm-def int-gcd-def int-lcm-def Frac-def*  
*zero-frac-def one-frac-def num-def denom-def norm-frac-def frac-def*  
*plus-frac-def times-frac-def uminus-frac-def number-of-frac-def*

*inverse-frac-def less-eq-frac-def of-frac-def*

**end**

## 58 SMT: Bindings to Satisfiability Modulo Theories (SMT) solvers

```
theory SMT
imports List
uses
  ~~/src/Tools/cache-io.ML
  (Tools/SMT/smt-monomorph.ML)
  (Tools/SMT/smt-normalize.ML)
  (Tools/SMT/smt-translate.ML)
  (Tools/SMT/smt-solver.ML)
  (Tools/SMT/smtlib-interface.ML)
  (Tools/SMT/z3-proof-parser.ML)
  (Tools/SMT/z3-proof-tools.ML)
  (Tools/SMT/z3-proof-literals.ML)
  (Tools/SMT/z3-proof-reconstruction.ML)
  (Tools/SMT/z3-model.ML)
  (Tools/SMT/z3-interface.ML)
  (Tools/SMT/z3-solver.ML)
  (Tools/SMT/cvc3-solver.ML)
  (Tools/SMT/yices-solver.ML)
begin
```

### 58.1 Triggers for quantifier instantiation

Some SMT solvers support triggers for quantifier instantiation. Each trigger consists of one or more patterns. A pattern may either be a list of positive subterms (each being tagged by "pat"), or a list of negative subterms (each being tagged by "nopat").

When an SMT solver finds a term matching a positive pattern (a pattern with positive subterms only), it instantiates the corresponding quantifier accordingly. Negative patterns inhibit quantifier instantiations. Each pattern should mention all preceding bound variables.

**datatype** *pattern* = *Pattern*

**definition** *pat* :: 'a  $\Rightarrow$  *pattern* **where** *pat* - = *Pattern*

**definition** *nopat* :: 'a  $\Rightarrow$  *pattern* **where** *nopat* - = *Pattern*

**definition** *trigger* :: *pattern list list*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool*

**where** *trigger* - *P* = *P*

## 58.2 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

**definition** *fun-app* **where** *fun-app*  $f\ x = f\ x$

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

**lemmas** *array-rules* = *ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def*

## 58.3 First-order logic

Some SMT solvers require a strict separation between formulas and terms. When translating higher-order into first-order problems, all uninterpreted constants (those not builtin in the target solver) are treated as function symbols in the first-order sense. Their occurrences as head symbols in atoms (i.e., as predicate symbols) is turned into terms by equating such atoms with *True* using the following term-level equation symbol.

**definition** *term-eq* :: *bool*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool* **where** *term-eq*  $x\ y = (x = y)$

## 58.4 Integer division and modulo for Z3

**definition** *z3div* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int* **where**  
*z3div*  $k\ l = (\text{if } 0 \leq l \text{ then } k \text{ div } l \text{ else } -(k \text{ div } (-l)))$

**definition** *z3mod* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int* **where**  
*z3mod*  $k\ l = (\text{if } 0 \leq l \text{ then } k \text{ mod } l \text{ else } k \text{ mod } (-l))$

**lemma** *div-by-z3div*:  $k \text{ div } l =$  (  
   *if*  $k = 0 \vee l = 0$  *then* 0  
   *else if*  $(0 < k \wedge 0 < l) \vee (k < 0 \wedge 0 < l)$  *then* *z3div*  $k\ l$   
   *else* *z3div*  $(-k)\ (-l)$ )  
**by** (*auto simp add: z3div-def*)

**lemma** *mod-by-z3mod*:  $k \text{ mod } l =$  (  
   *if*  $l = 0$  *then*  $k$   
   *else if*  $k = 0$  *then* 0  
   *else if*  $(0 < k \wedge 0 < l) \vee (k < 0 \wedge 0 < l)$  *then* *z3mod*  $k\ l$   
   *else*  $- \text{z3mod } (-k)\ (-l)$ )  
**by** (*auto simp add: z3mod-def*)

## 58.5 Setup

**use** *Tools/SMT/smt-monomorph.ML*  
**use** *Tools/SMT/smt-normalize.ML*



```

use Tools/SMT/smt-translate.ML
use Tools/SMT/smt-solver.ML
use Tools/SMT/smtlib-interface.ML
use Tools/SMT/z3-interface.ML
use Tools/SMT/z3-proof-parser.ML
use Tools/SMT/z3-proof-tools.ML
use Tools/SMT/z3-proof-literals.ML
use Tools/SMT/z3-proof-reconstruction.ML
use Tools/SMT/z3-model.ML
use Tools/SMT/z3-solver.ML
use Tools/SMT/cvc3-solver.ML
use Tools/SMT/yices-solver.ML

setup ⟨⟨
  SMT-Solver.setup #>
  Z3-Proof-Reconstruction.setup #>
  Z3-Solver.setup #>
  CVC3-Solver.setup #>
  Yices-Solver.setup
⟩⟩

```

## 58.6 Configuration

The current configuration can be printed by the command *smt-status*, which shows the values of most options.

## 58.7 General configuration options

The option *smt-solver* can be used to change the target SMT solver. The possible values are *cvc3*, *yices*, and *z3*. It is advisable to locally install the selected solver, although this is not necessary for *cvc3* and *z3*, which can also be used over an Internet-based service.

When using local SMT solvers, the path to their binaries should be declared by setting the following environment variables: *CVC3-SOLVER*, *YICES-SOLVER*, and *Z3-SOLVER*.

```
declare [[ smt-solver = z3 ]]
```

Since SMT solvers are potentially non-terminating, there is a timeout (given in seconds) to restrict their runtime. A value greater than 120 (seconds) is in most cases not advisable.

```
declare [[ smt-timeout = 20 ]]
```

## 58.8 Certificates

By setting the option *smt-certificates* to the name of a file, all following applications of an SMT solver are cached in that file. Any further application

of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending *.certs* instead of *.thy*) as the certificates file.

```
declare [[ smt-certificates =  ]]
```

The option *smt-fixed* controls whether only stored certificates are should be used or invocation of an SMT solver is allowed. When set to *true*, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to *false* and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

```
declare [[ smt-fixed = false ]]
```

## 58.9 Tracing

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option *smt-trace* should be set to *true*.

```
declare [[ smt-trace = false ]]
```

## 58.10 Z3-specific options

Z3 is the only SMT solver whose proofs are checked (or reconstructed) in Isabelle (all other solvers are implemented as oracles). Enabling or disabling proof reconstruction for Z3 is controlled by the option *z3-proofs*.

```
declare [[ z3-proofs = true ]]
```

From the set of assumptions given to Z3, those assumptions used in the proof are traced when the option *z3-trace-assms* is set to *true*.

```
declare [[ z3-trace-assms = false ]]
```

Z3 provides several commandline options to tweak its behaviour. They can be configured by writing them literally as value for the option *z3-options*.

```
declare [[ z3-options =  ]]
```

## 58.11 Schematic rules for Z3 proof reconstruction

Several proof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in *z3-rule* are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in *z3-simp* are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

**lemmas** [z3-rule] =  
*refl eq-commute conj-commute disj-commute simp-thms nnf-simps*  
*ring-distrib field-simps times-divide-eq-right times-divide-eq-left*  
*if-True if-False not-not*

**lemma** [z3-rule]:  
 $(P \longrightarrow Q) = (Q \vee \neg P)$   
 $(\neg P \longrightarrow Q) = (P \vee Q)$   
 $(\neg P \longrightarrow Q) = (Q \vee P)$   
**by** *auto*

**lemma** [z3-rule]:  
 $((P = Q) \longrightarrow R) = (R \mid (Q = (\neg P)))$   
**by** *auto*

**lemma** [z3-rule]:  
 $((\neg P) = P) = \text{False}$   
 $(P = (\neg P)) = \text{False}$   
 $(P \neq Q) = (Q = (\neg P))$   
 $(P = Q) = ((\neg P \vee Q) \wedge (P \vee \neg Q))$   
 $(P \neq Q) = ((\neg P \vee \neg Q) \wedge (P \vee Q))$   
**by** *auto*

**lemma** [z3-rule]:  
 $(\text{if } P \text{ then } P \text{ else } \neg P) = \text{True}$   
 $(\text{if } \neg P \text{ then } \neg P \text{ else } P) = \text{True}$   
 $(\text{if } P \text{ then True else False}) = P$   
 $(\text{if } P \text{ then False else True}) = (\neg P)$   
 $(\text{if } \neg P \text{ then } x \text{ else } y) = (\text{if } P \text{ then } y \text{ else } x)$   
**by** *auto*

**lemma** [z3-rule]:  
 $P = Q \vee P \vee Q$   
 $P = Q \vee \neg P \vee \neg Q$   
 $(\neg P) = Q \vee \neg P \vee Q$   
 $(\neg P) = Q \vee P \vee \neg Q$   
 $P = (\neg Q) \vee \neg P \vee Q$   
 $P = (\neg Q) \vee P \vee \neg Q$   
 $P \neq Q \vee P \vee \neg Q$   
 $P \neq Q \vee \neg P \vee Q$   
 $P \neq (\neg Q) \vee P \vee Q$   
 $(\neg P) \neq Q \vee P \vee Q$   
 $P \vee Q \vee P \neq (\neg Q)$   
 $P \vee Q \vee (\neg P) \neq Q$   
 $P \vee \neg Q \vee P \neq Q$   
 $\neg P \vee Q \vee P \neq Q$   
**by** *auto*

**lemma** [z3-rule]:

```

0 + (x::int) = x
x + 0 = x
0 * x = 0
1 * x = x
x + y = y + x
by auto

```

```

hide-type (open) pattern
hide-const Pattern term-eq
hide-const (open) trigger pat nopat fun-app z3div z3mod

end

```

## 59 Main: Main HOL

```

theory Main
imports Plain Predicate-Compile Nitpick SMT
begin

```

Classical Higher-order Logic – only “Main”, excluding real and complex numbers etc.

See further [\[1\]](#)

```
end
```

## 60 Lubs: Definitions of Upper Bounds and Least Upper Bounds

```

theory Lubs
imports Main
begin

```

Thanks to suggestions by James Margetson

```

definition
  settle :: ['a set, 'a::ord] => bool (infixl *<= 70) where
    S *<= x = (ALL y: S. y <= x)

```

```

definition
  setge :: ['a::ord, 'a set] => bool (infixl <=* 70) where
    x <=* S = (ALL y: S. x <= y)

```

```

definition
  leastP :: ['a => bool, 'a::ord] => bool where

```

$leastP\ P\ x = (P\ x \ \&\ x \leq^* Collect\ P)$

**definition**

$isUb :: ['a\ set, 'a\ set, 'a::ord] ==> bool$  **where**  
 $isUb\ R\ S\ x = (S \leq x \ \&\ x: R)$

**definition**

$isLub :: ['a\ set, 'a\ set, 'a::ord] ==> bool$  **where**  
 $isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

**definition**

$ubs :: ['a\ set, 'a::ord\ set] ==> 'a\ set$  **where**  
 $ubs\ R\ S = Collect\ (isUb\ R\ S)$

## 60.1 Rules for the Relations $\leq^*$ and $\leq$

**lemma** *settleI*:  $ALL\ y: S. y \leq x ==> S \leq^* x$   
**by** (*simp add: settle-def*)

**lemma** *settleD*:  $[| S \leq^* x; y: S |] ==> y \leq x$   
**by** (*simp add: settle-def*)

**lemma** *setgeI*:  $ALL\ y: S. x \leq y ==> x \leq^* S$   
**by** (*simp add: setge-def*)

**lemma** *setgeD*:  $[| x \leq^* S; y: S |] ==> x \leq y$   
**by** (*simp add: setge-def*)

## 60.2 Rules about the Operators *leastP*, *ub* and *lub*

**lemma** *leastPD1*:  $leastP\ P\ x ==> P\ x$   
**by** (*simp add: leastP-def*)

**lemma** *leastPD2*:  $leastP\ P\ x ==> x \leq^* Collect\ P$   
**by** (*simp add: leastP-def*)

**lemma** *leastPD3*:  $[| leastP\ P\ x; y: Collect\ P |] ==> x \leq y$   
**by** (*blast dest!: leastPD2 setgeD*)

**lemma** *isLubD1*:  $isLub\ R\ S\ x ==> S \leq^* x$   
**by** (*simp add: isLub-def isUb-def leastP-def*)

**lemma** *isLubD1a*:  $isLub\ R\ S\ x ==> x: R$   
**by** (*simp add: isLub-def isUb-def leastP-def*)

**lemma** *isLub-isUb*:  $isLub\ R\ S\ x ==> isUb\ R\ S\ x$   
**apply** (*simp add: isUb-def*)  
**apply** (*blast dest: isLubD1 isLubD1a*)  
**done**

```

lemma isLubD2: [| isLub R S x; y : S |] ==> y <= x
by (blast dest!: isLubD1 setleD)

lemma isLubD3: isLub R S x ==> leastP(isUb R S) x
by (simp add: isLub-def)

lemma isLubI1: leastP(isUb R S) x ==> isLub R S x
by (simp add: isLub-def)

lemma isLubI2: [| isUb R S x; x <=* Collect (isUb R S) |] ==> isLub R S x
by (simp add: isLub-def leastP-def)

lemma isUbD: [| isUb R S x; y : S |] ==> y <= x
by (simp add: isUb-def setle-def)

lemma isUbD2: isUb R S x ==> S *<= x
by (simp add: isUb-def)

lemma isUbD2a: isUb R S x ==> x: R
by (simp add: isUb-def)

lemma isUbI: [| S *<= x; x: R |] ==> isUb R S x
by (simp add: isUb-def)

lemma isLub-le-isUb: [| isLub R S x; isUb R S y |] ==> x <= y
apply (simp add: isLub-def)
apply (blast intro!: leastPD3)
done

lemma isLub-ubs: isLub R S x ==> x <=* ubs R S
apply (simp add: ubs-def isLub-def)
apply (erule leastPD2)
done

end

```

## 61 Fact: Factorial Function

```

theory Fact
imports Main
begin

class fact =

fixes
  fact :: 'a ⇒ 'a

instantiation nat :: fact

```

**begin**

**fun**

*fact-nat* :: *nat*  $\Rightarrow$  *nat*

**where**

*fact-0-nat*: *fact-nat* 0 = *Suc* 0

| *fact-Suc*: *fact-nat* (*Suc* *x*) = *Suc* *x* \* *fact* *x*

**instance proof qed**

**end**

**instantiation** *int* :: *fact*

**begin**

**definition**

*fact-int* :: *int*  $\Rightarrow$  *int*

**where**

*fact-int* *x* = (if *x*  $\geq$  0 then *int* (*fact* (*nat* *x*)) else 0)

**instance proof qed**

**end**

## 61.1 Set up Transfer

**lemma** *transfer-nat-int-factorial*:

(*x*::*int*)  $\geq$  0  $\implies$  *fact* (*nat* *x*) = *nat* (*fact* *x*)

**unfolding** *fact-int-def*

**by** *auto*

**lemma** *transfer-nat-int-factorial-closure*:

*x*  $\geq$  (0::*int*)  $\implies$  *fact* *x*  $\geq$  0

**by** (*auto simp add: fact-int-def*)

**declare** *transfer-morphism-nat-int*[*transfer add return*:

*transfer-nat-int-factorial transfer-nat-int-factorial-closure*]

**lemma** *transfer-int-nat-factorial*:

*fact* (*int* *x*) = *int* (*fact* *x*)

**unfolding** *fact-int-def* **by** *auto*

**lemma** *transfer-int-nat-factorial-closure*:

*is-nat* *x*  $\implies$  *fact* *x*  $\geq$  0

by (auto simp add: fact-int-def)

declare transfer-morphism-int-nat[transfer add return:  
transfer-int-nat-factorial transfer-int-nat-factorial-closure]

## 61.2 Factorial

lemma fact-0-int [simp]: fact (0::int) = 1  
by (simp add: fact-int-def)

lemma fact-1-nat [simp]: fact (1::nat) = 1  
by simp

lemma fact-Suc-0-nat [simp]: fact (Suc 0) = Suc 0  
by simp

lemma fact-1-int [simp]: fact (1::int) = 1  
by (simp add: fact-int-def)

lemma fact-plus-one-nat: fact ((n::nat) + 1) = (n + 1) \* fact n  
by simp

lemma fact-plus-one-int:  
assumes n >= 0  
shows fact ((n::int) + 1) = (n + 1) \* fact n

using prems unfolding fact-int-def  
by (simp add: nat-add-distrib algebra-simps int-mult)

lemma fact-reduce-nat: (n::nat) > 0  $\implies$  fact n = n \* fact (n - 1)  
apply (subgoal-tac n = Suc (n - 1))  
apply (erule ssubst)  
apply (subst fact-Suc)  
apply simp-all  
done

lemma fact-reduce-int: (n::int) > 0  $\implies$  fact n = n \* fact (n - 1)  
apply (subgoal-tac n = (n - 1) + 1)  
apply (erule ssubst)  
apply (subst fact-plus-one-int)  
apply simp-all  
done

lemma fact-nonzero-nat [simp]: fact (n::nat)  $\neq$  0  
apply (induct n)  
apply (auto simp add: fact-plus-one-nat)  
done

lemma fact-nonzero-int [simp]: n >= 0  $\implies$  fact (n::int)  $\sim$  0



```

    by (simp add: fact-int-def)

lemma fact-gt-zero-nat [simp]: fact (n :: nat) > 0
  by (insert fact-nonzero-nat [of n], arith)

lemma fact-gt-zero-int [simp]: n >= 0  $\implies$  fact (n :: int) > 0
  by (auto simp add: fact-int-def)

lemma fact-ge-one-nat [simp]: fact (n :: nat) >= 1
  by (insert fact-nonzero-nat [of n], arith)

lemma fact-ge-Suc-0-nat [simp]: fact (n :: nat) >= Suc 0
  by (insert fact-nonzero-nat [of n], arith)

lemma fact-ge-one-int [simp]: n >= 0  $\implies$  fact (n :: int) >= 1
  apply (auto simp add: fact-int-def)
  apply (subgoal-tac 1 = int 1)
  apply (erule ssubst)
  apply (subst zle-int)
  apply auto
done

lemma dvd-fact-nat [rule-format]: 1 <= m  $\longrightarrow$  m <= n  $\longrightarrow$  m dvd fact (n::nat)
  apply (induct n)
  apply force
  apply (auto simp only: fact-Suc)
  apply (subgoal-tac m = Suc n)
  apply (erule ssubst)
  apply (rule dvd-triv-left)
  apply auto
done

lemma dvd-fact-int [rule-format]: 1 <= m  $\longrightarrow$  m <= n  $\longrightarrow$  m dvd fact (n::int)
  apply (case-tac 1 <= n)
  apply (induct n rule: int-ge-induct)
  apply (auto simp add: fact-plus-one-int)
  apply (subgoal-tac m = i + 1)
  apply auto
done

lemma interval-plus-one-nat: (i::nat) <= j + 1  $\implies$ 
  {i..j+1} = {i..j} Un {j+1}
  by auto

lemma interval-Suc: i <= Suc j  $\implies$  {i..Suc j} = {i..j} Un {Suc j}
  by auto

lemma interval-plus-one-int: (i::int) <= j + 1  $\implies$  {i..j+1} = {i..j} Un {j+1}
  by auto

```

```

lemma fact-altdef-nat: fact (n::nat) = (PROD i::{1..n}. i)
  apply (induct n)
  apply force
  apply (subst fact-Suc)
  apply (subst interval-Suc)
  apply auto
done

```

```

lemma fact-altdef-int: n >= 0  $\implies$  fact (n::int) = (PROD i::{1..n}. i)
  apply (induct n rule: int-ge-induct)
  apply force
  apply (subst fact-plus-one-int, assumption)
  apply (subst interval-plus-one-int)
  apply auto
done

```

```

lemma fact-mono-nat: (m::nat)  $\leq$  n  $\implies$  fact m  $\leq$  fact n
  apply (drule le-imp-less-or-eq)
  apply (auto dest!: less-imp-Suc-add)
  apply (induct-tac k, auto)
done

```

```

lemma fact-neg-int [simp]: m < (0::int)  $\implies$  fact m = 0
  unfolding fact-int-def by auto

```

```

lemma fact-ge-zero-int [simp]: fact m >= (0::int)
  apply (case-tac m >= 0)
  apply auto
  apply (frule fact-gt-zero-int)
  apply arith
done

```

```

lemma fact-mono-int-aux [rule-format]: k >= (0::int)  $\implies$ 
  fact (m + k) >= fact m
  apply (case-tac m < 0)
  apply auto
  apply (induct k rule: int-ge-induct)
  apply auto
  apply (subst add-assoc [symmetric])
  apply (subst fact-plus-one-int)
  apply auto
  apply (erule order-trans)
  apply (subst mult-le-cancel-right1)
  apply (subgoal-tac fact (m + i) >= 0)
  apply arith
  apply auto
done

```

```

lemma fact-mono-int:  $(m::int) \leq n \implies \text{fact } m \leq \text{fact } n$ 
  apply (insert fact-mono-int-aux [of  $n - m$   $m$ ])
  apply auto
done

```

Note that  $\text{fact } (0::'a) = \text{fact } (1::'a)$

```

lemma fact-less-mono-nat:  $[(0::nat) < m; m < n] \implies \text{fact } m < \text{fact } n$ 
  apply (drule-tac  $m = m$  in less-imp-Suc-add, auto)
  apply (induct-tac  $k$ , auto)
done

```

```

lemma fact-less-mono-int-aux:  $k \geq 0 \implies (0::int) < m \implies$ 
   $\text{fact } m < \text{fact } ((m + 1) + k)$ 
  apply (induct  $k$  rule: int-ge-induct)
  apply (simp add: fact-plus-one-int)
  apply (subst mult-less-cancel-right1)
  apply (insert fact-gt-zero-int [of  $m$ ], arith)
  apply (subst (2) fact-reduce-int)
  apply (auto simp add: add-ac)
  apply (erule order-less-le-trans)
  apply (subst mult-le-cancel-right1)
  apply auto
  apply (subgoal-tac  $\text{fact } (i + (1 + m)) \geq 0$ )
  apply force
  apply (rule fact-ge-zero-int)
done

```

```

lemma fact-less-mono-int:  $(0::int) < m \implies m < n \implies \text{fact } m < \text{fact } n$ 
  apply (insert fact-less-mono-int-aux [of  $n - (m + 1)$   $m$ ])
  apply auto
done

```

```

lemma fact-num-eq-if-nat:  $\text{fact } (m::nat) =$ 
   $(\text{if } m=0 \text{ then } 1 \text{ else } m * \text{fact } (m - 1))$ 
by (cases  $m$ ) auto

```

```

lemma fact-add-num-eq-if-nat:
   $\text{fact } ((m::nat) + n) = (\text{if } m + n = 0 \text{ then } 1 \text{ else } (m + n) * \text{fact } (m + n - 1))$ 
by (cases  $m + n$ ) auto

```

```

lemma fact-add-num-eq-if2-nat:
   $\text{fact } ((m::nat) + n) =$ 
   $(\text{if } m = 0 \text{ then } \text{fact } n \text{ else } (m + n) * \text{fact } ((m - 1) + n))$ 
by (cases  $m$ ) auto

```

### 61.3 *fact* and *of-nat*

```

lemma of-nat-fact-not-zero [simp]:  $\text{of-nat } (\text{fact } n) \neq (0::'a::\text{semiring-char-0})$ 
by auto

```

**lemma** *of-nat-fact-gt-zero* [*simp*]:  $(0 :: 'a :: \{\text{linordered-semidom}\}) < \text{of-nat}(\text{fact } n)$   
**by** *auto*

**lemma** *of-nat-fact-ge-zero* [*simp*]:  $(0 :: 'a :: \text{linordered-semidom}) \leq \text{of-nat}(\text{fact } n)$   
**by** *simp*

**lemma** *inv-of-nat-fact-gt-zero* [*simp*]:  $(0 :: 'a :: \text{linordered-field}) < \text{inverse}(\text{of-nat}(\text{fact } n))$   
**by** (*auto simp add: positive-imp-inverse-positive*)

**lemma** *inv-of-nat-fact-ge-zero* [*simp*]:  $(0 :: 'a :: \text{linordered-field}) \leq \text{inverse}(\text{of-nat}(\text{fact } n))$   
**by** (*auto intro: order-less-imp-le*)

**end**

## 62 Parity: Even and Odd for int and nat

**theory** *Parity*  
**imports** *Main*  
**begin**

**class** *even-odd* =  
  **fixes** *even* ::  $'a \Rightarrow \text{bool}$

**abbreviation**  
   $\text{odd} :: 'a :: \text{even-odd} \Rightarrow \text{bool}$  **where**  
   $\text{odd } x \equiv \neg \text{even } x$

**instantiation** *nat* **and** *int* :: *even-odd*  
**begin**

**definition**  
   $\text{even-def}$  [*presburger*]:  $\text{even } x \longleftrightarrow (x :: \text{int}) \bmod 2 = 0$

**definition**  
   $\text{even-nat-def}$  [*presburger*]:  $\text{even } x \longleftrightarrow \text{even}(\text{int } x)$

**instance** ..

**end**

**lemma** *transfer-int-nat-relations*:  
   $\text{even}(\text{int } x) \longleftrightarrow \text{even } x$   
  **by** (*simp add: even-nat-def*)

**declare** *transfer-morphism-int-nat*[*transfer add return*:

```

  transfer-int-nat-relations
]

```

```

lemma even-zero-int[simp]: even (0::int) by presburger

```

```

lemma odd-one-int[simp]: odd (1::int) by presburger

```

```

lemma even-zero-nat[simp]: even (0::nat) by presburger

```

```

lemma odd-1-nat [simp]: odd (1::nat) by presburger

```

```

declare even-def[of number-of v, standard, simp]

```

```

declare even-nat-def[of number-of v, standard, simp]

```

### 62.1 Even and odd are mutually exclusive

```

lemma int-pos-lt-two-imp-zero-or-one:

```

```

  0 <= x ==> (x::int) < 2 ==> x = 0 | x = 1

```

```

by presburger

```

```

lemma neq-one-mod-two [simp, presburger]:

```

```

  ((x::int) mod 2 ~ = 0) = (x mod 2 = 1) by presburger

```

### 62.2 Behavior under integer arithmetic operations

```

declare dvd-def[algebra]

```

```

lemma nat-even-iff-2-dvd[algebra]: even (x::nat) <math>\longleftrightarrow 2 \text{ dvd } x</math>

```

```

by presburger

```

```

lemma int-even-iff-2-dvd[algebra]: even (x::int) <math>\longleftrightarrow 2 \text{ dvd } x</math>

```

```

by presburger

```

```

lemma even-times-anything: even (x::int) ==> even (x * y)

```

```

by algebra

```

```

lemma anything-times-even: even (y::int) ==> even (x * y) by algebra

```

```

lemma odd-times-odd: odd (x::int) ==> odd y ==> odd (x * y)

```

```

by (simp add: even-def zmod-zmult1-eq)

```

```

lemma even-product[simp,presburger]: even((x::int) * y) = (even x | even y)

```

```

apply (auto simp add: even-times-anything anything-times-even)

```

```

apply (rule ccontr)

```

```

apply (auto simp add: odd-times-odd)

```

```

done

```

```

lemma even-plus-even: even (x::int) ==> even y ==> even (x + y)

```

```

by presburger

```

```

lemma even-plus-odd: even (x::int) ==> odd y ==> odd (x + y)

```

**by** *presburger*

**lemma** *odd-plus-even*:  $\text{odd } (x::\text{int}) \implies \text{even } y \implies \text{odd } (x + y)$   
**by** *presburger*

**lemma** *odd-plus-odd*:  $\text{odd } (x::\text{int}) \implies \text{odd } y \implies \text{even } (x + y)$  **by** *presburger*

**lemma** *even-sum*[*simp,presburger*]:  
 $\text{even } ((x::\text{int}) + y) = ((\text{even } x \ \& \ \text{even } y) \mid (\text{odd } x \ \& \ \text{odd } y))$   
**by** *presburger*

**lemma** *even-neg*[*simp,presburger,algebra*]:  $\text{even } (-(x::\text{int})) = \text{even } x$   
**by** *presburger*

**lemma** *even-difference*[*simp*]:  
 $\text{even } ((x::\text{int}) - y) = ((\text{even } x \ \& \ \text{even } y) \mid (\text{odd } x \ \& \ \text{odd } y))$  **by** *presburger*

**lemma** *even-power*[*simp,presburger*]:  $\text{even } ((x::\text{int})^n) = (\text{even } x \ \& \ n \neq 0)$   
**by** (*induct n*) *auto*

**lemma** *odd-pow*:  $\text{odd } x \implies \text{odd } ((x::\text{int})^n)$  **by** *simp*

### 62.3 Equivalent definitions

**lemma** *two-times-even-div-two*:  $\text{even } (x::\text{int}) \implies 2 * (x \text{ div } 2) = x$   
**by** *presburger*

**lemma** *two-times-odd-div-two-plus-one*:  
 $\text{odd } (x::\text{int}) \implies 2 * (x \text{ div } 2) + 1 = x$   
**by** *presburger*

**lemma** *even-equiv-def*:  $\text{even } (x::\text{int}) = (\exists x. x = 2 * y)$  **by** *presburger*

**lemma** *odd-equiv-def*:  $\text{odd } (x::\text{int}) = (\exists x. x = 2 * y + 1)$  **by** *presburger*

### 62.4 even and odd for nats

**lemma** *pos-int-even-equiv-nat-even*:  $0 \leq x \implies \text{even } x = \text{even } (\text{nat } x)$   
**by** (*simp add: even-nat-def*)

**lemma** *even-product-nat*[*simp,presburger,algebra*]:  
 $\text{even } ((x::\text{nat}) * y) = (\text{even } x \mid \text{even } y)$   
**by** (*simp add: even-nat-def int-mult*)

**lemma** *even-sum-nat*[*simp,presburger,algebra*]:  
 $\text{even } ((x::\text{nat}) + y) = ((\text{even } x \ \& \ \text{even } y) \mid (\text{odd } x \ \& \ \text{odd } y))$   
**by** *presburger*

**lemma** *even-difference-nat*[*simp,presburger,algebra*]:  
 $\text{even } ((x::\text{nat}) - y) = (x < y \mid (\text{even } x \ \& \ \text{even } y) \mid (\text{odd } x \ \& \ \text{odd } y))$

**by** *presburger*

**lemma** *even-Suc*[*simp,presburger,algebra*]:  $\text{even } (\text{Suc } x) = \text{odd } x$   
**by** *presburger*

**lemma** *even-power-nat*[*simp,presburger,algebra*]:  
 $\text{even } ((x::\text{nat})^y) = (\text{even } x \ \& \ 0 < y)$   
**by** (*simp add: even-nat-def int-power*)

## 62.5 Equivalent definitions

**lemma** *nat-lt-two-imp-zero-or-one*:  
 $(x::\text{nat}) < \text{Suc } (\text{Suc } 0) \implies x = 0 \mid x = \text{Suc } 0$   
**by** *presburger*

**lemma** *even-nat-mod-two-eq-zero*:  $\text{even } (x::\text{nat}) \implies x \bmod (\text{Suc } (\text{Suc } 0)) = 0$   
**by** *presburger*

**lemma** *odd-nat-mod-two-eq-one*:  $\text{odd } (x::\text{nat}) \implies x \bmod (\text{Suc } (\text{Suc } 0)) = \text{Suc } 0$   
**by** *presburger*

**lemma** *even-nat-equiv-def*:  $\text{even } (x::\text{nat}) = (x \bmod \text{Suc } (\text{Suc } 0) = 0)$   
**by** *presburger*

**lemma** *odd-nat-equiv-def*:  $\text{odd } (x::\text{nat}) = (x \bmod \text{Suc } (\text{Suc } 0) = \text{Suc } 0)$   
**by** *presburger*

**lemma** *even-nat-div-two-times-two*:  $\text{even } (x::\text{nat}) \implies$   
 $\text{Suc } (\text{Suc } 0) * (x \text{ div } \text{Suc } (\text{Suc } 0)) = x$  **by** *presburger*

**lemma** *odd-nat-div-two-times-two-plus-one*:  $\text{odd } (x::\text{nat}) \implies$   
 $\text{Suc } (\text{Suc } (\text{Suc } 0) * (x \text{ div } \text{Suc } (\text{Suc } 0))) = x$  **by** *presburger*

**lemma** *even-nat-equiv-def2*:  $\text{even } (x::\text{nat}) = (\exists y. x = \text{Suc } (\text{Suc } 0) * y)$   
**by** *presburger*

**lemma** *odd-nat-equiv-def2*:  $\text{odd } (x::\text{nat}) = (\exists y. x = \text{Suc } (\text{Suc } (\text{Suc } 0) * y))$   
**by** *presburger*

## 62.6 Parity and powers

**lemma** *minus-one-even-odd-power*:  
 $(\text{even } x \implies (-1::'a::\{\text{comm-ring-1}\})^x = 1) \ \&$   
 $(\text{odd } x \implies (-1::'a)^x = -1)$   
**apply** (*induct x*)  
**apply** (*rule conjI*)  
**apply** *simp*  
**apply** (*insert even-zero-nat, blast*)  
**apply** *simp*  
**done**

**lemma** *minus-one-even-power* [simp]:  
 $even\ x \implies (-1::'a::\{comm-ring-1\})^x = 1$   
**using** *minus-one-even-odd-power* **by** *blast*

**lemma** *minus-one-odd-power* [simp]:  
 $odd\ x \implies (-1::'a::\{comm-ring-1\})^x = -1$   
**using** *minus-one-even-odd-power* **by** *blast*

**lemma** *neg-one-even-odd-power*:  
 $(even\ x \implies (-1::'a::\{number-ring\})^x = 1) \ \&$   
 $(odd\ x \implies (-1::'a)^x = -1)$   
**apply** (*induct* *x*)  
**apply** (*simp*, *simp*)  
**done**

**lemma** *neg-one-even-power* [simp]:  
 $even\ x \implies (-1::'a::\{number-ring\})^x = 1$   
**using** *neg-one-even-odd-power* **by** *blast*

**lemma** *neg-one-odd-power* [simp]:  
 $odd\ x \implies (-1::'a::\{number-ring\})^x = -1$   
**using** *neg-one-even-odd-power* **by** *blast*

**lemma** *neg-power-if*:  
 $(-x::'a::\{comm-ring-1\})^n =$   
 $(if\ even\ n\ then\ (x^{\wedge} n)\ else\ -(x^{\wedge} n))$   
**apply** (*induct* *n*)  
**apply** *simp-all*  
**done**

**lemma** *zero-le-even-power*:  $even\ n \implies$   
 $0 \leq (x::'a::\{linordered-ring, monoid-mult\})^n$   
**apply** (*simp* *add*: *even-nat-equiv-def2*)  
**apply** (*erule* *exE*)  
**apply** (*erule* *ssubst*)  
**apply** (*subst* *power-add*)  
**apply** (*rule* *zero-le-square*)  
**done**

**lemma** *zero-le-odd-power*:  $odd\ n \implies$   
 $(0 \leq (x::'a::\{linordered-idom\})^n) = (0 \leq x)$   
**apply** (*auto* *simp*: *odd-nat-equiv-def2* *power-add* *zero-le-mult-iff*)  
**apply** (*metis* *field-power-not-zero* *divisors-zero* *order-antisym-conv* *zero-le-square*)  
**done**

**lemma** *zero-le-power-eq[presburger]*:  $(0 \leq (x::'a::\{linordered-idom\})^n) =$   
 $(even\ n \mid (odd\ n \ \&\ 0 \leq x))$   
**apply** *auto*



```

apply (subst zero-le-odd-power [symmetric])
apply assumption+
apply (erule zero-le-even-power)
done

lemma zero-less-power-eq[presburger]:  $(0 < (x::'a::\{\text{linordered-idom}\}) ^ n) =$ 
 $(n = 0 \mid (\text{even } n \ \& \ x \sim= 0) \mid (\text{odd } n \ \& \ 0 < x))$ 

unfolding order-less-le zero-le-power-eq by auto

lemma power-less-zero-eq[presburger]:  $((x::'a::\{\text{linordered-idom}\}) ^ n < 0) =$ 
 $(\text{odd } n \ \& \ x < 0)$ 
apply (subst linorder-not-le [symmetric])+
apply (subst zero-le-power-eq)
apply auto
done

lemma power-le-zero-eq[presburger]:  $((x::'a::\{\text{linordered-idom}\}) ^ n \leq 0) =$ 
 $(n \sim= 0 \ \& \ ((\text{odd } n \ \& \ x \leq 0) \mid (\text{even } n \ \& \ x = 0)))$ 
apply (subst linorder-not-less [symmetric])+
apply (subst zero-less-power-eq)
apply auto
done

lemma power-even-abs:  $\text{even } n ==>$ 
 $(\text{abs } (x::'a::\{\text{linordered-idom}\})) ^ n = x ^ n$ 
apply (subst power-abs [symmetric])
apply (simp add: zero-le-even-power)
done

lemma zero-less-power-nat-eq[presburger]:  $(0 < (x::\text{nat}) ^ n) = (n = 0 \mid 0 < x)$ 
by (induct n) auto

lemma power-minus-even [simp]:  $\text{even } n ==>$ 
 $(- x) ^ n = (x ^ n::'a::\{\text{comm-ring-1}\})$ 
apply (subst power-minus)
apply simp
done

lemma power-minus-odd [simp]:  $\text{odd } n ==>$ 
 $(- x) ^ n = - (x ^ n::'a::\{\text{comm-ring-1}\})$ 
apply (subst power-minus)
apply simp
done

lemma power-mono-even: fixes  $x \ y :: 'a :: \{\text{linordered-idom}\}$ 
assumes  $\text{even } n$  and  $|x| \leq |y|$ 
shows  $x ^ n \leq y ^ n$ 
proof –

```

```

have  $0 \leq |x|$  by auto
with  $\langle |x| \leq |y| \rangle$ 
have  $|x|^\wedge n \leq |y|^\wedge n$  by (rule power-mono)
thus ?thesis unfolding power-even-abs[OF  $\langle \text{even } n \rangle$ ] .
qed

```

```

lemma odd-pos: odd (n::nat)  $\implies 0 < n$  by presburger

```

```

lemma power-mono-odd: fixes x y :: 'a :: {linordered-idom}
  assumes odd n and  $x \leq y$ 
  shows  $x^\wedge n \leq y^\wedge n$ 
proof (cases y < 0)
  case True with  $\langle x \leq y \rangle$  have  $-y \leq -x$  and  $0 \leq -y$  by auto
  hence  $(-y)^\wedge n \leq (-x)^\wedge n$  by (rule power-mono)
  thus ?thesis unfolding power-minus-odd[OF  $\langle \text{odd } n \rangle$ ] by auto
next
  case False
  show ?thesis
  proof (cases x < 0)
    case True hence  $n \neq 0$  and  $x \leq 0$  using  $\langle \text{odd } n \rangle$ [THEN odd-pos] by auto
    hence  $x^\wedge n \leq 0$  unfolding power-le-zero-eq using  $\langle \text{odd } n \rangle$  by auto
    moreover
    from  $\langle \neg y < 0 \rangle$  have  $0 \leq y$  by auto
    hence  $0 \leq y^\wedge n$  by auto
    ultimately show ?thesis by auto
  next
    case False hence  $0 \leq x$  by auto
    with  $\langle x \leq y \rangle$  show ?thesis using power-mono by auto
  qed
qed

```

## 62.7 More Even/Odd Results

```

lemma even-mult-two-ex: even(n) =  $(\exists m::nat. n = 2*m)$  by presburger

```

```

lemma odd-Suc-mult-two-ex: odd(n) =  $(\exists m. n = \text{Suc } (2*m))$  by presburger

```

```

lemma even-add [simp]: even(m + n::nat) = (even m = even n) by presburger

```

```

lemma odd-add [simp]: odd(m + n::nat) = (odd m  $\neq$  odd n) by presburger

```

```

lemma div-Suc: Suc a div c = a div c + Suc 0 div c +
  (a mod c + Suc 0 mod c) div c
  apply (subgoal-tac Suc a = a + Suc 0)
  apply (erule ssubst)
  apply (rule div-add1-eq, simp)
  done

```

```

lemma lemma-even-div2 [simp]: even (n::nat)  $\implies (n + 1) \text{ div } 2 = n \text{ div } 2$  by
presburger

```

**lemma** *lemma-not-even-div2* [simp]:  $\sim \text{even } n \implies (n + 1) \text{ div } 2 = \text{Suc } (n \text{ div } 2)$

**by** *presburger*

**lemma** *even-num-iff*:  $0 < n \implies \text{even } n = (\sim \text{even}(n - 1 :: \text{nat}))$  **by** *presburger*

**lemma** *even-even-mod-4-iff*:  $\text{even } (n :: \text{nat}) = \text{even } (n \bmod 4)$  **by** *presburger*

**lemma** *lemma-odd-mod-4-div-2*:  $n \bmod 4 = (3 :: \text{nat}) \implies \text{odd}((n - 1) \text{ div } 2)$  **by** *presburger*

**lemma** *lemma-even-mod-4-div-2*:  $n \bmod 4 = (1 :: \text{nat}) \implies \text{even}((n - 1) \text{ div } 2)$  **by** *presburger*

Simplify, when the exponent is a numeral

**lemmas** *power-0-left-number-of* = *power-0-left* [of *number-of w, standard*]

**declare** *power-0-left-number-of* [simp]

**lemmas** *zero-le-power-eq-number-of* [simp] =  
*zero-le-power-eq* [of *- number-of w, standard*]

**lemmas** *zero-less-power-eq-number-of* [simp] =  
*zero-less-power-eq* [of *- number-of w, standard*]

**lemmas** *power-le-zero-eq-number-of* [simp] =  
*power-le-zero-eq* [of *- number-of w, standard*]

**lemmas** *power-less-zero-eq-number-of* [simp] =  
*power-less-zero-eq* [of *- number-of w, standard*]

**lemmas** *zero-less-power-nat-eq-number-of* [simp] =  
*zero-less-power-nat-eq* [of *- number-of w, standard*]

**lemmas** *power-eq-0-iff-number-of* [simp] = *power-eq-0-iff* [of *- number-of w, standard*]

**lemmas** *power-even-abs-number-of* [simp] = *power-even-abs* [of *number-of w -, standard*]

## 62.8 An Equivalence for $0 \leq a^n$

**lemma** *even-power-le-0-imp-0*:

$a^n \leq (0 :: 'a :: \{\text{linordered-idom}\}) \implies a = 0$

**by** (*induct k*) (*auto simp add: zero-le-mult-iff mult-le-0-iff*)

**lemma** *zero-le-power-iff* [*presburger*]:

$(0 \leq a^n) = (0 \leq (a :: 'a :: \{\text{linordered-idom}\}) \mid \text{even } n)$

**proof** *cases*

**assume** *even*: *even n*

**then obtain** *k* **where**  $n = 2 * k$

```

    by (auto simp add: even-nat-equiv-def2 numeral-2-eq-2)
  thus ?thesis by (simp add: zero-le-even-power even)
next
  assume odd: odd n
  then obtain k where n = Suc(2*k)
    by (auto simp add: odd-nat-equiv-def2 numeral-2-eq-2)
  thus ?thesis
    by (auto simp add: zero-le-mult-iff zero-le-even-power
        dest!: even-power-le-0-imp-0)
qed

```

## 62.9 Miscellaneous

```

lemma [presburger]: (x + 1) div 2 = x div 2  $\longleftrightarrow$  even (x::int) by presburger
lemma [presburger]: (x + 1) div 2 = x div 2 + 1  $\longleftrightarrow$  odd (x::int) by presburger
lemma even-plus-one-div-two: even (x::int)  $\implies$  (x + 1) div 2 = x div 2 by
presburger
lemma odd-plus-one-div-two: odd (x::int)  $\implies$  (x + 1) div 2 = x div 2 + 1 by
presburger

lemma [presburger]: (Suc x) div Suc (Suc 0) = x div Suc (Suc 0)  $\longleftrightarrow$  even x by
presburger
lemma [presburger]: (Suc x) div Suc (Suc 0) = x div Suc (Suc 0)  $\longleftrightarrow$  even x by
presburger
lemma even-nat-plus-one-div-two: even (x::nat)  $\implies$ 
  (Suc x) div Suc (Suc 0) = x div Suc (Suc 0) by presburger

lemma odd-nat-plus-one-div-two: odd (x::nat)  $\implies$ 
  (Suc x) div Suc (Suc 0) = Suc (x div Suc (Suc 0)) by presburger

end

```

## 63 GCD: Greatest common divisor and least common multiple

```

theory GCD
imports Fact Parity
begin

```

```

declare One-nat-def [simp del]

```

### 63.1 GCD and LCM definitions

```

class gcd = zero + one + dvd +

fixes
  gcd :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a and
  lcm :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a

```

**begin**

**abbreviation**

*coprime* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

**where**

*coprime* x y == (gcd x y = 1)

**end**

**instantiation** nat :: gcd

**begin**

**fun**

*gcd-nat* :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat

**where**

*gcd-nat* x y =  
(if y = 0 then x else gcd y (x mod y))

**definition**

*lcm-nat* :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat

**where**

*lcm-nat* x y = x \* y div (gcd x y)

**instance** proof qed

**end**

**instantiation** int :: gcd

**begin**

**definition**

*gcd-int* :: int  $\Rightarrow$  int  $\Rightarrow$  int

**where**

*gcd-int* x y = int (gcd (nat (abs x)) (nat (abs y)))

**definition**

*lcm-int* :: int  $\Rightarrow$  int  $\Rightarrow$  int

**where**

*lcm-int* x y = int (lcm (nat (abs x)) (nat (abs y)))

**instance** proof qed

**end**

## 63.2 Transfer setup

**lemma** *transfer-nat-int-gcd*:

(x::int) >= 0  $\implies$  y >= 0  $\implies$  gcd (nat x) (nat y) = nat (gcd x y)

$(x::int) \geq 0 \implies y \geq 0 \implies \text{lcm } (nat\ x) (nat\ y) = nat\ (\text{lcm } x\ y)$   
**unfolding** *gcd-int-def lcm-int-def*  
**by** *auto*

**lemma** *transfer-nat-int-gcd-closures*:  
 $x \geq (0::int) \implies y \geq 0 \implies \text{gcd } x\ y \geq 0$   
 $x \geq (0::int) \implies y \geq 0 \implies \text{lcm } x\ y \geq 0$   
**by** (*auto simp add: gcd-int-def lcm-int-def*)

**declare** *transfer-morphism-nat-int*[*transfer add return:*  
*transfer-nat-int-gcd transfer-nat-int-gcd-closures*]

**lemma** *transfer-int-nat-gcd*:  
 $\text{gcd } (int\ x) (int\ y) = int\ (\text{gcd } x\ y)$   
 $\text{lcm } (int\ x) (int\ y) = int\ (\text{lcm } x\ y)$   
**by** (*unfold gcd-int-def lcm-int-def, auto*)

**lemma** *transfer-int-nat-gcd-closures*:  
 $is\_nat\ x \implies is\_nat\ y \implies \text{gcd } x\ y \geq 0$   
 $is\_nat\ x \implies is\_nat\ y \implies \text{lcm } x\ y \geq 0$   
**by** (*auto simp add: gcd-int-def lcm-int-def*)

**declare** *transfer-morphism-int-nat*[*transfer add return:*  
*transfer-int-nat-gcd transfer-int-nat-gcd-closures*]

### 63.3 GCD properties

**lemma** *gcd-nat-induct*:  
**fixes**  $m\ n :: nat$   
**assumes**  $\bigwedge m. P\ m\ 0$   
**and**  $\bigwedge m\ n. 0 < n \implies P\ n\ (m\ mod\ n) \implies P\ m\ n$   
**shows**  $P\ m\ n$   
**apply** (*rule gcd-nat.induct*)  
**apply** (*case-tac y = 0*)  
**using** *assms apply simp-all*  
**done**

**lemma** *gcd-neg1-int* [*simp*]:  $\text{gcd } (-x::int)\ y = \text{gcd } x\ y$   
**by** (*simp add: gcd-int-def*)

**lemma** *gcd-neg2-int* [*simp*]:  $\text{gcd } (x::int)\ (-y) = \text{gcd } x\ y$   
**by** (*simp add: gcd-int-def*)

**lemma** *abs-gcd-int*[*simp*]:  $\text{abs}(\text{gcd } (x::int)\ y) = \text{gcd } x\ y$   
**by**(*simp add: gcd-int-def*)

**lemma** *gcd-abs-int*:  $\text{gcd } (x::int)\ y = \text{gcd } (\text{abs } x)\ (\text{abs } y)$

**by** (*simp add: gcd-int-def*)

**lemma** *gcd-abs1-int*[*simp*]:  $\text{gcd } (\text{abs } x) (y::\text{int}) = \text{gcd } x y$   
**by** (*metis abs-idempotent gcd-abs-int*)

**lemma** *gcd-abs2-int*[*simp*]:  $\text{gcd } x (\text{abs } y::\text{int}) = \text{gcd } x y$   
**by** (*metis abs-idempotent gcd-abs-int*)

**lemma** *gcd-cases-int*:

**fixes**  $x :: \text{int}$  **and**  $y$   
**assumes**  $x \geq 0 \implies y \geq 0 \implies P (\text{gcd } x y)$   
**and**  $x \geq 0 \implies y \leq 0 \implies P (\text{gcd } x (-y))$   
**and**  $x \leq 0 \implies y \geq 0 \implies P (\text{gcd } (-x) y)$   
**and**  $x \leq 0 \implies y \leq 0 \implies P (\text{gcd } (-x) (-y))$   
**shows**  $P (\text{gcd } x y)$   
**by** (*insert assms, auto, arith*)

**lemma** *gcd-ge-0-int* [*simp*]:  $\text{gcd } (x::\text{int}) y \geq 0$   
**by** (*simp add: gcd-int-def*)

**lemma** *lcm-neg1-int*:  $\text{lcm } (-x::\text{int}) y = \text{lcm } x y$   
**by** (*simp add: lcm-int-def*)

**lemma** *lcm-neg2-int*:  $\text{lcm } (x::\text{int}) (-y) = \text{lcm } x y$   
**by** (*simp add: lcm-int-def*)

**lemma** *lcm-abs-int*:  $\text{lcm } (x::\text{int}) y = \text{lcm } (\text{abs } x) (\text{abs } y)$   
**by** (*simp add: lcm-int-def*)

**lemma** *abs-lcm-int* [*simp*]:  $\text{abs } (\text{lcm } i j::\text{int}) = \text{lcm } i j$   
**by** (*simp add: lcm-int-def*)

**lemma** *lcm-abs1-int*[*simp*]:  $\text{lcm } (\text{abs } x) (y::\text{int}) = \text{lcm } x y$   
**by** (*metis abs-idempotent lcm-int-def*)

**lemma** *lcm-abs2-int*[*simp*]:  $\text{lcm } x (\text{abs } y::\text{int}) = \text{lcm } x y$   
**by** (*metis abs-idempotent lcm-int-def*)

**lemma** *lcm-cases-int*:

**fixes**  $x :: \text{int}$  **and**  $y$   
**assumes**  $x \geq 0 \implies y \geq 0 \implies P (\text{lcm } x y)$   
**and**  $x \geq 0 \implies y \leq 0 \implies P (\text{lcm } x (-y))$   
**and**  $x \leq 0 \implies y \geq 0 \implies P (\text{lcm } (-x) y)$   
**and**  $x \leq 0 \implies y \leq 0 \implies P (\text{lcm } (-x) (-y))$   
**shows**  $P (\text{lcm } x y)$   
**by** (*insert prems, auto simp add: lcm-neg1-int lcm-neg2-int, arith*)

**lemma** *lcm-ge-0-int* [*simp*]:  $\text{lcm } (x::\text{int}) y \geq 0$   
**by** (*simp add: lcm-int-def*)

**lemma** *gcd-0-nat* [*simp*]:  $\text{gcd } (x::\text{nat}) \ 0 = x$   
**by** *simp*

**lemma** *gcd-0-int* [*simp*]:  $\text{gcd } (x::\text{int}) \ 0 = \text{abs } x$   
**by** (*unfold gcd-int-def, auto*)

**lemma** *gcd-0-left-nat* [*simp*]:  $\text{gcd } 0 \ (x::\text{nat}) = x$   
**by** *simp*

**lemma** *gcd-0-left-int* [*simp*]:  $\text{gcd } 0 \ (x::\text{int}) = \text{abs } x$   
**by** (*unfold gcd-int-def, auto*)

**lemma** *gcd-red-nat*:  $\text{gcd } (x::\text{nat}) \ y = \text{gcd } y \ (x \bmod y)$   
**by** (*case-tac y = 0, auto*)

**lemma** *gcd-non-0-nat*:  $y \sim = (0::\text{nat}) \implies \text{gcd } (x::\text{nat}) \ y = \text{gcd } y \ (x \bmod y)$   
**by** *simp*

**lemma** *gcd-1-nat* [*simp*]:  $\text{gcd } (m::\text{nat}) \ 1 = 1$   
**by** *simp*

**lemma** *gcd-Suc-0* [*simp*]:  $\text{gcd } (m::\text{nat}) \ (\text{Suc } 0) = \text{Suc } 0$   
**by** (*simp add: One-nat-def*)

**lemma** *gcd-1-int* [*simp*]:  $\text{gcd } (m::\text{int}) \ 1 = 1$   
**by** (*simp add: gcd-int-def*)

**lemma** *gcd-idem-nat*:  $\text{gcd } (x::\text{nat}) \ x = x$   
**by** *simp*

**lemma** *gcd-idem-int*:  $\text{gcd } (x::\text{int}) \ x = \text{abs } x$   
**by** (*auto simp add: gcd-int-def*)

**declare** *gcd-nat.simps* [*simp del*]

*gcd m n* divides *m* and *n*. The conjunctions don’t seem provable separately.

**lemma** *gcd-dvd1-nat* [*iff*]:  $(\text{gcd } (m::\text{nat})) \ n \ \text{dvd } m$   
**and** *gcd-dvd2-nat* [*iff*]:  $(\text{gcd } m \ n) \ \text{dvd } n$   
**apply** (*induct m n rule: gcd-nat-induct*)  
**apply** (*simp-all add: gcd-non-0-nat*)  
**apply** (*blast dest: dvd-mod-imp-dvd*)  
**done**

**lemma** *gcd-dvd1-int* [*iff*]:  $\text{gcd } (x::\text{int}) \ y \ \text{dvd } x$



**by** (*metis gcd-int-def int-dvd-iff gcd-dvd1-nat*)

**lemma** *gcd-dvd2-int* [*iff*]:  $\text{gcd } (x::\text{int}) \ y \ \text{dvd} \ y$   
**by** (*metis gcd-int-def int-dvd-iff gcd-dvd2-nat*)

**lemma** *dvd-gcd-D1-nat*:  $k \ \text{dvd} \ \text{gcd} \ m \ n \implies (k::\text{nat}) \ \text{dvd} \ m$   
**by**(*metis gcd-dvd1-nat dvd-trans*)

**lemma** *dvd-gcd-D2-nat*:  $k \ \text{dvd} \ \text{gcd} \ m \ n \implies (k::\text{nat}) \ \text{dvd} \ n$   
**by**(*metis gcd-dvd2-nat dvd-trans*)

**lemma** *dvd-gcd-D1-int*:  $i \ \text{dvd} \ \text{gcd} \ m \ n \implies (i::\text{int}) \ \text{dvd} \ m$   
**by**(*metis gcd-dvd1-int dvd-trans*)

**lemma** *dvd-gcd-D2-int*:  $i \ \text{dvd} \ \text{gcd} \ m \ n \implies (i::\text{int}) \ \text{dvd} \ n$   
**by**(*metis gcd-dvd2-int dvd-trans*)

**lemma** *gcd-le1-nat* [*simp*]:  $a \neq 0 \implies \text{gcd} \ (a::\text{nat}) \ b \leq a$   
**by** (*rule dvd-imp-le, auto*)

**lemma** *gcd-le2-nat* [*simp*]:  $b \neq 0 \implies \text{gcd} \ (a::\text{nat}) \ b \leq b$   
**by** (*rule dvd-imp-le, auto*)

**lemma** *gcd-le1-int* [*simp*]:  $a > 0 \implies \text{gcd} \ (a::\text{int}) \ b \leq a$   
**by** (*rule zdvd-imp-le, auto*)

**lemma** *gcd-le2-int* [*simp*]:  $b > 0 \implies \text{gcd} \ (a::\text{int}) \ b \leq b$   
**by** (*rule zdvd-imp-le, auto*)

**lemma** *gcd-greatest-nat*:  $(k::\text{nat}) \ \text{dvd} \ m \implies k \ \text{dvd} \ n \implies k \ \text{dvd} \ \text{gcd} \ m \ n$   
**by** (*induct m n rule: gcd-nat-induct*) (*simp-all add: gcd-non-0-nat dvd-mod*)

**lemma** *gcd-greatest-int*:  
 $(k::\text{int}) \ \text{dvd} \ m \implies k \ \text{dvd} \ n \implies k \ \text{dvd} \ \text{gcd} \ m \ n$   
**apply** (*subst gcd-abs-int*)  
**apply** (*subst abs-dvd-iff [symmetric]*)  
**apply** (*rule gcd-greatest-nat [transferred]*)  
**apply** *auto*  
**done**

**lemma** *gcd-greatest-iff-nat* [*iff*]:  $(k \ \text{dvd} \ \text{gcd} \ (m::\text{nat}) \ n) =$   
 $(k \ \text{dvd} \ m \ \& \ k \ \text{dvd} \ n)$   
**by** (*blast intro!: gcd-greatest-nat intro: dvd-trans*)

**lemma** *gcd-greatest-iff-int*:  $((k::\text{int}) \ \text{dvd} \ \text{gcd} \ m \ n) = (k \ \text{dvd} \ m \ \& \ k \ \text{dvd} \ n)$   
**by** (*blast intro!: gcd-greatest-int intro: dvd-trans*)

**lemma** *gcd-zero-nat* [*simp*]:  $(\text{gcd} \ (m::\text{nat}) \ n = 0) = (m = 0 \ \& \ n = 0)$   
**by** (*simp only: dvd-0-left-iff [symmetric] gcd-greatest-iff-nat*)

**lemma** *gcd-zero-int* [*simp*]:  $(\text{gcd } (m::\text{int}) \ n = 0) = (m = 0 \ \& \ n = 0)$   
**by** (*auto simp add: gcd-int-def*)

**lemma** *gcd-pos-nat* [*simp*]:  $(\text{gcd } (m::\text{nat}) \ n > 0) = (m \sim = 0 \mid n \sim = 0)$   
**by** (*insert gcd-zero-nat [of m n], arith*)

**lemma** *gcd-pos-int* [*simp*]:  $(\text{gcd } (m::\text{int}) \ n > 0) = (m \sim = 0 \mid n \sim = 0)$   
**by** (*insert gcd-zero-int [of m n], insert gcd-ge-0-int [of m n], arith*)

**interpretation** *gcd-nat!*: *abel-semigroup gcd :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat*  
**proof**  
**qed** (*auto intro: dvd-antisym dvd-trans*)

**interpretation** *gcd-int!*: *abel-semigroup gcd :: int  $\Rightarrow$  int  $\Rightarrow$  int*  
**proof**  
**qed** (*simp-all add: gcd-int-def gcd-nat.assoc gcd-nat.commute gcd-nat.left-commute*)

**lemmas** *gcd-assoc-nat* = *gcd-nat.assoc*  
**lemmas** *gcd-commute-nat* = *gcd-nat.commute*  
**lemmas** *gcd-left-commute-nat* = *gcd-nat.left-commute*  
**lemmas** *gcd-assoc-int* = *gcd-int.assoc*  
**lemmas** *gcd-commute-int* = *gcd-int.commute*  
**lemmas** *gcd-left-commute-int* = *gcd-int.left-commute*

**lemmas** *gcd-ac-nat* = *gcd-assoc-nat gcd-commute-nat gcd-left-commute-nat*

**lemmas** *gcd-ac-int* = *gcd-assoc-int gcd-commute-int gcd-left-commute-int*

**lemma** *gcd-unique-nat*:  $(d::\text{nat}) \ \text{dvd } a \ \wedge \ d \ \text{dvd } b \ \wedge$   
 $(\forall e. \ e \ \text{dvd } a \ \wedge \ e \ \text{dvd } b \longrightarrow e \ \text{dvd } d) \longleftrightarrow d = \text{gcd } a \ b$   
**apply** *auto*  
**apply** (*rule dvd-antisym*)  
**apply** (*erule (1) gcd-greatest-nat*)  
**apply** *auto*  
**done**

**lemma** *gcd-unique-int*:  $d \geq 0 \ \& \ (d::\text{int}) \ \text{dvd } a \ \wedge \ d \ \text{dvd } b \ \wedge$   
 $(\forall e. \ e \ \text{dvd } a \ \wedge \ e \ \text{dvd } b \longrightarrow e \ \text{dvd } d) \longleftrightarrow d = \text{gcd } a \ b$   
**apply** (*case-tac d = 0*)  
**apply** *simp*  
**apply** (*rule iffI*)  
**apply** (*rule zdvd-antisym-nonneg*)  
**apply** (*auto intro: gcd-greatest-int*)  
**done**

**lemma** *gcd-proj1-if-dvd-nat* [*simp*]:  $(x::\text{nat}) \ \text{dvd } y \Longrightarrow \text{gcd } x \ y = x$   
**by** (*metis dvd.eq-iff gcd-unique-nat*)

**lemma** *gcd-proj2-if-dvd-nat* [simp]:  $(y::nat) \text{ dvd } x \implies \text{gcd } x \ y = y$   
**by** (*metis* *dvd.eq-iff* *gcd-unique-nat*)

**lemma** *gcd-proj1-if-dvd-int* [simp]:  $x \text{ dvd } y \implies \text{gcd } (x::int) \ y = \text{abs } x$   
**by** (*metis* *abs-dvd-iff* *abs-eq-0* *gcd-0-left-int* *gcd-abs-int* *gcd-unique-int*)

**lemma** *gcd-proj2-if-dvd-int* [simp]:  $y \text{ dvd } x \implies \text{gcd } (x::int) \ y = \text{abs } y$   
**by** (*metis* *gcd-proj1-if-dvd-int* *gcd-commute-int*)

Multiplication laws

**lemma** *gcd-mult-distrib-nat*:  $(k::nat) * \text{gcd } m \ n = \text{gcd } (k * m) \ (k * n)$   
 — [?, page 27]  
**apply** (*induct* *m n* *rule*: *gcd-nat-induct*)  
**apply** *simp*  
**apply** (*case-tac* *k = 0*)  
**apply** (*simp-all* *add*: *mod-geq* *gcd-non-0-nat* *mod-mult-distrib2*)  
**done**

**lemma** *gcd-mult-distrib-int*:  $\text{abs } (k::int) * \text{gcd } m \ n = \text{gcd } (k * m) \ (k * n)$   
**apply** (*subst* (1 2) *gcd-abs-int*)  
**apply** (*subst* (1 2) *abs-mult*)  
**apply** (*rule* *gcd-mult-distrib-nat* [*transferred*])  
**apply** *auto*  
**done**

**lemma** *coprime-dvd-mult-nat*:  $\text{coprime } (k::nat) \ n \implies k \text{ dvd } m * n \implies k \text{ dvd } m$   
**apply** (*insert* *gcd-mult-distrib-nat* [*of m k n*])  
**apply** *simp*  
**apply** (*erule-tac* *t = m* **in** *ssubst*)  
**apply** *simp*  
**done**

**lemma** *coprime-dvd-mult-int*:  
 $\text{coprime } (k::int) \ n \implies k \text{ dvd } m * n \implies k \text{ dvd } m$   
**apply** (*subst* *abs-dvd-iff* [*symmetric*])  
**apply** (*subst* *dvd-abs-iff* [*symmetric*])  
**apply** (*subst* (*asm*) *gcd-abs-int*)  
**apply** (*rule* *coprime-dvd-mult-nat* [*transferred*])  
**prefer** 4 **apply** *assumption*  
**apply** *auto*  
**apply** (*subst* *abs-mult* [*symmetric*], *auto*)  
**done**

**lemma** *coprime-dvd-mult-iff-nat*:  $\text{coprime } (k::nat) \ n \implies$   
 $(k \text{ dvd } m * n) = (k \text{ dvd } m)$   
**by** (*auto* *intro*: *coprime-dvd-mult-nat*)

**lemma** *coprime-dvd-mult-iff-int*:  $\text{coprime } (k::int) \ n \implies$   
 $(k \text{ dvd } m * n) = (k \text{ dvd } m)$

```

by (auto intro: coprime-dvd-mult-int)

lemma gcd-mult-cancel-nat: coprime k n  $\implies$  gcd ((k::nat) * m) n = gcd m n
  apply (rule dvd-antisym)
  apply (rule gcd-greatest-nat)
  apply (rule-tac n = k in coprime-dvd-mult-nat)
  apply (simp add: gcd-assoc-nat)
  apply (simp add: gcd-commute-nat)
  apply (simp-all add: mult-commute)
done

lemma gcd-mult-cancel-int:
  coprime (k::int) n  $\implies$  gcd (k * m) n = gcd m n
  apply (subst (1 2) gcd-abs-int)
  apply (subst abs-mult)
  apply (rule gcd-mult-cancel-nat [transferred], auto)
done

lemma coprime-crossproduct-nat:
  fixes a b c d :: nat
  assumes coprime a d and coprime b c
  shows a * c = b * d  $\longleftrightarrow$  a = b  $\wedge$  c = d (is ?lhs  $\longleftrightarrow$  ?rhs)
proof
  assume ?rhs then show ?lhs by simp
next
  assume ?lhs
  from <?lhs> have a dvd b * d by (auto intro: dvdI dest: sym)
  with <coprime a d> have a dvd b by (simp add: coprime-dvd-mult-iff-nat)
  from <?lhs> have b dvd a * c by (auto intro: dvdI dest: sym)
  with <coprime b c> have b dvd a by (simp add: coprime-dvd-mult-iff-nat)
  from <?lhs> have c dvd d * b by (auto intro: dvdI dest: sym simp add: mult-commute)
  with <coprime b c> have c dvd d by (simp add: coprime-dvd-mult-iff-nat gcd-commute-nat)
  from <?lhs> have d dvd c * a by (auto intro: dvdI dest: sym simp add: mult-commute)
  with <coprime a d> have d dvd c by (simp add: coprime-dvd-mult-iff-nat gcd-commute-nat)
  from <a dvd b> <b dvd a> have a = b by (rule Nat.dvd.antisym)
  moreover from <c dvd d> <d dvd c> have c = d by (rule Nat.dvd.antisym)
  ultimately show ?rhs ..
qed

lemma coprime-crossproduct-int:
  fixes a b c d :: int
  assumes coprime a d and coprime b c
  shows |a| * |c| = |b| * |d|  $\longleftrightarrow$  |a| = |b|  $\wedge$  |c| = |d|
  using assms by (intro coprime-crossproduct-nat [transferred]) auto

Addition laws

lemma gcd-add1-nat [simp]: gcd ((m::nat) + n) n = gcd m n
  apply (case-tac n = 0)
  apply (simp-all add: gcd-non-0-nat)

```

done

```
lemma gcd-add2-nat [simp]: gcd (m::nat) (m + n) = gcd m n
  apply (subst (1 2) gcd-commute-nat)
  apply (subst add-commute)
  apply simp
done
```

```
lemma gcd-diff1-nat: (m::nat) >= n  $\implies$  gcd (m - n) n = gcd m n
  by (subst gcd-add1-nat [symmetric], auto)
```

```
lemma gcd-diff2-nat: (n::nat) >= m  $\implies$  gcd (n - m) n = gcd m n
  apply (subst gcd-commute-nat)
  apply (subst gcd-diff1-nat [symmetric])
  apply auto
  apply (subst gcd-commute-nat)
  apply (subst gcd-diff1-nat)
  apply assumption
  apply (rule gcd-commute-nat)
done
```

```
lemma gcd-non-0-int: (y::int) > 0  $\implies$  gcd x y = gcd y (x mod y)
  apply (frule-tac b = y and a = x in pos-mod-sign)
  apply (simp del: pos-mod-sign add: gcd-int-def abs-if nat-mod-distrib)
  apply (auto simp add: gcd-non-0-nat nat-mod-distrib [symmetric]
    zmod-zminus1-eq-if)
  apply (frule-tac a = x in pos-mod-bound)
  apply (subst (1 2) gcd-commute-nat)
  apply (simp del: pos-mod-bound add: nat-diff-distrib gcd-diff2-nat
    nat-le-eq-zle)
done
```

```
lemma gcd-red-int: gcd (x::int) y = gcd y (x mod y)
  apply (case-tac y = 0)
  apply force
  apply (case-tac y > 0)
  apply (subst gcd-non-0-int, auto)
  apply (insert gcd-non-0-int [of -y -x])
  apply auto
done
```

```
lemma gcd-add1-int [simp]: gcd ((m::int) + n) n = gcd m n
  by (metis gcd-red-int mod-add-self1 zadd-commute)
```

```
lemma gcd-add2-int [simp]: gcd m ((m::int) + n) = gcd m n
  by (metis gcd-add1-int gcd-commute-int zadd-commute)
```

**lemma** *gcd-add-mult-nat*:  $\text{gcd } (m::\text{nat}) (k * m + n) = \text{gcd } m \ n$   
**by** (*metis mod-mult-self3 gcd-commute-nat gcd-red-nat*)

**lemma** *gcd-add-mult-int*:  $\text{gcd } (m::\text{int}) (k * m + n) = \text{gcd } m \ n$   
**by** (*metis gcd-commute-int gcd-red-int mod-mult-self1 zadd-commute*)

**lemma** *gcd-dvd-prod-nat* [*iff*]:  $\text{gcd } (m::\text{nat}) \ n \ \text{dvd} \ k * n$   
**using** *mult-dvd-mono* [*of 1*] **by** *auto*

**lemma** *finite-divisors-nat*[*simp*]:  
**assumes**  $(m::\text{nat}) \sim= 0$  **shows**  $\text{finite}\{d. \ d \ \text{dvd} \ m\}$   
**proof**–  
**have**  $\text{finite}\{d. \ d \leq m\}$  **by**(*blast intro: bounded-nat-set-is-finite*)  
**from** *finite-subset[OF - this]* **show** *?thesis* **using** *assms*  
**by**(*bestsimp intro!: dvd-imp-le*)  
**qed**

**lemma** *finite-divisors-int*[*simp*]:  
**assumes**  $(i::\text{int}) \sim= 0$  **shows**  $\text{finite}\{d. \ d \ \text{dvd} \ i\}$   
**proof**–  
**have**  $\{d. \ \text{abs } d \leq \text{abs } i\} = \{- \ \text{abs } i \ .. \ \text{abs } i\}$  **by**(*auto simp: abs-if*)  
**hence**  $\text{finite}\{d. \ \text{abs } d \leq \text{abs } i\}$  **by** *simp*  
**from** *finite-subset[OF - this]* **show** *?thesis* **using** *assms*  
**by**(*bestsimp intro!: dvd-imp-le-int*)  
**qed**

**lemma** *Max-divisors-self-nat*[*simp*]:  $n \neq 0 \implies \text{Max}\{d::\text{nat}. \ d \ \text{dvd} \ n\} = n$   
**apply**(*rule antisym*)  
**apply** (*fastsimp intro: Max-le-iff[THEN iffD2] simp: dvd-imp-le*)  
**apply** *simp*  
**done**

**lemma** *Max-divisors-self-int*[*simp*]:  $n \neq 0 \implies \text{Max}\{d::\text{int}. \ d \ \text{dvd} \ n\} = \text{abs } n$   
**apply**(*rule antisym*)  
**apply**(*rule Max-le-iff[THEN iffD2]*)  
**apply** *simp*  
**apply** *fastsimp*  
**apply** (*metis Collect-def abs-ge-self dvd-imp-le-int mem-def zle-trans*)  
**apply** *simp*  
**done**

**lemma** *gcd-is-Max-divisors-nat*:  
 $m \sim= 0 \implies n \sim= 0 \implies \text{gcd } (m::\text{nat}) \ n = (\text{Max } \{d. \ d \ \text{dvd} \ m \ \& \ d \ \text{dvd} \ n\})$

```

apply(rule Max-eqI[THEN sym])
  apply (metis finite-Collect-conjI finite-divisors-nat)
  apply simp
  apply(metis Suc-diff-1 Suc-neq-Zero dvd-imp-le gcd-greatest-iff-nat gcd-pos-nat)
apply simp
done

```

```

lemma gcd-is-Max-divisors-int:
   $m \sim 0 \implies n \sim 0 \implies \text{gcd } (m::\text{int}) \ n = (\text{Max } \{d. d \text{ dvd } m \ \& \ d \text{ dvd } n\})$ 
apply(rule Max-eqI[THEN sym])
  apply (metis finite-Collect-conjI finite-divisors-int)
  apply simp
  apply (metis gcd-greatest-iff-int gcd-pos-int zdvd-imp-le)
apply simp
done

```

```

lemma gcd-code-int [code]:
   $\text{gcd } k \ l = \text{if } l = (0::\text{int}) \text{ then } k \text{ else } \text{gcd } l \ (|k| \bmod |l|)$ 
  by (simp add: gcd-int-def nat-mod-distrib gcd-non-0-nat)

```

### 63.4 Coprimality

```

lemma div-gcd-coprime-nat:
  assumes nz:  $(a::\text{nat}) \neq 0 \vee b \neq 0$ 
  shows coprime  $(a \text{ div gcd } a \ b) \ (b \text{ div gcd } a \ b)$ 
proof –
  let ?g = gcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = gcd ?a' ?b'
  have dvdg: ?g dvd a ?g dvd b by simp-all
  have dvdg': ?g' dvd ?a' ?g' dvd ?b' by simp-all
  from dvdg dvdg' obtain ka kb ka' kb' where
    kab:  $a = ?g * ka \ b = ?g * kb \ ?a' = ?g' * ka' \ ?b' = ?g' * kb'$ 
  unfolding dvd-def by blast
  then have ?g * ?a' = (?g * ?g') * ka' ?g * ?b' = (?g * ?g') * kb'
    by simp-all
  then have dvdgg': ?g * ?g' dvd a ?g * ?g' dvd b
    by (auto simp add: dvd-mult-div-cancel [OF dvdg(1)]
      dvd-mult-div-cancel [OF dvdg(2)] dvd-def)
  have ?g  $\neq 0$  using nz by simp
  then have gp: ?g > 0 by arith
  from gcd-greatest-nat [OF dvdgg'] have ?g * ?g' dvd ?g .
  with dvd-mult-cancel1 [OF gp] show ?g' = 1 by simp
qed

```

```

lemma div-gcd-coprime-int:
  assumes nz:  $(a::\text{int}) \neq 0 \vee b \neq 0$ 
  shows coprime  $(a \text{ div gcd } a \ b) \ (b \text{ div gcd } a \ b)$ 

```

```

apply (subst (1 2 3) gcd-abs-int)
apply (subst (1 2) abs-div)
  apply simp
  apply simp
apply (subst (1 2) abs-gcd-int)
apply (rule div-gcd-coprime-nat [transferred])
using nz apply (auto simp add: gcd-abs-int [symmetric])
done

```

```

lemma coprime-nat: coprime (a::nat) b  $\longleftrightarrow$  ( $\forall d. d \text{ dvd } a \wedge d \text{ dvd } b \longleftrightarrow d = 1$ )
  using gcd-unique-nat[of 1 a b, simplified] by auto

```

```

lemma coprime-Suc-0-nat:
  coprime (a::nat) b  $\longleftrightarrow$  ( $\forall d. d \text{ dvd } a \wedge d \text{ dvd } b \longleftrightarrow d = \text{Suc } 0$ )
  using coprime-nat by (simp add: One-nat-def)

```

```

lemma coprime-int: coprime (a::int) b  $\longleftrightarrow$ 
  ( $\forall d. d \geq 0 \wedge d \text{ dvd } a \wedge d \text{ dvd } b \longleftrightarrow d = 1$ )
  using gcd-unique-int [of 1 a b]
  apply clarsimp
  apply (erule subst)
  apply (rule iffI)
  apply force
  apply (drule-tac x = abs e in exI)
  apply (case-tac e  $\geq 0$ )
  apply force
  apply force
done

```

```

lemma gcd-coprime-nat:
  assumes z: gcd (a::nat) b  $\neq 0$  and a: a = a' * gcd a b and
    b: b = b' * gcd a b
  shows coprime a' b'

```

```

  apply (subgoal-tac a' = a div gcd a b)
  apply (erule ssubst)
  apply (subgoal-tac b' = b div gcd a b)
  apply (erule ssubst)
  apply (rule div-gcd-coprime-nat)
  using prems
  apply force
  apply (subst (1) b)
  using z apply force
  apply (subst (1) a)
  using z apply force
done

```

```

lemma gcd-coprime-int:
  assumes z: gcd (a::int) b  $\neq 0$  and a: a = a' * gcd a b and

```



$b: b = b' * \text{gcd } a \ b$   
**shows**  $\text{coprime } a' \ b'$

**apply** (*subgoal-tac*  $a' = a \text{ div } \text{gcd } a \ b$ )  
**apply** (*erule* *ssubst*)  
**apply** (*subgoal-tac*  $b' = b \text{ div } \text{gcd } a \ b$ )  
**apply** (*erule* *ssubst*)  
**apply** (*rule* *div-gcd-coprime-int*)  
**using** *prems*  
**apply** *force*  
**apply** (*subst* (1) *b*)  
**using** *z* **apply** *force*  
**apply** (*subst* (1) *a*)  
**using** *z* **apply** *force*  
**done**

**lemma** *coprime-mult-nat*: **assumes** *da*:  $\text{coprime } (d::\text{nat}) \ a$  **and** *db*:  $\text{coprime } d \ b$   
**shows**  $\text{coprime } d \ (a * b)$   
**apply** (*subst* *gcd-commute-nat*)  
**using** *da* **apply** (*subst* *gcd-mult-cancel-nat*)  
**apply** (*subst* *gcd-commute-nat*, *assumption*)  
**apply** (*subst* *gcd-commute-nat*, *rule* *db*)  
**done**

**lemma** *coprime-mult-int*: **assumes** *da*:  $\text{coprime } (d::\text{int}) \ a$  **and** *db*:  $\text{coprime } d \ b$   
**shows**  $\text{coprime } d \ (a * b)$   
**apply** (*subst* *gcd-commute-int*)  
**using** *da* **apply** (*subst* *gcd-mult-cancel-int*)  
**apply** (*subst* *gcd-commute-int*, *assumption*)  
**apply** (*subst* *gcd-commute-int*, *rule* *db*)  
**done**

**lemma** *coprime-lmult-nat*:  
**assumes** *dab*:  $\text{coprime } (d::\text{nat}) \ (a * b)$  **shows**  $\text{coprime } d \ a$   
**proof** –  
**have**  $\text{gcd } d \ a \ \text{dvd } \text{gcd } d \ (a * b)$   
**by** (*rule* *gcd-greatest-nat*, *auto*)  
**with** *dab* **show** *?thesis*  
**by** *auto*  
**qed**

**lemma** *coprime-lmult-int*:  
**assumes**  $\text{coprime } (d::\text{int}) \ (a * b)$  **shows**  $\text{coprime } d \ a$   
**proof** –  
**have**  $\text{gcd } d \ a \ \text{dvd } \text{gcd } d \ (a * b)$   
**by** (*rule* *gcd-greatest-int*, *auto*)  
**with** *assms* **show** *?thesis*  
**by** *auto*  
**qed**

```

lemma coprime-rmult-nat:
  assumes coprime (d::nat) (a * b) shows coprime d b
proof –
  have gcd d b dvd gcd d (a * b)
    by (rule gcd-greatest-nat, auto intro: dvd-mult)
  with assms show ?thesis
    by auto
qed

lemma coprime-rmult-int:
  assumes dab: coprime (d::int) (a * b) shows coprime d b
proof –
  have gcd d b dvd gcd d (a * b)
    by (rule gcd-greatest-int, auto intro: dvd-mult)
  with dab show ?thesis
    by auto
qed

lemma coprime-mul-eq-nat: coprime (d::nat) (a * b)  $\longleftrightarrow$ 
  coprime d a  $\wedge$  coprime d b
  using coprime-rmult-nat[of d a b] coprime-lmult-nat[of d a b]
  coprime-mult-nat[of d a b]
  by blast

lemma coprime-mul-eq-int: coprime (d::int) (a * b)  $\longleftrightarrow$ 
  coprime d a  $\wedge$  coprime d b
  using coprime-rmult-int[of d a b] coprime-lmult-int[of d a b]
  coprime-mult-int[of d a b]
  by blast

lemma gcd-coprime-exists-nat:
  assumes nz: gcd (a::nat) b  $\neq$  0
  shows  $\exists a' b'. a = a' * \text{gcd } a \ b \wedge b = b' * \text{gcd } a \ b \wedge \text{coprime } a' \ b'$ 
  apply (rule-tac x = a div gcd a b in exI)
  apply (rule-tac x = b div gcd a b in exI)
  using nz apply (auto simp add: div-gcd-coprime-nat dvd-div-mult)
done

lemma gcd-coprime-exists-int:
  assumes nz: gcd (a::int) b  $\neq$  0
  shows  $\exists a' b'. a = a' * \text{gcd } a \ b \wedge b = b' * \text{gcd } a \ b \wedge \text{coprime } a' \ b'$ 
  apply (rule-tac x = a div gcd a b in exI)
  apply (rule-tac x = b div gcd a b in exI)
  using nz apply (auto simp add: div-gcd-coprime-int dvd-div-mult-self)
done

lemma coprime-exp-nat: coprime (d::nat) a  $\implies$  coprime d (a^n)
  by (induct n, simp-all add: coprime-mult-nat)

```

**lemma** *coprime-exp-int*:  $\text{coprime } (d::\text{int}) \ a \implies \text{coprime } d \ (a^n)$   
**by** (*induct n*, *simp-all add: coprime-mult-int*)

**lemma** *coprime-exp2-nat* [*intro*]:  $\text{coprime } (a::\text{nat}) \ b \implies \text{coprime } (a^n) \ (b^m)$   
**apply** (*rule coprime-exp-nat*)  
**apply** (*subst gcd-commute-nat*)  
**apply** (*rule coprime-exp-nat*)  
**apply** (*subst gcd-commute-nat, assumption*)  
**done**

**lemma** *coprime-exp2-int* [*intro*]:  $\text{coprime } (a::\text{int}) \ b \implies \text{coprime } (a^n) \ (b^m)$   
**apply** (*rule coprime-exp-int*)  
**apply** (*subst gcd-commute-int*)  
**apply** (*rule coprime-exp-int*)  
**apply** (*subst gcd-commute-int, assumption*)  
**done**

**lemma** *gcd-exp-nat*:  $\text{gcd } ((a::\text{nat})^n) \ (b^n) = (\text{gcd } a \ b)^n$   
**proof** (*cases*)  
**assume**  $a = 0 \ \& \ b = 0$   
**thus** *?thesis* **by** *simp*  
**next assume**  $\sim(a = 0 \ \& \ b = 0)$   
**hence**  $\text{coprime } ((a \ \text{div} \ \text{gcd } a \ b)^n) \ ((b \ \text{div} \ \text{gcd } a \ b)^n)$   
**by** (*auto simp:div-gcd-coprime-nat*)  
**hence**  $\text{gcd } ((a \ \text{div} \ \text{gcd } a \ b)^n * (\text{gcd } a \ b)^n)$   
 $((b \ \text{div} \ \text{gcd } a \ b)^n * (\text{gcd } a \ b)^n) = (\text{gcd } a \ b)^n$   
**apply** (*subst (1 2) mult-commute*)  
**apply** (*subst gcd-mult-distrib-nat [symmetric]*)  
**apply** *simp*  
**done**  
**also have**  $(a \ \text{div} \ \text{gcd } a \ b)^n * (\text{gcd } a \ b)^n = a^n$   
**apply** (*subst div-power*)  
**apply** *auto*  
**apply** (*rule dvd-div-mult-self*)  
**apply** (*rule dvd-power-same*)  
**apply** *auto*  
**done**  
**also have**  $(b \ \text{div} \ \text{gcd } a \ b)^n * (\text{gcd } a \ b)^n = b^n$   
**apply** (*subst div-power*)  
**apply** *auto*  
**apply** (*rule dvd-div-mult-self*)  
**apply** (*rule dvd-power-same*)  
**apply** *auto*  
**done**  
**finally show** *?thesis* .  
**qed**

**lemma** *gcd-exp-int*:  $\text{gcd } ((a::\text{int})^n) \ (b^n) = (\text{gcd } a \ b)^n$

```

apply (subst (1 2) gcd-abs-int)
apply (subst (1 2) power-abs)
apply (rule gcd-exp-nat [where n = n, transferred])
apply auto
done

```

```

lemma division-decomp-nat: assumes dc: (a::nat) dvd b * c
  shows  $\exists b' c'. a = b' * c' \wedge b' \text{ dvd } b \wedge c' \text{ dvd } c$ 
proof–
  let ?g = gcd a b
  {assume ?g = 0 with dc have ?thesis by auto}
  moreover
  {assume z: ?g  $\neq$  0
    from gcd-coprime-exists-nat[OF z]
    obtain a' b' where ab': a = a' * ?g b = b' * ?g coprime a' b'
      by blast
    have thb: ?g dvd b by auto
    from ab'(1) have a' dvd a unfolding dvd-def by blast
    with dc have th0: a' dvd b*c using dvd-trans[of a' a b*c] by simp
    from dc ab'(1,2) have a'*?g dvd (b'*?g)*c by auto
    hence ?g*a' dvd ?g*(b'*c) by (simp add: mult-assoc)
    with z have th-1: a' dvd b'*c by auto
    from coprime-dvd-mult-nat[OF ab'(3)] th-1
    have thc: a' dvd c by (subst (asm) mult-commute, blast)
    from ab' have a = ?g*a' by algebra
    with thb thc have ?thesis by blast }
  ultimately show ?thesis by blast
qed

```

```

lemma division-decomp-int: assumes dc: (a::int) dvd b * c
  shows  $\exists b' c'. a = b' * c' \wedge b' \text{ dvd } b \wedge c' \text{ dvd } c$ 
proof–
  let ?g = gcd a b
  {assume ?g = 0 with dc have ?thesis by auto}
  moreover
  {assume z: ?g  $\neq$  0
    from gcd-coprime-exists-int[OF z]
    obtain a' b' where ab': a = a' * ?g b = b' * ?g coprime a' b'
      by blast
    have thb: ?g dvd b by auto
    from ab'(1) have a' dvd a unfolding dvd-def by blast
    with dc have th0: a' dvd b*c
      using dvd-trans[of a' a b*c] by simp
    from dc ab'(1,2) have a'*?g dvd (b'*?g)*c by auto
    hence ?g*a' dvd ?g*(b'*c) by (simp add: mult-assoc)
    with z have th-1: a' dvd b'*c by auto
    from coprime-dvd-mult-int[OF ab'(3)] th-1
    have thc: a' dvd c by (subst (asm) mult-commute, blast)
    from ab' have a = ?g*a' by algebra
  }

```

with *thb thc* have *?thesis* by *blast* }  
 ultimately show *?thesis* by *blast*  
 qed

lemma *pow-divides-pow-nat*:

assumes *ab*:  $(a::nat) \wedge n \text{ dvd } b \wedge n \neq 0$   
 shows  $a \text{ dvd } b$

proof—

let  $?g = \text{gcd } a \ b$   
 from *n* obtain *m* where  $m: n = \text{Suc } m$  by (*cases n, simp-all*)  
 {assume  $?g = 0$  with *ab n* have *?thesis* by *auto* }  
 moreover  
 {assume  $z: ?g \neq 0$   
 hence  $zn: ?g \wedge n \neq 0$  using *n* by *simp*  
 from *gcd-coprime-exists-nat*[*OF z*]  
 obtain *a' b'* where *ab'*:  $a = a' * ?g \wedge b = b' * ?g$  coprime *a' b'*  
 by *blast*  
 from *ab* have  $(a' * ?g) \wedge n \text{ dvd } (b' * ?g) \wedge n$   
 by (*simp add: ab'(1,2)[symmetric]*)  
 hence  $?g \wedge n * a' \wedge n \text{ dvd } ?g \wedge n * b' \wedge n$   
 by (*simp only: power-mult-distrib mult-commute*)  
 with  $zn \ z \ n$  have  $th0: a' \wedge n \text{ dvd } b' \wedge n$  by *auto*  
 have  $a' \text{ dvd } a' \wedge n$  by (*simp add: m*)  
 with *th0* have  $a' \text{ dvd } b' \wedge n$  using *dvd-trans*[*of a' a' ^ n b' ^ n*] by *simp*  
 hence  $th1: a' \text{ dvd } b' \wedge m * b'$  by (*simp add: m mult-commute*)  
 from *coprime-dvd-mult-nat*[*OF coprime-exp-nat [OF ab'(3), of m]*] *th1*  
 have  $a' \text{ dvd } b'$  by (*subst (asm) mult-commute, blast*)  
 hence  $a' * ?g \text{ dvd } b' * ?g$  by *simp*  
 with *ab'(1,2)* have *?thesis* by *simp* }  
 ultimately show *?thesis* by *blast*  
 qed

lemma *pow-divides-pow-int*:

assumes *ab*:  $(a::int) \wedge n \text{ dvd } b \wedge n \neq 0$   
 shows  $a \text{ dvd } b$

proof—

let  $?g = \text{gcd } a \ b$   
 from *n* obtain *m* where  $m: n = \text{Suc } m$  by (*cases n, simp-all*)  
 {assume  $?g = 0$  with *ab n* have *?thesis* by *auto* }  
 moreover  
 {assume  $z: ?g \neq 0$   
 hence  $zn: ?g \wedge n \neq 0$  using *n* by *simp*  
 from *gcd-coprime-exists-int*[*OF z*]  
 obtain *a' b'* where *ab'*:  $a = a' * ?g \wedge b = b' * ?g$  coprime *a' b'*  
 by *blast*  
 from *ab* have  $(a' * ?g) \wedge n \text{ dvd } (b' * ?g) \wedge n$   
 by (*simp add: ab'(1,2)[symmetric]*)  
 hence  $?g \wedge n * a' \wedge n \text{ dvd } ?g \wedge n * b' \wedge n$   
 by (*simp only: power-mult-distrib mult-commute*)

```

with zn z n have th0: a'^n dvd b'^n by auto
have a' dvd a'^n by (simp add: m)
with th0 have a' dvd b'^n
  using dvd-trans[of a' a'^n b'^n] by simp
hence th1: a' dvd b'^m * b' by (simp add: m mult-commute)
from coprime-dvd-mult-int[OF coprime-exp-int [OF ab'(3), of m]] th1
have a' dvd b' by (subst (asm) mult-commute, blast)
hence a'*?g dvd b'*?g by simp
with ab'(1,2) have ?thesis by simp }
ultimately show ?thesis by blast
qed

```

```

lemma pow-divides-eq-nat [simp]: n ~ = 0 ==> ((a::nat)^n dvd b^n) = (a dvd b)
  by (auto intro: pow-divides-pow-nat dvd-power-same)

```

```

lemma pow-divides-eq-int [simp]: n ~ = 0 ==> ((a::int)^n dvd b^n) = (a dvd b)
  by (auto intro: pow-divides-pow-int dvd-power-same)

```

```

lemma divides-mult-nat:
  assumes mr: (m::nat) dvd r and nr: n dvd r and mn: coprime m n
  shows m * n dvd r
proof-
  from mr nr obtain m' n' where m': r = m*m' and n': r = n*n'
  unfolding dvd-def by blast
  from mr n' have m dvd n'*n by (simp add: mult-commute)
  hence m dvd n' using coprime-dvd-mult-iff-nat[OF mn] by simp
  then obtain k where k: n' = m*k unfolding dvd-def by blast
  from n' k show ?thesis unfolding dvd-def by auto
qed

```

```

lemma divides-mult-int:
  assumes mr: (m::int) dvd r and nr: n dvd r and mn: coprime m n
  shows m * n dvd r
proof-
  from mr nr obtain m' n' where m': r = m*m' and n': r = n*n'
  unfolding dvd-def by blast
  from mr n' have m dvd n'*n by (simp add: mult-commute)
  hence m dvd n' using coprime-dvd-mult-iff-int[OF mn] by simp
  then obtain k where k: n' = m*k unfolding dvd-def by blast
  from n' k show ?thesis unfolding dvd-def by auto
qed

```

```

lemma coprime-plus-one-nat [simp]: coprime ((n::nat) + 1) n
  apply (subgoal-tac gcd (n + 1) n dvd (n + 1 - n))
  apply force
  apply (rule dvd-diff-nat)
  apply auto
done

```

```

lemma coprime-Suc-nat [simp]: coprime (Suc n) n
  using coprime-plus-one-nat by (simp add: One-nat-def)

lemma coprime-plus-one-int [simp]: coprime ((n::int) + 1) n
  apply (subgoal-tac gcd (n + 1) n dvd (n + 1 - n))
  apply force
  apply (rule dvd-diff)
  apply auto
done

lemma coprime-minus-one-nat: (n::nat) ≠ 0 ⇒ coprime (n - 1) n
  using coprime-plus-one-nat [of n - 1]
  gcd-commute-nat [of n - 1 n] by auto

lemma coprime-minus-one-int: coprime ((n::int) - 1) n
  using coprime-plus-one-int [of n - 1]
  gcd-commute-int [of n - 1 n] by auto

lemma setprod-coprime-nat [rule-format]:
  (ALL i: A. coprime (f i) (x::nat)) --> coprime (PROD i:A. f i) x
  apply (case-tac finite A)
  apply (induct set: finite)
  apply (auto simp add: gcd-mult-cancel-nat)
done

lemma setprod-coprime-int [rule-format]:
  (ALL i: A. coprime (f i) (x::int)) --> coprime (PROD i:A. f i) x
  apply (case-tac finite A)
  apply (induct set: finite)
  apply (auto simp add: gcd-mult-cancel-int)
done

lemma coprime-common-divisor-nat: coprime (a::nat) b ⇒ x dvd a ⇒
  x dvd b ⇒ x = 1
  apply (subgoal-tac x dvd gcd a b)
  apply simp
  apply (erule (1) gcd-greatest-nat)
done

lemma coprime-common-divisor-int: coprime (a::int) b ⇒ x dvd a ⇒
  x dvd b ⇒ abs x = 1
  apply (subgoal-tac x dvd gcd a b)
  apply simp
  apply (erule (1) gcd-greatest-int)
done

lemma coprime-divisors-nat: (d::int) dvd a ⇒ e dvd b ⇒ coprime a b ⇒
  coprime d e
  apply (auto simp add: dvd-def)

```

```

apply (frule coprime-lmult-int)
apply (subst gcd-commute-int)
apply (subst (asm) (2) gcd-commute-int)
apply (erule coprime-lmult-int)
done

```

```

lemma invertible-coprime-nat:  $(x::nat) * y \bmod m = 1 \implies \text{coprime } x \ m$ 
apply (metis coprime-lmult-nat gcd-1-nat gcd-commute-nat gcd-red-nat)
done

```

```

lemma invertible-coprime-int:  $(x::int) * y \bmod m = 1 \implies \text{coprime } x \ m$ 
apply (metis coprime-lmult-int gcd-1-int gcd-commute-int gcd-red-int)
done

```

### 63.5 Bezout’s theorem

```

fun
  bezw :: nat  $\Rightarrow$  nat  $\Rightarrow$  int * int
where
  bezw x y =
    (if y = 0 then (1, 0) else
     (snd (bezw y (x mod y)),
      fst (bezw y (x mod y)) - snd (bezw y (x mod y)) * int(x div y)))

```

```

lemma bezw-0 [simp]: bezw x 0 = (1, 0) by simp

```

```

lemma bezw-non-0:  $y > 0 \implies \text{bezw } x \ y = (\text{snd } (\text{bezw } y \ (x \bmod y)),$ 
   $\text{fst } (\text{bezw } y \ (x \bmod y)) - \text{snd } (\text{bezw } y \ (x \bmod y)) * \text{int}(x \text{ div } y))$ 
by simp

```

```

declare bezw.simps [simp del]

```

```

lemma bezw-aux [rule-format]:
   $\text{fst } (\text{bezw } x \ y) * \text{int } x + \text{snd } (\text{bezw } x \ y) * \text{int } y = \text{int } (\text{gcd } x \ y)$ 
proof (induct x y rule: gcd-nat-induct)
  fix m :: nat
  show  $\text{fst } (\text{bezw } m \ 0) * \text{int } m + \text{snd } (\text{bezw } m \ 0) * \text{int } 0 = \text{int } (\text{gcd } m \ 0)$ 
  by auto
  next fix m :: nat and n
  assume ngt0:  $n > 0$  and
    ih:  $\text{fst } (\text{bezw } n \ (m \bmod n)) * \text{int } n +$ 
       $\text{snd } (\text{bezw } n \ (m \bmod n)) * \text{int } (m \bmod n) =$ 
       $\text{int } (\text{gcd } n \ (m \bmod n))$ 
  thus  $\text{fst } (\text{bezw } m \ n) * \text{int } m + \text{snd } (\text{bezw } m \ n) * \text{int } n = \text{int } (\text{gcd } m \ n)$ 
  apply (simp add: bezw-non-0 gcd-non-0-nat)
  apply (erule subst)
  apply (simp add: field-simps)
  apply (subst mod-div-equality [of m n, symmetric])

```



```

    apply (simp only: field-simps zadd-int [symmetric]
      zmult-int [symmetric])
  done
qed

lemma bezout-int:
  fixes x y
  shows  $EX\ u\ v.\ u * (x::int) + v * y = gcd\ x\ y$ 
proof -
  have bezout-aux:  $!!x\ y.\ x \geq (0::int) \implies y \geq 0 \implies$ 
     $EX\ u\ v.\ u * x + v * y = gcd\ x\ y$ 
  apply (rule-tac x = fst (bezw (nat x) (nat y)) in exI)
  apply (rule-tac x = snd (bezw (nat x) (nat y)) in exI)
  apply (unfold gcd-int-def)
  apply simp
  apply (subst bezw-aux [symmetric])
  apply auto
  done
  have  $(x \geq 0 \wedge y \geq 0) \mid (x \geq 0 \wedge y \leq 0) \mid (x \leq 0 \wedge y \geq 0) \mid$ 
     $(x \leq 0 \wedge y \leq 0)$ 
  by auto
  moreover have  $x \geq 0 \implies y \geq 0 \implies ?thesis$ 
  by (erule (1) bezout-aux)
  moreover have  $x >= 0 \implies y <= 0 \implies ?thesis$ 
  apply (insert bezout-aux [of x -y])
  apply auto
  apply (rule-tac x = u in exI)
  apply (rule-tac x = -v in exI)
  apply (subst gcd-neg2-int [symmetric])
  apply auto
  done
  moreover have  $x <= 0 \implies y >= 0 \implies ?thesis$ 
  apply (insert bezout-aux [of -x y])
  apply auto
  apply (rule-tac x = -u in exI)
  apply (rule-tac x = v in exI)
  apply (subst gcd-neg1-int [symmetric])
  apply auto
  done
  moreover have  $x <= 0 \implies y <= 0 \implies ?thesis$ 
  apply (insert bezout-aux [of -x -y])
  apply auto
  apply (rule-tac x = -u in exI)
  apply (rule-tac x = -v in exI)
  apply (subst gcd-neg1-int [symmetric])
  apply (subst gcd-neg2-int [symmetric])
  apply auto
  done
  ultimately show ?thesis by blast

```

qed

versions of Bezout for nat, by Amine Chaieb

**lemma** *ind-euclid*:

**assumes** *c*:  $\forall a\ b. P\ (a::nat)\ b \longleftrightarrow P\ b\ a$  **and** *z*:  $\forall a. P\ a\ 0$

**and** *add*:  $\forall a\ b. P\ a\ b \longrightarrow P\ a\ (a + b)$

**shows**  $P\ a\ b$

**proof**(*induct*  $a + b$  *arbitrary*:  $a\ b$  *rule*: *less-induct*)

**case** *less*

**have**  $a = b \vee a < b \vee b < a$  **by** *arith*

**moreover** {**assume** *eq*:  $a = b$

**from** *add*[*rule-format*, *OF* *z*[*rule-format*, *of* *a*]] **have**  $P\ a\ b$  **using** *eq*

**by** *simp*}

**moreover**

{**assume** *lt*:  $a < b$

**hence**  $a + b - a < a + b \vee a = 0$  **by** *arith*

**moreover**

{**assume**  $a = 0$  **with** *z* *c* **have**  $P\ a\ b$  **by** *blast* }

**moreover**

{**assume**  $a + b - a < a + b$

**also** **have** *th0*:  $a + b - a = a + (b - a)$  **using** *lt* **by** *arith*

**finally** **have**  $a + (b - a) < a + b$  .

**then** **have**  $P\ a\ (a + (b - a))$  **by** (*rule* *add*[*rule-format*, *OF* *less*])

**then** **have**  $P\ a\ b$  **by** (*simp* *add*: *th0*[*symmetric*])}

**ultimately** **have**  $P\ a\ b$  **by** *blast*}

**moreover**

{**assume** *lt*:  $a > b$

**hence**  $b + a - b < a + b \vee b = 0$  **by** *arith*

**moreover**

{**assume**  $b = 0$  **with** *z* *c* **have**  $P\ a\ b$  **by** *blast* }

**moreover**

{**assume**  $b + a - b < a + b$

**also** **have** *th0*:  $b + a - b = b + (a - b)$  **using** *lt* **by** *arith*

**finally** **have**  $b + (a - b) < a + b$  .

**then** **have**  $P\ b\ (b + (a - b))$  **by** (*rule* *add*[*rule-format*, *OF* *less*])

**then** **have**  $P\ b\ a$  **by** (*simp* *add*: *th0*[*symmetric*])

**hence**  $P\ a\ b$  **using** *c* **by** *blast* }

**ultimately** **have**  $P\ a\ b$  **by** *blast*}

**ultimately** **show**  $P\ a\ b$  **by** *blast*

qed

**lemma** *bezout-lemma-nat*:

**assumes** *ex*:  $\exists (d::nat)\ x\ y. d\ dvd\ a \wedge d\ dvd\ b \wedge$

$(a * x = b * y + d \vee b * x = a * y + d)$

**shows**  $\exists d\ x\ y. d\ dvd\ a \wedge d\ dvd\ a + b \wedge$

$(a * x = (a + b) * y + d \vee (a + b) * x = a * y + d)$

**using** *ex*

**apply** *clarsimp*

**apply** (*rule-tac*  $x=d$  **in** *exI*, *simp*)

```

apply (case-tac  $a * x = b * y + d$  , simp-all)
apply (rule-tac  $x=x + y$  in exI)
apply (rule-tac  $x=y$  in exI)
apply algebra
apply (rule-tac  $x=x$  in exI)
apply (rule-tac  $x=x + y$  in exI)
apply algebra
done

```

```

lemma bezout-add-nat:  $\exists (d::nat) \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge$ 
  ( $a * x = b * y + d \vee b * x = a * y + d$ )
apply(induct a b rule: ind-euclid)
apply blast
apply clarify
apply (rule-tac  $x=a$  in exI, simp)
apply clarsimp
apply (rule-tac  $x=d$  in exI)
apply (case-tac  $a * x = b * y + d$  , simp-all)
apply (rule-tac  $x=x+y$  in exI)
apply (rule-tac  $x=y$  in exI)
apply algebra
apply (rule-tac  $x=x$  in exI)
apply (rule-tac  $x=x+y$  in exI)
apply algebra
done

```

```

lemma bezout1-nat:  $\exists (d::nat) \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge$ 
  ( $a * x - b * y = d \vee b * x - a * y = d$ )
using bezout-add-nat[of a b]
apply clarsimp
apply (rule-tac  $x=d$  in exI, simp)
apply (rule-tac  $x=x$  in exI)
apply (rule-tac  $x=y$  in exI)
apply auto
done

```

```

lemma bezout-add-strong-nat: assumes nz:  $a \neq (0::nat)$ 
shows  $\exists d \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge a * x = b * y + d$ 
proof–
from nz have ap:  $a > 0$  by simp
from bezout-add-nat[of a b]
have ( $\exists d \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge a * x = b * y + d$ )  $\vee$ 
  ( $\exists d \ x \ y. \ d \ \text{dvd} \ a \wedge d \ \text{dvd} \ b \wedge b * x = a * y + d$ ) by blast
moreover
  {fix  $d \ x \ y$  assume H:  $d \ \text{dvd} \ a \ d \ \text{dvd} \ b \ a * x = b * y + d$ 
   from H have ?thesis by blast }
moreover
  {fix  $d \ x \ y$  assume H:  $d \ \text{dvd} \ a \ d \ \text{dvd} \ b \ b * x = a * y + d$ 
   {assume b0:  $b = 0$  with H have ?thesis by simp}}

```

```

moreover
{assume  $b: b \neq 0$  hence  $bp: b > 0$  by simp
  from  $b \text{ dvd-imp-le } [OF\ H(2)]$  have  $d < b \vee d = b$ 
    by auto
  moreover
  {assume  $db: d=b$ 
    from prems have ?thesis apply simp
      apply (rule exI[where  $x = b$ ], simp)
      apply (rule exI[where  $x = b$ ])
      by (rule exI[where  $x = a - 1$ ], simp add: diff-mult-distrib2)
    moreover
    {assume  $db: d < b$ 
      {assume  $x=0$  hence ?thesis using prems by simp }
      moreover
      {assume  $x0: x \neq 0$  hence  $xp: x > 0$  by simp
        from  $db$  have  $d \leq b - 1$  by simp
        hence  $d*b \leq b*(b - 1)$  by simp
        with  $xp \text{ mult-mono}[of\ 1\ x\ d*b\ b*(b - 1)]$ 
        have  $dble: d*b \leq x*b*(b - 1)$  using  $bp$  by simp
        from  $H\ (3)$  have  $d + (b - 1) * (b*x) = d + (b - 1) * (a*y + d)$ 
          by simp
        hence  $d + (b - 1) * a * y + (b - 1) * d = d + (b - 1) * b * x$ 
          by (simp only: mult-assoc right-distrib)
        hence  $a * ((b - 1) * y) + d * (b - 1 + 1) = d + x*b*(b - 1)$ 
          by algebra
        hence  $a * ((b - 1) * y) = d + x*b*(b - 1) - d*b$  using  $bp$  by simp
        hence  $a * ((b - 1) * y) = d + (x*b*(b - 1) - d*b)$ 
          by (simp only: diff-add-assoc[OF dble, of d, symmetric])
        hence  $a * ((b - 1) * y) = b*(x*(b - 1) - d) + d$ 
          by (simp only: diff-mult-distrib2 add-commute mult-ac)
        hence ?thesis using  $H(1,2)$ 
          apply -
          apply (rule exI[where  $x=d$ ], simp)
          apply (rule exI[where  $x=(b - 1) * y$ ])
          by (rule exI[where  $x=x*(b - 1) - d$ ], simp)
        ultimately have ?thesis by blast
      }
    }
  }
ultimately have ?thesis by blast
ultimately have ?thesis by blast
ultimately show ?thesis by blast
qed

```

**lemma bezout-nat:** assumes  $a: (a::nat) \neq 0$

shows  $\exists x\ y. a * x = b * y + \text{gcd } a\ b$

**proof**—

let  $?g = \text{gcd } a\ b$

from *bezout-add-strong-nat*[*OF a, of b*]

obtain  $d\ x\ y$  where  $d: d \text{ dvd } a\ d \text{ dvd } b\ a * x = b * y + d$  by *blast*

from  $d(1,2)$  have  $d \text{ dvd } ?g$  by *simp*

then obtain  $k$  where  $k: ?g = d*k$  unfolding *dvd-def* by *blast*

```

from  $d(\mathcal{J})$  have  $a * x * k = (b * y + d) * k$  by auto
hence  $a * (x * k) = b * (y * k) + ?g$  by (algebra add: k)
thus ?thesis by blast
qed

```

### 63.6 LCM properties

```

lemma lcm-altdef-int [code]:  $\text{lcm } (a::\text{int}) \ b = (\text{abs } a) * (\text{abs } b) \ \text{div} \ \text{gcd } a \ b$ 
by (simp add: lcm-int-def lcm-nat-def zdiv-int
    zmult-int [symmetric] gcd-int-def)

```

```

lemma prod-gcd-lcm-nat:  $(m::\text{nat}) * n = \text{gcd } m \ n * \text{lcm } m \ n$ 
unfolding lcm-nat-def
by (simp add: dvd-mult-div-cancel [OF gcd-dvd-prod-nat])

```

```

lemma prod-gcd-lcm-int:  $\text{abs}(m::\text{int}) * \text{abs } n = \text{gcd } m \ n * \text{lcm } m \ n$ 
unfolding lcm-int-def gcd-int-def
apply (subst int-mult [symmetric])
apply (subst prod-gcd-lcm-nat [symmetric])
apply (subst nat-abs-mult-distrib [symmetric])
apply (simp, simp add: abs-mult)
done

```

```

lemma lcm-0-nat [simp]:  $\text{lcm } (m::\text{nat}) \ 0 = 0$ 
unfolding lcm-nat-def by simp

```

```

lemma lcm-0-int [simp]:  $\text{lcm } (m::\text{int}) \ 0 = 0$ 
unfolding lcm-int-def by simp

```

```

lemma lcm-0-left-nat [simp]:  $\text{lcm } (0::\text{nat}) \ n = 0$ 
unfolding lcm-nat-def by simp

```

```

lemma lcm-0-left-int [simp]:  $\text{lcm } (0::\text{int}) \ n = 0$ 
unfolding lcm-int-def by simp

```

```

lemma lcm-pos-nat:
   $(m::\text{nat}) > 0 \implies n > 0 \implies \text{lcm } m \ n > 0$ 
by (metis gr0I mult-is-0 prod-gcd-lcm-nat)

```

```

lemma lcm-pos-int:
   $(m::\text{int}) \sim 0 \implies n \sim 0 \implies \text{lcm } m \ n > 0$ 
apply (subst lcm-abs-int)
apply (rule lcm-pos-nat [transferred])
apply auto
done

```

```

lemma dvd-pos-nat:
  fixes  $n \ m :: \text{nat}$ 
  assumes  $n > 0$  and  $m \ \text{dvd} \ n$ 

```

```

  shows  $m > 0$ 
using assms by (cases m) auto

lemma lcm-least-nat:
  assumes ( $m::nat$ ) dvd k and n dvd k
  shows lcm m n dvd k
proof (cases k)
  case 0 then show ?thesis by auto
next
  case (Suc -) then have pos-k:  $k > 0$  by auto
  from assms dvd-pos-nat [OF this] have pos-mn:  $m > 0$   $n > 0$  by auto
  with gcd-zero-nat [of m n] have pos-gcd:  $\gcd m n > 0$  by simp
  from assms obtain p where k-m:  $k = m * p$  using dvd-def by blast
  from assms obtain q where k-n:  $k = n * q$  using dvd-def by blast
  from pos-k k-m have pos-p:  $p > 0$  by auto
  from pos-k k-n have pos-q:  $q > 0$  by auto
  have  $k * k * \gcd q p = k * \gcd (k * q) (k * p)$ 
    by (simp add: mult-ac gcd-mult-distrib-nat)
  also have  $\dots = k * \gcd (m * p * q) (n * q * p)$ 
    by (simp add: k-m [symmetric] k-n [symmetric])
  also have  $\dots = k * p * q * \gcd m n$ 
    by (simp add: mult-ac gcd-mult-distrib-nat)
  finally have  $(m * p) * (n * q) * \gcd q p = k * p * q * \gcd m n$ 
    by (simp only: k-m [symmetric] k-n [symmetric])
  then have  $p * q * m * n * \gcd q p = p * q * k * \gcd m n$ 
    by (simp add: mult-ac)
  with pos-p pos-q have  $m * n * \gcd q p = k * \gcd m n$ 
    by simp
  with prod-gcd-lcm-nat [of m n]
  have  $\text{lcm } m n * \gcd q p * \gcd m n = k * \gcd m n$ 
    by (simp add: mult-ac)
  with pos-gcd have  $\text{lcm } m n * \gcd q p = k$  by auto
  then show ?thesis using dvd-def by auto
qed

lemma lcm-least-int:
  ( $m::int$ ) dvd k  $\implies$  n dvd k  $\implies$  lcm m n dvd k
apply (subst lcm-abs-int)
apply (rule dvd-trans)
apply (rule lcm-least-nat [transferred, of - abs k -])
apply auto
done

lemma lcm-dvd1-nat: ( $m::nat$ ) dvd lcm m n
proof (cases m)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have mpos:  $m > 0$  by simp

```

```

show ?thesis
proof (cases n)
  case 0 then show ?thesis by simp
next
  case (Suc -)
  then have npos:  $n > 0$  by simp
  have gcd m n dvd n by simp
  then obtain k where  $n = \text{gcd } m \ n * k$  using dvd-def by auto
  then have  $m * n \text{ div } \text{gcd } m \ n = m * (\text{gcd } m \ n * k) \text{ div } \text{gcd } m \ n$ 
    by (simp add: mult-ac)
  also have  $\dots = m * k$  using mpos npos gcd-zero-nat by simp
  finally show ?thesis by (simp add: lcm-nat-def)
qed
qed

```

```

lemma lcm-dvd1-int:  $(m::\text{int}) \text{ dvd lcm } m \ n$ 
  apply (subst lcm-abs-int)
  apply (rule dvd-trans)
  prefer 2
  apply (rule lcm-dvd1-nat [transferred])
  apply auto
done

```

```

lemma lcm-dvd2-nat:  $(n::\text{nat}) \text{ dvd lcm } m \ n$ 
  using lcm-dvd1-nat [of n m] by (simp only: lcm-nat-def mult.commute gcd-nat.commute)

```

```

lemma lcm-dvd2-int:  $(n::\text{int}) \text{ dvd lcm } m \ n$ 
  using lcm-dvd1-int [of n m] by (simp only: lcm-int-def lcm-nat-def mult.commute
gcd-nat.commute)

```

```

lemma dvd-lcm-I1-nat[simp]:  $(k::\text{nat}) \text{ dvd } m \implies k \text{ dvd lcm } m \ n$ 
  by (metis lcm-dvd1-nat dvd-trans)

```

```

lemma dvd-lcm-I2-nat[simp]:  $(k::\text{nat}) \text{ dvd } n \implies k \text{ dvd lcm } m \ n$ 
  by (metis lcm-dvd2-nat dvd-trans)

```

```

lemma dvd-lcm-I1-int[simp]:  $(i::\text{int}) \text{ dvd } m \implies i \text{ dvd lcm } m \ n$ 
  by (metis lcm-dvd1-int dvd-trans)

```

```

lemma dvd-lcm-I2-int[simp]:  $(i::\text{int}) \text{ dvd } n \implies i \text{ dvd lcm } m \ n$ 
  by (metis lcm-dvd2-int dvd-trans)

```

```

lemma lcm-unique-nat:  $(a::\text{nat}) \text{ dvd } d \wedge b \text{ dvd } d \wedge$ 
   $(\forall e. a \text{ dvd } e \wedge b \text{ dvd } e \implies d \text{ dvd } e) \iff d = \text{lcm } a \ b$ 
  by (auto intro: dvd-antisym lcm-least-nat lcm-dvd1-nat lcm-dvd2-nat)

```

```

lemma lcm-unique-int:  $d \geq 0 \wedge (a::\text{int}) \text{ dvd } d \wedge b \text{ dvd } d \wedge$ 
   $(\forall e. a \text{ dvd } e \wedge b \text{ dvd } e \implies d \text{ dvd } e) \iff d = \text{lcm } a \ b$ 
  by (auto intro: dvd-antisym [transferred] lcm-least-int)

```

**interpretation** *lcm-nat!*: *abel-semigroup lcm :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat*

**proof**

fix *n m p :: nat*

show *lcm (lcm n m) p = lcm n (lcm m p)*

by (*rule lcm-unique-nat [THEN iffD1]*) (*metis dvd.order-trans lcm-unique-nat*)

show *lcm m n = lcm n m*

by (*simp add: lcm-nat-def gcd-commute-nat field-simps*)

**qed**

**interpretation** *lcm-int!*: *abel-semigroup lcm :: int  $\Rightarrow$  int  $\Rightarrow$  int*

**proof**

fix *n m p :: int*

show *lcm (lcm n m) p = lcm n (lcm m p)*

by (*rule lcm-unique-int [THEN iffD1]*) (*metis dvd-trans lcm-unique-int*)

show *lcm m n = lcm n m*

by (*simp add: lcm-int-def lcm-nat.commute*)

**qed**

**lemmas** *lcm-assoc-nat = lcm-nat.assoc*

**lemmas** *lcm-commute-nat = lcm-nat.commute*

**lemmas** *lcm-left-commute-nat = lcm-nat.left-commute*

**lemmas** *lcm-assoc-int = lcm-int.assoc*

**lemmas** *lcm-commute-int = lcm-int.commute*

**lemmas** *lcm-left-commute-int = lcm-int.left-commute*

**lemmas** *lcm-ac-nat = lcm-assoc-nat lcm-commute-nat lcm-left-commute-nat*

**lemmas** *lcm-ac-int = lcm-assoc-int lcm-commute-int lcm-left-commute-int*

**lemma** *lcm-proj2-if-dvd-nat [simp]: (x::nat) dvd y  $\Longrightarrow$  lcm x y = y*

apply (*rule sym*)

apply (*subst lcm-unique-nat [symmetric]*)

apply *auto*

**done**

**lemma** *lcm-proj2-if-dvd-int [simp]: (x::int) dvd y  $\Longrightarrow$  lcm x y = abs y*

apply (*rule sym*)

apply (*subst lcm-unique-int [symmetric]*)

apply *auto*

**done**

**lemma** *lcm-proj1-if-dvd-nat [simp]: (x::nat) dvd y  $\Longrightarrow$  lcm y x = y*

by (*subst lcm-commute-nat, erule lcm-proj2-if-dvd-nat*)

**lemma** *lcm-proj1-if-dvd-int [simp]: (x::int) dvd y  $\Longrightarrow$  lcm y x = abs y*

by (*subst lcm-commute-int, erule lcm-proj2-if-dvd-int*)

**lemma** *lcm-proj1-iff-nat[simp]: lcm m n = (m::nat)  $\longleftrightarrow$  n dvd m*

by (*metis lcm-proj1-if-dvd-nat lcm-unique-nat*)



**lemma** *lcm-proj2-iff-nat[simp]*:  $\text{lcm } m \ n = (n::\text{nat}) \longleftrightarrow m \text{ dvd } n$   
**by** (*metis lcm-proj2-if-dvd-nat lcm-unique-nat*)

**lemma** *lcm-proj1-iff-int[simp]*:  $\text{lcm } m \ n = \text{abs}(m::\text{int}) \longleftrightarrow n \text{ dvd } m$   
**by** (*metis dvd-abs-iff lcm-proj1-if-dvd-int lcm-unique-int*)

**lemma** *lcm-proj2-iff-int[simp]*:  $\text{lcm } m \ n = \text{abs}(n::\text{int}) \longleftrightarrow m \text{ dvd } n$   
**by** (*metis dvd-abs-iff lcm-proj2-if-dvd-int lcm-unique-int*)

**lemma** *fun-left-comm-idem-gcd-nat*: *fun-left-comm-idem* (*gcd* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*)  
**proof** **qed** (*auto simp add: gcd-ac-nat*)

**lemma** *fun-left-comm-idem-gcd-int*: *fun-left-comm-idem* (*gcd* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int*)  
**proof** **qed** (*auto simp add: gcd-ac-int*)

**lemma** *fun-left-comm-idem-lcm-nat*: *fun-left-comm-idem* (*lcm* :: *nat*  $\Rightarrow$  *nat*  $\Rightarrow$  *nat*)  
**proof** **qed** (*auto simp add: lcm-ac-nat*)

**lemma** *fun-left-comm-idem-lcm-int*: *fun-left-comm-idem* (*lcm* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int*)  
**proof** **qed** (*auto simp add: lcm-ac-int*)

**lemma** *lcm-0-iff-nat[simp]*:  $\text{lcm } (m::\text{nat}) \ n = 0 \longleftrightarrow m=0 \vee n=0$   
**by** (*metis lcm-0-left-nat lcm-0-nat mult-is-0 prod-gcd-lcm-nat*)

**lemma** *lcm-0-iff-int[simp]*:  $\text{lcm } (m::\text{int}) \ n = 0 \longleftrightarrow m=0 \vee n=0$   
**by** (*metis lcm-0-int lcm-0-left-int lcm-pos-int zless-le*)

**lemma** *lcm-1-iff-nat[simp]*:  $\text{lcm } (m::\text{nat}) \ n = 1 \longleftrightarrow m=1 \wedge n=1$   
**by** (*metis gcd-1-nat lcm-unique-nat nat-mult-1 prod-gcd-lcm-nat*)

**lemma** *lcm-1-iff-int[simp]*:  $\text{lcm } (m::\text{int}) \ n = 1 \longleftrightarrow (m=1 \vee m=-1) \wedge (n=1 \vee n=-1)$   
**by** (*auto simp add: abs-mult-self trans [OF lcm-unique-int eq-commute, symmetric] zmult-eq-1-iff*)

### 63.6.1 The complete divisibility lattice

**interpretation** *gcd-semilattice-nat*: *semilattice-inf* *op* *dvd* ( $\%m \ n::\text{nat}. m \text{ dvd } n$  &  $\sim n \text{ dvd } m$ ) *gcd*

**proof**

**case** *goal3* **thus** ?*case* **by**(*metis gcd-unique-nat*)  
**qed** *auto*

**interpretation** *lcm-semilattice-nat*: *semilattice-sup* *op* *dvd* ( $\%m \ n::\text{nat}. m \text{ dvd } n$  &  $\sim n \text{ dvd } m$ ) *lcm*

**proof**

**case** *goal3* **thus** ?*case* **by**(*metis lcm-unique-nat*)

**qed** *auto*

**interpretation** *gcd-lcm-lattice-nat*: *lattice op dvd (%m n::nat. m dvd n & ~ n dvd m) gcd lcm ..*

Lifting gcd and lcm to finite (Gcd/Lcm) and infinite sets (GCD/LCM). GCD is defined via LCM to facilitate the proof that we have a complete lattice. Later on we show that GCD and Gcd coincide on finite sets.

**context** *gcd*

**begin**

**definition** *Gcd* :: 'a set  $\Rightarrow$  'a

**where** *Gcd* = *fold gcd 0*

**definition** *Lcm* :: 'a set  $\Rightarrow$  'a

**where** *Lcm* = *fold lcm 1*

**definition** *LCM* :: 'a set  $\Rightarrow$  'a **where**

*LCM* *M* = (*if finite M then Lcm M else 0*)

**definition** *GCD* :: 'a set  $\Rightarrow$  'a **where**

*GCD* *M* = *LCM*(*INT m:M. {d. d dvd m}*)

**end**

**lemma** *Gcd-empty[simp]*: *Gcd* {} = 0

**by**(*simp add:Gcd-def*)

**lemma** *Lcm-empty[simp]*: *Lcm* {} = 1

**by**(*simp add:Lcm-def*)

**lemma** *GCD-empty-nat[simp]*: *GCD* {} = (0::nat)

**by**(*simp add:GCD-def LCM-def*)

**lemma** *LCM-eq-Lcm[simp]*: *finite M*  $\Longrightarrow$  *LCM* *M* = *Lcm* *M*

**by**(*simp add:LCM-def*)

**lemma** *Lcm-insert-nat [simp]*:

**assumes** *finite N*

**shows** *Lcm* (*insert* (*n::nat*) *N*) = *lcm n* (*Lcm* *N*)

**proof** –

**interpret** *fun-left-comm-idem lcm::nat $\Rightarrow$ nat $\Rightarrow$ nat*

**by** (*rule fun-left-comm-idem-lcm-nat*)

**from** *assms* **show** ?*thesis* **by**(*simp add: Lcm-def*)

**qed**

**lemma** *Lcm-insert-int [simp]*:

```

assumes finite N
shows  $Lcm\ (insert\ (n::int)\ N) = lcm\ n\ (Lcm\ N)$ 
proof –
  interpret fun-left-comm-idem lcm::int $\Rightarrow$ int $\Rightarrow$ int
    by (rule fun-left-comm-idem-lcm-int)
  from assms show ?thesis by(simp add: Lcm-def)
qed

```

```

lemma Gcd-insert-nat [simp]:
  assumes finite N
  shows  $Gcd\ (insert\ (n::nat)\ N) = gcd\ n\ (Gcd\ N)$ 
proof –
  interpret fun-left-comm-idem gcd::nat $\Rightarrow$ nat $\Rightarrow$ nat
    by (rule fun-left-comm-idem-gcd-nat)
  from assms show ?thesis by(simp add: Gcd-def)
qed

```

```

lemma Gcd-insert-int [simp]:
  assumes finite N
  shows  $Gcd\ (insert\ (n::int)\ N) = gcd\ n\ (Gcd\ N)$ 
proof –
  interpret fun-left-comm-idem gcd::int $\Rightarrow$ int $\Rightarrow$ int
    by (rule fun-left-comm-idem-gcd-int)
  from assms show ?thesis by(simp add: Gcd-def)
qed

```

```

lemma Lcm0-iff[simp]: finite (M::nat set)  $\Rightarrow M \neq \{\}$   $\Rightarrow Lcm\ M = 0 \longleftrightarrow 0$ 
  : M
by(induct rule:finite-ne-induct) auto

```

```

lemma Lcm-eq-0[simp]: finite (M::nat set)  $\Rightarrow 0 : M \Rightarrow Lcm\ M = 0$ 
by (metis Lcm0-iff empty-iff)

```

```

lemma Gcd-dvd-nat [simp]:
  assumes finite M and (m::nat)  $\in M$ 
  shows  $Gcd\ M\ dvd\ m$ 
proof –
  show ?thesis using gcd-semilattice-nat.fold-inf-le-inf[OF assms, of 0] by (simp
    add: Gcd-def)
qed

```

```

lemma dvd-Gcd-nat[simp]:
  assumes finite M and ALL (m::nat) : M. n dvd m
  shows  $n\ dvd\ Gcd\ M$ 
proof –
  show ?thesis using gcd-semilattice-nat.inf-le-fold-inf[OF assms, of 0] by (simp
    add: Gcd-def)
qed

```

```

lemma dvd-Lcm-nat [simp]:
  assumes finite M and (m::nat) ∈ M
  shows m dvd Lcm M
proof –
  show ?thesis using lcm-semilattice-nat.sup-le-fold-sup[OF assms, of 1] by (simp
add: Lcm-def)
qed

```

```

lemma Lcm-dvd-nat[simp]:
  assumes finite M and ALL (m::nat) : M. m dvd n
  shows Lcm M dvd n
proof –
  show ?thesis using lcm-semilattice-nat.fold-sup-le-sup[OF assms, of 1] by (simp
add: Lcm-def)
qed

```

```

interpretation gcd-lcm-complete-lattice-nat:
  complete-lattice GCD LCM op dvd %m n::nat. m dvd n & ~ n dvd m gcd lcm 1
  0
proof
  case goal1 show ?case by simp
next
  case goal2 show ?case by simp
next
  case goal5 thus ?case by (auto simp: LCM-def)
next
  case goal6 thus ?case
    by(auto simp: LCM-def)(metis finite-nat-set-iff-bounded-le gcd-proj2-if-dvd-nat
gcd-le1-nat)
next
  case goal3 thus ?case by (auto simp: GCD-def LCM-def)(metis finite-INT
finite-divisors-nat)
next
  case goal4 thus ?case by(auto simp: LCM-def GCD-def)
qed

```

Alternative characterizations of Gcd and GCD:

```

lemma Gcd-eq-Max: finite(M::nat set) ⇒ M ≠ {} ⇒ 0 ∉ M ⇒ Gcd M =
Max(⋂ m∈M. {d. d dvd m})
apply(rule antisym)
apply(rule Max-ge)
apply (metis all-not-in-conv finite-divisors-nat finite-INT)
apply simp
apply (rule Max-le-iff[THEN iffD2])
apply (metis all-not-in-conv finite-divisors-nat finite-INT)
apply fastsimp
apply clarsimp
apply (metis Gcd-dvd-nat Max-in dvd-0-left dvd-Gcd-nat dvd-imp-le linorder-antisym-conv3
not-less0)

```

done

```

lemma Gcd-remove0-nat: finite M  $\implies$  Gcd M = Gcd (M - {0::nat})
apply (induct pred:finite)
apply simp
apply (case-tac x=0)
apply simp
apply (subgoal-tac insert x F - {0} = insert x (F - {0}))
apply simp
apply blast
done

```

```

lemma Lcm-in-lcm-closed-set-nat:
  finite M  $\implies$  M  $\neq$  {}  $\implies$  ALL m n :: nat. m:M  $\longrightarrow$  n:M  $\longrightarrow$  lcm m n : M  $\implies$ 
  Lcm M : M
apply (induct rule:finite-linorder-min-induct)
apply simp
apply simp
apply (subgoal-tac ALL m n :: nat. m:A  $\longrightarrow$  n:A  $\longrightarrow$  lcm m n : A)
apply simp
apply (case-tac A={})
apply simp
apply simp
apply (metis lcm-pos-nat lcm-unique-nat linorder-neq-iff nat-dvd-not-less not-less0)
done

```

```

lemma Lcm-eq-Max-nat:
  finite M  $\implies$  M  $\neq$  {}  $\implies$  0  $\notin$  M  $\implies$  ALL m n :: nat. m:M  $\longrightarrow$  n:M  $\longrightarrow$  lcm
  m n : M  $\implies$  Lcm M = Max M
apply (rule antisym)
apply (rule Max-ge, assumption)
apply (erule (2) Lcm-in-lcm-closed-set-nat)
apply clarsimp
apply (metis Lcm0-iff dvd-Lcm-nat dvd-imp-le neq0-conv)
done

```

Finally GCD is Gcd:

```

lemma GCD-eq-Gcd[simp]: assumes finite(M::nat set) shows GCD M = Gcd M
proof –
  have divisors-remove0-nat: ( $\bigcap m \in M. \{d::nat. d \text{ dvd } m\}$ ) = ( $\bigcap m \in M - \{0\}. \{d::nat. d \text{ dvd } m\}$ ) by auto
  show ?thesis
proof cases
  assume M={0} thus ?thesis by simp
next
  assume M $\neq$ {0}
  show ?thesis
proof cases
  assume M={0} thus ?thesis by (simp add:GCD-def LCM-def)

```

```

next
  assume  $M \neq \{0\}$ 
  with  $\langle M \neq \{0\} \rangle$  assms show ?thesis
    apply(subst Gcd-remove0-nat[OF assms])
    apply(simp add:GCD-def)
    apply(subst divisors-remove0-nat)
    apply(simp add:LCM-def)
    apply rule
    apply rule
    apply(subst Gcd-eq-Max)
      apply simp
      apply blast
      apply blast
    apply(rule Lcm-eq-Max-nat)
      apply simp
      apply blast
      apply fastsimp
      apply clarsimp
    apply(fastsimp intro: finite-divisors-nat intro!: finite-INT)
  done
qed
qed
qed

lemma Lcm-set-nat [code-unfold]:
  Lcm (set ns) = foldl lcm (1::nat) ns
proof -
  interpret fun-left-comm-idem lcm::nat $\Rightarrow$ nat $\Rightarrow$ nat by (rule fun-left-comm-idem-lcm-nat)
  show ?thesis by(simp add: Lcm-def fold-set lcm-commute-nat)
qed

lemma Lcm-set-int [code-unfold]:
  Lcm (set is) = foldl lcm (1::int) is
proof -
  interpret fun-left-comm-idem lcm::int $\Rightarrow$ int $\Rightarrow$ int by (rule fun-left-comm-idem-lcm-int)
  show ?thesis by(simp add: Lcm-def fold-set lcm-commute-int)
qed

lemma Gcd-set-nat [code-unfold]:
  Gcd (set ns) = foldl gcd (0::nat) ns
proof -
  interpret fun-left-comm-idem gcd::nat $\Rightarrow$ nat $\Rightarrow$ nat by (rule fun-left-comm-idem-gcd-nat)
  show ?thesis by(simp add: Gcd-def fold-set gcd-commute-nat)
qed

lemma Gcd-set-int [code-unfold]:
  Gcd (set ns) = foldl gcd (0::int) ns
proof -
  interpret fun-left-comm-idem gcd::int $\Rightarrow$ int $\Rightarrow$ int by (rule fun-left-comm-idem-gcd-int)

```

```

  show ?thesis by(simp add: Gcd-def fold-set gcd-commute-int)
qed

```

```

lemma mult-inj-if-coprime-nat:
  inj-on f A  $\implies$  inj-on g B  $\implies$  ALL a:A. ALL b:B. coprime (f a) (g b)
   $\implies$  inj-on (%(a,b). f a * g b::nat) (A  $\times$  B)
apply(auto simp add:inj-on-def)
apply (metis coprime-dvd-mult-iff-nat dvd.neq-le-trans dvd-triv-left)
apply (metis gcd-semilattice-nat.inf-commute coprime-dvd-mult-iff-nat
  dvd.neq-le-trans dvd-triv-right mult-commute)
done

```

Nitpick:

```

lemma gcd-eq-nitpick-gcd [nitpick-def]: gcd x y  $\equiv$  Nitpick.nat-gcd x y
apply (rule eq-reflection)
apply (induct x y rule: nat-gcd.induct)
by (simp add: gcd-nat.simps Nitpick.nat-gcd.simps)

```

```

lemma lcm-eq-nitpick-lcm [nitpick-def]: lcm x y  $\equiv$  Nitpick.nat-lcm x y
by (simp only: lcm-nat-def Nitpick.nat-lcm-def gcd-eq-nitpick-gcd)

```

end

## 64 Archimedean-Field: Archimedean Fields, Floor and Ceiling Functions

```

theory Archimedean-Field
imports Main
begin

```

### 64.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

```

class archimedean-field = linordered-field + number-ring +
  assumes ex-le-of-int:  $\exists z. x \leq \text{of-int } z$ 

```

```

lemma ex-less-of-int:
  fixes x :: 'a::archimedean-field shows  $\exists z. x < \text{of-int } z$ 
proof -
  from ex-le-of-int obtain z where  $x \leq \text{of-int } z$  ..
  then have  $x < \text{of-int } (z + 1)$  by simp
  then show ?thesis ..
qed

```

```

lemma ex-of-int-less:

```

```

fixes  $x :: 'a::\text{archimedean-field}$  shows  $\exists z. \text{of-int } z < x$ 
proof –
  from  $\text{ex-less-of-int}$  obtain  $z$  where  $-x < \text{of-int } z ..$ 
  then have  $\text{of-int } (-z) < x$  by  $\text{simp}$ 
  then show  $?thesis ..$ 
qed

```

```

lemma  $\text{ex-less-of-nat}$ :
  fixes  $x :: 'a::\text{archimedean-field}$  shows  $\exists n. x < \text{of-nat } n$ 
proof –
  obtain  $z$  where  $x < \text{of-int } z$  using  $\text{ex-less-of-int} ..$ 
  also have  $\dots \leq \text{of-int } (\text{int } (\text{nat } z))$  by  $\text{simp}$ 
  also have  $\dots = \text{of-nat } (\text{nat } z)$  by  $(\text{simp only: of-int-of-nat-eq})$ 
  finally show  $?thesis ..$ 
qed

```

```

lemma  $\text{ex-le-of-nat}$ :
  fixes  $x :: 'a::\text{archimedean-field}$  shows  $\exists n. x \leq \text{of-nat } n$ 
proof –
  obtain  $n$  where  $x < \text{of-nat } n$  using  $\text{ex-less-of-nat} ..$ 
  then have  $x \leq \text{of-nat } n$  by  $\text{simp}$ 
  then show  $?thesis ..$ 
qed

```

Archimedean fields have no infinitesimal elements.

```

lemma  $\text{ex-inverse-of-nat-Suc-less}$ :
  fixes  $x :: 'a::\text{archimedean-field}$ 
  assumes  $0 < x$  shows  $\exists n. \text{inverse } (\text{of-nat } (\text{Suc } n)) < x$ 
proof –
  from  $\langle 0 < x \rangle$  have  $0 < \text{inverse } x$ 
    by  $(\text{rule positive-imp-inverse-positive})$ 
  obtain  $n$  where  $\text{inverse } x < \text{of-nat } n$ 
    using  $\text{ex-less-of-nat} ..$ 
  then obtain  $m$  where  $\text{inverse } x < \text{of-nat } (\text{Suc } m)$ 
    using  $\langle 0 < \text{inverse } x \rangle$  by  $(\text{cases } n) (\text{simp-all del: of-nat-Suc})$ 
  then have  $\text{inverse } (\text{of-nat } (\text{Suc } m)) < \text{inverse } (\text{inverse } x)$ 
    using  $\langle 0 < \text{inverse } x \rangle$  by  $(\text{rule less-imp-inverse-less})$ 
  then have  $\text{inverse } (\text{of-nat } (\text{Suc } m)) < x$ 
    using  $\langle 0 < x \rangle$  by  $(\text{simp add: nonzero-inverse-inverse-eq})$ 
  then show  $?thesis ..$ 
qed

```

```

lemma  $\text{ex-inverse-of-nat-less}$ :
  fixes  $x :: 'a::\text{archimedean-field}$ 
  assumes  $0 < x$  shows  $\exists n > 0. \text{inverse } (\text{of-nat } n) < x$ 
  using  $\text{ex-inverse-of-nat-Suc-less}$   $[OF \langle 0 < x \rangle]$  by  $\text{auto}$ 

```

```

lemma  $\text{ex-less-of-nat-mult}$ :
  fixes  $x :: 'a::\text{archimedean-field}$ 

```



```

    assumes  $0 < x$  shows  $\exists n. y < \text{of-nat } n * x$ 
  proof -
    obtain  $n$  where  $y / x < \text{of-nat } n$  using ex-less-of-nat ..
    with  $\langle 0 < x \rangle$  have  $y < \text{of-nat } n * x$  by (simp add: pos-divide-less-eq)
    then show ?thesis ..
  qed

```

## 64.2 Existence and uniqueness of floor function

```

lemma exists-least-lemma:
  assumes  $\neg P\ 0$  and  $\exists n. P\ n$ 
  shows  $\exists n. \neg P\ n \wedge P\ (\text{Suc } n)$ 
  proof -
    from  $\langle \exists n. P\ n \rangle$  have  $P\ (\text{Least } P)$  by (rule LeastI-ex)
    with  $\langle \neg P\ 0 \rangle$  obtain  $n$  where  $\text{Least } P = \text{Suc } n$ 
      by (cases Least P) auto
    then have  $n < \text{Least } P$  by simp
    then have  $\neg P\ n$  by (rule not-less-Least)
    then have  $\neg P\ n \wedge P\ (\text{Suc } n)$ 
      using  $\langle P\ (\text{Least } P) \rangle \langle \text{Least } P = \text{Suc } n \rangle$  by simp
    then show ?thesis ..
  qed

```

```

lemma floor-exists:
  fixes  $x :: 'a::\text{archimedean-field}$ 
  shows  $\exists z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$ 
  proof (cases)
    assume  $0 \leq x$ 
    then have  $\neg x < \text{of-nat } 0$  by simp
    then have  $\exists n. \neg x < \text{of-nat } n \wedge x < \text{of-nat } (\text{Suc } n)$ 
      using ex-less-of-nat by (rule exists-least-lemma)
    then obtain  $n$  where  $\neg x < \text{of-nat } n \wedge x < \text{of-nat } (\text{Suc } n)$  ..
    then have  $\text{of-int } (\text{int } n) \leq x \wedge x < \text{of-int } (\text{int } n + 1)$  by simp
    then show ?thesis ..
  next
    assume  $\neg 0 \leq x$ 
    then have  $\neg -x \leq \text{of-nat } 0$  by simp
    then have  $\exists n. \neg -x \leq \text{of-nat } n \wedge -x \leq \text{of-nat } (\text{Suc } n)$ 
      using ex-le-of-nat by (rule exists-least-lemma)
    then obtain  $n$  where  $\neg -x \leq \text{of-nat } n \wedge -x \leq \text{of-nat } (\text{Suc } n)$  ..
    then have  $\text{of-int } (-\text{int } n - 1) \leq x \wedge x < \text{of-int } (-\text{int } n - 1 + 1)$  by simp
    then show ?thesis ..
  qed

```

```

lemma floor-exists1:
  fixes  $x :: 'a::\text{archimedean-field}$ 
  shows  $\exists! z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$ 
  proof (rule ex-ex1I)
    show  $\exists z. \text{of-int } z \leq x \wedge x < \text{of-int } (z + 1)$ 

```

```

    by (rule floor-exists)
next
  fix y z assume
    of-int y ≤ x ∧ x < of-int (y + 1)
    of-int z ≤ x ∧ x < of-int (z + 1)
  then have
    of-int y ≤ x x < of-int (y + 1)
    of-int z ≤ x x < of-int (z + 1)
  by simp-all
  from le-less-trans [OF ⟨of-int y ≤ x⟩ ⟨x < of-int (z + 1)⟩]
    le-less-trans [OF ⟨of-int z ≤ x⟩ ⟨x < of-int (y + 1)⟩]
  show y = z by (simp del: of-int-add)
qed

```

### 64.3 Floor function

#### definition

```

floor :: 'a::archimedean-field ⇒ int where
[code del]: floor x = (THE z. of-int z ≤ x ∧ x < of-int (z + 1))

```

#### notation (*xsymbols*)

```

floor ([_])

```

#### notation (*HTML output*)

```

floor ([_])

```

```

lemma floor-correct: of-int (floor x) ≤ x ∧ x < of-int (floor x + 1)
  unfolding floor-def using floor-exists1 by (rule theI')

```

```

lemma floor-unique: [of-int z ≤ x; x < of-int z + 1] ⇒ floor x = z
  using floor-correct [of x] floor-exists1 [of x] by auto

```

```

lemma of-int-floor-le: of-int (floor x) ≤ x
  using floor-correct ..

```

```

lemma le-floor-iff: z ≤ floor x ⟷ of-int z ≤ x

```

#### proof

```

  assume z ≤ floor x
  then have (of-int z :: 'a) ≤ of-int (floor x) by simp
  also have of-int (floor x) ≤ x by (rule of-int-floor-le)
  finally show of-int z ≤ x .

```

#### next

```

  assume of-int z ≤ x
  also have x < of-int (floor x + 1) using floor-correct ..
  finally show z ≤ floor x by (simp del: of-int-add)

```

#### qed

```

lemma floor-less-iff: floor x < z ⟷ x < of-int z
  by (simp add: not-le [symmetric] le-floor-iff)

```

**lemma** *less-floor-iff*:  $z < \text{floor } x \iff \text{of-int } z + 1 \leq x$   
**using** *le-floor-iff* [*of*  $z + 1$   $x$ ] **by** *auto*

**lemma** *floor-le-iff*:  $\text{floor } x \leq z \iff x < \text{of-int } z + 1$   
**by** (*simp* *add*: *not-less* [*symmetric*] *less-floor-iff*)

**lemma** *floor-mono*: **assumes**  $x \leq y$  **shows**  $\text{floor } x \leq \text{floor } y$   
**proof** –  
**have**  $\text{of-int } (\text{floor } x) \leq x$  **by** (*rule* *of-int-floor-le*)  
**also note**  $\langle x \leq y \rangle$   
**finally show** *?thesis* **by** (*simp* *add*: *le-floor-iff*)  
**qed**

**lemma** *floor-less-cancel*:  $\text{floor } x < \text{floor } y \implies x < y$   
**by** (*auto* *simp* *add*: *not-le* [*symmetric*] *floor-mono*)

**lemma** *floor-of-int* [*simp*]:  $\text{floor } (\text{of-int } z) = z$   
**by** (*rule* *floor-unique*) *simp-all*

**lemma** *floor-of-nat* [*simp*]:  $\text{floor } (\text{of-nat } n) = \text{int } n$   
**using** *floor-of-int* [*of*  $\text{of-nat } n$ ] **by** *simp*

Floor with numerals

**lemma** *floor-zero* [*simp*]:  $\text{floor } 0 = 0$   
**using** *floor-of-int* [*of*  $0$ ] **by** *simp*

**lemma** *floor-one* [*simp*]:  $\text{floor } 1 = 1$   
**using** *floor-of-int* [*of*  $1$ ] **by** *simp*

**lemma** *floor-number-of* [*simp*]:  $\text{floor } (\text{number-of } v) = \text{number-of } v$   
**using** *floor-of-int* [*of*  $\text{number-of } v$ ] **by** *simp*

**lemma** *zero-le-floor* [*simp*]:  $0 \leq \text{floor } x \iff 0 \leq x$   
**by** (*simp* *add*: *le-floor-iff*)

**lemma** *one-le-floor* [*simp*]:  $1 \leq \text{floor } x \iff 1 \leq x$   
**by** (*simp* *add*: *le-floor-iff*)

**lemma** *number-of-le-floor* [*simp*]:  $\text{number-of } v \leq \text{floor } x \iff \text{number-of } v \leq x$   
**by** (*simp* *add*: *le-floor-iff*)

**lemma** *zero-less-floor* [*simp*]:  $0 < \text{floor } x \iff 1 \leq x$   
**by** (*simp* *add*: *less-floor-iff*)

**lemma** *one-less-floor* [*simp*]:  $1 < \text{floor } x \iff 2 \leq x$   
**by** (*simp* *add*: *less-floor-iff*)

**lemma** *number-of-less-floor* [*simp*]:

$number-of\ v < floor\ x \longleftrightarrow number-of\ v + 1 \leq x$   
**by** (*simp add: less-floor-iff*)

**lemma** *floor-le-zero* [*simp*]:  $floor\ x \leq 0 \longleftrightarrow x < 1$   
**by** (*simp add: floor-le-iff*)

**lemma** *floor-le-one* [*simp*]:  $floor\ x \leq 1 \longleftrightarrow x < 2$   
**by** (*simp add: floor-le-iff*)

**lemma** *floor-le-number-of* [*simp*]:  
 $floor\ x \leq number-of\ v \longleftrightarrow x < number-of\ v + 1$   
**by** (*simp add: floor-le-iff*)

**lemma** *floor-less-zero* [*simp*]:  $floor\ x < 0 \longleftrightarrow x < 0$   
**by** (*simp add: floor-less-iff*)

**lemma** *floor-less-one* [*simp*]:  $floor\ x < 1 \longleftrightarrow x < 1$   
**by** (*simp add: floor-less-iff*)

**lemma** *floor-less-number-of* [*simp*]:  
 $floor\ x < number-of\ v \longleftrightarrow x < number-of\ v$   
**by** (*simp add: floor-less-iff*)

Addition and subtraction of integers

**lemma** *floor-add-of-int* [*simp*]:  $floor\ (x + of-int\ z) = floor\ x + z$   
**using** *floor-correct* [*of x*] **by** (*simp add: floor-unique*)

**lemma** *floor-add-number-of* [*simp*]:  
 $floor\ (x + number-of\ v) = floor\ x + number-of\ v$   
**using** *floor-add-of-int* [*of x number-of v*] **by** *simp*

**lemma** *floor-add-one* [*simp*]:  $floor\ (x + 1) = floor\ x + 1$   
**using** *floor-add-of-int* [*of x 1*] **by** *simp*

**lemma** *floor-diff-of-int* [*simp*]:  $floor\ (x - of-int\ z) = floor\ x - z$   
**using** *floor-add-of-int* [*of x - z*] **by** (*simp add: algebra-simps*)

**lemma** *floor-diff-number-of* [*simp*]:  
 $floor\ (x - number-of\ v) = floor\ x - number-of\ v$   
**using** *floor-diff-of-int* [*of x number-of v*] **by** *simp*

**lemma** *floor-diff-one* [*simp*]:  $floor\ (x - 1) = floor\ x - 1$   
**using** *floor-diff-of-int* [*of x 1*] **by** *simp*

## 64.4 Ceiling function

**definition**

*ceiling* :: 'a::archimedean-field  $\Rightarrow$  int **where**  
~~[code del]: ceiling~~  $x = - floor\ (-x)$

**notation** (*xsymbols*)

*ceiling* ( $\lceil \cdot \rceil$ )

**notation** (*HTML output*)

*ceiling* ( $\lceil \cdot \rceil$ )

**lemma** *ceiling-correct*:  $\text{of-int } (\text{ceiling } x) - 1 < x \wedge x \leq \text{of-int } (\text{ceiling } x)$   
**unfolding** *ceiling-def* **using** *floor-correct* [*of*  $- x$ ] **by** *simp*

**lemma** *ceiling-unique*:  $\llbracket \text{of-int } z - 1 < x; x \leq \text{of-int } z \rrbracket \implies \text{ceiling } x = z$   
**unfolding** *ceiling-def* **using** *floor-unique* [*of*  $- z - x$ ] **by** *simp*

**lemma** *le-of-int-ceiling*:  $x \leq \text{of-int } (\text{ceiling } x)$   
**using** *ceiling-correct* ..

**lemma** *ceiling-le-iff*:  $\text{ceiling } x \leq z \longleftrightarrow x \leq \text{of-int } z$   
**unfolding** *ceiling-def* **using** *le-floor-iff* [*of*  $- z - x$ ] **by** *auto*

**lemma** *less-ceiling-iff*:  $z < \text{ceiling } x \longleftrightarrow \text{of-int } z < x$   
**by** (*simp* *add: not-le [symmetric] ceiling-le-iff*)

**lemma** *ceiling-less-iff*:  $\text{ceiling } x < z \longleftrightarrow x \leq \text{of-int } z - 1$   
**using** *ceiling-le-iff* [*of*  $x \ z - 1$ ] **by** *simp*

**lemma** *le-ceiling-iff*:  $z \leq \text{ceiling } x \longleftrightarrow \text{of-int } z - 1 < x$   
**by** (*simp* *add: not-less [symmetric] ceiling-less-iff*)

**lemma** *ceiling-mono*:  $x \geq y \implies \text{ceiling } x \geq \text{ceiling } y$   
**unfolding** *ceiling-def* **by** (*simp* *add: floor-mono*)

**lemma** *ceiling-less-cancel*:  $\text{ceiling } x < \text{ceiling } y \implies x < y$   
**by** (*auto* *simp* *add: not-le [symmetric] ceiling-mono*)

**lemma** *ceiling-of-int [simp]*:  $\text{ceiling } (\text{of-int } z) = z$   
**by** (*rule* *ceiling-unique*) *simp-all*

**lemma** *ceiling-of-nat [simp]*:  $\text{ceiling } (\text{of-nat } n) = \text{int } n$   
**using** *ceiling-of-int* [*of*  $\text{of-nat } n$ ] **by** *simp*

Ceiling with numerals

**lemma** *ceiling-zero [simp]*:  $\text{ceiling } 0 = 0$   
**using** *ceiling-of-int* [*of*  $0$ ] **by** *simp*

**lemma** *ceiling-one [simp]*:  $\text{ceiling } 1 = 1$   
**using** *ceiling-of-int* [*of*  $1$ ] **by** *simp*

**lemma** *ceiling-number-of [simp]*:  $\text{ceiling } (\text{number-of } v) = \text{number-of } v$   
**using** *ceiling-of-int* [*of*  $\text{number-of } v$ ] **by** *simp*

**lemma** *ceiling-le-zero* [simp]:  $\text{ceiling } x \leq 0 \longleftrightarrow x \leq 0$   
**by** (simp add: ceiling-le-iff)

**lemma** *ceiling-le-one* [simp]:  $\text{ceiling } x \leq 1 \longleftrightarrow x \leq 1$   
**by** (simp add: ceiling-le-iff)

**lemma** *ceiling-le-number-of* [simp]:  
 $\text{ceiling } x \leq \text{number-of } v \longleftrightarrow x \leq \text{number-of } v$   
**by** (simp add: ceiling-le-iff)

**lemma** *ceiling-less-zero* [simp]:  $\text{ceiling } x < 0 \longleftrightarrow x \leq -1$   
**by** (simp add: ceiling-less-iff)

**lemma** *ceiling-less-one* [simp]:  $\text{ceiling } x < 1 \longleftrightarrow x \leq 0$   
**by** (simp add: ceiling-less-iff)

**lemma** *ceiling-less-number-of* [simp]:  
 $\text{ceiling } x < \text{number-of } v \longleftrightarrow x \leq \text{number-of } v - 1$   
**by** (simp add: ceiling-less-iff)

**lemma** *zero-le-ceiling* [simp]:  $0 \leq \text{ceiling } x \longleftrightarrow -1 < x$   
**by** (simp add: le-ceiling-iff)

**lemma** *one-le-ceiling* [simp]:  $1 \leq \text{ceiling } x \longleftrightarrow 0 < x$   
**by** (simp add: le-ceiling-iff)

**lemma** *number-of-le-ceiling* [simp]:  
 $\text{number-of } v \leq \text{ceiling } x \longleftrightarrow \text{number-of } v - 1 < x$   
**by** (simp add: le-ceiling-iff)

**lemma** *zero-less-ceiling* [simp]:  $0 < \text{ceiling } x \longleftrightarrow 0 < x$   
**by** (simp add: less-ceiling-iff)

**lemma** *one-less-ceiling* [simp]:  $1 < \text{ceiling } x \longleftrightarrow 1 < x$   
**by** (simp add: less-ceiling-iff)

**lemma** *number-of-less-ceiling* [simp]:  
 $\text{number-of } v < \text{ceiling } x \longleftrightarrow \text{number-of } v < x$   
**by** (simp add: less-ceiling-iff)

Addition and subtraction of integers

**lemma** *ceiling-add-of-int* [simp]:  $\text{ceiling } (x + \text{of-int } z) = \text{ceiling } x + z$   
**using** ceiling-correct [of  $x$ ] **by** (simp add: ceiling-unique)

**lemma** *ceiling-add-number-of* [simp]:  
 $\text{ceiling } (x + \text{number-of } v) = \text{ceiling } x + \text{number-of } v$   
**using** ceiling-add-of-int [of  $x$  number-of  $v$ ] **by** simp

**lemma** *ceiling-add-one* [simp]:  $\text{ceiling } (x + 1) = \text{ceiling } x + 1$   
**using** *ceiling-add-of-int* [of  $x$  1] **by** *simp*

**lemma** *ceiling-diff-of-int* [simp]:  $\text{ceiling } (x - \text{of-int } z) = \text{ceiling } x - z$   
**using** *ceiling-add-of-int* [of  $x - z$ ] **by** (*simp add: algebra-simps*)

**lemma** *ceiling-diff-number-of* [simp]:  
 $\text{ceiling } (x - \text{number-of } v) = \text{ceiling } x - \text{number-of } v$   
**using** *ceiling-diff-of-int* [of  $x$  *number-of*  $v$ ] **by** *simp*

**lemma** *ceiling-diff-one* [simp]:  $\text{ceiling } (x - 1) = \text{ceiling } x - 1$   
**using** *ceiling-diff-of-int* [of  $x$  1] **by** *simp*

## 64.5 Negation

**lemma** *floor-minus*:  $\text{floor } (-x) = -\text{ceiling } x$   
**unfolding** *ceiling-def* **by** *simp*

**lemma** *ceiling-minus*:  $\text{ceiling } (-x) = -\text{floor } x$   
**unfolding** *ceiling-def* **by** *simp*

**end**

## 65 Rat: Rational numbers

**theory** *Rat*  
**imports** *GCD Archimedean-Field*  
**uses** (*Tools/float-syntax.ML*)  
**begin**

### 65.1 Rational numbers as quotient

#### 65.1.1 Construction of the type of rational numbers

**definition**  
 $\text{ratrel} :: ((\text{int} \times \text{int}) \times (\text{int} \times \text{int})) \text{ set}$  **where**  
 $\text{ratrel} = \{(x, y). \text{snd } x \neq 0 \wedge \text{snd } y \neq 0 \wedge \text{fst } x * \text{snd } y = \text{fst } y * \text{snd } x\}$

**lemma** *ratrel-iff* [simp]:  
 $(x, y) \in \text{ratrel} \longleftrightarrow \text{snd } x \neq 0 \wedge \text{snd } y \neq 0 \wedge \text{fst } x * \text{snd } y = \text{fst } y * \text{snd } x$   
**by** (*simp add: ratrel-def*)

**lemma** *refl-on-ratrel*:  $\text{refl-on } \{x. \text{snd } x \neq 0\} \text{ ratrel}$   
**by** (*auto simp add: refl-on-def ratrel-def*)

**lemma** *sym-ratrel*:  $\text{sym ratrel}$   
**by** (*simp add: ratrel-def sym-def*)

**lemma** *trans-ratrel*: *trans ratrel*  
**proof** (*rule transI, unfold split-paired-all*)  
 fix  $a\ b\ a'\ b'\ a''\ b'' :: \text{int}$   
 assume  $A: ((a, b), (a', b')) \in \text{ratrel}$   
 assume  $B: ((a', b'), (a'', b'')) \in \text{ratrel}$   
 have  $b' * (a * b'') = b'' * (a * b')$  **by** *simp*  
 also from  $A$  have  $a * b' = a' * b$  **by** *auto*  
 also have  $b'' * (a' * b) = b * (a' * b'')$  **by** *simp*  
 also from  $B$  have  $a' * b'' = a'' * b'$  **by** *auto*  
 also have  $b * (a'' * b') = b' * (a'' * b)$  **by** *simp*  
 finally have  $b' * (a * b'') = b' * (a'' * b)$  .  
 moreover from  $B$  have  $b' \neq 0$  **by** *auto*  
 ultimately have  $a * b'' = a'' * b$  **by** *simp*  
 with  $A\ B$  show  $((a, b), (a'', b'')) \in \text{ratrel}$  **by** *auto*  
**qed**

**lemma** *equiv-ratrel*: *equiv  $\{x. \text{snd } x \neq 0\}$  ratrel*  
**by** (*rule equiv.intro [OF refl-on-ratrel sym-ratrel trans-ratrel]*)

**lemmas** *UN-ratrel* = *UN-equiv-class [OF equiv-ratrel]*  
**lemmas** *UN-ratrel2* = *UN-equiv-class2 [OF equiv-ratrel equiv-ratrel]*

**lemma** *equiv-ratrel-iff [iff]*:  
 assumes  $\text{snd } x \neq 0$  **and**  $\text{snd } y \neq 0$   
 shows *ratrel “ $\{x\} = \text{ratrel “ $\{y\} \longleftrightarrow (x, y) \in \text{ratrel}$ ”$*   
**by** (*rule eq-equiv-class-iff, rule equiv-ratrel*) (*auto simp add: assms*)

**typedef** (*Rat*) *rat* =  $\{x. \text{snd } x \neq 0\}$  // *ratrel*  
**proof**  
 have  $(0::\text{int}, 1::\text{int}) \in \{x. \text{snd } x \neq 0\}$  **by** *simp*  
 then show *ratrel “ $\{(0, 1)\} \in \{x. \text{snd } x \neq 0\}$  // ratrel* **by** (*rule quotientI*)  
**qed**

**lemma** *ratrel-in-Rat [simp]*:  $\text{snd } x \neq 0 \implies \text{ratrel “ $\{x\} \in \text{Rat}$ ”$   
**by** (*simp add: Rat-def quotientI*)

**declare** *Abs-Rat-inject [simp]* *Abs-Rat-inverse [simp]*

### 65.1.2 Representation and basic operations

**definition**

*Fract* ::  $\text{int} \Rightarrow \text{int} \Rightarrow \text{rat}$  **where**  
*Fract*  $a\ b = \text{Abs-Rat } (\text{ratrel “ $\{\text{if } b = 0 \text{ then } (0, 1) \text{ else } (a, b)\}$ ”})$

**lemma** *eq-rat*:  
 shows  $\bigwedge a\ b\ c\ d. b \neq 0 \implies d \neq 0 \implies \text{Fract } a\ b = \text{Fract } c\ d \longleftrightarrow a * d = c * b$   
 and  $\bigwedge a. \text{Fract } a\ 0 = \text{Fract } 0\ 1$   
 and  $\bigwedge a\ c. \text{Fract } 0\ a = \text{Fract } 0\ c$   
**by** (*simp-all add: Fract-def*)



```

lemma Rat-cases [case-names Fract, cases type: rat]:
  assumes  $\bigwedge a\ b. q = \text{Fract } a\ b \implies b > 0 \implies \text{coprime } a\ b \implies C$ 
  shows  $C$ 
proof –
  obtain  $a\ b :: \text{int}$  where  $q = \text{Fract } a\ b$  and  $b \neq 0$ 
    by (cases q) (clarsimp simp add: Fract-def Rat-def quotient-def)
  let  $?a = a \text{ div } \text{gcd } a\ b$ 
  let  $?b = b \text{ div } \text{gcd } a\ b$ 
  from  $\langle b \neq 0 \rangle$  have  $?b * \text{gcd } a\ b = b$ 
    by (simp add: dvd-div-mult-self)
  with  $\langle b \neq 0 \rangle$  have  $?b \neq 0$  by auto
  from  $\langle q = \text{Fract } a\ b \rangle \langle b \neq 0 \rangle \langle ?b \neq 0 \rangle$  have  $q = \text{Fract } ?a\ ?b$ 
    by (simp add: eq-rat dvd-div-mult mult-commute [of a])
  from  $\langle b \neq 0 \rangle$  have coprime: coprime  $?a\ ?b$ 
    by (auto intro: div-gcd-coprime-int)
  show  $C$  proof (cases b > 0)
    case True
    note assms
    moreover note q
    moreover from True have  $?b > 0$  by (simp add: nonneg1-imp-zdiv-pos-iff)
    moreover note coprime
    ultimately show  $C$  .
  next
    case False
    note assms
    moreover from q have  $q = \text{Fract } (-\ ?a)\ (-\ ?b)$  by (simp add: Fract-def)
    moreover from False  $\langle b \neq 0 \rangle$  have  $-\ ?b > 0$  by (simp add: pos-imp-zdiv-neg-iff)
    moreover from coprime have coprime  $(-\ ?a)\ (-\ ?b)$  by simp
    ultimately show  $C$  .
  qed
qed

lemma Rat-induct [case-names Fract, induct type: rat]:
  assumes  $\bigwedge a\ b. b > 0 \implies \text{coprime } a\ b \implies P\ (\text{Fract } a\ b)$ 
  shows  $P\ q$ 
  using assms by (cases q) simp

instantiation rat :: comm-ring-1
begin

definition
  Zero-rat-def:  $0 = \text{Fract } 0\ 1$ 

definition
  One-rat-def:  $1 = \text{Fract } 1\ 1$ 

definition
  add-rat-def:

```

$q + r = \text{Abs-Rat } (\bigcup x \in \text{Rep-Rat } q. \bigcup y \in \text{Rep-Rat } r.$   
 $\text{ratrel } “ \{(\text{fst } x * \text{snd } y + \text{fst } y * \text{snd } x, \text{snd } x * \text{snd } y)\}$

**lemma** *add-rat* [*simp*]:  
 assumes  $b \neq 0$  and  $d \neq 0$   
 shows  $\text{Fract } a \ b + \text{Fract } c \ d = \text{Fract } (a * d + c * b) \ (b * d)$   
**proof** –  
 have  $(\lambda x \ y. \text{ratrel } “ \{(\text{fst } x * \text{snd } y + \text{fst } y * \text{snd } x, \text{snd } x * \text{snd } y)\})$   
 $\text{respects2 } \text{ratrel}$   
 by (rule *equiv-ratrel* [THEN *congruent2-commuteI*]) (simp-all add: *left-distrib*)  
 with *assms* show ?thesis by (simp add: *Fract-def add-rat-def UN-ratrel2*)  
**qed**

**definition**  
*minus-rat-def*:  
 $- \ q = \text{Abs-Rat } (\bigcup x \in \text{Rep-Rat } q. \text{ratrel } “ \{(- \ \text{fst } x, \text{snd } x)\})$

**lemma** *minus-rat* [*simp*]:  $- \ \text{Fract } a \ b = \text{Fract } (- \ a) \ b$   
**proof** –  
 have  $(\lambda x. \text{ratrel } “ \{(- \ \text{fst } x, \text{snd } x)\}) \text{ respects } \text{ratrel}$   
 by (simp add: *congruent-def*)  
 then show ?thesis by (simp add: *Fract-def minus-rat-def UN-ratrel*)  
**qed**

**lemma** *minus-rat-cancel* [*simp*]:  $\text{Fract } (- \ a) \ (- \ b) = \text{Fract } a \ b$   
 by (cases  $b = 0$ ) (simp-all add: *eq-rat*)

**definition**  
*diff-rat-def*:  $q - r = q + - \ (r::\text{rat})$

**lemma** *diff-rat* [*simp*]:  
 assumes  $b \neq 0$  and  $d \neq 0$   
 shows  $\text{Fract } a \ b - \text{Fract } c \ d = \text{Fract } (a * d - c * b) \ (b * d)$   
 using *assms* by (simp add: *diff-rat-def*)

**definition**  
*mult-rat-def*:  
 $q * r = \text{Abs-Rat } (\bigcup x \in \text{Rep-Rat } q. \bigcup y \in \text{Rep-Rat } r.$   
 $\text{ratrel } “ \{(\text{fst } x * \text{fst } y, \text{snd } x * \text{snd } y)\}$

**lemma** *mult-rat* [*simp*]:  $\text{Fract } a \ b * \text{Fract } c \ d = \text{Fract } (a * c) \ (b * d)$   
**proof** –  
 have  $(\lambda x \ y. \text{ratrel } “ \{(\text{fst } x * \text{fst } y, \text{snd } x * \text{snd } y)\}) \text{ respects2 } \text{ratrel}$   
 by (rule *equiv-ratrel* [THEN *congruent2-commuteI*]) (simp-all)  
 then show ?thesis by (simp add: *Fract-def mult-rat-def UN-ratrel2*)  
**qed**

**lemma** *mult-rat-cancel*:  
 assumes  $c \neq 0$

shows  $\text{Fract } (c * a) (c * b) = \text{Fract } a b$   
**proof** –  
 from *assms* have  $\text{Fract } c c = \text{Fract } 1 1$  **by** (*simp add: Fract-def*)  
 then show *?thesis* **by** (*simp add: mult-rat [symmetric]*)  
**qed**

**instance proof**  
 fix  $q r s :: \text{rat}$  **show**  $(q * r) * s = q * (r * s)$   
 by (*cases q, cases r, cases s*) (*simp add: eq-rat*)  
**next**  
 fix  $q r :: \text{rat}$  **show**  $q * r = r * q$   
 by (*cases q, cases r*) (*simp add: eq-rat*)  
**next**  
 fix  $q :: \text{rat}$  **show**  $1 * q = q$   
 by (*cases q*) (*simp add: One-rat-def eq-rat*)  
**next**  
 fix  $q r s :: \text{rat}$  **show**  $(q + r) + s = q + (r + s)$   
 by (*cases q, cases r, cases s*) (*simp add: eq-rat algebra-simps*)  
**next**  
 fix  $q r :: \text{rat}$  **show**  $q + r = r + q$   
 by (*cases q, cases r*) (*simp add: eq-rat*)  
**next**  
 fix  $q :: \text{rat}$  **show**  $0 + q = q$   
 by (*cases q*) (*simp add: Zero-rat-def eq-rat*)  
**next**  
 fix  $q :: \text{rat}$  **show**  $- q + q = 0$   
 by (*cases q*) (*simp add: Zero-rat-def eq-rat*)  
**next**  
 fix  $q r :: \text{rat}$  **show**  $q - r = q + - r$   
 by (*cases q, cases r*) (*simp add: eq-rat*)  
**next**  
 fix  $q r s :: \text{rat}$  **show**  $(q + r) * s = q * s + r * s$   
 by (*cases q, cases r, cases s*) (*simp add: eq-rat algebra-simps*)  
**next**  
 show  $(0::\text{rat}) \neq 1$  **by** (*simp add: Zero-rat-def One-rat-def eq-rat*)  
**qed**  
**end**

**lemma** *of-nat-rat*:  $\text{of-nat } k = \text{Fract } (\text{of-nat } k) 1$   
 by (*induct k*) (*simp-all add: Zero-rat-def One-rat-def*)

**lemma** *of-int-rat*:  $\text{of-int } k = \text{Fract } k 1$   
 by (*cases k rule: int-diff-cases*) (*simp add: of-nat-rat*)

**lemma** *Fract-of-nat-eq*:  $\text{Fract } (\text{of-nat } k) 1 = \text{of-nat } k$   
 by (*rule of-nat-rat [symmetric]*)

**lemma** *Fract-of-int-eq*:  $\text{Fract } k 1 = \text{of-int } k$

```

    by (rule of-int-rat [symmetric])

instantiation rat :: number-ring
begin

definition
  rat-number-of-def: number-of  $w = \text{Fract } w \ 1$ 

instance proof
qed (simp add: rat-number-of-def of-int-rat)

end

lemma rat-number-collapse:
  Fract 0  $k = 0$ 
  Fract 1 1 = 1
  Fract (number-of  $k$ ) 1 = number-of  $k$ 
  Fract  $k$  0 = 0
  by (cases  $k = 0$ )
    (simp-all add: Zero-rat-def One-rat-def number-of-is-id number-of-eq of-int-rat
eq-rat Fract-def)

lemma rat-number-expand [code-unfold]:
  0 = Fract 0 1
  1 = Fract 1 1
  number-of  $k = \text{Fract } (\text{number-of } k) \ 1$ 
  by (simp-all add: rat-number-collapse)

lemma iszero-rat [simp]:
  iszero (number-of  $k :: \text{rat}$ )  $\longleftrightarrow$  iszero (number-of  $k :: \text{int}$ )
  by (simp add: iszero-def rat-number-expand number-of-is-id eq-rat)

lemma Rat-cases-nonzero [case-names Fract 0]:
  assumes Fract:  $\bigwedge a \ b. q = \text{Fract } a \ b \implies b > 0 \implies a \neq 0 \implies \text{coprime } a \ b \implies$ 
  C
  assumes 0:  $q = 0 \implies C$ 
  shows C
proof (cases  $q = 0$ )
  case True then show C using 0 by auto
next
  case False
  then obtain  $a \ b$  where  $q = \text{Fract } a \ b$  and  $b > 0$  and coprime  $a \ b$  by (cases
 $q$ ) auto
  moreover with False have  $0 \neq \text{Fract } a \ b$  by simp
  with  $\langle b > 0 \rangle$  have  $a \neq 0$  by (simp add: Zero-rat-def eq-rat)
  with Fract  $\langle q = \text{Fract } a \ b \rangle \langle b > 0 \rangle \langle \text{coprime } a \ b \rangle$  show C by blast
qed

```

**65.1.3 Function *normalize***

**lemma** *Fract-coprime*:  $\text{Fract } (a \text{ div } \text{gcd } a \ b) \ (b \text{ div } \text{gcd } a \ b) = \text{Fract } a \ b$

**proof** (*cases*  $b = 0$ )

case *True* then show ?thesis by (simp add: eq-rat)

next

case *False*

moreover have  $b \text{ div } \text{gcd } a \ b * \text{gcd } a \ b = b$

by (rule dvd-div-mult-self) simp

ultimately have  $b \text{ div } \text{gcd } a \ b \neq 0$  by auto

with *False* show ?thesis by (simp add: eq-rat dvd-div-mult mult-commute [of  $a$ ])

qed

**definition** *normalize* ::  $\text{int} \times \text{int} \Rightarrow \text{int} \times \text{int}$  where

*normalize*  $p = (\text{if } \text{snd } p > 0 \text{ then } (\text{let } a = \text{gcd } (\text{fst } p) \ (\text{snd } p) \text{ in } (\text{fst } p \text{ div } a, \text{snd } p \text{ div } a))$

*else if*  $\text{snd } p = 0 \text{ then } (0, 1)$

*else*  $(\text{let } a = -\text{gcd } (\text{fst } p) \ (\text{snd } p) \text{ in } (\text{fst } p \text{ div } a, \text{snd } p \text{ div } a)))$

**lemma** *normalize-crossproduct*:

assumes  $q \neq 0 \ s \neq 0$

assumes *normalize*  $(p, q) = \text{normalize } (r, s)$

shows  $p * s = r * q$

**proof** –

have *aux*:  $p * \text{gcd } r \ s = \text{sgn } (q * s) * r * \text{gcd } p \ q \Longrightarrow q * \text{gcd } r \ s = \text{sgn } (q * s) * s * \text{gcd } p \ q \Longrightarrow p * s = q * r$

**proof** –

assume  $p * \text{gcd } r \ s = \text{sgn } (q * s) * r * \text{gcd } p \ q$  and  $q * \text{gcd } r \ s = \text{sgn } (q * s) * s * \text{gcd } p \ q$

then have  $(p * \text{gcd } r \ s) * (\text{sgn } (q * s) * s * \text{gcd } p \ q) = (q * \text{gcd } r \ s) * (\text{sgn } (q * s) * r * \text{gcd } p \ q)$  by simp

with *assms* show  $p * s = q * r$  by (auto simp add: mult-ac sgn-times sgn-0-0)

qed

from *assms* show ?thesis

by (auto simp add: normalize-def Let-def dvd-div-div-eq-mult mult-commute sgn-times split: if-splits intro: *aux*)

qed

**lemma** *normalize-eq*:  $\text{normalize } (a, b) = (p, q) \Longrightarrow \text{Fract } p \ q = \text{Fract } a \ b$

by (auto simp add: normalize-def Let-def Fract-coprime dvd-div-neg rat-number-collapse split:split-if-asm)

**lemma** *normalize-denom-pos*:  $\text{normalize } r = (p, q) \Longrightarrow q > 0$

by (auto simp add: normalize-def Let-def dvd-div-neg pos-imp-zdiv-neg-iff nonneg1-imp-zdiv-pos-iff split:split-if-asm)

**lemma** *normalize-coprime*:  $\text{normalize } r = (p, q) \Longrightarrow \text{coprime } p \ q$

by (auto simp add: normalize-def Let-def dvd-div-neg div-gcd-coprime-int split:split-if-asm)

**lemma** *normalize-stable* [simp]:

$q > 0 \implies \text{coprime } p \ q \implies \text{normalize } (p, q) = (p, q)$   
**by** (simp add: normalize-def)

**lemma** *normalize-denom-zero* [simp]:

$\text{normalize } (p, 0) = (0, 1)$   
**by** (simp add: normalize-def)

**lemma** *normalize-negative* [simp]:

$q < 0 \implies \text{normalize } (p, q) = \text{normalize } (-p, -q)$   
**by** (simp add: normalize-def Let-def dvd-div-neg dvd-neg-div)

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

**definition** *quotient-of* ::  $\text{rat} \Rightarrow \text{int} \times \text{int}$  **where**

$\text{quotient-of } x = (\text{THE pair. } x = \text{Fract } (\text{fst pair}) (\text{snd pair}) \ \& \ \text{snd pair} > 0 \ \& \ \text{coprime } (\text{fst pair}) (\text{snd pair}))$

**lemma** *quotient-of-unique*:

$\exists! p. r = \text{Fract } (\text{fst } p) (\text{snd } p) \wedge \text{snd } p > 0 \wedge \text{coprime } (\text{fst } p) (\text{snd } p)$

**proof** (cases r)

**case** (Fract a b)

**then have**  $r = \text{Fract } (\text{fst } (a, b)) (\text{snd } (a, b)) \wedge \text{snd } (a, b) > 0 \wedge \text{coprime } (\text{fst } (a, b)) (\text{snd } (a, b))$  **by** auto

**then show** ?thesis **proof** (rule ex1I)

**fix** p

**obtain** c d :: int **where** p:  $p = (c, d)$  **by** (cases p)

**assume**  $r = \text{Fract } (\text{fst } p) (\text{snd } p) \wedge \text{snd } p > 0 \wedge \text{coprime } (\text{fst } p) (\text{snd } p)$

**with** p **have**  $\text{Fract}' : r = \text{Fract } c \ d \ d > 0 \ \text{coprime } c \ d$  **by** simp-all

**have**  $c = a \wedge d = b$

**proof** (cases a = 0)

**case** True **with** Fract Fract' **show** ?thesis **by** (simp add: eq-rat)

**next**

**case** False

**with** Fract Fract' **have** \*:  $c * b = a * d$  **and**  $c \neq 0$  **by** (auto simp add: eq-rat)

**then have**  $c * b > 0 \iff a * d > 0$  **by** auto

**with**  $\langle b > 0 \rangle \langle d > 0 \rangle$  **have**  $a > 0 \iff c > 0$  **by** (simp add: zero-less-mult-iff)

**with**  $\langle a \neq 0 \rangle \langle c \neq 0 \rangle$  **have** sgn:  $\text{sgn } a = \text{sgn } c$  **by** (auto simp add: not-less)

**from**  $\langle \text{coprime } a \ b \rangle \langle \text{coprime } c \ d \rangle$  **have**  $|a| * |d| = |c| * |b| \iff |a| = |c| \wedge |d| = |b|$

**by** (simp add: coprime-crossproduct-int)

**with**  $\langle b > 0 \rangle \langle d > 0 \rangle$  **have**  $|a| * d = |c| * b \iff |a| = |c| \wedge d = b$  **by** simp

**then have**  $a * \text{sgn } a * d = c * \text{sgn } c * b \iff a * \text{sgn } a = c * \text{sgn } c \wedge d = b$

**by** (simp add: abs-sgn)

**with** sgn \* **show** ?thesis **by** (auto simp add: sgn-0-0)

**qed**

**with** p **show**  $p = (a, b)$  **by** simp

qed  
qed

**lemma** *quotient-of-Fract* [code]:

*quotient-of* (Fract *a b*) = *normalize* (*a, b*)

**proof** –

**have** *Fract a b* = *Fract* (*fst* (*normalize* (*a, b*))) (*snd* (*normalize* (*a, b*))) (**is** *?Fract*)

**by** (*rule sym*) (*auto intro: normalize-eq*)

**moreover have**  $0 < \text{snd } (\text{normalize } (a, b))$  (**is** *?denom-pos*)

**by** (*cases normalize* (*a, b*)) (*rule normalize-denom-pos, simp*)

**moreover have** *coprime* (*fst* (*normalize* (*a, b*))) (*snd* (*normalize* (*a, b*))) (**is** *?coprime*)

**by** (*rule normalize-coprime*) *simp*

**ultimately have** *?Fract*  $\wedge$  *?denom-pos*  $\wedge$  *?coprime* **by** *blast*

**with** *quotient-of-unique* **have**

(*THE p. Fract a b* = *Fract* (*fst p*) (*snd p*)  $\wedge$   $0 < \text{snd } p \wedge \text{coprime } (\text{fst } p) (\text{snd } p)$ ) = *normalize* (*a, b*)

**by** (*rule the1-equality*)

**then show** *?thesis* **by** (*simp add: quotient-of-def*)

qed

**lemma** *quotient-of-number* [*simp*]:

*quotient-of*  $0 = (0, 1)$

*quotient-of*  $1 = (1, 1)$

*quotient-of* (*number-of k*) = (*number-of k*,  $1$ )

**by** (*simp-all add: rat-number-expand quotient-of-Fract*)

**lemma** *quotient-of-eq*: *quotient-of* (Fract *a b*) = (*p, q*)  $\implies$  Fract *p q* = Fract *a b*

**by** (*simp add: quotient-of-Fract normalize-eq*)

**lemma** *quotient-of-denom-pos*: *quotient-of* *r* = (*p, q*)  $\implies q > 0$

**by** (*cases r*) (*simp add: quotient-of-Fract normalize-denom-pos*)

**lemma** *quotient-of-coprime*: *quotient-of* *r* = (*p, q*)  $\implies \text{coprime } p q$

**by** (*cases r*) (*simp add: quotient-of-Fract normalize-coprime*)

**lemma** *quotient-of-inject*:

**assumes** *quotient-of a* = *quotient-of b*

**shows** *a* = *b*

**proof** –

**obtain** *p q r s* **where** *a* = Fract *p q*

**and** *b*: *b* = Fract *r s*

**and**  $q > 0$  **and**  $s > 0$  **by** (*cases a, cases b*)

**with** *assms* **show** *?thesis* **by** (*simp add: eq-rat quotient-of-Fract normalize-crossproduct*)

qed

**lemma** *quotient-of-inject-eq*:

*quotient-of a* = *quotient-of b*  $\longleftrightarrow a = b$

by (auto simp add: quotient-of-inject)

#### 65.1.4 The field of rational numbers

**instantiation** *rat* :: *field-inverse-zero*  
**begin**

**definition**

*inverse-rat-def*:  
 $\text{inverse } q = \text{Abs-Rat } (\bigcup x \in \text{Rep-Rat } q.$   
 $\text{ratrel } “ \{ \text{if } \text{fst } x = 0 \text{ then } (0, 1) \text{ else } (\text{snd } x, \text{fst } x) \} )$

**lemma** *inverse-rat* [*simp*]:  $\text{inverse } (\text{Fract } a \ b) = \text{Fract } b \ a$

**proof** –

**have**  $(\lambda x. \text{ratrel } “ \{ \text{if } \text{fst } x = 0 \text{ then } (0, 1) \text{ else } (\text{snd } x, \text{fst } x) \} )$  respects *ratrel*  
**by** (auto simp add: congruent-def mult-commute)  
**then show** ?thesis **by** (simp add: *Fract-def inverse-rat-def UN-ratrel*)

**qed**

**definition**

*divide-rat-def*:  $q \ / \ r = q \ * \ \text{inverse } (r::\text{rat})$

**lemma** *divide-rat* [*simp*]:  $\text{Fract } a \ b \ / \ \text{Fract } c \ d = \text{Fract } (a \ * \ d) \ (b \ * \ c)$   
**by** (simp add: *divide-rat-def*)

**instance proof**

**fix** *q* :: *rat*  
**assume**  $q \neq 0$   
**then show**  $\text{inverse } q \ * \ q = 1$  **by** (cases *q* rule: *Rat-cases-nonzero*)  
 (simp-all add: *rat-number-expand eq-rat*)

**next**

**fix** *q r* :: *rat*  
**show**  $q \ / \ r = q \ * \ \text{inverse } r$  **by** (simp add: *divide-rat-def*)

**next**

**show**  $\text{inverse } 0 = (0::\text{rat})$  **by** (simp add: *rat-number-expand*, *simp add: rat-number-collapse*)

**qed**

**end**

#### 65.1.5 Various

**lemma** *Fract-add-one*:  $n \neq 0 \implies \text{Fract } (m + n) \ n = \text{Fract } m \ n + 1$   
**by** (simp add: *rat-number-expand*)

**lemma** *Fract-of-int-quotient*:  $\text{Fract } k \ l = \text{of-int } k \ / \ \text{of-int } l$   
**by** (simp add: *Fract-of-int-eq [symmetric]*)

**lemma** *Fract-number-of-quotient*:

$\text{Fract } (\text{number-of } k) \ (\text{number-of } l) = \text{number-of } k \ / \ \text{number-of } l$   
**unfolding** *Fract-of-int-quotient number-of-is-id number-of-eq ..*



**lemma** *Fract-1-number-of*:  
*Fract 1 (number-of k) = 1 / number-of k*  
**unfolding** *Fract-of-int-quotient number-of-eq* **by** *simp*

### 65.1.6 The ordered field of rational numbers

**instantiation** *rat* :: *linorder*  
**begin**

**definition**

*le-rat-def*:  
 $q \leq r \longleftrightarrow \text{contents } (\bigcup x \in \text{Rep-Rat } q. \bigcup y \in \text{Rep-Rat } r. \\ \{(fst\ x * snd\ y) * (snd\ x * snd\ y) \leq (fst\ y * snd\ x) * (snd\ x * snd\ y)\})$

**lemma** *le-rat [simp]*:  
**assumes**  $b \neq 0$  **and**  $d \neq 0$   
**shows**  $\text{Fract } a\ b \leq \text{Fract } c\ d \longleftrightarrow (a * d) * (b * d) \leq (c * b) * (b * d)$   
**proof** –  
**have**  $(\lambda x\ y. \{(fst\ x * snd\ y) * (snd\ x * snd\ y) \leq (fst\ y * snd\ x) * (snd\ x * snd\ y)\})$   
*respects2 ratrel*  
**proof** (*clarsimp simp add: congruent2-def*)  
**fix**  $a\ b\ a'\ b'\ c\ d\ c'\ d' :: \text{int}$   
**assume** *neg*:  $b \neq 0\ b' \neq 0\ d \neq 0\ d' \neq 0$   
**assume** *eq1*:  $a * b' = a' * b$   
**assume** *eq2*:  $c * d' = c' * d$   
  
**let**  $?le = \lambda a\ b\ c\ d. ((a * d) * (b * d) \leq (c * b) * (b * d))$   
**{**  
**fix**  $a\ b\ c\ d\ x :: \text{int}$  **assume**  $x \neq 0$   
**have**  $?le\ a\ b\ c\ d = ?le\ (a * x)\ (b * x)\ c\ d$   
**proof** –  
**from**  $x$  **have**  $0 < x * x$  **by** (*auto simp add: zero-less-mult-iff*)  
**hence**  $?le\ a\ b\ c\ d =$   
 $((a * d) * (b * d) * (x * x) \leq (c * b) * (b * d) * (x * x))$   
**by** (*simp add: mult-le-cancel-right*)  
**also have**  $\dots = ?le\ (a * x)\ (b * x)\ c\ d$   
**by** (*simp add: mult-ac*)  
**finally show** *?thesis* .  
**qed**  
**}** **note** *le-factor = this*

**let**  $?D = b * d$  **and**  $?D' = b' * d'$   
**from** *neg* **have**  $D: ?D \neq 0$  **by** *simp*  
**from** *neg* **have**  $?D' \neq 0$  **by** *simp*  
**hence**  $?le\ a\ b\ c\ d = ?le\ (a * ?D')\ (b * ?D')\ c\ d$   
**by** (*rule le-factor*)  
**also have**  $\dots = ((a * b') * ?D * ?D' * d * d' \leq (c * d') * ?D * ?D' * b * b')$

```

    by (simp add: mult-ac)
  also have ... = ((a' * b) * ?D * ?D' * d * d' ≤ (c' * d) * ?D * ?D' * b * b')
    by (simp only: eq1 eq2)
  also have ... = ?le (a' * ?D) (b' * ?D) c' d'
    by (simp add: mult-ac)
  also from D have ... = ?le a' b' c' d'
    by (rule le-factor [symmetric])
  finally show ?le a b c d = ?le a' b' c' d' .
qed
with assms show ?thesis by (simp add: Fract-def le-rat-def UN-ratrel2)
qed

```

**definition**

*less-rat-def*:  $z < (w::rat) \longleftrightarrow z \leq w \wedge z \neq w$

**lemma** *less-rat* [simp]:

assumes  $b \neq 0$  and  $d \neq 0$   
 shows  $\text{Fract } a \ b < \text{Fract } c \ d \longleftrightarrow (a * d) * (b * d) < (c * b) * (b * d)$   
 using assms by (simp add: less-rat-def eq-rat order-less-le)

**instance proof**

```

fix q r s :: rat
{
  assume q ≤ r and r ≤ s
  then show q ≤ s
  proof (induct q, induct r, induct s)
    fix a b c d e f :: int
    assume neq: b > 0 d > 0 f > 0
    assume 1: Fract a b ≤ Fract c d and 2: Fract c d ≤ Fract e f
    show Fract a b ≤ Fract e f
    proof -
      from neq obtain bb: 0 < b * b and dd: 0 < d * d and ff: 0 < f * f
        by (auto simp add: zero-less-mult-iff linorder-neq-iff)
      have (a * d) * (b * d) * (f * f) ≤ (c * b) * (b * d) * (f * f)
      proof -
        from neq 1 have (a * d) * (b * d) ≤ (c * b) * (b * d)
          by simp
        with ff show ?thesis by (simp add: mult-le-cancel-right)
      qed
      also have ... = (c * f) * (d * f) * (b * b) by algebra
      also have ... ≤ (e * d) * (d * f) * (b * b)
      proof -
        from neq 2 have (c * f) * (d * f) ≤ (e * d) * (d * f)
          by simp
        with bb show ?thesis by (simp add: mult-le-cancel-right)
      qed
      finally have (a * f) * (b * f) * (d * d) ≤ e * b * (b * f) * (d * d)
        by (simp only: mult-ac)
      with dd have (a * f) * (b * f) ≤ (e * b) * (b * f)

```

```

      by (simp add: mult-le-cancel-right)
    with neq show ?thesis by simp
  qed
qed
next
  assume  $q \leq r$  and  $r \leq q$ 
  then show  $q = r$ 
  proof (induct q, induct r)
    fix a b c d :: int
    assume neq:  $b > 0$   $d > 0$ 
    assume 1:  $\text{Fract } a \ b \leq \text{Fract } c \ d$  and 2:  $\text{Fract } c \ d \leq \text{Fract } a \ b$ 
    show  $\text{Fract } a \ b = \text{Fract } c \ d$ 
    proof -
      from neq 1 have  $(a * d) * (b * d) \leq (c * b) * (b * d)$ 
      by simp
      also have  $\dots \leq (a * d) * (b * d)$ 
      proof -
        from neq 2 have  $(c * b) * (d * b) \leq (a * d) * (d * b)$ 
        by simp
        thus ?thesis by (simp only: mult-ac)
      qed
      finally have  $(a * d) * (b * d) = (c * b) * (b * d)$  .
      moreover from neq have  $b * d \neq 0$  by simp
      ultimately have  $a * d = c * b$  by simp
      with neq show ?thesis by (simp add: eq-rat)
    qed
  qed
qed
next
  show  $q \leq q$ 
  by (induct q) simp
  show  $(q < r) = (q \leq r \wedge \neg r \leq q)$ 
  by (induct q, induct r) (auto simp add: le-less mult-commute)
  show  $q \leq r \vee r \leq q$ 
  by (induct q, induct r)
    (simp add: mult-commute, rule linorder-linear)
}
qed
end

instantiation rat :: {distrib-lattice, abs-if, sgn-if}
begin

definition
  abs-rat-def:  $|q| = (\text{if } q < 0 \text{ then } -q \text{ else } (q::rat))$ 

lemma abs-rat [simp, code]:  $|\text{Fract } a \ b| = \text{Fract } |a| \ |b|$ 
  by (auto simp add: abs-rat-def zabs-def Zero-rat-def not-less le-less eq-rat zero-less-mult-iff)

```

**definition**

*sgn-rat-def*:  $\text{sgn } (q::\text{rat}) = (\text{if } q = 0 \text{ then } 0 \text{ else if } 0 < q \text{ then } 1 \text{ else } -1)$

**lemma** *sgn-rat* [*simp*, *code*]:  $\text{sgn } (\text{Fract } a \ b) = \text{of-int } (\text{sgn } a * \text{sgn } b)$

**unfolding** *Fract-of-int-eq*

**by** (*auto simp: zsgn-def sgn-rat-def Zero-rat-def eq-rat*)

(*auto simp: rat-number-collapse not-less le-less zero-less-mult-iff*)

**definition**

(*inf :: rat  $\Rightarrow$  rat  $\Rightarrow$  rat*) = *min*

**definition**

(*sup :: rat  $\Rightarrow$  rat  $\Rightarrow$  rat*) = *max*

**instance by** *intro-classes*

(*auto simp add: abs-rat-def sgn-rat-def min-max.sup-inf-distrib1 inf-rat-def sup-rat-def*)

**end**

**instance** *rat :: linordered-field-inverse-zero*

**proof**

**fix** *q r s :: rat*

**show**  $q \leq r \implies s + q \leq s + r$

**proof** (*induct q, induct r, induct s*)

**fix** *a b c d e f :: int*

**assume** *neq*:  $b > 0 \ d > 0 \ f > 0$

**assume** *le*:  $\text{Fract } a \ b \leq \text{Fract } c \ d$

**show**  $\text{Fract } e \ f + \text{Fract } a \ b \leq \text{Fract } e \ f + \text{Fract } c \ d$

**proof** –

**let**  $?F = f * f$  **from** *neq* **have**  $0 < ?F$

**by** (*auto simp add: zero-less-mult-iff*)

**from** *neq le* **have**  $(a * d) * (b * d) \leq (c * b) * (b * d)$

**by** *simp*

**with** *F* **have**  $(a * d) * (b * d) * ?F * ?F \leq (c * b) * (b * d) * ?F * ?F$

**by** (*simp add: mult-le-cancel-right*)

**with** *neq* **show** *?thesis* **by** (*simp add: mult-ac int-distrib*)

**qed**

**qed**

**show**  $q < r \implies 0 < s \implies s * q < s * r$

**proof** (*induct q, induct r, induct s*)

**fix** *a b c d e f :: int*

**assume** *neq*:  $b > 0 \ d > 0 \ f > 0$

**assume** *le*:  $\text{Fract } a \ b < \text{Fract } c \ d$

**assume** *gt*:  $0 < \text{Fract } e \ f$

**show**  $\text{Fract } e \ f * \text{Fract } a \ b < \text{Fract } e \ f * \text{Fract } c \ d$

**proof** –

**let**  $?E = e * f$  **and**  $?F = f * f$

**from** *neq gt* **have**  $0 < ?E$

**by** (*auto simp add: Zero-rat-def order-less-le eq-rat*)

```

moreover from neq have  $0 < ?F$ 
  by (auto simp add: zero-less-mult-iff)
moreover from neq le have  $(a * d) * (b * d) < (c * b) * (b * d)$ 
  by simp
ultimately have  $(a * d) * (b * d) * ?E * ?F < (c * b) * (b * d) * ?E * ?F$ 
  by (simp add: mult-less-cancel-right)
with neq show ?thesis
  by (simp add: mult-ac)
qed
qed
qed auto

```

```

lemma Rat-induct-pos [case-names Fract, induct type: rat]:
  assumes step:  $\bigwedge a b. 0 < b \implies P \text{ (Fract } a \text{ } b)$ 
  shows  $P \text{ } q$ 
proof (cases q)
  have step':  $\bigwedge a b. b < 0 \implies P \text{ (Fract } a \text{ } b)$ 
  proof –
    fix a::int and b::int
    assume b:  $b < 0$ 
    hence  $0 < -b$  by simp
    hence  $P \text{ (Fract } (-a) \text{ } (-b))$  by (rule step)
    thus  $P \text{ (Fract } a \text{ } b)$  by (simp add: order-less-imp-not-eq [OF b])
  qed
  case (Fract a b)
  thus  $P \text{ } q$  by (force simp add: linorder-neq-iff step step')
qed

```

```

lemma zero-less-Fract-iff:
   $0 < b \implies 0 < \text{Fract } a \text{ } b \longleftrightarrow 0 < a$ 
  by (simp add: Zero-rat-def zero-less-mult-iff)

```

```

lemma Fract-less-zero-iff:
   $0 < b \implies \text{Fract } a \text{ } b < 0 \longleftrightarrow a < 0$ 
  by (simp add: Zero-rat-def mult-less-0-iff)

```

```

lemma zero-le-Fract-iff:
   $0 < b \implies 0 \leq \text{Fract } a \text{ } b \longleftrightarrow 0 \leq a$ 
  by (simp add: Zero-rat-def zero-le-mult-iff)

```

```

lemma Fract-le-zero-iff:
   $0 < b \implies \text{Fract } a \text{ } b \leq 0 \longleftrightarrow a \leq 0$ 
  by (simp add: Zero-rat-def mult-le-0-iff)

```

```

lemma one-less-Fract-iff:
   $0 < b \implies 1 < \text{Fract } a \text{ } b \longleftrightarrow b < a$ 
  by (simp add: One-rat-def mult-less-cancel-right-disj)

```

```

lemma Fract-less-one-iff:

```

$0 < b \implies \text{Fract } a \ b < 1 \iff a < b$   
**by** (*simp add: One-rat-def mult-less-cancel-right-disj*)

**lemma** *one-le-Fract-iff*:  
 $0 < b \implies 1 \leq \text{Fract } a \ b \iff b \leq a$   
**by** (*simp add: One-rat-def mult-le-cancel-right*)

**lemma** *Fract-le-one-iff*:  
 $0 < b \implies \text{Fract } a \ b \leq 1 \iff a \leq b$   
**by** (*simp add: One-rat-def mult-le-cancel-right*)

### 65.1.7 Rationals are an Archimedean field

**lemma** *rat-floor-lemma*:  
**shows**  $\text{of-int } (a \text{ div } b) \leq \text{Fract } a \ b \wedge \text{Fract } a \ b < \text{of-int } (a \text{ div } b + 1)$   
**proof** –  
**have**  $\text{Fract } a \ b = \text{of-int } (a \text{ div } b) + \text{Fract } (a \text{ mod } b) \ b$   
**by** (*cases b = 0, simp, simp add: of-int-rat*)  
**moreover have**  $0 \leq \text{Fract } (a \text{ mod } b) \ b \wedge \text{Fract } (a \text{ mod } b) \ b < 1$   
**unfolding** *Fract-of-int-quotient*  
**by** (*rule linorder-cases [of b 0]*) (*simp add: divide-nonpos-neg, simp, simp add: divide-nonneg-pos*)  
**ultimately show** *?thesis* **by** *simp*  
**qed**

**instance** *rat :: archimedean-field*  
**proof**  
**fix**  $r :: \text{rat}$   
**show**  $\exists z. r \leq \text{of-int } z$   
**proof** (*induct r*)  
**case** ( $\text{Fract } a \ b$ )  
**have**  $\text{Fract } a \ b \leq \text{of-int } (a \text{ div } b + 1)$   
**using** *rat-floor-lemma [of a b]* **by** *simp*  
**then show**  $\exists z. \text{Fract } a \ b \leq \text{of-int } z$ ..  
**qed**  
**qed**

**lemma** *floor-Fract*:  $\text{floor } (\text{Fract } a \ b) = a \text{ div } b$   
**using** *rat-floor-lemma [of a b]*  
**by** (*simp add: floor-unique*)

## 65.2 Linear arithmetic setup

**declaration**  $\ll$   
 $K \ (\text{Lin-Arith.add-inj-thms } [\text{@}\{\text{thm of-nat-le-iff}\} \text{ RS iffD2}, \text{@}\{\text{thm of-nat-eq-iff}\} \text{ RS iffD2}]$   
*(\* not needed because  $x < (y::\text{nat})$  can be rewritten as  $\text{Suc } x <= y$ : of-nat-less-iff*  
 $\text{RS iffD2 } *)$   
 $\#> \text{Lin-Arith.add-inj-thms } [\text{@}\{\text{thm of-int-le-iff}\} \text{ RS iffD2}, \text{@}\{\text{thm of-int-eq-iff}\} \text{ RS iffD2}]$

(*\* not needed because  $x < (y::int)$  can be rewritten as  $x + 1 \leq y$ : of-int-less-iff RS iffD2 \**)

```

#> Lin-Arith.add-simps [@{thm neg-less-iff-less},
  @{thm True-implies-equals},
  read-instantiate @{context} [((a, 0), (number-of ?v))] @{thm right-distrib},
  @{thm divide-1}, @{thm divide-zero-left},
  @{thm times-divide-eq-right}, @{thm times-divide-eq-left},
  @{thm minus-divide-left} RS sym, @{thm minus-divide-right} RS sym,
  @{thm of-int-minus}, @{thm of-int-diff},
  @{thm of-int-of-nat-eq}]
#> Lin-Arith.add-simprocs Numeral-Simprocs.field-cancel-numeral-factors
#> Lin-Arith.add-inj-const (@{const-name of-nat}, @{typ nat => rat})
#> Lin-Arith.add-inj-const (@{const-name of-int}, @{typ int => rat}))
>>

```

### 65.3 Embedding from Rationals to other Fields

**class** *field-char-0* = *field* + *ring-char-0*

**subclass** (in *linordered-field*) *field-char-0* ..

**context** *field-char-0*

**begin**

**definition** *of-rat* :: *rat*  $\Rightarrow$  'a **where**

*of-rat* *q* = *contents* ( $\bigcup (a,b) \in \text{Rep-Rat } q. \{ \text{of-int } a / \text{of-int } b \}$ )

**end**

**lemma** *of-rat-congruent*:

( $\lambda(a, b). \{ \text{of-int } a / \text{of-int } b :: 'a::\text{field-char-0} \}$ ) *respects ratrel*

**apply** (*rule congruent.intro*)

**apply** (*clarsimp simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq*)

**apply** (*simp only: of-int-mult [symmetric]*)

**done**

**lemma** *of-rat-rat*:  $b \neq 0 \implies \text{of-rat } (\text{Fract } a \ b) = \text{of-int } a / \text{of-int } b$

**unfolding** *Fract-def of-rat-def* **by** (*simp add: UN-ratrel of-rat-congruent*)

**lemma** *of-rat-0* [*simp*]: *of-rat* 0 = 0

**by** (*simp add: Zero-rat-def of-rat-rat*)

**lemma** *of-rat-1* [*simp*]: *of-rat* 1 = 1

**by** (*simp add: One-rat-def of-rat-rat*)

**lemma** *of-rat-add*: *of-rat* (*a* + *b*) = *of-rat* *a* + *of-rat* *b*

**by** (*induct a, induct b, simp add: of-rat-rat add-frac-eq*)

**lemma** *of-rat-minus*: *of-rat* (− *a*) = − *of-rat* *a*

**by** (*induct a, simp add: of-rat-rat*)

**lemma** *of-rat-diff*:  $\text{of-rat } (a - b) = \text{of-rat } a - \text{of-rat } b$   
**by** (*simp only: diff-minus of-rat-add of-rat-minus*)

**lemma** *of-rat-mult*:  $\text{of-rat } (a * b) = \text{of-rat } a * \text{of-rat } b$   
**apply** (*induct a, induct b, simp add: of-rat-rat*)  
**apply** (*simp add: divide-inverse nonzero-inverse-mult-distrib mult-ac*)  
**done**

**lemma** *nonzero-of-rat-inverse*:  
 $a \neq 0 \implies \text{of-rat } (\text{inverse } a) = \text{inverse } (\text{of-rat } a)$   
**apply** (*rule inverse-unique [symmetric]*)  
**apply** (*simp add: of-rat-mult [symmetric]*)  
**done**

**lemma** *of-rat-inverse*:  
 $(\text{of-rat } (\text{inverse } a) :: 'a :: \{\text{field-char-0, field-inverse-zero}\}) = \text{inverse } (\text{of-rat } a)$   
**by** (*cases a = 0, simp-all add: nonzero-of-rat-inverse*)

**lemma** *nonzero-of-rat-divide*:  
 $b \neq 0 \implies \text{of-rat } (a / b) = \text{of-rat } a / \text{of-rat } b$   
**by** (*simp add: divide-inverse of-rat-mult nonzero-of-rat-inverse*)

**lemma** *of-rat-divide*:  
 $(\text{of-rat } (a / b) :: 'a :: \{\text{field-char-0, field-inverse-zero}\}) = \text{of-rat } a / \text{of-rat } b$   
**by** (*cases b = 0 (simp-all add: nonzero-of-rat-divide)*)

**lemma** *of-rat-power*:  
 $(\text{of-rat } (a ^ n) :: 'a :: \text{field-char-0}) = \text{of-rat } a ^ n$   
**by** (*induct n (simp-all add: of-rat-mult)*)

**lemma** *of-rat-eq-iff* [*simp*]:  $(\text{of-rat } a = \text{of-rat } b) = (a = b)$   
**apply** (*induct a, induct b*)  
**apply** (*simp add: of-rat-rat eq-rat*)  
**apply** (*simp add: nonzero-divide-eq-eq nonzero-eq-divide-eq*)  
**apply** (*simp only: of-int-mult [symmetric] of-int-eq-iff*)  
**done**

**lemma** *of-rat-less*:  
 $(\text{of-rat } r :: 'a :: \text{linordered-field}) < \text{of-rat } s \iff r < s$   
**proof** (*induct r, induct s*)  
**fix** *a b c d :: int*  
**assume** *not-zero: b > 0 d > 0*  
**then have**  $b * d > 0$  **by** (*rule mult-pos-pos*)  
**have** *of-int-divide-less-eq*:  
 $(\text{of-int } a :: 'a) / \text{of-int } b < \text{of-int } c / \text{of-int } d$



```

     $\longleftrightarrow (of\text{-}int\ a :: 'a) * of\text{-}int\ d < of\text{-}int\ c * of\text{-}int\ b$ 
  using not-zero by (simp add: pos-less-divide-eq pos-divide-less-eq)
  show (of-rat (Fract a b) :: 'a::linordered-field) < of-rat (Fract c d)
     $\longleftrightarrow Fract\ a\ b < Fract\ c\ d$ 
  using not-zero  $\langle b * d > 0 \rangle$ 
  by (simp add: of-rat-rat of-int-divide-less-eq of-int-mult [symmetric] del: of-int-mult)
qed

```

**lemma** *of-rat-less-eq*:

```

  (of-rat r :: 'a::linordered-field)  $\leq$  of-rat s  $\longleftrightarrow r \leq s$ 
  unfolding le-less by (auto simp add: of-rat-less)

```

**lemmas** *of-rat-eq-0-iff* [simp] = *of-rat-eq-iff* [of - 0, simplified]

**lemma** *of-rat-eq-id* [simp]: *of-rat* = *id*

**proof**

fix *a*

show *of-rat a* = *id a*

by (induct *a*)

(simp add: of-rat-rat Fract-of-int-eq [symmetric])

qed

Collapse nested embeddings

**lemma** *of-rat-of-nat-eq* [simp]: *of-rat* (*of-nat n*) = *of-nat n*

by (induct *n*) (simp-all add: of-rat-add)

**lemma** *of-rat-of-int-eq* [simp]: *of-rat* (*of-int z*) = *of-int z*

by (cases *z* rule: int-diff-cases) (simp add: of-rat-diff)

**lemma** *of-rat-number-of-eq* [simp]:

*of-rat* (*number-of w*) = (*number-of w* :: 'a::{number-ring,field-char-0})

by (simp add: number-of-eq)

**lemmas** *zero-rat* = *Zero-rat-def*

**lemmas** *one-rat* = *One-rat-def*

**abbreviation**

*rat-of-nat* :: *nat*  $\Rightarrow$  *rat*

**where**

*rat-of-nat*  $\equiv$  *of-nat*

**abbreviation**

*rat-of-int* :: *int*  $\Rightarrow$  *rat*

**where**

*rat-of-int*  $\equiv$  *of-int*

## 65.4 The Set of Rational Numbers

**context** *field-char-0*

**begin**

**definition**

*Rats* :: 'a set **where**  
*Rats* = range of-rat

**notation** (*xsymbols*)

*Rats* ( $\mathbb{Q}$ )

**end**

**lemma** *Rats-of-rat* [*simp*]: of-rat  $r \in Rats$   
**by** (*simp add: Rats-def*)

**lemma** *Rats-of-int* [*simp*]: of-int  $z \in Rats$   
**by** (*subst of-rat-of-int-eq [symmetric]*, rule *Rats-of-rat*)

**lemma** *Rats-of-nat* [*simp*]: of-nat  $n \in Rats$   
**by** (*subst of-rat-of-nat-eq [symmetric]*, rule *Rats-of-rat*)

**lemma** *Rats-number-of* [*simp*]:  
 (*number-of w::'a::{number-ring,field-char-0}*)  $\in Rats$   
**by** (*subst of-rat-number-of-eq [symmetric]*, rule *Rats-of-rat*)

**lemma** *Rats-0* [*simp*]:  $0 \in Rats$   
**apply** (*unfold Rats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-rat-0 [symmetric]*)  
**done**

**lemma** *Rats-1* [*simp*]:  $1 \in Rats$   
**apply** (*unfold Rats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-rat-1 [symmetric]*)  
**done**

**lemma** *Rats-add* [*simp*]:  $\llbracket a \in Rats; b \in Rats \rrbracket \implies a + b \in Rats$   
**apply** (*auto simp add: Rats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-rat-add [symmetric]*)  
**done**

**lemma** *Rats-minus* [*simp*]:  $a \in Rats \implies - a \in Rats$   
**apply** (*auto simp add: Rats-def*)  
**apply** (*rule range-eqI*)  
**apply** (*rule of-rat-minus [symmetric]*)  
**done**

**lemma** *Rats-diff* [*simp*]:  $\llbracket a \in Rats; b \in Rats \rrbracket \implies a - b \in Rats$

```

apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-diff [symmetric])
done

```

```

lemma Rats-mult [simp]:  $\llbracket a \in \text{Rats}; b \in \text{Rats} \rrbracket \implies a * b \in \text{Rats}$ 
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-mult [symmetric])
done

```

```

lemma nonzero-Rats-inverse:
  fixes a :: 'a::field-char-0
  shows  $\llbracket a \in \text{Rats}; a \neq 0 \rrbracket \implies \text{inverse } a \in \text{Rats}$ 
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (erule nonzero-of-rat-inverse [symmetric])
done

```

```

lemma Rats-inverse [simp]:
  fixes a :: 'a::{field-char-0, field-inverse-zero}
  shows  $a \in \text{Rats} \implies \text{inverse } a \in \text{Rats}$ 
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-inverse [symmetric])
done

```

```

lemma nonzero-Rats-divide:
  fixes a b :: 'a::field-char-0
  shows  $\llbracket a \in \text{Rats}; b \in \text{Rats}; b \neq 0 \rrbracket \implies a / b \in \text{Rats}$ 
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (erule nonzero-of-rat-divide [symmetric])
done

```

```

lemma Rats-divide [simp]:
  fixes a b :: 'a::{field-char-0, field-inverse-zero}
  shows  $\llbracket a \in \text{Rats}; b \in \text{Rats} \rrbracket \implies a / b \in \text{Rats}$ 
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-divide [symmetric])
done

```

```

lemma Rats-power [simp]:
  fixes a :: 'a::field-char-0
  shows  $a \in \text{Rats} \implies a ^ n \in \text{Rats}$ 
apply (auto simp add: Rats-def)
apply (rule range-eqI)
apply (rule of-rat-power [symmetric])

```

done

**lemma** *Rats-cases* [*cases set: Rats*]:  
 assumes  $q \in \mathbb{Q}$   
 obtains (*of-rat*)  $r$  **where**  $q = \text{of-rat } r$   
 unfolding *Rats-def*  
**proof** –  
 from  $\langle q \in \mathbb{Q} \rangle$  **have**  $q \in \text{range of-rat}$  **unfolding** *Rats-def* .  
**then obtain**  $r$  **where**  $q = \text{of-rat } r$  ..  
**then show** *thesis* ..  
**qed**

**lemma** *Rats-induct* [*case-names of-rat, induct set: Rats*]:  
 $q \in \mathbb{Q} \implies (\bigwedge r. P (\text{of-rat } r)) \implies P q$   
**by** (*rule Rats-cases*) *auto*

## 65.5 Implementation of rational numbers as pairs of integers

**definition** *Frct* ::  $\text{int} \times \text{int} \Rightarrow \text{rat}$  **where**  
 [*simp*]:  $\text{Frct } p = \text{Fract } (\text{fst } p) (\text{snd } p)$

**lemma** [*code abstype*]:  
 $\text{Frct } (\text{quotient-of } q) = q$   
**by** (*cases q*) (*auto intro: quotient-of-eq*)

**lemma** *Frct-code-post* [*code-post*]:  
 $\text{Frct } (0, k) = 0$   
 $\text{Frct } (k, 0) = 0$   
 $\text{Frct } (1, 1) = 1$   
 $\text{Frct } (\text{number-of } k, 1) = \text{number-of } k$   
 $\text{Frct } (1, \text{number-of } k) = 1 / \text{number-of } k$   
 $\text{Frct } (\text{number-of } k, \text{number-of } l) = \text{number-of } k / \text{number-of } l$   
**by** (*simp-all add: rat-number-collapse Fract-number-of-quotient Fract-1-number-of*)

**declare** *quotient-of-Fract* [*code abstract*]

**lemma** *rat-zero-code* [*code abstract*]:  
 $\text{quotient-of } 0 = (0, 1)$   
**by** (*simp add: Zero-rat-def quotient-of-Fract normalize-def*)

**lemma** *rat-one-code* [*code abstract*]:  
 $\text{quotient-of } 1 = (1, 1)$   
**by** (*simp add: One-rat-def quotient-of-Fract normalize-def*)

**lemma** *rat-plus-code* [*code abstract*]:  
 $\text{quotient-of } (p + q) = (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q$   
   *in normalize*  $(a * d + b * c, c * d))$   
**by** (*cases p, cases q*) (*simp add: quotient-of-Fract*)

```

lemma rat-uminus-code [code abstract]:
  quotient-of ( $- p$ ) = (let ( $a, b$ ) = quotient-of  $p$  in ( $- a, b$ ))
  by (cases  $p$ ) (simp add: quotient-of-Fract)

lemma rat-minus-code [code abstract]:
  quotient-of ( $p - q$ ) = (let ( $a, c$ ) = quotient-of  $p$ ; ( $b, d$ ) = quotient-of  $q$ 
    in normalize ( $a * d - b * c, c * d$ ))
  by (cases  $p, cases$   $q$ ) (simp add: quotient-of-Fract)

lemma rat-times-code [code abstract]:
  quotient-of ( $p * q$ ) = (let ( $a, c$ ) = quotient-of  $p$ ; ( $b, d$ ) = quotient-of  $q$ 
    in normalize ( $a * b, c * d$ ))
  by (cases  $p, cases$   $q$ ) (simp add: quotient-of-Fract)

lemma rat-inverse-code [code abstract]:
  quotient-of (inverse  $p$ ) = (let ( $a, b$ ) = quotient-of  $p$ 
    in if  $a = 0$  then ( $0, 1$ ) else (sgn  $a * b, |a|$ ))
proof (cases  $p$ )
  case (Fract  $a b$ ) then show ?thesis
    by (cases  $0::int$  a rule: linorder-cases) (simp-all add: quotient-of-Fract gcd-int commute)
qed

lemma rat-divide-code [code abstract]:
  quotient-of ( $p / q$ ) = (let ( $a, c$ ) = quotient-of  $p$ ; ( $b, d$ ) = quotient-of  $q$ 
    in normalize ( $a * d, c * b$ ))
  by (cases  $p, cases$   $q$ ) (simp add: quotient-of-Fract)

lemma rat-abs-code [code abstract]:
  quotient-of  $|p|$  = (let ( $a, b$ ) = quotient-of  $p$  in ( $|a|, b$ ))
  by (cases  $p$ ) (simp add: quotient-of-Fract)

lemma rat-sgn-code [code abstract]:
  quotient-of (sgn  $p$ ) = (sgn (fst (quotient-of  $p$ )),  $1$ )
proof (cases  $p$ )
  case (Fract  $a b$ ) then show ?thesis
    by (cases  $0::int$  a rule: linorder-cases) (simp-all add: quotient-of-Fract)
qed

instantiation rat :: eq
begin

definition [code]:
  eq-class.eq  $a b \longleftrightarrow$  quotient-of  $a =$  quotient-of  $b$ 

instance proof
qed (simp add: eq-rat-def quotient-of-inject-eq)

lemma rat-eq-refl [code nbe]:
  eq-class.eq ( $r::rat$ )  $r \longleftrightarrow$  True

```

```

by (rule HOL.eq-refl)

end

lemma rat-less-eq-code [code]:
   $p \leq q \longleftrightarrow (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a * d \leq c * b)$ 
by (cases p, cases q) (simp add: quotient-of-Fract mult.commute)

lemma rat-less-code [code]:
   $p < q \longleftrightarrow (\text{let } (a, c) = \text{quotient-of } p; (b, d) = \text{quotient-of } q \text{ in } a * d < c * b)$ 
by (cases p, cases q) (simp add: quotient-of-Fract mult.commute)

lemma [code]:
   $\text{of-rat } p = (\text{let } (a, b) = \text{quotient-of } p \text{ in of-int } a / \text{of-int } b)$ 
by (cases p) (simp add: quotient-of-Fract of-rat-rat)

definition (in term-syntax)
  valterm-fract :: int  $\times$  (unit  $\Rightarrow$  Code-Evaluation.term)  $\Rightarrow$  int  $\times$  (unit  $\Rightarrow$  Code-Evaluation.term)
   $\Rightarrow$  rat  $\times$  (unit  $\Rightarrow$  Code-Evaluation.term) where
    [code-unfold]: valterm-fract k l = Code-Evaluation.valtermify Fract {·} k {·} l

notation fcomp (infixl o> 60)
notation scomp (infixl o $\rightarrow$  60)

instantiation rat :: random
begin

definition
  Quickcheck.random i = Quickcheck.random i o $\rightarrow$  ( $\lambda$ num. Random.range i o $\rightarrow$ 
    ( $\lambda$ denom. Pair (
      let j = Code-Numeral.int-of (denom + 1)
      in valterm-fract num (j,  $\lambda$ u. Code-Evaluation.term-of j))))

instance ..

end

no-notation fcomp (infixl o> 60)
no-notation scomp (infixl o $\rightarrow$  60)

Setup for SML code generator

types-code
  rat ((int */ int))
attach (term-of)  $\ll$ 
fun term-of-rat (p, q) =
  let
    val rT = Type (Rat.rat, [])
  in
    if q = 1 orelse p = 0 then HOLogic.mk-number rT p

```

```

    else @{term op / :: rat ⇒ rat ⇒ rat} $
      HLogic.mk-number rT p $ HLogic.mk-number rT q
  end;
>>
attach (test) <<
fun gen-rat i =
  let
    val p = random-range 0 i;
    val q = random-range 1 (i + 1);
    val g = Integer.gcd p q;
    val p' = p div g;
    val q' = q div g;
    val r = (if one-of [true, false] then p' else ~ p',
              if p' = 0 then 1 else q')
  in
    (r, fn () => term-of-rat r)
  end;
>>

consts-code
  Fract ((-, / -))

consts-code
  quotient-of ({*normalize*})

consts-code
  of-int :: int ⇒ rat (<module>rat'-of'-int)
attach <<
fun rat-of-int i = (i, 1);
>>

setup <<
  Nitpick.register-frac-type @{type-name rat}
  [(@{const-name zero-rat-inst.zero-rat}, @{const-name Nitpick.zero-frac}),
   (@{const-name one-rat-inst.one-rat}, @{const-name Nitpick.one-frac}),
   (@{const-name plus-rat-inst.plus-rat}, @{const-name Nitpick.plus-frac}),
   (@{const-name times-rat-inst.times-rat}, @{const-name Nitpick.times-frac}),
   (@{const-name uminus-rat-inst.uminus-rat}, @{const-name Nitpick.uminus-frac}),
   (@{const-name number-rat-inst.number-of-rat}, @{const-name Nitpick.number-of-frac}),
   (@{const-name inverse-rat-inst.inverse-rat}, @{const-name Nitpick.inverse-frac}),
   (@{const-name ord-rat-inst.less-eq-rat}, @{const-name Nitpick.less-eq-frac}),
   (@{const-name field-char-0-class.of-rat}, @{const-name Nitpick.of-frac}),
   (@{const-name field-char-0-class.Rats}, @{const-abbrev UNIV})]
>>

lemmas [nitpick-def] = inverse-rat-inst.inverse-rat
  number-rat-inst.number-of-rat one-rat-inst.one-rat ord-rat-inst.less-eq-rat
  plus-rat-inst.plus-rat times-rat-inst.times-rat uminus-rat-inst.uminus-rat
  zero-rat-inst.zero-rat

```

## 65.6 Float syntax

**syntax** *-Float* :: *float-const*  $\Rightarrow$  'a    (-)

**use** *Tools/float-syntax.ML*

**setup** *Float-Syntax.setup*

Test:

**lemma**  $123.456 = -111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::rat)$   
**by** *simp*

**hide-const** (open) *normalize*

**end**

## 66 RealDef: Development of the Reals using Cauchy Sequences

**theory** *RealDef*

**imports** *Rat*

**begin**

This theory contains a formalization of the real numbers as equivalence classes of Cauchy sequences of rationals. See [HOL/ex/Dedekind\\_Real.thy](#) for an alternative construction using Dedekind cuts.

### 66.1 Preliminary lemmas

**lemma** *add-diff-add*:

**fixes** *a b c d* :: 'a::ab-group-add

**shows**  $(a + c) - (b + d) = (a - b) + (c - d)$

**by** *simp*

**lemma** *minus-diff-minus*:

**fixes** *a b* :: 'a::ab-group-add

**shows**  $- a - - b = - (a - b)$

**by** *simp*

**lemma** *mult-diff-mult*:

**fixes** *x y a b* :: 'a::ring

**shows**  $(x * y - a * b) = x * (y - b) + (x - a) * b$

**by** (*simp add: algebra-simps*)

**lemma** *inverse-diff-inverse*:

**fixes** *a b* :: 'a::division-ring

**assumes**  $a \neq 0$  **and**  $b \neq 0$



**shows**  $\text{inverse } a - \text{inverse } b = - (\text{inverse } a * (a - b) * \text{inverse } b)$   
**using** *assms* **by** (*simp add: algebra-simps*)

**lemma** *obtain-pos-sum*:

**fixes**  $r :: \text{rat}$  **assumes**  $r: 0 < r$

**obtains**  $s\ t$  **where**  $0 < s$  **and**  $0 < t$  **and**  $r = s + t$

**proof**

**from**  $r$  **show**  $0 < r/2$  **by** *simp*

**from**  $r$  **show**  $0 < r/2$  **by** *simp*

**show**  $r = r/2 + r/2$  **by** *simp*

**qed**

## 66.2 Sequences that converge to zero

**definition**

$\text{vanishes} :: (\text{nat} \Rightarrow \text{rat}) \Rightarrow \text{bool}$

**where**

$\text{vanishes } X = (\forall r > 0. \exists k. \forall n \geq k. |X\ n| < r)$

**lemma** *vanishesI*:  $(\bigwedge r. 0 < r \implies \exists k. \forall n \geq k. |X\ n| < r) \implies \text{vanishes } X$

**unfolding** *vanishes-def* **by** *simp*

**lemma** *vanishesD*:  $\llbracket \text{vanishes } X; 0 < r \rrbracket \implies \exists k. \forall n \geq k. |X\ n| < r$

**unfolding** *vanishes-def* **by** *simp*

**lemma** *vanishes-const* [*simp*]:  $\text{vanishes } (\lambda n. c) \longleftrightarrow c = 0$

**unfolding** *vanishes-def*

**apply** (*cases*  $c = 0$ , *auto*)

**apply** (*rule* *exI* [**where**  $x = |c|$ ], *auto*)

**done**

**lemma** *vanishes-minus*:  $\text{vanishes } X \implies \text{vanishes } (\lambda n. - X\ n)$

**unfolding** *vanishes-def* **by** *simp*

**lemma** *vanishes-add*:

**assumes**  $X: \text{vanishes } X$  **and**  $Y: \text{vanishes } Y$

**shows**  $\text{vanishes } (\lambda n. X\ n + Y\ n)$

**proof** (*rule* *vanishesI*)

**fix**  $r :: \text{rat}$  **assume**  $0 < r$

**then obtain**  $s\ t$  **where**  $s: 0 < s$  **and**  $t: 0 < t$  **and**  $r: r = s + t$

**by** (*rule* *obtain-pos-sum*)

**obtain**  $i$  **where**  $i: \forall n \geq i. |X\ n| < s$

**using** *vanishesD* [*OF*  $X\ s$ ] **..**

**obtain**  $j$  **where**  $j: \forall n \geq j. |Y\ n| < t$

**using** *vanishesD* [*OF*  $Y\ t$ ] **..**

**have**  $\forall n \geq \max i\ j. |X\ n + Y\ n| < r$

**proof** (*clarsimp*)

**fix**  $n$  **assume**  $n: i \leq n\ j \leq n$

**have**  $|X\ n + Y\ n| \leq |X\ n| + |Y\ n|$  **by** (*rule* *abs-triangle-ineq*)

also have  $\dots < s + t$  by (simp add: add-strict-mono i j n)  
 finally show  $|X\ n + Y\ n| < r$  unfolding r .  
 qed  
 thus  $\exists k. \forall n \geq k. |X\ n + Y\ n| < r$  ..  
 qed

**lemma** *vanishes-diff*:  
 assumes  $X$ : *vanishes*  $X$  and  $Y$ : *vanishes*  $Y$   
 shows *vanishes*  $(\lambda n. X\ n - Y\ n)$   
 unfolding *diff-minus* by (intro *vanishes-add vanishes-minus*  $X\ Y$ )

**lemma** *vanishes-mult-bounded*:  
 assumes  $X$ :  $\exists a > 0. \forall n. |X\ n| < a$   
 assumes  $Y$ : *vanishes*  $(\lambda n. Y\ n)$   
 shows *vanishes*  $(\lambda n. X\ n * Y\ n)$   
**proof** (rule *vanishesI*)  
 fix  $r :: \text{rat}$  assume  $r$ :  $0 < r$   
 obtain  $a$  where  $a$ :  $0 < a \ \forall n. |X\ n| < a$   
 using  $X$  by fast  
 obtain  $b$  where  $b$ :  $0 < b \ r = a * b$   
**proof**  
 show  $0 < r / a$  using  $r\ a$  by (simp add: divide-pos-pos)  
 show  $r = a * (r / a)$  using  $a$  by simp  
 qed  
 obtain  $k$  where  $k$ :  $\forall n \geq k. |Y\ n| < b$   
 using *vanishesD* [OF  $Y\ b(1)$ ] ..  
 have  $\forall n \geq k. |X\ n * Y\ n| < r$   
 by (simp add:  $b(2)$  *abs-mult mult-strict-mono'*  $a\ k$ )  
 thus  $\exists k. \forall n \geq k. |X\ n * Y\ n| < r$  ..  
 qed

### 66.3 Cauchy sequences

**definition**  
*cauchy* ::  $(\text{nat} \Rightarrow \text{rat})$  set  
**where**  
*cauchy*  $X \longleftrightarrow (\forall r > 0. \exists k. \forall m \geq k. \forall n \geq k. |X\ m - X\ n| < r)$

**lemma** *cauchyI*:  
 $(\bigwedge r. 0 < r \implies \exists k. \forall m \geq k. \forall n \geq k. |X\ m - X\ n| < r) \implies \text{cauchy } X$   
 unfolding *cauchy-def* by simp

**lemma** *cauchyD*:  
 $\llbracket \text{cauchy } X; 0 < r \rrbracket \implies \exists k. \forall m \geq k. \forall n \geq k. |X\ m - X\ n| < r$   
 unfolding *cauchy-def* by simp

**lemma** *cauchy-const* [simp]: *cauchy*  $(\lambda n. x)$   
 unfolding *cauchy-def* by simp

```

lemma cauchy-add [simp]:
  assumes X: cauchy X and Y: cauchy Y
  shows cauchy ( $\lambda n. X\ n + Y\ n$ )
proof (rule cauchyI)
  fix r :: rat assume  $0 < r$ 
  then obtain s t where s:  $0 < s$  and t:  $0 < t$  and r:  $r = s + t$ 
    by (rule obtain-pos-sum)
  obtain i where i:  $\forall m \geq i. \forall n \geq i. |X\ m - X\ n| < s$ 
    using cauchyD [OF X s] ..
  obtain j where j:  $\forall m \geq j. \forall n \geq j. |Y\ m - Y\ n| < t$ 
    using cauchyD [OF Y t] ..
  have  $\forall m \geq \max\ i\ j. \forall n \geq \max\ i\ j. |(X\ m + Y\ m) - (X\ n + Y\ n)| < r$ 
proof (clarsimp)
  fix m n assume *:  $i \leq m\ j \leq m\ i \leq n\ j \leq n$ 
  have  $|(X\ m + Y\ m) - (X\ n + Y\ n)| \leq |X\ m - X\ n| + |Y\ m - Y\ n|$ 
    unfolding add-diff-add by (rule abs-triangle-ineq)
  also have  $\dots < s + t$ 
    by (rule add-strict-mono, simp-all add: i j *)
  finally show  $|(X\ m + Y\ m) - (X\ n + Y\ n)| < r$  unfolding r .
qed
thus  $\exists k. \forall m \geq k. \forall n \geq k. |(X\ m + Y\ m) - (X\ n + Y\ n)| < r$  ..
qed

```

```

lemma cauchy-minus [simp]:
  assumes X: cauchy X
  shows cauchy ( $\lambda n. - X\ n$ )
using assms unfolding cauchy-def
unfolding minus-diff-minus abs-minus-cancel .

```

```

lemma cauchy-diff [simp]:
  assumes X: cauchy X and Y: cauchy Y
  shows cauchy ( $\lambda n. X\ n - Y\ n$ )
using assms unfolding diff-minus by simp

```

```

lemma cauchy-imp-bounded:
  assumes cauchy X shows  $\exists b > 0. \forall n. |X\ n| < b$ 
proof -
  obtain k where k:  $\forall m \geq k. \forall n \geq k. |X\ m - X\ n| < 1$ 
    using cauchyD [OF assms zero-less-one] ..
  show  $\exists b > 0. \forall n. |X\ n| < b$ 
proof (intro exI conjI allI)
  have  $0 \leq |X\ 0|$  by simp
  also have  $|X\ 0| \leq \text{Max}\ (\text{abs}\ 'X'\ \{\dots k\})$  by simp
  finally have  $0 \leq \text{Max}\ (\text{abs}\ 'X'\ \{\dots k\})$  .
  thus  $0 < \text{Max}\ (\text{abs}\ 'X'\ \{\dots k\}) + 1$  by simp
next
  fix n :: nat
  show  $|X\ n| < \text{Max}\ (\text{abs}\ 'X'\ \{\dots k\}) + 1$ 
proof (rule linorder-le-cases)

```

```

    assume  $n \leq k$ 
    hence  $|X\ n| \leq \text{Max } (\text{abs } 'X' \{..k\})$  by simp
    thus  $|X\ n| < \text{Max } (\text{abs } 'X' \{..k\}) + 1$  by simp
  next
    assume  $k \leq n$ 
    have  $|X\ n| = |X\ k + (X\ n - X\ k)|$  by simp
    also have  $|X\ k + (X\ n - X\ k)| \leq |X\ k| + |X\ n - X\ k|$ 
      by (rule abs-triangle-ineq)
    also have  $\dots < \text{Max } (\text{abs } 'X' \{..k\}) + 1$ 
      by (rule add-le-less-mono, simp, simp add:  $k \leq n$ )
    finally show  $|X\ n| < \text{Max } (\text{abs } 'X' \{..k\}) + 1$  .
  qed
qed
qed

lemma cauchy-mult [simp]:
  assumes  $X$ : cauchy  $X$  and  $Y$ : cauchy  $Y$ 
  shows cauchy  $(\lambda n. X\ n * Y\ n)$ 
proof (rule cauchyI)
  fix  $r :: \text{rat}$  assume  $0 < r$ 
  then obtain  $u\ v$  where  $u: 0 < u$  and  $v: 0 < v$  and  $r = u + v$ 
    by (rule obtain-pos-sum)
  obtain  $a$  where  $a: 0 < a \ \forall n. |X\ n| < a$ 
    using cauchy-imp-bounded [OF  $X$ ] by fast
  obtain  $b$  where  $b: 0 < b \ \forall n. |Y\ n| < b$ 
    using cauchy-imp-bounded [OF  $Y$ ] by fast
  obtain  $s\ t$  where  $s: 0 < s$  and  $t: 0 < t$  and  $r: r = a * t + s * b$ 
  proof
    show  $0 < v/b$  using  $v\ b(1)$  by (rule divide-pos-pos)
    show  $0 < u/a$  using  $u\ a(1)$  by (rule divide-pos-pos)
    show  $r = a * (u/a) + (v/b) * b$ 
      using  $a(1)\ b(1)\ (r = u + v)$  by simp
  qed
  obtain  $i$  where  $i: \forall m \geq i. \forall n \geq i. |X\ m - X\ n| < s$ 
    using cauchyD [OF  $X\ s$ ] ..
  obtain  $j$  where  $j: \forall m \geq j. \forall n \geq j. |Y\ m - Y\ n| < t$ 
    using cauchyD [OF  $Y\ t$ ] ..
  have  $\forall m \geq \max i\ j. \forall n \geq \max i\ j. |X\ m * Y\ m - X\ n * Y\ n| < r$ 
  proof (clarsimp)
    fix  $m\ n$  assume  $*: i \leq m\ j \leq m\ i \leq n\ j \leq n$ 
    have  $|X\ m * Y\ m - X\ n * Y\ n| = |X\ m * (Y\ m - Y\ n) + (X\ m - X\ n) * Y\ n|$ 
      by (rule abs-triangle-ineq)
    also have  $\dots \leq |X\ m| * |Y\ m - Y\ n| + |X\ m - X\ n| * |Y\ n|$ 
      by (rule abs-mult) ..
    also have  $\dots < a * t + s * b$ 
      by (simp-all add: add-strict-mono mult-strict-mono'  $a\ b\ i\ j\ *$ )
  qed

```

finally show  $|X\ m * Y\ m - X\ n * Y\ n| < r$  **unfolding**  $r$  .  
**qed**  
 thus  $\exists k. \forall m \geq k. \forall n \geq k. |X\ m * Y\ m - X\ n * Y\ n| < r$  ..  
**qed**

**lemma** *cauchy-not-vanishes-cases*:

assumes  $X$ : *cauchy*  $X$   
 assumes  $nz$ :  $\neg$  *vanishes*  $X$   
 shows  $\exists b > 0. \exists k. (\forall n \geq k. b < -X\ n) \vee (\forall n \geq k. b < X\ n)$   
**proof** –  
 obtain  $r$  where  $0 < r$  and  $r$ :  $\forall k. \exists n \geq k. r \leq |X\ n|$   
 using  $nz$  **unfolding** *vanishes-def* **by** (*auto simp add: not-less*)  
 obtain  $s\ t$  where  $s$ :  $0 < s$  and  $t$ :  $0 < t$  and  $r = s + t$   
 using  $\langle 0 < r \rangle$  **by** (*rule obtain-pos-sum*)  
 obtain  $i$  where  $i$ :  $\forall m \geq i. \forall n \geq i. |X\ m - X\ n| < s$   
 using *cauchyD* [*OF*  $X\ s$ ] ..  
 obtain  $k$  where  $i \leq k$  and  $r \leq |X\ k|$   
 using  $r$  **by** *fast*  
 have  $k$ :  $\forall n \geq k. |X\ n - X\ k| < s$   
 using  $i\ \langle i \leq k \rangle$  **by** *auto*  
 have  $X\ k \leq -r \vee r \leq X\ k$   
 using  $\langle r \leq |X\ k| \rangle$  **by** *auto*  
 hence  $(\forall n \geq k. t < -X\ n) \vee (\forall n \geq k. t < X\ n)$   
**unfolding**  $\langle r = s + t \rangle$  **using**  $k$  **by** *auto*  
 hence  $\exists k. (\forall n \geq k. t < -X\ n) \vee (\forall n \geq k. t < X\ n)$  ..  
 thus  $\exists t > 0. \exists k. (\forall n \geq k. t < -X\ n) \vee (\forall n \geq k. t < X\ n)$   
 using  $t$  **by** *auto*  
**qed**

**lemma** *cauchy-not-vanishes*:

assumes  $X$ : *cauchy*  $X$   
 assumes  $nz$ :  $\neg$  *vanishes*  $X$   
 shows  $\exists b > 0. \exists k. \forall n \geq k. b < |X\ n|$   
**using** *cauchy-not-vanishes-cases* [*OF* *assms*]  
**by** *clarify* (*rule exI, erule conjI, rule-tac x=k in exI, auto*)

**lemma** *cauchy-inverse* [*simp*]:

assumes  $X$ : *cauchy*  $X$   
 assumes  $nz$ :  $\neg$  *vanishes*  $X$   
 shows *cauchy*  $(\lambda n. \text{inverse } (X\ n))$   
**proof** (*rule cauchyI*)  
 fix  $r :: \text{rat}$  **assume**  $0 < r$   
 obtain  $b\ i$  where  $b$ :  $0 < b$  and  $i$ :  $\forall n \geq i. b < |X\ n|$   
 using *cauchy-not-vanishes* [*OF*  $X\ nz$ ] **by** *fast*  
 from  $b\ i$  **have**  $nz$ :  $\forall n \geq i. X\ n \neq 0$  **by** *auto*  
 obtain  $s$  where  $s$ :  $0 < s$  and  $r$ :  $r = \text{inverse } b * s * \text{inverse } b$   
**proof**  
 show  $0 < b * r * b$   
**by** (*simp add: \langle 0 < r \rangle b mult-pos-pos*)

```

  show  $r = \text{inverse } b * (b * r * b) * \text{inverse } b$ 
  using  $b$  by simp
qed
obtain  $j$  where  $j: \forall m \geq j. \forall n \geq j. |X\ m - X\ n| < s$ 
  using cauchyD [OF X s] ..
have  $\forall m \geq \max i\ j. \forall n \geq \max i\ j. |\text{inverse } (X\ m) - \text{inverse } (X\ n)| < r$ 
proof (clarsimp)
  fix  $m\ n$  assume  $*: i \leq m\ j \leq m\ i \leq n\ j \leq n$ 
  have  $|\text{inverse } (X\ m) - \text{inverse } (X\ n)| =$ 
     $\text{inverse } |X\ m| * |X\ m - X\ n| * \text{inverse } |X\ n|$ 
  by (simp add: inverse-diff-inverse nz * abs-mult)
  also have  $\dots < \text{inverse } b * s * \text{inverse } b$ 
  by (simp add: mult-strict-mono less-imp-inverse-less
     $\text{mult-pos-pos } i\ j\ b * s$ )
  finally show  $|\text{inverse } (X\ m) - \text{inverse } (X\ n)| < r$  unfolding  $r$  .
qed
thus  $\exists k. \forall m \geq k. \forall n \geq k. |\text{inverse } (X\ m) - \text{inverse } (X\ n)| < r$  ..
qed

```

## 66.4 Equivalence relation on Cauchy sequences

**definition**

$\text{realrel} :: ((\text{nat} \Rightarrow \text{rat}) \times (\text{nat} \Rightarrow \text{rat})) \text{ set}$

**where**

$\text{realrel} = \{(X, Y). \text{cauchy } X \wedge \text{cauchy } Y \wedge \text{vanishes } (\lambda n. X\ n - Y\ n)\}$

**lemma** *refl-realrel*:  $\text{refl-on } \{X. \text{cauchy } X\} \text{ realrel}$

**unfolding** *realrel-def* **by** (*rule refl-onI, clarsimp, simp*)

**lemma** *sym-realrel*:  $\text{sym realrel}$

**unfolding** *realrel-def*

**by** (*rule symI, clarify, drule vanishes-minus, simp*)

**lemma** *trans-realrel*:  $\text{trans realrel}$

**unfolding** *realrel-def*

**apply** (*rule transI, clarify*)

**apply** (*drule (1) vanishes-add*)

**apply** (*simp add: algebra-simps*)

**done**

**lemma** *equiv-realrel*:  $\text{equiv } \{X. \text{cauchy } X\} \text{ realrel}$

**using** *refl-realrel sym-realrel trans-realrel*

**by** (*rule equiv.intro*)

## 66.5 The field of real numbers

**typedef** (**open**)  $\text{real} = \{X. \text{cauchy } X\} // \text{realrel}$

**by** (*fast intro: quotientI cauchy-const*)

**definition**

$Real :: (nat \Rightarrow rat) \Rightarrow real$   
**where**  
 $Real\ X = Abs\text{-}real\ (realrel\ \{\{X\}\})$

**definition**

$real\text{-}case :: ((nat \Rightarrow rat) \Rightarrow 'a) \Rightarrow real \Rightarrow 'a$   
**where**  
 $real\text{-}case\ f\ x = (THE\ y. \forall X \in Rep\text{-}real\ x. y = f\ X)$

**lemma** *Real-induct* [induct type: real]:  
 $(\bigwedge X. cauchy\ X \Longrightarrow P\ (Real\ X)) \Longrightarrow P\ x$   
**unfolding** *Real-def*  
**apply** (*induct*  $x$ )  
**apply** (*erule* *quotientE*)  
**apply** (*simp*)  
**done**

**lemma** *real-case-1*:  
**assumes**  $f$ : *congruent* *realrel*  $f$   
**assumes**  $X$ : *cauchy*  $X$   
**shows**  $real\text{-}case\ f\ (Real\ X) = f\ X$   
**unfolding** *real-case-def* *Real-def*  
**apply** (*subst* *Abs-real-inverse*)  
**apply** (*simp* *add*: *quotientI*  $X$ )  
**apply** (*rule* *the-equality*)  
**apply** *clarsimp*  
**apply** (*erule* *congruent.congruent* [*OF*  $f$ ])  
**apply** (*erule* *bspec*)  
**apply** *simp*  
**apply** (*rule* *refl-onD* [*OF* *refl-realrel*])  
**apply** (*simp* *add*:  $X$ )  
**done**

**lemma** *real-case-2*:  
**assumes**  $f$ : *congruent2* *realrel* *realrel*  $f$   
**assumes**  $X$ : *cauchy*  $X$  **and**  $Y$ : *cauchy*  $Y$   
**shows**  $real\text{-}case\ (\lambda X. real\text{-}case\ (\lambda Y. f\ X\ Y)\ (Real\ Y))\ (Real\ X) = f\ X\ Y$   
**apply** (*subst* *real-case-1* [*OF* -  $X$ ])  
**apply** (*rule* *congruent.intro*)  
**apply** (*subst* *real-case-1* [*OF* -  $Y$ ])  
**apply** (*rule* *congruent2-implies-congruent* [*OF* *equiv-realrel*  $f$ ])  
**apply** (*simp* *add*: *realrel-def*)  
**apply** (*subst* *real-case-1* [*OF* -  $Y$ ])  
**apply** (*rule* *congruent2-implies-congruent* [*OF* *equiv-realrel*  $f$ ])  
**apply** (*simp* *add*: *realrel-def*)  
**apply** (*erule* *congruent2.congruent2* [*OF*  $f$ ])  
**apply** (*rule* *refl-onD* [*OF* *refl-realrel*])  
**apply** (*simp* *add*:  $Y$ )  
**apply** (*rule* *real-case-1* [*OF* -  $Y$ ])

```

apply (rule congruent2-implies-congruent [OF equiv-realrel f])
apply (simp add: X)
done

```

**lemma** *eq-Real*:

```

  cauchy X  $\implies$  cauchy Y  $\implies$  Real X = Real Y  $\longleftrightarrow$  vanishes ( $\lambda n. X\ n - Y\ n$ )
unfolding Real-def
apply (subst Abs-real-inject)
apply (simp add: quotientI)
apply (simp add: quotientI)
apply (simp add: eq-equiv-class-iff [OF equiv-realrel])
apply (simp add: realrel-def)
done

```

**lemma** *add-respects2-realrel*:

```

  ( $\lambda X\ Y. \text{Real } (\lambda n. X\ n + Y\ n)$ ) respects2 realrel
proof (rule congruent2-commuteI [OF equiv-realrel, unfolded mem-Collect-eq])
  fix X Y show Real ( $\lambda n. X\ n + Y\ n$ ) = Real ( $\lambda n. Y\ n + X\ n$ )
    by (simp add: add-commute)
next
  fix X assume X: cauchy X
  fix Y Z assume (Y, Z)  $\in$  realrel
  hence Y: cauchy Y and Z: cauchy Z and YZ: vanishes ( $\lambda n. Y\ n - Z\ n$ )
    unfolding realrel-def by simp-all
  show Real ( $\lambda n. X\ n + Y\ n$ ) = Real ( $\lambda n. X\ n + Z\ n$ )
  proof (rule eq-Real [THEN iffD2])
    show cauchy ( $\lambda n. X\ n + Y\ n$ ) using X Y by (rule cauchy-add)
    show cauchy ( $\lambda n. X\ n + Z\ n$ ) using X Z by (rule cauchy-add)
    show vanishes ( $\lambda n. (X\ n + Y\ n) - (X\ n + Z\ n)$ )
      unfolding add-diff-add using YZ by simp
  qed
qed

```

**lemma** *minus-respects-realrel*:

```

  ( $\lambda X. \text{Real } (\lambda n. - X\ n)$ ) respects realrel
proof (rule congruent.intro)
  fix X Y assume (X, Y)  $\in$  realrel
  hence X: cauchy X and Y: cauchy Y and XY: vanishes ( $\lambda n. X\ n - Y\ n$ )
    unfolding realrel-def by simp-all
  show Real ( $\lambda n. - X\ n$ ) = Real ( $\lambda n. - Y\ n$ )
  proof (rule eq-Real [THEN iffD2])
    show cauchy ( $\lambda n. - X\ n$ ) using X by (rule cauchy-minus)
    show cauchy ( $\lambda n. - Y\ n$ ) using Y by (rule cauchy-minus)
    show vanishes ( $\lambda n. (- X\ n) - (- Y\ n)$ )
      unfolding minus-diff-minus using XY by (rule vanishes-minus)
  qed
qed

```

**lemma** *mult-respects2-realrel*:



```

( $\lambda X Y. \text{Real } (\lambda n. X n * Y n)$ ) respects2 realrel
proof (rule congruent2-commuteI [OF equiv-realrel, unfolded mem-Collect-eq])
  fix X Y
  show Real ( $\lambda n. X n * Y n$ ) = Real ( $\lambda n. Y n * X n$ )
    by (simp add: mult-commute)
next
  fix X assume X: cauchy X
  fix Y Z assume (Y, Z)  $\in$  realrel
  hence Y: cauchy Y and Z: cauchy Z and YZ: vanishes ( $\lambda n. Y n - Z n$ )
    unfolding realrel-def by simp-all
  show Real ( $\lambda n. X n * Y n$ ) = Real ( $\lambda n. X n * Z n$ )
proof (rule eq-Real [THEN iffD2])
  show cauchy ( $\lambda n. X n * Y n$ ) using X Y by (rule cauchy-mult)
  show cauchy ( $\lambda n. X n * Z n$ ) using X Z by (rule cauchy-mult)
  have vanishes ( $\lambda n. X n * (Y n - Z n)$ )
    by (intro vanishes-mult-bounded cauchy-imp-bounded X YZ)
  thus vanishes ( $\lambda n. X n * Y n - X n * Z n$ )
    by (simp add: right-diff-distrib)
qed
qed

lemma vanishes-diff-inverse:
  assumes X: cauchy X  $\neg$  vanishes X
  assumes Y: cauchy Y  $\neg$  vanishes Y
  assumes XY: vanishes ( $\lambda n. X n - Y n$ )
  shows vanishes ( $\lambda n. \text{inverse } (X n) - \text{inverse } (Y n)$ )
proof (rule vanishesI)
  fix r :: rat assume r:  $0 < r$ 
  obtain a i where a:  $0 < a$  and i:  $\forall n \geq i. a < |X n|$ 
    using cauchy-not-vanishes [OF X] by fast
  obtain b j where b:  $0 < b$  and j:  $\forall n \geq j. b < |Y n|$ 
    using cauchy-not-vanishes [OF Y] by fast
  obtain s where s:  $0 < s$  and inverse a * s * inverse b = r
proof
  show  $0 < a * r * b$ 
    using a r b by (simp add: mult-pos-pos)
  show inverse a * (a * r * b) * inverse b = r
    using a r b by simp
qed
  obtain k where k:  $\forall n \geq k. |X n - Y n| < s$ 
    using vanishesD [OF XY s] ..
  have  $\forall n \geq \max i j. k. |\text{inverse } (X n) - \text{inverse } (Y n)| < r$ 
proof (clarsimp)
  fix n assume n:  $i \leq n \wedge j \leq n \wedge k \leq n$ 
  have  $X n \neq 0$  and  $Y n \neq 0$ 
    using i j a b n by auto
  hence  $|\text{inverse } (X n) - \text{inverse } (Y n)| =$ 
     $\text{inverse } |X n| * |X n - Y n| * \text{inverse } |Y n|$ 
    by (simp add: inverse-diff-inverse abs-mult)

```

```

also have ... < inverse a * s * inverse b
apply (intro mult-strict-mono' less-imp-inverse-less)
apply (simp-all add: a b i j k n mult-nonneg-nonneg)
done
also note (inverse a * s * inverse b = r)
finally show |inverse (X n) - inverse (Y n)| < r .
qed
thus  $\exists k. \forall n \geq k. |\text{inverse } (X\ n) - \text{inverse } (Y\ n)| < r ..$ 
qed

```

**lemma** *inverse-respects-realrel*:

```

( $\lambda X. \text{if vanishes } X \text{ then } c \text{ else } \text{Real } (\lambda n. \text{inverse } (X\ n))$ ) respects realrel
(is ?inv respects realrel)
proof (rule congruent.intro)
fix X Y assume (X, Y)  $\in$  realrel
hence X: cauchy X and Y: cauchy Y and XY: vanishes ( $\lambda n. X\ n - Y\ n$ )
unfolding realrel-def by simp-all
have vanishes X  $\longleftrightarrow$  vanishes Y
proof
assume vanishes X
from vanishes-diff [OF this XY] show vanishes Y by simp
next
assume vanishes Y
from vanishes-add [OF this XY] show vanishes X by simp
qed
thus ?inv X = ?inv Y
by (simp add: vanishes-diff-inverse eq-Real X Y XY)
qed

```

**instantiation** *real* :: *field-inverse-zero*  
**begin**

**definition**  
 $0 = \text{Real } (\lambda n. 0)$

**definition**  
 $1 = \text{Real } (\lambda n. 1)$

**definition**  
 $x + y = \text{real-case } (\lambda X. \text{real-case } (\lambda Y. \text{Real } (\lambda n. X\ n + Y\ n)))\ y)\ x$

**definition**  
 $- x = \text{real-case } (\lambda X. \text{Real } (\lambda n. - X\ n))\ x$

**definition**  
 $x - y = (x :: \text{real}) + - y$

**definition**  
 $x * y = \text{real-case } (\lambda X. \text{real-case } (\lambda Y. \text{Real } (\lambda n. X\ n * Y\ n)))\ y)\ x$

**definition**

*inverse* =  
*real-case* ( $\lambda X$ . if vanishes  $X$  then 0 else *Real* ( $\lambda n$ . *inverse* ( $X\ n$ )))

**definition**

$x / y = (x :: \text{real}) * \text{inverse } y$

**lemma** *add-Real*:

**assumes**  $X$ : *cauchy*  $X$  **and**  $Y$ : *cauchy*  $Y$   
**shows** *Real*  $X + \text{Real } Y = \text{Real } (\lambda n$ .  $X\ n + Y\ n$ )  
**unfolding** *plus-real-def*  
**by** (*rule real-case-2* [*OF add-respects2-realrel*  $X\ Y$ ])

**lemma** *minus-Real*:

**assumes**  $X$ : *cauchy*  $X$   
**shows**  $-\text{Real } X = \text{Real } (\lambda n$ .  $-X\ n$ )  
**unfolding** *uminus-real-def*  
**by** (*rule real-case-1* [*OF minus-respects-realrel*  $X$ ])

**lemma** *diff-Real*:

**assumes**  $X$ : *cauchy*  $X$  **and**  $Y$ : *cauchy*  $Y$   
**shows** *Real*  $X - \text{Real } Y = \text{Real } (\lambda n$ .  $X\ n - Y\ n$ )  
**unfolding** *minus-real-def* *diff-minus*  
**by** (*simp add: minus-Real add-Real*  $X\ Y$ )

**lemma** *mult-Real*:

**assumes**  $X$ : *cauchy*  $X$  **and**  $Y$ : *cauchy*  $Y$   
**shows** *Real*  $X * \text{Real } Y = \text{Real } (\lambda n$ .  $X\ n * Y\ n$ )  
**unfolding** *times-real-def*  
**by** (*rule real-case-2* [*OF mult-respects2-realrel*  $X\ Y$ ])

**lemma** *inverse-Real*:

**assumes**  $X$ : *cauchy*  $X$   
**shows** *inverse* (*Real*  $X$ ) =  
 (*if* vanishes  $X$  then 0 else *Real* ( $\lambda n$ . *inverse* ( $X\ n$ )))  
**unfolding** *inverse-real-def*  
**by** (*rule real-case-1* [*OF inverse-respects-realrel*  $X$ ])

**instance** *proof*

**fix**  $a\ b\ c :: \text{real}$   
**show**  $a + b = b + a$   
**by** (*induct*  $a$ , *induct*  $b$ ) (*simp add: add-Real add-ac*)  
**show**  $(a + b) + c = a + (b + c)$   
**by** (*induct*  $a$ , *induct*  $b$ , *induct*  $c$ ) (*simp add: add-Real add-ac*)  
**show**  $0 + a = a$   
**unfolding** *zero-real-def*  
**by** (*induct*  $a$ ) (*simp add: add-Real*)  
**show**  $-a + a = 0$

```

    unfolding zero-real-def
    by (induct a) (simp add: minus-Real add-Real)
show  $a - b = a + - b$ 
    by (rule minus-real-def)
show  $(a * b) * c = a * (b * c)$ 
    by (induct a, induct b, induct c) (simp add: mult-Real mult-ac)
show  $a * b = b * a$ 
    by (induct a, induct b) (simp add: mult-Real mult-ac)
show  $1 * a = a$ 
    unfolding one-real-def
    by (induct a) (simp add: mult-Real)
show  $(a + b) * c = a * c + b * c$ 
    by (induct a, induct b, induct c)
    (simp add: mult-Real add-Real algebra-simps)
show  $(0::real) \neq (1::real)$ 
    unfolding zero-real-def one-real-def
    by (simp add: eq-Real)
show  $a \neq 0 \implies \text{inverse } a * a = 1$ 
    unfolding zero-real-def one-real-def
    apply (induct a)
    apply (simp add: eq-Real inverse-Real mult-Real)
    apply (rule vanishesI)
    apply (frule (1) cauchy-not-vanishes, clarify)
    apply (rule-tac x=k in exI, clarify)
    apply (drule-tac x=n in spec, simp)
    done
show  $a / b = a * \text{inverse } b$ 
    by (rule divide-real-def)
show  $\text{inverse } (0::real) = 0$ 
    by (simp add: zero-real-def inverse-Real)
qed

end

```

## 66.6 Positive reals

### definition

$\text{positive} :: \text{real} \Rightarrow \text{bool}$

### where

$\text{positive} = \text{real-case } (\lambda X. \exists r > 0. \exists k. \forall n \geq k. r < X \ n)$

### lemma bool-congruentI:

**assumes**  $\text{sym}: \text{sym } r$

**assumes**  $P: \bigwedge x \ y. (x, y) \in r \implies P \ x \implies P \ y$

**shows**  $P$  respects  $r$

**apply** (rule congruent.intro)

**apply** (rule iffI)

**apply** (erule (1) P)

**apply** (erule (1) P [OF symD [OF sym]])

done

**lemma** *positive-respects-realrel*:

$(\lambda X. \exists r > 0. \exists k. \forall n \geq k. r < X\ n)$  respects realrel

**proof** (rule bool-congruentI)

show sym realrel by (rule sym-realrel)

next

fix  $X\ Y$  assume  $(X, Y) \in \text{realrel}$

hence  $XY$ : vanishes  $(\lambda n. X\ n - Y\ n)$

unfolding realrel-def by simp-all

assume  $\exists r > 0. \exists k. \forall n \geq k. r < X\ n$

then obtain  $r\ i$  where  $0 < r$  and  $i: \forall n \geq i. r < X\ n$

by fast

obtain  $s\ t$  where  $s: 0 < s$  and  $t: 0 < t$  and  $r: r = s + t$

using  $\langle 0 < r \rangle$  by (rule obtain-pos-sum)

obtain  $j$  where  $j: \forall n \geq j. |X\ n - Y\ n| < s$

using vanishesD [OF XY s] ..

have  $\forall n \geq \max i\ j. t < Y\ n$

**proof** (clarsimp)

fix  $n$  assume  $n: i \leq n\ j \leq n$

have  $|X\ n - Y\ n| < s$  and  $r < X\ n$

using  $i\ j\ n$  by simp-all

thus  $t < Y\ n$  unfolding  $r$  by simp

qed

thus  $\exists r > 0. \exists k. \forall n \geq k. r < Y\ n$  using  $t$  by fast

qed

**lemma** *positive-Real*:

assumes  $X$ : *cauchy*  $X$

shows *positive*  $(\text{Real } X) \longleftrightarrow (\exists r > 0. \exists k. \forall n \geq k. r < X\ n)$

unfolding *positive-def*

by (rule real-case-1 [OF *positive-respects-realrel*  $X$ ])

**lemma** *positive-zero*:  $\neg$  *positive* 0

unfolding *zero-real-def* by (auto simp add: *positive-Real*)

**lemma** *positive-add*:

*positive*  $x \implies$  *positive*  $y \implies$  *positive*  $(x + y)$

**apply** (induct  $x$ , induct  $y$ , rename-tac  $Y\ X$ )

**apply** (simp add: *add-Real* *positive-Real*)

**apply** (clarify, rename-tac  $a\ b\ i\ j$ )

**apply** (rule-tac  $x = a + b$  in  $exI$ , simp)

**apply** (rule-tac  $x = \max i\ j$  in  $exI$ , clarsimp)

**apply** (simp add: *add-strict-mono*)

done

**lemma** *positive-mult*:

*positive*  $x \implies$  *positive*  $y \implies$  *positive*  $(x * y)$

**apply** (induct  $x$ , induct  $y$ , rename-tac  $Y\ X$ )

```

apply (simp add: mult-Real positive-Real)
apply (clarify, rename-tac a b i j)
apply (rule-tac x=a * b in exI, simp add: mult-pos-pos)
apply (rule-tac x=max i j in exI, clarsimp)
apply (rule mult-strict-mono, auto)
done

lemma positive-minus:
   $\neg \text{positive } x \implies x \neq 0 \implies \text{positive } (-x)$ 
apply (induct x, rename-tac X)
apply (simp add: zero-real-def eq-Real minus-Real positive-Real)
apply (drule (1) cauchy-not-vanishes-cases, safe, fast, fast)
done

instantiation real :: linordered-field-inverse-zero
begin

definition
   $x < y \iff \text{positive } (y - x)$ 

definition
   $x \leq (y::\text{real}) \iff x < y \vee x = y$ 

definition
   $\text{abs } (a::\text{real}) = (\text{if } a < 0 \text{ then } -a \text{ else } a)$ 

definition
   $\text{sgn } (a::\text{real}) = (\text{if } a = 0 \text{ then } 0 \text{ else if } 0 < a \text{ then } 1 \text{ else } -1)$ 

instance proof
  fix a b c :: real
  show |a| = (if a < 0 then -a else a)
    by (rule abs-real-def)
  show a < b  $\iff$  a  $\leq$  b  $\wedge$   $\neg$  b  $\leq$  a
    unfolding less-eq-real-def less-real-def
    by (auto, drule (1) positive-add, simp add: positive-zero)
  show a  $\leq$  a
    unfolding less-eq-real-def by simp
  show a  $\leq$  b  $\implies$  b  $\leq$  c  $\implies$  a  $\leq$  c
    unfolding less-eq-real-def less-real-def
    by (auto, drule (1) positive-add, simp add: algebra-simps)
  show a  $\leq$  b  $\implies$  b  $\leq$  a  $\implies$  a = b
    unfolding less-eq-real-def less-real-def
    by (auto, drule (1) positive-add, simp add: positive-zero)
  show a  $\leq$  b  $\implies$  c + a  $\leq$  c + b
    unfolding less-eq-real-def less-real-def by auto
  show sgn a = (if a = 0 then 0 else if 0 < a then 1 else -1)
    by (rule sgn-real-def)
  show a  $\leq$  b  $\vee$  b  $\leq$  a

```

```

    unfolding less-eq-real-def less-real-def
    by (auto dest!: positive-minus)
  show  $a < b \implies 0 < c \implies c * a < c * b$ 
    unfolding less-real-def
    by (drule (1) positive-mult, simp add: algebra-simps)
qed

end

instantiation real :: distrib-lattice
begin

definition
  ( $inf :: real \Rightarrow real \Rightarrow real$ ) = min

definition
  ( $sup :: real \Rightarrow real \Rightarrow real$ ) = max

instance proof
qed (auto simp add: inf-real-def sup-real-def min-max.sup-inf-distrib1)

end

instantiation real :: number-ring
begin

definition
  ( $number-of\ x :: real$ ) = of-int x

instance proof
qed (rule number-of-real-def)

end

lemma of-nat-Real:  $of-nat\ x = Real\ (\lambda n. of-nat\ x)$ 
apply (induct x)
apply (simp add: zero-real-def)
apply (simp add: one-real-def add-Real)
done

lemma of-int-Real:  $of-int\ x = Real\ (\lambda n. of-int\ x)$ 
apply (cases x rule: int-diff-cases)
apply (simp add: of-nat-Real diff-Real)
done

lemma of-rat-Real:  $of-rat\ x = Real\ (\lambda n. x)$ 
apply (induct x)
apply (simp add: Fract-of-int-quotient of-rat-divide)
apply (simp add: of-int-Real divide-inverse)

```

```

apply (simp add: inverse-Real mult-Real)
done

```

```

instance real :: archimedean-field

```

```

proof

```

```

  fix x :: real

```

```

  show  $\exists z. x \leq \text{of-int } z$ 

```

```

    apply (induct x)

```

```

    apply (frule cauchy-imp-bounded, clarify)

```

```

    apply (rule-tac x=ceiling b + 1 in exI)

```

```

    apply (rule less-imp-le)

```

```

    apply (simp add: of-int-Real less-real-def diff-Real positive-Real)

```

```

    apply (rule-tac x=1 in exI, simp add: algebra-simps)

```

```

    apply (rule-tac x=0 in exI, clarsimp)

```

```

    apply (rule le-less-trans [OF abs-ge-self])

```

```

    apply (rule less-le-trans [OF - le-of-int-ceiling])

```

```

    apply simp

```

```

  done

```

```

qed

```

## 66.7 Completeness

```

lemma not-positive-Real:

```

```

  assumes X: cauchy X

```

```

  shows  $\neg \text{positive (Real } X) \longleftrightarrow (\forall r>0. \exists k. \forall n\geq k. X\ n \leq r)$ 

```

```

unfolding positive-Real [OF X]

```

```

apply (auto, unfold not-less)

```

```

apply (erule obtain-pos-sum)

```

```

apply (drule-tac x=s in spec, simp)

```

```

apply (drule-tac r=t in cauchyD [OF X], clarify)

```

```

apply (drule-tac x=k in spec, clarsimp)

```

```

apply (rule-tac x=n in exI, clarify, rename-tac m)

```

```

apply (drule-tac x=m in spec, simp)

```

```

apply (drule-tac x=n in spec, simp)

```

```

apply (drule spec, drule (1) mp, clarify, rename-tac i)

```

```

apply (rule-tac x=max i k in exI, simp)

```

```

done

```

```

lemma le-Real:

```

```

  assumes X: cauchy X and Y: cauchy Y

```

```

  shows  $\text{Real } X \leq \text{Real } Y = (\forall r>0. \exists k. \forall n\geq k. X\ n \leq Y\ n + r)$ 

```

```

unfolding not-less [symmetric, where 'a=real] less-real-def

```

```

apply (simp add: diff-Real not-positive-Real X Y)

```

```

apply (simp add: diff-le-eq add-ac)

```

```

done

```

```

lemma le-RealI:

```

```

  assumes Y: cauchy Y

```

```

  shows  $\forall n. x \leq \text{of-rat } (Y\ n) \implies x \leq \text{Real } Y$ 

```



**proof** (*induct x*)  
**fix**  $X$  **assume**  $X$ : *cauchy*  $X$  **and**  $\forall n. \text{Real } X \leq \text{of-rat } (Y\ n)$   
**hence**  $le$ :  $\bigwedge m\ r. 0 < r \implies \exists k. \forall n \geq k. X\ n \leq Y\ m + r$   
**by** (*simp add: of-rat-Real le-Real*)  
{  
**fix**  $r :: \text{rat}$  **assume**  $0 < r$   
**then obtain**  $s\ t$  **where**  $s$ :  $0 < s$  **and**  $t$ :  $0 < t$  **and**  $r$ :  $r = s + t$   
**by** (*rule obtain-pos-sum*)  
**obtain**  $i$  **where**  $i$ :  $\forall m \geq i. \forall n \geq i. |Y\ m - Y\ n| < s$   
**using** *cauchyD [OF Y s]* ..  
**obtain**  $j$  **where**  $j$ :  $\forall n \geq j. X\ n \leq Y\ i + t$   
**using**  $le$  [*OF t*] ..  
**have**  $\forall n \geq \max\ i\ j. X\ n \leq Y\ n + r$   
**proof** (*clarsimp*)  
**fix**  $n$  **assume**  $n$ :  $i \leq n\ j \leq n$   
**have**  $X\ n \leq Y\ i + t$  **using**  $n\ j$  **by** *simp*  
**moreover** **have**  $|Y\ i - Y\ n| < s$  **using**  $n\ i$  **by** *simp*  
**ultimately show**  $X\ n \leq Y\ n + r$  **unfolding**  $r$  **by** *simp*  
**qed**  
**hence**  $\exists k. \forall n \geq k. X\ n \leq Y\ n + r$  ..  
}  
**thus**  $\text{Real } X \leq \text{Real } Y$   
**by** (*simp add: of-rat-Real le-Real X Y*)  
**qed**

**lemma** *Real-leI*:  
**assumes**  $X$ : *cauchy*  $X$   
**assumes**  $le$ :  $\forall n. \text{of-rat } (X\ n) \leq y$   
**shows**  $\text{Real } X \leq y$   
**proof** –  
**have**  $-y \leq -\text{Real } X$   
**by** (*simp add: minus-Real X le-RealI of-rat-minus le*)  
**thus** *?thesis* **by** *simp*  
**qed**

**lemma** *less-RealD*:  
**assumes**  $Y$ : *cauchy*  $Y$   
**shows**  $x < \text{Real } Y \implies \exists n. x < \text{of-rat } (Y\ n)$   
**by** (*erule contrapos-pp, simp add: not-less, erule Real-leI [OF Y]*)

**lemma** *of-nat-less-two-power*:  
 $\text{of-nat } n < (2 :: 'a :: \{\text{linordered-idom, number-ring}\}) ^ n$   
**apply** (*induct n*)  
**apply** *simp*  
**apply** (*subgoal-tac*  $(1 :: 'a) \leq 2 ^ n$ )  
**apply** (*drule* (1) *add-le-less-mono, simp*)  
**apply** *simp*  
**done**

**lemma** *complete-real*:

**fixes**  $S :: \text{real set}$

**assumes**  $\exists x. x \in S$  **and**  $\exists z. \forall x \in S. x \leq z$

**shows**  $\exists y. (\forall x \in S. x \leq y) \wedge (\forall z. (\forall x \in S. x \leq z) \longrightarrow y \leq z)$

**proof** –

**obtain**  $x$  **where**  $x: x \in S$  **using** *assms(1)* ..

**obtain**  $z$  **where**  $z: \forall x \in S. x \leq z$  **using** *assms(2)* ..

**def**  $P \equiv \lambda x. \forall y \in S. y \leq \text{of-rat } x$

**obtain**  $a$  **where**  $a: \neg P a$

**proof**

**have**  $\text{of-int } (\text{floor } (x - 1)) \leq x - 1$  **by** (*rule of-int-floor-le*)

**also have**  $x - 1 < x$  **by** *simp*

**finally have**  $\text{of-int } (\text{floor } (x - 1)) < x$  .

**hence**  $\neg x \leq \text{of-int } (\text{floor } (x - 1))$  **by** (*simp only: not-le*)

**then show**  $\neg P (\text{of-int } (\text{floor } (x - 1)))$

**unfolding**  $P\text{-def of-rat-of-int-eq}$  **using**  $x$  **by** *fast*

**qed**

**obtain**  $b$  **where**  $b: P b$

**proof**

**show**  $P (\text{of-int } (\text{ceiling } z))$

**unfolding**  $P\text{-def of-rat-of-int-eq}$

**proof**

**fix**  $y$  **assume**  $y \in S$

**hence**  $y \leq z$  **using**  $z$  **by** *simp*

**also have**  $z \leq \text{of-int } (\text{ceiling } z)$  **by** (*rule le-of-int-ceiling*)

**finally show**  $y \leq \text{of-int } (\text{ceiling } z)$  .

**qed**

**qed**

**def**  $\text{avg} \equiv \lambda x y :: \text{rat}. x/2 + y/2$

**def**  $\text{bisect} \equiv \lambda(x, y). \text{if } P (\text{avg } x y) \text{ then } (x, \text{avg } x y) \text{ else } (\text{avg } x y, y)$

**def**  $A \equiv \lambda n. \text{fst } ((\text{bisect} \wedge^n) (a, b))$

**def**  $B \equiv \lambda n. \text{snd } ((\text{bisect} \wedge^n) (a, b))$

**def**  $C \equiv \lambda n. \text{avg } (A n) (B n)$

**have**  $A\ 0$  [*simp*]:  $A\ 0 = a$  **unfolding**  $A\text{-def}$  **by** *simp*

**have**  $B\ 0$  [*simp*]:  $B\ 0 = b$  **unfolding**  $B\text{-def}$  **by** *simp*

**have**  $A\text{-Suc}$  [*simp*]:  $\bigwedge n. A (\text{Suc } n) = (\text{if } P (C n) \text{ then } A n \text{ else } C n)$

**unfolding**  $A\text{-def } B\text{-def } C\text{-def bisect-def split-def}$  **by** *simp*

**have**  $B\text{-Suc}$  [*simp*]:  $\bigwedge n. B (\text{Suc } n) = (\text{if } P (C n) \text{ then } C n \text{ else } B n)$

**unfolding**  $A\text{-def } B\text{-def } C\text{-def bisect-def split-def}$  **by** *simp*

**have**  $\text{width}$ :  $\bigwedge n. B n - A n = (b - a) / 2^n$

**apply** (*simp add: eq-divide-eq*)

**apply** (*induct-tac n, simp*)

**apply** (*simp add: C-def avg-def algebra-simps*)

**done**

**have**  $\text{twos}$ :  $\bigwedge y r :: \text{rat}. 0 < r \implies \exists n. y / 2^n < r$

```

apply (simp add: divide-less-eq)
apply (subst mult-commute)
apply (frule-tac y=y in ex-less-of-nat-mult)
apply clarify
apply (rule-tac x=n in exI)
apply (erule less-trans)
apply (rule mult-strict-right-mono)
apply (rule le-less-trans [OF - of-nat-less-two-power])
apply simp
apply assumption
done

have PA:  $\bigwedge n. \neg P (A\ n)$ 
  by (induct-tac n, simp-all add: a)
have PB:  $\bigwedge n. P (B\ n)$ 
  by (induct-tac n, simp-all add: b)
have ab:  $a < b$ 
  using a b unfolding P-def
  apply (clarsimp simp add: not-le)
  apply (drule (1) bspec)
  apply (drule (1) less-le-trans)
  apply (simp add: of-rat-less)
  done
have AB:  $\bigwedge n. A\ n < B\ n$ 
  by (induct-tac n, simp add: ab, simp add: C-def avg-def)
have A-mono:  $\bigwedge i\ j. i \leq j \implies A\ i \leq A\ j$ 
  apply (auto simp add: le-less [where 'a=nat])
  apply (erule less-Suc-induct)
  apply (clarsimp simp add: C-def avg-def)
  apply (simp add: add-divide-distrib [symmetric])
  apply (rule AB [THEN less-imp-le])
  apply simp
  done
have B-mono:  $\bigwedge i\ j. i \leq j \implies B\ j \leq B\ i$ 
  apply (auto simp add: le-less [where 'a=nat])
  apply (erule less-Suc-induct)
  apply (clarsimp simp add: C-def avg-def)
  apply (simp add: add-divide-distrib [symmetric])
  apply (rule AB [THEN less-imp-le])
  apply simp
  done
have cauchy-lemma:
   $\bigwedge X. \forall n. \forall i \geq n. A\ n \leq X\ i \wedge X\ i \leq B\ n \implies \text{cauchy } X$ 
  apply (rule cauchyI)
  apply (drule twos [where y=b - a])
  apply (erule exE)
  apply (rule-tac x=n in exI, clarify, rename-tac i j)
  apply (rule-tac y=B n - A n in le-less-trans) defer
  apply (simp add: width)

```

```

  apply (drule-tac x=n in spec)
  apply (frule-tac x=i in spec, drule (1) mp)
  apply (frule-tac x=j in spec, drule (1) mp)
  apply (frule A-mono, drule B-mono)
  apply (frule A-mono, drule B-mono)
  apply arith
done
have cauchy A
  apply (rule cauchy-lemma [rule-format])
  apply (simp add: A-mono)
  apply (erule order-trans [OF less-imp-le [OF AB] B-mono])
done
have cauchy B
  apply (rule cauchy-lemma [rule-format])
  apply (simp add: B-mono)
  apply (erule order-trans [OF A-mono less-imp-le [OF AB]])
done
have 1:  $\forall x \in S. x \leq \text{Real } B$ 
proof
  fix x assume x  $\in S$ 
  then show  $x \leq \text{Real } B$ 
    using PB [unfolded P-def] ⟨cauchy B⟩
    by (simp add: le-RealI)
qed
have 2:  $\forall z. (\forall x \in S. x \leq z) \longrightarrow \text{Real } A \leq z$ 
  apply clarify
  apply (erule contrapos-pp)
  apply (simp add: not-le)
  apply (drule less-RealD [OF ⟨cauchy A⟩, clarify])
  apply (subgoal-tac  $\neg P (A \ n)$ )
  apply (simp add: P-def not-le, clarify)
  apply (erule rev-bexI)
  apply (erule (1) less-trans)
  apply (simp add: PA)
done
have vanishes  $(\lambda n. (b - a) / 2 ^ n)$ 
proof (rule vanishesI)
  fix r :: rat assume  $0 < r$ 
  then obtain k where  $|b - a| / 2 ^ k < r$ 
    using twos by fast
  have  $\forall n \geq k. |(b - a) / 2 ^ n| < r$ 
  proof (clarify)
    fix n assume n:  $k \leq n$ 
    have  $|(b - a) / 2 ^ n| = |b - a| / 2 ^ n$ 
      by simp
    also have  $\dots \leq |b - a| / 2 ^ k$ 
      using n by (simp add: divide-left-mono mult-pos-pos)
    also note k
    finally show  $|(b - a) / 2 ^ n| < r$  .
  end
end

```

```

qed
thus  $\exists k. \forall n \geq k. |(b - a) / 2^n| < r ..$ 
qed
hence  $\exists$ : Real B = Real A
  by (simp add: eq-Real <cauchy A> <cauchy B> width)
show  $\exists y. (\forall x \in S. x \leq y) \wedge (\forall z. (\forall x \in S. x \leq z) \longrightarrow y \leq z)$ 
  using 1 2 3 by (rule-tac x=Real B in exI, simp)
qed

```

## 66.8 Hiding implementation details

**hide-const** (**open**) *vanishes cauchy positive Real real-case*

```

declare Real-induct [induct del]
declare Abs-real-induct [induct del]
declare Abs-real-cases [cases del]

```

## 66.9 Legacy theorem names

TODO: Could we have a way to mark theorem names as deprecated, and have Isabelle issue a warning and display the preferred name?

```

lemmas real-diff-def = minus-real-def [of r s, standard]
lemmas real-divide-def = divide-real-def [of R S, standard]
lemmas real-less-def = less-le [of x::real y, standard]
lemmas real-abs-def = abs-real-def [of r, standard]
lemmas real-sgn-def = sgn-real-def [of x, standard]
lemmas real-mult-commute = mult-commute [of z::real w, standard]
lemmas real-mult-assoc = mult-assoc [of z1::real z2 z3, standard]
lemmas real-mult-1 = mult-1-left [of z::real, standard]
lemmas real-add-mult-distrib = left-distrib [of z1::real z2 w, standard]
lemmas real-zero-not-eq-one = zero-neq-one [where 'a=real]
lemmas real-mult-inverse-left = left-inverse [of x::real, standard]
lemmas INVERSE-ZERO = inverse-zero [where 'a=real]
lemmas real-le-refl = order-refl [of w::real, standard]
lemmas real-le-antisym = order-antisym [of z::real w, standard]
lemmas real-le-trans = order-trans [of i::real j k, standard]
lemmas real-le-linear = linear [of z::real w, standard]
lemmas real-le-eq-diff = le-iff-diff-le-0 [of x::real y, standard]
lemmas real-add-left-mono = add-left-mono [of x::real y z, standard]
lemmas real-mult-order = mult-pos-pos [of x::real y, standard]
lemmas real-mult-less-mono2 =
  mult-strict-left-mono [of x::real y z, COMP swap-prems-rl, standard]

```

## 66.10 More Lemmas

BH: These lemmas should not be necessary; they should be covered by existing simp rules and simplification procedures.

**lemma** *real-mult-left-cancel*:  $(c::real) \neq 0 \implies (c*a=c*b) = (a=b)$

by *simp*

**lemma** *real-mult-right-cancel*:  $(c::\text{real}) \neq 0 \implies (a*c=b*c) = (a=b)$   
 by *simp*

**lemma** *real-mult-less-iff1* [*simp*]:  $(0::\text{real}) < z \implies (x*z < y*z) = (x < y)$   
 by *simp*

**lemma** *real-mult-le-cancel-iff1* [*simp*]:  $(0::\text{real}) < z \implies (x*z \leq y*z) = (x \leq y)$   
 by *simp*

**lemma** *real-mult-le-cancel-iff2* [*simp*]:  $(0::\text{real}) < z \implies (z*x \leq z*y) = (x \leq y)$   
 by (rule *mult-le-cancel-left-pos*)

## 66.11 Embedding numbers into the Reals

**abbreviation**

*real-of-nat* :: *nat*  $\Rightarrow$  *real*

**where**

*real-of-nat*  $\equiv$  *of-nat*

**abbreviation**

*real-of-int* :: *int*  $\Rightarrow$  *real*

**where**

*real-of-int*  $\equiv$  *of-int*

**abbreviation**

*real-of-rat* :: *rat*  $\Rightarrow$  *real*

**where**

*real-of-rat*  $\equiv$  *of-rat*

**consts**

*real* :: 'a  $\Rightarrow$  *real*

**defs (overloaded)**

*real-of-nat-def* [*code-unfold*]: *real* == *real-of-nat*

*real-of-int-def* [*code-unfold*]: *real* == *real-of-int*

**lemma** *real-eq-of-nat*: *real* = *of-nat*  
**unfolding** *real-of-nat-def* ..

**lemma** *real-eq-of-int*: *real* = *of-int*  
**unfolding** *real-of-int-def* ..

**lemma** *real-of-int-zero* [*simp*]: *real* (0::*int*) = 0  
 by (*simp add: real-of-int-def*)

**lemma** *real-of-one* [*simp*]: *real* (1::*int*) = (1::*real*)

**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-add* [*simp*]:  $\text{real}(x + y) = \text{real } (x::\text{int}) + \text{real } y$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-minus* [*simp*]:  $\text{real}(-x) = -\text{real } (x::\text{int})$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-diff* [*simp*]:  $\text{real}(x - y) = \text{real } (x::\text{int}) - \text{real } y$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-mult* [*simp*]:  $\text{real}(x * y) = \text{real } (x::\text{int}) * \text{real } y$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-power* [*simp*]:  $\text{real } (x \wedge n) = \text{real } (x::\text{int}) \wedge n$   
**by** (*simp add: real-of-int-def of-int-power*)

**lemmas** *power-real-of-int* = *real-of-int-power* [*symmetric*]

**lemma** *real-of-int-setsum* [*simp*]:  $\text{real } ((\text{SUM } x:A. f x)::\text{int}) = (\text{SUM } x:A. \text{real}(f x))$   
**apply** (*subst real-eq-of-int*) +  
**apply** (*rule of-int-setsum*)  
**done**

**lemma** *real-of-int-setprod* [*simp*]:  $\text{real } ((\text{PROD } x:A. f x)::\text{int}) = (\text{PROD } x:A. \text{real}(f x))$   
**apply** (*subst real-eq-of-int*) +  
**apply** (*rule of-int-setprod*)  
**done**

**lemma** *real-of-int-zero-cancel* [*simp, algebra, presburger*]:  $(\text{real } x = 0) = (x = (0::\text{int}))$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-inject* [*iff, algebra, presburger*]:  $(\text{real } (x::\text{int}) = \text{real } y) = (x = y)$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-less-iff* [*iff, presburger*]:  $(\text{real } (x::\text{int}) < \text{real } y) = (x < y)$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-le-iff* [*simp, presburger*]:  $(\text{real } (x::\text{int}) \leq \text{real } y) = (x \leq y)$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-gt-zero-cancel-iff* [*simp, presburger*]:  $(0 < \text{real } (n::\text{int})) = (0 < n)$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-ge-zero-cancel-iff* [*simp*, *presburger*]:  $(0 \leq \text{real } (n::\text{int})) = (0 \leq n)$

**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-lt-zero-cancel-iff* [*simp*, *presburger*]:  $(\text{real } (n::\text{int}) < 0) = (n < 0)$

**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-le-zero-cancel-iff* [*simp*, *presburger*]:  $(\text{real } (n::\text{int}) \leq 0) = (n \leq 0)$

**by** (*simp add: real-of-int-def*)

**lemma** *real-of-int-abs* [*simp*]:  $\text{real } (\text{abs } x) = \text{abs}(\text{real } (x::\text{int}))$

**by** (*auto simp add: abs-if*)

**lemma** *int-less-real-le*:  $((n::\text{int}) < m) = (\text{real } n + 1 \leq \text{real } m)$

**apply** (*subgoal-tac real n + 1 = real (n + 1)*)

**apply** (*simp del: real-of-int-add*)

**apply** *auto*

**done**

**lemma** *int-le-real-less*:  $((n::\text{int}) \leq m) = (\text{real } n < \text{real } m + 1)$

**apply** (*subgoal-tac real m + 1 = real (m + 1)*)

**apply** (*simp del: real-of-int-add*)

**apply** *simp*

**done**

**lemma** *real-of-int-div-aux*:  $d \sim 0 \implies (\text{real } (x::\text{int})) / (\text{real } d) = \text{real } (x \text{ div } d) + (\text{real } (x \text{ mod } d)) / (\text{real } d)$

**proof** –

**assume**  $d \sim 0$

**have**  $x = (x \text{ div } d) * d + x \text{ mod } d$

**by** *auto*

**then have**  $\text{real } x = \text{real } (x \text{ div } d) * \text{real } d + \text{real } (x \text{ mod } d)$

**by** (*simp only: real-of-int-mult [THEN sym] real-of-int-add [THEN sym]*)

**then have**  $\text{real } x / \text{real } d = \dots / \text{real } d$

**by** *simp*

**then show** *?thesis*

**by** (*auto simp add: add-divide-distrib algebra-simps prems*)

**qed**

**lemma** *real-of-int-div*:  $(d::\text{int}) \sim 0 \implies d \text{ dvd } n \implies$

$\text{real } (n \text{ div } d) = \text{real } n / \text{real } d$

**apply** (*frule real-of-int-div-aux [of d n]*)

**apply** *simp*

**apply** (*simp add: dvd-eq-mod-eq-0*)

**done**

**lemma** *real-of-int-div2*:



```

0 <= real (n::int) / real (x) - real (n div x)
apply (case-tac x = 0)
apply simp
apply (case-tac 0 < x)
apply (simp add: algebra-simps)
apply (subst real-of-int-div-aux)
apply simp
apply simp
apply (subst zero-le-divide-iff)
apply auto
apply (simp add: algebra-simps)
apply (subst real-of-int-div-aux)
apply simp
apply simp
apply (subst zero-le-divide-iff)
apply auto
done

```

```

lemma real-of-int-div3:
  real (n::int) / real (x) - real (n div x) <= 1
  apply (case-tac x = 0)
  apply simp
  apply (simp add: algebra-simps)
  apply (subst real-of-int-div-aux)
  apply assumption
  apply simp
  apply (subst divide-le-eq)
  apply clarsimp
  apply (rule conjI)
  apply (rule impI)
  apply (rule order-less-imp-le)
  apply simp
  apply (rule impI)
  apply (rule order-less-imp-le)
  apply simp
done

```

```

lemma real-of-int-div4: real (n div x) <= real (n::int) / real x
by (insert real-of-int-div2 [of n x], simp)

```

```

lemma Ints-real-of-int [simp]: real (x::int) ∈ Ints
unfolding real-of-int-def by (rule Ints-of-int)

```

## 66.12 Embedding the Naturals into the Reals

```

lemma real-of-nat-zero [simp]: real (0::nat) = 0
by (simp add: real-of-nat-def)

```

```

lemma real-of-nat-1 [simp]: real (1::nat) = 1

```

**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-one* [*simp*]:  $\text{real } (\text{Suc } 0) = (1::\text{real})$   
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-add* [*simp*]:  $\text{real } (m + n) = \text{real } (m::\text{nat}) + \text{real } n$   
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-Suc*:  $\text{real } (\text{Suc } n) = \text{real } n + (1::\text{real})$   
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-less-iff* [*iff*]:  
 $(\text{real } (n::\text{nat}) < \text{real } m) = (n < m)$   
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-le-iff* [*iff*]:  $(\text{real } (n::\text{nat}) \leq \text{real } m) = (n \leq m)$   
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-ge-zero* [*iff*]:  $0 \leq \text{real } (n::\text{nat})$   
**by** (*simp add: real-of-nat-def zero-le-imp-of-nat*)

**lemma** *real-of-nat-Suc-gt-zero*:  $0 < \text{real } (\text{Suc } n)$   
**by** (*simp add: real-of-nat-def del: of-nat-Suc*)

**lemma** *real-of-nat-mult* [*simp*]:  $\text{real } (m * n) = \text{real } (m::\text{nat}) * \text{real } n$   
**by** (*simp add: real-of-nat-def of-nat-mult*)

**lemma** *real-of-nat-power* [*simp*]:  $\text{real } (m ^ n) = \text{real } (m::\text{nat}) ^ n$   
**by** (*simp add: real-of-nat-def of-nat-power*)

**lemmas** *power-real-of-nat = real-of-nat-power* [*symmetric*]

**lemma** *real-of-nat-setsum* [*simp*]:  $\text{real } ((\text{SUM } x:A. f x)::\text{nat}) =$   
 $(\text{SUM } x:A. \text{real}(f x))$   
**apply** (*subst real-eq-of-nat*)  
**apply** (*rule of-nat-setsum*)  
**done**

**lemma** *real-of-nat-setprod* [*simp*]:  $\text{real } ((\text{PROD } x:A. f x)::\text{nat}) =$   
 $(\text{PROD } x:A. \text{real}(f x))$   
**apply** (*subst real-eq-of-nat*)  
**apply** (*rule of-nat-setprod*)  
**done**

**lemma** *real-of-card*:  $\text{real } (\text{card } A) = \text{setsum } (\%x.1) A$   
**apply** (*subst card-eq-setsum*)  
**apply** (*subst real-of-nat-setsum*)  
**apply** *simp*

done

**lemma** *real-of-nat-inject* [iff]: (*real* (*n::nat*) = *real m*) = (*n* = *m*)  
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-zero-iff* [iff]: (*real* (*n::nat*) = 0) = (*n* = 0)  
**by** (*simp add: real-of-nat-def*)

**lemma** *real-of-nat-diff*:  $n \leq m \implies \text{real } (m - n) = \text{real } (m::\text{nat}) - \text{real } n$   
**by** (*simp add: add: real-of-nat-def of-nat-diff*)

**lemma** *real-of-nat-gt-zero-cancel-iff* [simp]: ( $0 < \text{real } (n::\text{nat})$ ) = ( $0 < n$ )  
**by** (*auto simp: real-of-nat-def*)

**lemma** *real-of-nat-le-zero-cancel-iff* [simp]: ( $\text{real } (n::\text{nat}) \leq 0$ ) = ( $n = 0$ )  
**by** (*simp add: add: real-of-nat-def*)

**lemma** *not-real-of-nat-less-zero* [simp]:  $\sim \text{real } (n::\text{nat}) < 0$   
**by** (*simp add: add: real-of-nat-def*)

**lemma** *nat-less-real-le*: ( $(n::\text{nat}) < m$ ) = ( $\text{real } n + 1 \leq \text{real } m$ )  
**apply** (*subgoal-tac real n + 1 = real (Suc n)*)  
**apply** *simp*  
**apply** (*auto simp add: real-of-nat-Suc*)  
done

**lemma** *nat-le-real-less*: ( $(n::\text{nat}) \leq m$ ) = ( $\text{real } n < \text{real } m + 1$ )  
**apply** (*subgoal-tac real m + 1 = real (Suc m)*)  
**apply** (*simp add: less-Suc-eq-le*)  
**apply** (*simp add: real-of-nat-Suc*)  
done

**lemma** *real-of-nat-div-aux*:  $0 < d \implies (\text{real } (x::\text{nat})) / (\text{real } d) =$   
 $\text{real } (x \text{ div } d) + (\text{real } (x \text{ mod } d)) / (\text{real } d)$   
**proof** –  
**assume**  $0 < d$   
**have**  $x = (x \text{ div } d) * d + x \text{ mod } d$   
**by** *auto*  
**then have**  $\text{real } x = \text{real } (x \text{ div } d) * \text{real } d + \text{real } (x \text{ mod } d)$   
**by** (*simp only: real-of-nat-mult [THEN sym] real-of-nat-add [THEN sym]*)  
**then have**  $\text{real } x / \text{real } d = \dots / \text{real } d$   
**by** *simp*  
**then show** *?thesis*  
**by** (*auto simp add: add-divide-distrib algebra-simps prems*)  
**qed**

**lemma** *real-of-nat-div*:  $0 < (d::\text{nat}) \implies d \text{ dvd } n \implies$   
 $\text{real } (n \text{ div } d) = \text{real } n / \text{real } d$   
**apply** (*frule real-of-nat-div-aux [of d n]*)

```

apply simp
apply (subst dvd-eq-mod-eq-0 [THEN sym])
apply assumption
done

```

```

lemma real-of-nat-div2:
   $0 \leq \text{real } (n::\text{nat}) / \text{real } (x) - \text{real } (n \text{ div } x)$ 
apply(case-tac x = 0)
apply (simp)
apply (simp add: algebra-simps)
apply (subst real-of-nat-div-aux)
apply simp
apply simp
apply (subst zero-le-divide-iff)
apply simp
done

```

```

lemma real-of-nat-div3:
   $\text{real } (n::\text{nat}) / \text{real } (x) - \text{real } (n \text{ div } x) \leq 1$ 
apply(case-tac x = 0)
apply (simp)
apply (simp add: algebra-simps)
apply (subst real-of-nat-div-aux)
apply simp
apply simp
done

```

```

lemma real-of-nat-div4:  $\text{real } (n \text{ div } x) \leq \text{real } (n::\text{nat}) / \text{real } x$ 
by (insert real-of-nat-div2 [of n x], simp)

```

```

lemma real-of-int-real-of-nat:  $\text{real } (\text{int } n) = \text{real } n$ 
by (simp add: real-of-nat-def real-of-int-def int-eq-of-nat)

```

```

lemma real-of-int-of-nat-eq [simp]:  $\text{real } (\text{of-nat } n :: \text{int}) = \text{real } n$ 
by (simp add: real-of-int-def real-of-nat-def)

```

```

lemma real-nat-eq-real [simp]:  $0 \leq x \implies \text{real } (\text{nat } x) = \text{real } x$ 
apply (subgoal-tac real(int(nat x)) = real(nat x))
apply force
apply (simp only: real-of-int-real-of-nat)
done

```

```

lemma Nats-real-of-nat [simp]:  $\text{real } (n::\text{nat}) \in \text{Nats}$ 
unfolding real-of-nat-def by (rule of-nat-in-Nats)

```

```

lemma Ints-real-of-nat [simp]:  $\text{real } (n::\text{nat}) \in \text{Ints}$ 
unfolding real-of-nat-def by (rule Ints-of-nat)

```

## 66.13 Rationals

**lemma** *Rats-real-nat[simp]*:  $\text{real}(n::\text{nat}) \in \mathbb{Q}$   
**by** (*simp add: real-eq-of-nat*)

**lemma** *Rats-eq-int-div-int*:  
 $\mathbb{Q} = \{ \text{real}(i::\text{int})/\text{real}(j::\text{int}) \mid i\ j.\ j \neq 0 \}$  (**is**  $- = ?S$ )  
**proof**  
 show  $\mathbb{Q} \subseteq ?S$   
**proof**  
 fix  $x::\text{real}$  **assume**  $x : \mathbb{Q}$   
 then **obtain**  $r$  **where**  $x = \text{of-rat } r$  **unfolding** *Rats-def* ..  
 have  $\text{of-rat } r : ?S$   
 by (*cases r*)(*auto simp add: of-rat-rat real-eq-of-int*)  
 thus  $x : ?S$  **using**  $\langle x = \text{of-rat } r \rangle$  **by** *simp*  
**qed**  
**next**  
 show  $?S \subseteq \mathbb{Q}$   
**proof**(*auto simp: Rats-def*)  
 fix  $i\ j :: \text{int}$  **assume**  $j \neq 0$   
 hence  $\text{real } i / \text{real } j = \text{of-rat}(\text{Fract } i\ j)$   
 by (*simp add: of-rat-rat real-eq-of-int*)  
 thus  $\text{real } i / \text{real } j \in \text{range of-rat}$  **by** *blast*  
**qed**  
**qed**

**lemma** *Rats-eq-int-div-nat*:  
 $\mathbb{Q} = \{ \text{real}(i::\text{int})/\text{real}(n::\text{nat}) \mid i\ n.\ n \neq 0 \}$   
**proof**(*auto simp: Rats-eq-int-div-int*)  
 fix  $i\ j::\text{int}$  **assume**  $j \neq 0$   
 show  $\exists (i'::\text{int}) (n::\text{nat}). \text{real } i / \text{real } j = \text{real } i' / \text{real } n \wedge 0 < n$   
**proof** *cases*  
 assume  $j > 0$   
 hence  $\text{real } i / \text{real } j = \text{real } i / \text{real}(\text{nat } j) \wedge 0 < \text{nat } j$   
 by (*simp add: real-eq-of-int real-eq-of-nat of-nat-nat*)  
 thus  $?thesis$  **by** *blast*  
**next**  
 assume  $j < 0$   
 hence  $\text{real } i / \text{real } j = \text{real}(-i) / \text{real}(\text{nat}(-j)) \wedge 0 < \text{nat}(-j)$  **using**  $\langle j \neq 0 \rangle$   
 by (*simp add: real-eq-of-int real-eq-of-nat of-nat-nat*)  
 thus  $?thesis$  **by** *blast*  
**qed**  
**next**  
 fix  $i::\text{int}$  **and**  $n::\text{nat}$  **assume**  $0 < n$   
 hence  $\text{real } i / \text{real } n = \text{real } i / \text{real}(\text{int } n) \wedge \text{int } n \neq 0$  **by** *simp*  
 thus  $\exists (i'::\text{int})\ j::\text{int}. \text{real } i / \text{real } n = \text{real } i' / \text{real } j \wedge j \neq 0$  **by** *blast*  
**qed**

**lemma** *Rats-abs-nat-div-natE*:

```

assumes  $x \in \mathbb{Q}$ 
obtains  $m\ n :: \text{nat}$ 
where  $n \neq 0$  and  $|x| = \text{real } m / \text{real } n$  and  $\text{gcd } m\ n = 1$ 
proof –
  from  $\langle x \in \mathbb{Q} \rangle$  obtain  $i :: \text{int}$  and  $n :: \text{nat}$  where  $n \neq 0$  and  $x = \text{real } i / \text{real } n$ 
    by (auto simp add: Rats-eq-int-div-nat)
  hence  $|x| = \text{real}(\text{nat}(\text{abs } i)) / \text{real } n$  by simp
  then obtain  $m :: \text{nat}$  where  $x\text{-rat}: |x| = \text{real } m / \text{real } n$  by blast
  let  $?gcd = \text{gcd } m\ n$ 
  from  $\langle n \neq 0 \rangle$  have  $\text{gcd}: ?gcd \neq 0$  by simp
  let  $?k = m \text{ div } ?gcd$ 
  let  $?l = n \text{ div } ?gcd$ 
  let  $?gcd' = \text{gcd } ?k\ ?l$ 
  have  $?gcd \text{ dvd } m$  .. then have  $\text{gcd-k}: ?gcd * ?k = m$ 
    by (rule dvd-mult-div-cancel)
  have  $?gcd \text{ dvd } n$  .. then have  $\text{gcd-l}: ?gcd * ?l = n$ 
    by (rule dvd-mult-div-cancel)
  from  $\langle n \neq 0 \rangle$  and  $\text{gcd-l}$  have  $?l \neq 0$  by (auto iff del: neq0-conv)
  moreover
  have  $|x| = \text{real } ?k / \text{real } ?l$ 
  proof –
    from  $\text{gcd}$  have  $\text{real } ?k / \text{real } ?l =$ 
       $\text{real } (?gcd * ?k) / \text{real } (?gcd * ?l)$  by simp
    also from  $\text{gcd-k}$  and  $\text{gcd-l}$  have  $\dots = \text{real } m / \text{real } n$  by simp
    also from  $x\text{-rat}$  have  $\dots = |x|$  ..
    finally show  $?thesis$  ..
  qed
  moreover
  have  $?gcd' = 1$ 
  proof –
    have  $?gcd * ?gcd' = \text{gcd } (?gcd * ?k) (?gcd * ?l)$ 
      by (rule gcd-mult-distrib-nat)
    with  $\text{gcd-k}$   $\text{gcd-l}$  have  $?gcd * ?gcd' = ?gcd$  by simp
    with  $\text{gcd}$  show  $?thesis$  by auto
  qed
  ultimately show  $?thesis$  ..
qed

```

## 66.14 Numerals and Arithmetic

**declare** *number-of-real-def* [*code del*]

**lemma** [*code-unfold-post*]:  
 $\text{number-of } k = \text{real-of-int } (\text{number-of } k)$   
**unfolding** *number-of-is-id number-of-real-def* **..**

Collapse applications of *real* to *number-of*

**lemma** *real-number-of* [*simp*]:  $\text{real } (\text{number-of } v :: \text{int}) = \text{number-of } v$   
**by** (*simp add: real-of-int-def*)

**lemma** *real-of-nat-number-of* [simp]:  

$$\text{real } (\text{number-of } v :: \text{nat}) =$$

$$(\text{if } \text{neg } (\text{number-of } v :: \text{int}) \text{ then } 0$$

$$\text{else } (\text{number-of } v :: \text{real}))$$
**by** (simp add: real-of-int-real-of-nat [symmetric])

**declaration**  $\langle\langle$   
 $K$  (Lin-Arith.add-inj-thms [ $@\{\text{thm real-of-nat-le-iff}\}$  RS iffD2,  $@\{\text{thm real-of-nat-inject}\}$  RS iffD2]  
 (\* not needed because  $x < (y::\text{nat})$  can be rewritten as  $\text{Suc } x \leq y$ : real-of-nat-less-iff RS iffD2 \*)  
 $\#>$  Lin-Arith.add-inj-thms [ $@\{\text{thm real-of-int-le-iff}\}$  RS iffD2,  $@\{\text{thm real-of-int-inject}\}$  RS iffD2]  
 (\* not needed because  $x < (y::\text{int})$  can be rewritten as  $x + 1 \leq y$ : real-of-int-less-iff RS iffD2 \*)  
 $\#>$  Lin-Arith.add-simps [ $@\{\text{thm real-of-nat-zero}\}$ ,  $@\{\text{thm real-of-nat-Suc}\}$ ,  $@\{\text{thm real-of-nat-add}\}$ ,  
 $@\{\text{thm real-of-nat-mult}\}$ ,  $@\{\text{thm real-of-int-zero}\}$ ,  $@\{\text{thm real-of-one}\}$ ,  
 $@\{\text{thm real-of-int-add}\}$ ,  $@\{\text{thm real-of-int-minus}\}$ ,  $@\{\text{thm real-of-int-diff}\}$ ,  
 $@\{\text{thm real-of-int-mult}\}$ ,  $@\{\text{thm real-of-int-of-nat-eq}\}$ ,  
 $@\{\text{thm real-of-nat-number-of}\}$ ,  $@\{\text{thm real-number-of}\}]$   
 $\#>$  Lin-Arith.add-inj-const ( $@\{\text{const-name real}\}$ ,  $@\{\text{typ nat} \Rightarrow \text{real}\}$ )  
 $\#>$  Lin-Arith.add-inj-const ( $@\{\text{const-name real}\}$ ,  $@\{\text{typ int} \Rightarrow \text{real}\}$ )  
 $\rangle\rangle$

### 66.15 Simprules combining $x+y$ and 0: ARE THEY NEEDED?

Needed in this non-standard form by Hyperreal/Transcendental

**lemma** *real-0-le-divide-iff*:  

$$((0::\text{real}) \leq x/y) = ((x \leq 0 \mid 0 \leq y) \ \& \ (0 \leq x \mid y \leq 0))$$
**by** (auto simp add: zero-le-divide-iff)

**lemma** *real-add-minus-iff* [simp]:  $(x + - a = (0::\text{real})) = (x=a)$   
**by** arith

FIXME: redundant with *add-eq-0-iff* below

**lemma** *real-add-eq-0-iff*:  $(x+y = (0::\text{real})) = (y = -x)$   
**by** auto

**lemma** *real-add-less-0-iff*:  $(x+y < (0::\text{real})) = (y < -x)$   
**by** auto

**lemma** *real-0-less-add-iff*:  $((0::\text{real}) < x+y) = (-x < y)$   
**by** auto

**lemma** *real-add-le-0-iff*:  $(x+y \leq (0::\text{real})) = (y \leq -x)$   
**by** auto

**lemma** *real-0-le-add-iff*:  $((0::\text{real}) \leq x+y) = (-x \leq y)$   
**by** *auto*

## 66.16 Lemmas about powers

FIXME: declare this in Rings.thy or not at all

**declare** *abs-mult-self* [*simp*]

**lemma** *two-realpows-ge-one*:  $(1::\text{real}) \leq 2^n$   
**by** *simp*

**lemma** *two-realpows-gt* [*simp*]:  $\text{real } (n::\text{nat}) < 2^n$   
**apply** (*induct* *n*)  
**apply** (*auto simp add: real-of-nat-Suc*)  
**apply** (*subst mult-2*)  
**apply** (*erule add-less-le-mono*)  
**apply** (*rule two-realpows-ge-one*)  
**done**

TODO: no longer real-specific; rename and move elsewhere

**lemma** *realpows-Suc-le-self*:  
**fixes**  $r :: 'a::\text{linordered-semidom}$   
**shows**  $[| 0 \leq r; r \leq 1 |] \implies r^{Suc\ n} \leq r$   
**by** (*insert power-decreasing [of 1 Suc n r], simp*)

TODO: no longer real-specific; rename and move elsewhere

**lemma** *realpows-minus-mult*:  
**fixes**  $x :: 'a::\text{monoid-mult}$   
**shows**  $0 < n \implies x^{(n-1)} * x = x^n$   
**by** (*simp add: power-commutes split add: nat-diff-split*)

TODO: no longer real-specific; rename and move elsewhere

**lemma** *realpows-two-diff*:  
**fixes**  $x :: 'a::\text{comm-ring-1}$   
**shows**  $x^{Suc\ 0} - y^{Suc\ 0} = (x - y) * (x + y)$   
**by** (*simp add: algebra-simps*)

TODO: move elsewhere

**lemma** *add-eq-0-iff*:  
**fixes**  $x\ y :: 'a::\text{group-add}$   
**shows**  $x + y = 0 \iff y = -x$   
**by** (*auto dest: minus-unique*)

TODO: no longer real-specific; rename and move elsewhere

**lemma** *realpows-two-disj*:  
**fixes**  $x :: 'a::\text{idom}$



**shows**  $(x \text{^} \text{Suc } (\text{Suc } 0) = y \text{^} \text{Suc } (\text{Suc } 0)) = (x = y \mid x = -y)$   
**using** *realpow-two-diff* [of  $x \ y$ ]  
**by** (*auto simp add: add-eq-0-iff*)

FIXME: declare this [simp] for all types, or not at all

**lemma** *real-two-squares-add-zero-iff* [simp]:  
 $(x * x + y * y = 0) = ((x::\text{real}) = 0 \wedge y = 0)$   
**by** (*rule sum-squares-eq-zero-iff*)

TODO: no longer real-specific; rename and move elsewhere

**lemma** *real-squared-diff-one-factored*:  
**fixes**  $x :: 'a::\text{ring-1}$   
**shows**  $x * x - 1 = (x + 1) * (x - 1)$   
**by** (*simp add: algebra-simps*)

FIXME: declare this [simp] for all types, or not at all

**lemma** *realpow-two-sum-zero-iff* [simp]:  
 $(x \text{^} 2 + y \text{^} 2 = (0::\text{real})) = (x = 0 \ \& \ y = 0)$   
**by** (*rule sum-power2-eq-zero-iff*)

**lemma** *real-minus-mult-self-le* [simp]:  $-(u * u) \leq (x * (x::\text{real}))$   
**by** (*rule-tac y = 0 in order-trans, auto*)

**lemma** *realpow-square-minus-le* [simp]:  $-(u \text{^} 2) \leq (x::\text{real}) \text{^} 2$   
**by** (*auto simp add: power2-eq-square*)

## 66.17 Density of the Reals

**lemma** *real-lbound-gt-zero*:  
 $[| (0::\text{real}) < d1; 0 < d2 |] ==> \exists e. 0 < e \ \& \ e < d1 \ \& \ e < d2$   
**apply** (*rule-tac x = (min d1 d2) / 2 in exI*)  
**apply** (*simp add: min-def*)  
**done**

Similar results are proved in *Fields*

**lemma** *real-less-half-sum*:  $x < y ==> x < (x+y) / (2::\text{real})$   
**by** *auto*

**lemma** *real-gt-half-sum*:  $x < y ==> (x+y)/(2::\text{real}) < y$   
**by** *auto*

## 66.18 Absolute Value Function for the Reals

**lemma** *abs-minus-add-cancel*:  $\text{abs}(x + (-y)) = \text{abs}(y + -(x::\text{real}))$   
**by** (*simp add: abs-if*)

**lemma** *abs-le-interval-iff*:  $(\text{abs } x \leq r) = (-r \leq x \ \& \ x \leq (r::\text{real}))$   
**by** (*force simp add: abs-le-iff*)

**lemma** *abs-add-one-gt-zero* [simp]:  $(0::real) < 1 + abs(x)$   
**by** (simp add: abs-if)

**lemma** *abs-real-of-nat-cancel* [simp]:  $abs (real\ x) = real\ (x::nat)$   
**by** (rule abs-of-nonneg [OF real-of-nat-ge-zero])

**lemma** *abs-add-one-not-less-self* [simp]:  $\sim abs(x) + (1::real) < x$   
**by** simp

**lemma** *abs-sum-triangle-ineq*:  $abs ((x::real) + y + (-l + -m)) \leq abs(x + -l) + abs(y + -m)$   
**by** simp

## 66.19 Implementation of rational real numbers

**definition** *Ratreal* :: *rat*  $\Rightarrow$  *real* **where**  
 [simp]: *Ratreal* = of-rat

**code-datatype** *Ratreal*

**lemma** *Ratreal-number-collapse* [code-post]:  
   *Ratreal* 0 = 0  
   *Ratreal* 1 = 1  
   *Ratreal* (number-of *k*) = number-of *k*  
**by** simp-all

**lemma** *zero-real-code* [code, code-unfold]:  
   0 = *Ratreal* 0  
**by** simp

**lemma** *one-real-code* [code, code-unfold]:  
   1 = *Ratreal* 1  
**by** simp

**lemma** *number-of-real-code* [code-unfold]:  
   number-of *k* = *Ratreal* (number-of *k*)  
**by** simp

**lemma** *Ratreal-number-of-quotient* [code-post]:  
   *Ratreal* (number-of *r*) / *Ratreal* (number-of *s*) = number-of *r* / number-of *s*  
**by** simp

**lemma** *Ratreal-number-of-quotient2* [code-post]:  
   *Ratreal* (number-of *r* / number-of *s*) = number-of *r* / number-of *s*  
**unfolding** *Ratreal-number-of-quotient* [symmetric] *Ratreal-def* of-rat-divide ..

**instantiation** *real* :: *eq*  
**begin**

**definition** *eq-class.eq* (*x::real*) *y*  $\longleftrightarrow x - y = 0$

**instance by default** (*simp add: eq-real-def*)

**lemma** *real-eq-code* [*code*]: *eq-class.eq* (*Ratreal* *x*) (*Ratreal* *y*)  $\longleftrightarrow$  *eq-class.eq* *x* *y*  
**by** (*simp add: eq-real-def eq*)

**lemma** *real-eq-refl* [*code nbe*]:  
*eq-class.eq* (*x::real*) *x*  $\longleftrightarrow$  *True*  
**by** (*rule HOL.eq-refl*)

**end**

**lemma** *real-less-eq-code* [*code*]: *Ratreal* *x*  $\leq$  *Ratreal* *y*  $\longleftrightarrow x \leq y$   
**by** (*simp add: of-rat-less-eq*)

**lemma** *real-less-code* [*code*]: *Ratreal* *x*  $<$  *Ratreal* *y*  $\longleftrightarrow x < y$   
**by** (*simp add: of-rat-less*)

**lemma** *real-plus-code* [*code*]: *Ratreal* *x* + *Ratreal* *y* = *Ratreal* (*x* + *y*)  
**by** (*simp add: of-rat-add*)

**lemma** *real-times-code* [*code*]: *Ratreal* *x* \* *Ratreal* *y* = *Ratreal* (*x* \* *y*)  
**by** (*simp add: of-rat-mult*)

**lemma** *real-uminus-code* [*code*]:  $-$  *Ratreal* *x* = *Ratreal* ( $-$  *x*)  
**by** (*simp add: of-rat-minus*)

**lemma** *real-minus-code* [*code*]: *Ratreal* *x* - *Ratreal* *y* = *Ratreal* (*x* - *y*)  
**by** (*simp add: of-rat-diff*)

**lemma** *real-inverse-code* [*code*]: *inverse* (*Ratreal* *x*) = *Ratreal* (*inverse* *x*)  
**by** (*simp add: of-rat-inverse*)

**lemma** *real-divide-code* [*code*]: *Ratreal* *x* / *Ratreal* *y* = *Ratreal* (*x* / *y*)  
**by** (*simp add: of-rat-divide*)

**definition** (*in term-syntax*)  
*valterm-ratreal* :: *rat*  $\times$  (*unit*  $\Rightarrow$  *Code-Evaluation.term*)  $\Rightarrow$  *real*  $\times$  (*unit*  $\Rightarrow$  *Code-Evaluation.term*) **where**  
[*code-unfold*]: *valterm-ratreal* *k* = *Code-Evaluation.valtermify* *Ratreal*  $\{\cdot\}$  *k*

**notation** *fcomp* (**infixl** *o>* 60)

**notation** *scomp* (**infixl** *o $\rightarrow$*  60)

**instantiation** *real* :: *random*  
**begin**

**definition**

*Quickcheck.random i = Quickcheck.random i o → (λr. Pair (valterm-ratreal r))*

**instance ..**

**end**

**no-notation** *fcomp* (**infixl** *o* > 60)

**no-notation** *scomp* (**infixl** *o* → 60)

Setup for SML code generator

**types-code**

*real ((int \*/ int))*

**attach** (*term-of*) ⟨⟨

*fun term-of-real (p, q) =*

*let*

*val rT = HOLogic.realT*

*in*

*if q = 1 orelse p = 0 then HOLogic.mk-number rT p*

*else @{term op / :: real ⇒ real ⇒ real} \$*

*HOLogic.mk-number rT p \$ HOLogic.mk-number rT q*

*end;*

⟩⟩

**attach** (*test*) ⟨⟨

*fun gen-real i =*

*let*

*val p = random-range 0 i;*

*val q = random-range 1 (i + 1);*

*val g = Integer.gcd p q;*

*val p' = p div g;*

*val q' = q div g;*

*val r = (if one-of [true, false] then p' else ~ p',*

*if p' = 0 then 1 else q')*

*in*

*(r, fn () => term-of-real r)*

*end;*

⟩⟩

**consts-code**

*Ratreal ((-))*

**consts-code**

*of-int :: int ⇒ real (⟨module⟩real'-of'-int)*

**attach** ⟨⟨

*fun real-of-int i = (i, 1);*

⟩⟩

**setup** ⟨⟨

*Nitpick.register-frac-type @{type-name real}*

```

[(@{const-name zero-real-inst.zero-real}, @{const-name Nitpick.zero-frac}),
 (@{const-name one-real-inst.one-real}, @{const-name Nitpick.one-frac}),
 (@{const-name plus-real-inst.plus-real}, @{const-name Nitpick.plus-frac}),
 (@{const-name times-real-inst.times-real}, @{const-name Nitpick.times-frac}),
 (@{const-name uminus-real-inst.uminus-real}, @{const-name Nitpick.uminus-frac}),
 (@{const-name number-real-inst.number-of-real}, @{const-name Nitpick.number-of-frac}),
 (@{const-name inverse-real-inst.inverse-real}, @{const-name Nitpick.inverse-frac}),
 (@{const-name ord-real-inst.less-eq-real}, @{const-name Nitpick.less-eq-frac})]
>>

```

```

lemmas [nitpick-def] = inverse-real-inst.inverse-real
          number-real-inst.number-of-real one-real-inst.one-real
          ord-real-inst.less-eq-real plus-real-inst.plus-real
          times-real-inst.times-real uminus-real-inst.uminus-real
          zero-real-inst.zero-real

```

```

end

```

## 67 RComplete: Completeness of the Reals; Floor and Ceiling Functions

```

theory RComplete
imports Lubs RealDef
begin

```

```

lemma real-sum-of-halves:  $x/2 + x/2 = (x::real)$ 
by simp

```

```

lemma abs-diff-less-iff:
   $(|x - a| < (r::'a::linordered-idom)) = (a - r < x \wedge x < a + r)$ 
by auto

```

### 67.1 Completeness of Positive Reals

Supremum property for the set of positive reals

Let  $P$  be a non-empty set of positive reals, with an upper bound  $y$ . Then  $P$  has a least upper bound (written  $S$ ).

FIXME: Can the premise be weakened to  $\forall x \in P. x \leq y$ ?

Only used in HOL/Import/HOL4Compat.thy; delete?

```

lemma posreal-complete:
  assumes positive-P:  $\forall x \in P. (0::real) < x$ 
  and not-empty-P:  $\exists x. x \in P$ 
  and upper-bound-Ex:  $\exists y. \forall x \in P. x < y$ 
  shows  $\exists S. \forall y. (\exists x \in P. y < x) = (y < S)$ 
proof –

```

```

from upper-bound-Ex have  $\exists z. \forall x \in P. x \leq z$ 
  by (auto intro: less-imp-le)
from complete-real [OF not-empty-P this] obtain S
where S1:  $\bigwedge x. x \in P \implies x \leq S$  and S2:  $\bigwedge z. \forall x \in P. x \leq z \implies S \leq z$  by fast
have  $\forall y. (\exists x \in P. y < x) = (y < S)$ 
proof
  fix y show  $(\exists x \in P. y < x) = (y < S)$ 
    apply (cases  $\exists x \in P. y < x$ , simp-all)
    apply (clarify, drule S1, simp)
    apply (simp add: not-less S2)
    done
qed
thus ?thesis ..
qed

```

Completeness properties using *isUb*, *isLub* etc.

```

lemma real-isLub-unique:  $[[ \text{isLub } R \ S \ x; \text{isLub } R \ S \ y ] \implies x = (y::\text{real})$ 
  apply (frule isLub-isUb)
  apply (frule-tac  $x = y$  in isLub-isUb)
  apply (blast intro!: order-antisym dest!: isLub-le-isUb)
  done

```

reals Completeness (again!)

```

lemma reals-complete:
  assumes notempty-S:  $\exists X. X \in S$ 
    and exists-Ub:  $\exists Y. \text{isUb } (\text{UNIV}::\text{real set}) \ S \ Y$ 
  shows  $\exists t. \text{isLub } (\text{UNIV}::\text{real set}) \ S \ t$ 
proof –
  from assms have  $\exists X. X \in S$  and  $\exists Y. \forall x \in S. x \leq Y$ 
    unfolding isUb-def settle-def by simp-all
  from complete-real [OF this] show ?thesis
    unfolding isLub-def leastP-def settle-def setge-def Ball-def
      Collect-def mem-def isUb-def UNIV-def by simp
qed

```

A version of the same theorem without all those predicates!

```

lemma reals-complete2:
  fixes S :: (real set)
  assumes  $\exists y. y \in S$  and  $\exists (x::\text{real}). \forall y \in S. y \leq x$ 
  shows  $\exists x. (\forall y \in S. y \leq x) \ \& \$ 
     $(\forall z. ((\forall y \in S. y \leq z) \implies x \leq z))$ 
using assms by (rule complete-real)

```

## 67.2 The Archimedean Property of the Reals

```

theorem reals-Archimedean:
  assumes x-pos:  $0 < x$ 
  shows  $\exists n. \text{inverse } (\text{real } (\text{Suc } n)) < x$ 

```

```

unfolding real-of-nat-def using x-pos
by (rule ex-inverse-of-nat-Suc-less)

lemma reals-Archimedean2:  $\exists n. (x::real) < real (n::nat)$ 
unfolding real-of-nat-def by (rule ex-less-of-nat)

lemma reals-Archimedean3:
  assumes x-greater-zero:  $0 < x$ 
  shows  $\forall (y::real). \exists (n::nat). y < real\ n * x$ 
  unfolding real-of-nat-def using  $\langle 0 < x \rangle$ 
  by (auto intro: ex-less-of-nat-mult)

lemma reals-Archimedean6:
   $0 \leq r \implies \exists (n::nat). real\ (n - 1) \leq r \ \&\ r < real\ (n)$ 
unfolding real-of-nat-def
apply (rule exI [where x=nat (floor r + 1)])
apply (insert floor-correct [of r])
apply (simp add: nat-add-distrib of-nat-nat)
done

lemma reals-Archimedean6a:  $0 \leq r \implies \exists n. real\ (n) \leq r \ \&\ r < real\ (Suc\ n)$ 
by (drule reals-Archimedean6) auto

TODO: delete

lemma reals-Archimedean-6b-int:
   $0 \leq r \implies \exists n::int. real\ n \leq r \ \&\ r < real\ (n+1)$ 
unfolding real-of-int-def by (rule floor-exists)

TODO: delete

lemma reals-Archimedean-6c-int:
   $r < 0 \implies \exists n::int. real\ n \leq r \ \&\ r < real\ (n+1)$ 
unfolding real-of-int-def by (rule floor-exists)

```

### 67.3 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean property of the reals.

```

lemma Rats-dense-in-nn-real: fixes x::real
assumes  $0 \leq x$  and  $x < y$  shows  $\exists r \in \mathbb{Q}. x < r \ \&\ r < y$ 
proof –
  from  $\langle x < y \rangle$  have  $0 < y - x$  by simp
  with reals-Archimedean obtain q::nat
    where q: inverse (real q)  $< y - x$  and  $0 < real\ q$  by auto
  def p  $\equiv$  LEAST n.  $y \leq real\ (Suc\ n) / real\ q$ 
  from reals-Archimedean2 obtain n::nat where  $y * real\ q < real\ n$  by auto
  with  $\langle 0 < real\ q \rangle$  have ex:  $y \leq real\ n / real\ q$  (is ?P n)
    by (simp add: pos-less-divide-eq[THEN sym])

```

```

also from assms have  $\neg y \leq \text{real } (0::\text{nat}) / \text{real } q$  by simp
ultimately have main:  $(\text{LEAST } n. y \leq \text{real } n / \text{real } q) = \text{Suc } p$ 
  by (unfold p-def) (rule Least-Suc)
also from ex have  $?P (\text{LEAST } x. ?P x)$  by (rule LeastI)
ultimately have suc:  $y \leq \text{real } (\text{Suc } p) / \text{real } q$  by simp
def  $r \equiv \text{real } p / \text{real } q$ 
have  $x = y - (y - x)$  by simp
also from suc  $q$  have  $\dots < \text{real } (\text{Suc } p) / \text{real } q - \text{inverse } (\text{real } q)$  by arith
also have  $\dots = \text{real } p / \text{real } q$ 
  by (simp only: inverse-eq-divide diff-def real-of-nat-Suc
    minus-divide-left add-divide-distrib [THEN sym]) simp
finally have  $x < r$  by (unfold r-def)
have  $p < \text{Suc } p$  .. also note main [THEN sym]
finally have  $\neg ?P p$  by (rule not-less-Least)
hence  $r < y$  by (simp add: r-def)
from r-def have  $r \in \mathbb{Q}$  by simp
with  $\langle x < r \rangle$   $\langle r < y \rangle$  show ?thesis by fast
qed

```

```

theorem Rats-dense-in-real: fixes  $x y :: \text{real}$ 
assumes  $x < y$  shows  $\exists r \in \mathbb{Q}. x < r \wedge r < y$ 
proof -
  from reals-Archimedean2 obtain  $n::\text{nat}$  where  $-x < \text{real } n$  by auto
  hence  $0 \leq x + \text{real } n$  by arith
  also from  $\langle x < y \rangle$  have  $x + \text{real } n < y + \text{real } n$  by arith
  ultimately have  $\exists r \in \mathbb{Q}. x + \text{real } n < r \wedge r < y + \text{real } n$ 
    by (rule Rats-dense-in-nn-real)
  then obtain  $r$  where  $r \in \mathbb{Q}$  and  $r2: x + \text{real } n < r$ 
    and  $r3: r < y + \text{real } n$ 
  by blast
  have  $r - \text{real } n = r + \text{real } (\text{int } n) / \text{real } (-1::\text{int})$  by simp
  also from  $\langle r \in \mathbb{Q} \rangle$  have  $r + \text{real } (\text{int } n) / \text{real } (-1::\text{int}) \in \mathbb{Q}$  by simp
  also from  $r2$  have  $x < r - \text{real } n$  by arith
  moreover from  $r3$  have  $r - \text{real } n < y$  by arith
  ultimately show ?thesis by fast
qed

```

## 67.4 Floor and Ceiling Functions from the Reals to the Integers

```

lemma number-of-less-real-of-int-iff [simp]:
   $((\text{number-of } n) < \text{real } (m::\text{int})) = (\text{number-of } n < m)$ 
apply auto
apply (rule real-of-int-less-iff [THEN iffD1])
apply (drule-tac [2] real-of-int-less-iff [THEN iffD2], auto)
done

```

```

lemma number-of-less-real-of-int-iff2 [simp]:
   $(\text{real } (m::\text{int}) < (\text{number-of } n)) = (m < \text{number-of } n)$ 

```



```

apply auto
apply (rule real-of-int-less-iff [THEN iffD1])
apply (drule-tac [2] real-of-int-less-iff [THEN iffD2], auto)
done

```

```

lemma number-of-le-real-of-int-iff [simp]:
   $((\text{number-of } n) \leq \text{real } (m::\text{int})) = (\text{number-of } n \leq m)$ 
by (simp add: linorder-not-less [symmetric])

```

```

lemma number-of-le-real-of-int-iff2 [simp]:
   $(\text{real } (m::\text{int}) \leq (\text{number-of } n)) = (m \leq \text{number-of } n)$ 
by (simp add: linorder-not-less [symmetric])

```

```

lemma floor-real-of-nat [simp]:  $\text{floor } (\text{real } (n::\text{nat})) = \text{int } n$ 
unfolding real-of-nat-def by simp

```

```

lemma floor-minus-real-of-nat [simp]:  $\text{floor } (- \text{real } (n::\text{nat})) = - \text{int } n$ 
unfolding real-of-nat-def by (simp add: floor-minus)

```

```

lemma floor-real-of-int [simp]:  $\text{floor } (\text{real } (n::\text{int})) = n$ 
unfolding real-of-int-def by simp

```

```

lemma floor-minus-real-of-int [simp]:  $\text{floor } (- \text{real } (n::\text{int})) = - n$ 
unfolding real-of-int-def by (simp add: floor-minus)

```

```

lemma real-lb-ub-int:  $\exists n::\text{int}. \text{real } n \leq r \ \& \ r < \text{real } (n+1)$ 
unfolding real-of-int-def by (rule floor-exists)

```

```

lemma lemma-floor:
  assumes a1:  $\text{real } m \leq r$  and a2:  $r < \text{real } n + 1$ 
  shows  $m \leq (n::\text{int})$ 
proof –
  have  $\text{real } m < \text{real } n + 1$  using a1 a2 by (rule order-le-less-trans)
  also have  $\dots = \text{real } (n + 1)$  by simp
  finally have  $m < n + 1$  by (simp only: real-of-int-less-iff)
  thus ?thesis by arith
qed

```

```

lemma real-of-int-floor-le [simp]:  $\text{real } (\text{floor } r) \leq r$ 
unfolding real-of-int-def by (rule of-int-floor-le)

```

```

lemma lemma-floor2:  $\text{real } n < \text{real } (x::\text{int}) + 1 \implies n \leq x$ 
by (auto intro: lemma-floor)

```

```

lemma real-of-int-floor-cancel [simp]:
   $(\text{real } (\text{floor } x) = x) = (\exists n::\text{int}. x = \text{real } n)$ 
  using floor-real-of-int by metis

```

```

lemma floor-eq:  $[\text{real } n < x; x < \text{real } n + 1] \implies \text{floor } x = n$ 

```

```

unfolding real-of-int-def using floor-unique [of n x] by simp

lemma floor-eq2: [ $\text{real } n \leq x; x < \text{real } n + 1$ ]  $\implies \text{floor } x = n$ 
  unfolding real-of-int-def by (rule floor-unique)

lemma floor-eq3: [ $\text{real } n < x; x < \text{real } (\text{Suc } n)$ ]  $\implies \text{nat}(\text{floor } x) = n$ 
apply (rule inj-int [THEN injD])
apply (simp add: real-of-nat-Suc)
apply (simp add: real-of-nat-Suc floor-eq floor-eq [where n = int n])
done

lemma floor-eq4: [ $\text{real } n \leq x; x < \text{real } (\text{Suc } n)$ ]  $\implies \text{nat}(\text{floor } x) = n$ 
apply (drule order-le-imp-less-or-eq)
apply (auto intro: floor-eq3)
done

lemma real-of-int-floor-ge-diff-one [simp]:  $r - 1 \leq \text{real}(\text{floor } r)$ 
  unfolding real-of-int-def using floor-correct [of r] by simp

lemma real-of-int-floor-gt-diff-one [simp]:  $r - 1 < \text{real}(\text{floor } r)$ 
  unfolding real-of-int-def using floor-correct [of r] by simp

lemma real-of-int-floor-add-one-ge [simp]:  $r \leq \text{real}(\text{floor } r) + 1$ 
  unfolding real-of-int-def using floor-correct [of r] by simp

lemma real-of-int-floor-add-one-gt [simp]:  $r < \text{real}(\text{floor } r) + 1$ 
  unfolding real-of-int-def using floor-correct [of r] by simp

lemma le-floor:  $\text{real } a \leq x \implies a \leq \text{floor } x$ 
  unfolding real-of-int-def by (simp add: le-floor-iff)

lemma real-le-floor:  $a \leq \text{floor } x \implies \text{real } a \leq x$ 
  unfolding real-of-int-def by (simp add: le-floor-iff)

lemma le-floor-eq:  $(a \leq \text{floor } x) = (\text{real } a \leq x)$ 
  unfolding real-of-int-def by (rule le-floor-iff)

lemma floor-less-eq:  $(\text{floor } x < a) = (x < \text{real } a)$ 
  unfolding real-of-int-def by (rule floor-less-iff)

lemma less-floor-eq:  $(a < \text{floor } x) = (\text{real } a + 1 \leq x)$ 
  unfolding real-of-int-def by (rule less-floor-iff)

lemma floor-le-eq:  $(\text{floor } x \leq a) = (x < \text{real } a + 1)$ 
  unfolding real-of-int-def by (rule floor-le-iff)

lemma floor-add [simp]:  $\text{floor } (x + \text{real } a) = \text{floor } x + a$ 
  unfolding real-of-int-def by (rule floor-add-of-int)

```

**lemma** *floor-subtract* [simp]:  $\text{floor } (x - \text{real } a) = \text{floor } x - a$   
**unfolding** *real-of-int-def* **by** (rule *floor-diff-of-int*)

**lemma** *le-mult-floor*:  
**assumes**  $0 \leq (a :: \text{real})$  **and**  $0 \leq b$   
**shows**  $\text{floor } a * \text{floor } b \leq \text{floor } (a * b)$   
**proof** –  
**have**  $\text{real } (\text{floor } a) \leq a$   
**and**  $\text{real } (\text{floor } b) \leq b$  **by** *auto*  
**hence**  $\text{real } (\text{floor } a * \text{floor } b) \leq a * b$   
**using** *assms* **by** (auto *intro!*: *mult-mono*)  
**also have**  $a * b < \text{real } (\text{floor } (a * b) + 1)$  **by** *auto*  
**finally show** *?thesis* **unfolding** *real-of-int-less-iff* **by** *simp*  
**qed**

**lemma** *ceiling-real-of-nat* [simp]:  $\text{ceiling } (\text{real } (n :: \text{nat})) = \text{int } n$   
**unfolding** *real-of-nat-def* **by** *simp*

**lemma** *ceiling-floor* [simp]:  $\text{ceiling } (\text{real } (\text{floor } r)) = \text{floor } r$   
**unfolding** *real-of-int-def* **by** *simp*

**lemma** *floor-ceiling* [simp]:  $\text{floor } (\text{real } (\text{ceiling } r)) = \text{ceiling } r$   
**unfolding** *real-of-int-def* **by** *simp*

**lemma** *real-of-int-ceiling-ge* [simp]:  $r \leq \text{real } (\text{ceiling } r)$   
**unfolding** *real-of-int-def* **by** (rule *le-of-int-ceiling*)

**lemma** *ceiling-real-of-int* [simp]:  $\text{ceiling } (\text{real } (n :: \text{int})) = n$   
**unfolding** *real-of-int-def* **by** *simp*

**lemma** *real-of-int-ceiling-cancel* [simp]:  
 $(\text{real } (\text{ceiling } x) = x) = (\exists n :: \text{int}. x = \text{real } n)$   
**using** *ceiling-real-of-int* **by** *metis*

**lemma** *ceiling-eq*:  $[\text{real } n < x; x < \text{real } n + 1] \implies \text{ceiling } x = n + 1$   
**unfolding** *real-of-int-def* **using** *ceiling-unique* [of  $n + 1$   $x$ ] **by** *simp*

**lemma** *ceiling-eq2*:  $[\text{real } n < x; x \leq \text{real } n + 1] \implies \text{ceiling } x = n + 1$   
**unfolding** *real-of-int-def* **using** *ceiling-unique* [of  $n + 1$   $x$ ] **by** *simp*

**lemma** *ceiling-eq3*:  $[\text{real } n - 1 < x; x \leq \text{real } n] \implies \text{ceiling } x = n$   
**unfolding** *real-of-int-def* **using** *ceiling-unique* [of  $n$   $x$ ] **by** *simp*

**lemma** *real-of-int-ceiling-diff-one-le* [simp]:  $\text{real } (\text{ceiling } r) - 1 \leq r$   
**unfolding** *real-of-int-def* **using** *ceiling-correct* [of  $r$ ] **by** *simp*

**lemma** *real-of-int-ceiling-le-add-one* [simp]:  $\text{real } (\text{ceiling } r) \leq r + 1$   
**unfolding** *real-of-int-def* **using** *ceiling-correct* [of  $r$ ] **by** *simp*

**lemma** *ceiling-le*:  $x \leq \text{real } a \implies \text{ceiling } x \leq a$   
**unfolding** *real-of-int-def* **by** (*simp add: ceiling-le-iff*)

**lemma** *ceiling-le-real*:  $\text{ceiling } x \leq a \implies x \leq \text{real } a$   
**unfolding** *real-of-int-def* **by** (*simp add: ceiling-le-iff*)

**lemma** *ceiling-le-eq*:  $(\text{ceiling } x \leq a) = (x \leq \text{real } a)$   
**unfolding** *real-of-int-def* **by** (*rule ceiling-le-iff*)

**lemma** *less-ceiling-eq*:  $(a < \text{ceiling } x) = (\text{real } a < x)$   
**unfolding** *real-of-int-def* **by** (*rule less-ceiling-iff*)

**lemma** *ceiling-less-eq*:  $(\text{ceiling } x < a) = (x \leq \text{real } a - 1)$   
**unfolding** *real-of-int-def* **by** (*rule ceiling-less-iff*)

**lemma** *le-ceiling-eq*:  $(a \leq \text{ceiling } x) = (\text{real } a - 1 < x)$   
**unfolding** *real-of-int-def* **by** (*rule le-ceiling-iff*)

**lemma** *ceiling-add* [*simp*]:  $\text{ceiling } (x + \text{real } a) = \text{ceiling } x + a$   
**unfolding** *real-of-int-def* **by** (*rule ceiling-add-of-int*)

**lemma** *ceiling-subtract* [*simp*]:  $\text{ceiling } (x - \text{real } a) = \text{ceiling } x - a$   
**unfolding** *real-of-int-def* **by** (*rule ceiling-diff-of-int*)

## 67.5 Versions for the natural numbers

### definition

*natfloor* :: *real* => *nat* **where**  
*natfloor* *x* = *nat*(*floor* *x*)

### definition

*natceiling* :: *real* => *nat* **where**  
*natceiling* *x* = *nat*(*ceiling* *x*)

**lemma** *natfloor-zero* [*simp*]: *natfloor* 0 = 0  
**by** (*unfold natfloor-def, simp*)

**lemma** *natfloor-one* [*simp*]: *natfloor* 1 = 1  
**by** (*unfold natfloor-def, simp*)

**lemma** *zero-le-natfloor* [*simp*]: 0 ≤ *natfloor* *x*  
**by** (*unfold natfloor-def, simp*)

**lemma** *natfloor-number-of-eq* [*simp*]: *natfloor* (*number-of* *n*) = *number-of* *n*  
**by** (*unfold natfloor-def, simp*)

**lemma** *natfloor-real-of-nat* [*simp*]: *natfloor*(*real* *n*) = *n*  
**by** (*unfold natfloor-def, simp*)

**lemma** *real-natfloor-le*:  $0 \leq x \implies \text{real}(\text{natfloor } x) \leq x$   
**by** (*unfold natfloor-def*, *simp*)

**lemma** *natfloor-neg*:  $x \leq 0 \implies \text{natfloor } x = 0$   
**apply** (*unfold natfloor-def*)  
**apply** (*subgoal-tac floor x ≤ floor 0*)  
**apply** *simp*  
**apply** (*erule floor-mono*)  
**done**

**lemma** *natfloor-mono*:  $x \leq y \implies \text{natfloor } x \leq \text{natfloor } y$   
**apply** (*case-tac 0 ≤ x*)  
**apply** (*subst natfloor-def*)  
**apply** (*subst nat-le-eq-zle*)  
**apply** *force*  
**apply** (*erule floor-mono*)  
**apply** (*subst natfloor-neg*)  
**apply** *simp*  
**apply** *simp*  
**done**

**lemma** *le-natfloor*:  $\text{real } x \leq a \implies x \leq \text{natfloor } a$   
**apply** (*unfold natfloor-def*)  
**apply** (*subst nat-int [THEN sym]*)  
**apply** (*subst nat-le-eq-zle*)  
**apply** *simp*  
**apply** (*rule le-floor*)  
**apply** *simp*  
**done**

**lemma** *less-natfloor*:  
**assumes**  $0 \leq x$  **and**  $x < \text{real } (n :: \text{nat})$   
**shows**  $\text{natfloor } x < n$   
**proof** (*rule ccontr*)  
**assume**  $\neg ?thesis$  **hence**  $*: n \leq \text{natfloor } x$  **by** *simp*  
**note** *assms(2)*  
**also have**  $\text{real } n \leq \text{real } (\text{natfloor } x)$   
**using**  $*$  **unfolding** *real-of-nat-le-iff* .  
**finally have**  $x < \text{real } (\text{natfloor } x)$  .  
**with** *real-natfloor-le[OF assms(1)]*  
**show** *False* **by** *auto*  
**qed**

**lemma** *le-natfloor-eq*:  $0 \leq x \implies (a \leq \text{natfloor } x) = (\text{real } a \leq x)$   
**apply** (*rule iffI*)  
**apply** (*rule order-trans*)  
**prefer** 2  
**apply** (*erule real-natfloor-le*)  
**apply** (*subst real-of-nat-le-iff*)

```

  apply assumption
  apply (erule le-natfloor)
done

```

```

lemma le-natfloor-eq-number-of [simp]:
  ~ neg((number-of n)::int) ==> 0 <= x ==>
    (number-of n <= natfloor x) = (number-of n <= x)
  apply (subst le-natfloor-eq, assumption)
  apply simp
done

```

```

lemma le-natfloor-eq-one [simp]: (1 <= natfloor x) = (1 <= x)
  apply (case-tac 0 <= x)
  apply (subst le-natfloor-eq, assumption, simp)
  apply (rule iffI)
  apply (subgoal-tac natfloor x <= natfloor 0)
  apply simp
  apply (rule natfloor-mono)
  apply simp
  apply simp
done

```

```

lemma natfloor-eq: real n <= x ==> x < real n + 1 ==> natfloor x = n
  apply (unfold natfloor-def)
  apply (subst (2) nat-int [THEN sym])
  apply (subst eq-nat-nat-iff)
  apply simp
  apply simp
  apply (rule floor-eq2)
  apply auto
done

```

```

lemma real-natfloor-add-one-gt: x < real(natfloor x) + 1
  apply (case-tac 0 <= x)
  apply (unfold natfloor-def)
  apply simp
  apply simp-all
done

```

```

lemma real-natfloor-gt-diff-one: x - 1 < real(natfloor x)
using real-natfloor-add-one-gt by (simp add: algebra-simps)

```

```

lemma ge-natfloor-plus-one-imp-gt: natfloor z + 1 <= n ==> z < real n
  apply (subgoal-tac z < real(natfloor z) + 1)
  apply arith
  apply (rule real-natfloor-add-one-gt)
done

```

```

lemma natfloor-add [simp]: 0 <= x ==> natfloor (x + real a) = natfloor x + a

```

```

apply (unfold natfloor-def)
apply (subgoal-tac real a = real (int a))
apply (erule ssubst)
apply (simp add: nat-add-distrib del: real-of-int-of-nat-eq)
apply simp
done

```

```

lemma natfloor-add-number-of [simp]:
  ~neg ((number-of n)::int) ==> 0 <= x ==>
    natfloor (x + number-of n) = natfloor x + number-of n
apply (subst natfloor-add [THEN sym])
apply simp-all
done

```

```

lemma natfloor-add-one: 0 <= x ==> natfloor(x + 1) = natfloor x + 1
apply (subst natfloor-add [THEN sym])
apply assumption
apply simp
done

```

```

lemma natfloor-subtract [simp]: real a <= x ==>
  natfloor(x - real a) = natfloor x - a
apply (unfold natfloor-def)
apply (subgoal-tac real a = real (int a))
apply (erule ssubst)
apply (simp del: real-of-int-of-nat-eq)
apply simp
done

```

```

lemma natfloor-div-nat: 1 <= x ==> y > 0 ==>
  natfloor (x / real y) = natfloor x div y
proof -
  assume 1 <= (x::real) and (y::nat) > 0
  have natfloor x = (natfloor x) div y * y + (natfloor x) mod y
    by simp
  then have a: real(natfloor x) = real ((natfloor x) div y) * real y +
    real((natfloor x) mod y)
    by (simp only: real-of-nat-add [THEN sym] real-of-nat-mult [THEN sym])
  have x = real(natfloor x) + (x - real(natfloor x))
    by simp
  then have x = real ((natfloor x) div y) * real y +
    real((natfloor x) mod y) + (x - real(natfloor x))
    by (simp add: a)
  then have x / real y = ... / real y
    by simp
  also have ... = real((natfloor x) div y) + real((natfloor x) mod y) /
    real y + (x - real(natfloor x)) / real y
    by (auto simp add: algebra-simps add-divide-distrib
      diff-divide-distrib prems)

```

```

finally have  $\text{natfloor } (x / \text{real } y) = \text{natfloor}(\dots)$  by simp
also have  $\dots = \text{natfloor}(\text{real}((\text{natfloor } x) \bmod y) /$ 
 $\text{real } y + (x - \text{real}(\text{natfloor } x)) / \text{real } y + \text{real}((\text{natfloor } x) \text{ div } y))$ 
by (simp add: add-ac)
also have  $\dots = \text{natfloor}(\text{real}((\text{natfloor } x) \bmod y) /$ 
 $\text{real } y + (x - \text{real}(\text{natfloor } x)) / \text{real } y) + (\text{natfloor } x) \text{ div } y$ 
apply (rule natfloor-add)
apply (rule add-nonneg-nonneg)
apply (rule divide-nonneg-pos)
apply simp
apply (simp add: prems)
apply (rule divide-nonneg-pos)
apply (simp add: algebra-simps)
apply (rule real-natfloor-le)
apply (insert prems, auto)
done
also have  $\text{natfloor}(\text{real}((\text{natfloor } x) \bmod y) /$ 
 $\text{real } y + (x - \text{real}(\text{natfloor } x)) / \text{real } y) = 0$ 
apply (rule natfloor-eq)
apply simp
apply (rule add-nonneg-nonneg)
apply (rule divide-nonneg-pos)
apply force
apply (force simp add: prems)
apply (rule divide-nonneg-pos)
apply (simp add: algebra-simps)
apply (rule real-natfloor-le)
apply (auto simp add: prems)
apply (insert prems, arith)
apply (simp add: add-divide-distrib [THEN sym])
apply (subgoal-tac real y = real y - 1 + 1)
apply (erule ssubst)
apply (rule add-le-less-mono)
apply (simp add: algebra-simps)
apply (subgoal-tac 1 + real(natfloor x mod y) =
 $\text{real}(\text{natfloor } x \bmod y + 1)$ )
apply (erule ssubst)
apply (subst real-of-nat-le-iff)
apply (subgoal-tac natfloor x mod y < y)
apply arith
apply (rule mod-less-divisor)
apply auto
using real-natfloor-add-one-gt
apply (simp add: algebra-simps)
done
finally show ?thesis by simp
qed

```

**lemma** *le-mult-natfloor*:



```

assumes  $0 \leq (a :: \text{real})$  and  $0 \leq b$ 
shows  $\text{natfloor } a * \text{natfloor } b \leq \text{natfloor } (a * b)$ 
unfolding natfloor-def
apply (subst nat-mult-distrib[symmetric])
using assms apply simp
apply (subst nat-le-eq-zle)
using assms le-mult-floor by (auto intro!: mult-nonneg-nonneg)

lemma natceiling-zero [simp]:  $\text{natceiling } 0 = 0$ 
by (unfold natceiling-def, simp)

lemma natceiling-one [simp]:  $\text{natceiling } 1 = 1$ 
by (unfold natceiling-def, simp)

lemma zero-le-natceiling [simp]:  $0 \leq \text{natceiling } x$ 
by (unfold natceiling-def, simp)

lemma natceiling-number-of-eq [simp]:  $\text{natceiling } (\text{number-of } n) = \text{number-of } n$ 
by (unfold natceiling-def, simp)

lemma natceiling-real-of-nat [simp]:  $\text{natceiling}(\text{real } n) = n$ 
by (unfold natceiling-def, simp)

lemma real-natceiling-ge:  $x \leq \text{real}(\text{natceiling } x)$ 
apply (unfold natceiling-def)
apply (case-tac x < 0)
apply simp
apply (subst real-nat-eq-real)
apply (subgoal-tac ceiling 0 ≤ ceiling x)
apply simp
apply (rule ceiling-mono)
apply simp
apply simp
done

lemma natceiling-neg:  $x \leq 0 \implies \text{natceiling } x = 0$ 
apply (unfold natceiling-def)
apply simp
done

lemma natceiling-mono:  $x \leq y \implies \text{natceiling } x \leq \text{natceiling } y$ 
apply (case-tac 0 ≤ x)
apply (subst natceiling-def)+
apply (subst nat-le-eq-zle)
apply (rule disjI2)
apply (subgoal-tac real (0::int) ≤ real(ceiling y))
apply simp
apply (rule order-trans)
apply simp

```

```

  apply (erule order-trans)
  apply simp
  apply (erule ceiling-mono)
  apply (subst natceiling-neg)
  apply simp-all
done

```

```

lemma natceiling-le:  $x \leq \text{real } a \implies \text{natceiling } x \leq a$ 
  apply (unfold natceiling-def)
  apply (case-tac  $x < 0$ )
  apply simp
  apply (subst (2) nat-int [THEN sym])
  apply (subst nat-le-eq-zle)
  apply simp
  apply (rule ceiling-le)
  apply simp
done

```

```

lemma natceiling-le-eq:  $0 \leq x \implies (\text{natceiling } x \leq a) = (x \leq \text{real } a)$ 
  apply (rule iffI)
  apply (rule order-trans)
  apply (rule real-natceiling-ge)
  apply (subst real-of-nat-le-iff)
  apply assumption
  apply (erule natceiling-le)
done

```

```

lemma natceiling-le-eq-number-of [simp]:
   $\sim \text{neg}((\text{number-of } n)::\text{int}) \implies 0 \leq x \implies$ 
   $(\text{natceiling } x \leq \text{number-of } n) = (x \leq \text{number-of } n)$ 
  apply (subst natceiling-le-eq, assumption)
  apply simp
done

```

```

lemma natceiling-le-eq-one:  $(\text{natceiling } x \leq 1) = (x \leq 1)$ 
  apply (case-tac  $0 \leq x$ )
  apply (subst natceiling-le-eq)
  apply assumption
  apply simp
  apply (subst natceiling-neg)
  apply simp
  apply simp
done

```

```

lemma natceiling-eq:  $\text{real } n < x \implies x \leq \text{real } n + 1 \implies \text{natceiling } x = n + 1$ 
  apply (unfold natceiling-def)
  apply (simplesubst nat-int [THEN sym]) back back
  apply (subgoal-tac  $\text{nat}(\text{int } n) + 1 = \text{nat}(\text{int } n + 1)$ )

```

```

apply (erule ssubst)
apply (subst eq-nat-nat-iff)
apply (subgoal-tac ceiling 0 <= ceiling x)
apply simp
apply (rule ceiling-mono)
apply force
apply force
apply (rule ceiling-eq2)
apply (simp, simp)
apply (subst nat-add-distrib)
apply auto
done

```

```

lemma natceiling-add [simp]: 0 <= x ==>
  natceiling (x + real a) = natceiling x + a
apply (unfold natceiling-def)
apply (subgoal-tac real a = real (int a))
apply (erule ssubst)
apply (simp del: real-of-int-of-nat-eq)
apply (subst nat-add-distrib)
apply (subgoal-tac 0 = ceiling 0)
apply (erule ssubst)
apply (erule ceiling-mono)
apply simp-all
done

```

```

lemma natceiling-add-number-of [simp]:
  ~ neg ((number-of n)::int) ==> 0 <= x ==>
  natceiling (x + number-of n) = natceiling x + number-of n
apply (subst natceiling-add [THEN sym])
apply simp-all
done

```

```

lemma natceiling-add-one: 0 <= x ==> natceiling(x + 1) = natceiling x + 1
apply (subst natceiling-add [THEN sym])
apply assumption
apply simp
done

```

```

lemma natceiling-subtract [simp]: real a <= x ==>
  natceiling(x - real a) = natceiling x - a
apply (unfold natceiling-def)
apply (subgoal-tac real a = real (int a))
apply (erule ssubst)
apply (simp del: real-of-int-of-nat-eq)
apply simp
done

```

## 67.6 Exponentiation with floor

**lemma** *floor-power*:

**assumes**  $x = \text{real } (\text{floor } x)$

**shows**  $\text{floor } (x \wedge n) = \text{floor } x \wedge n$

**proof** –

**have**  $*$ :  $x \wedge n = \text{real } (\text{floor } x \wedge n)$

**using** *assms* **by** (*induct n arbitrary: x*) *simp-all*

**show** *?thesis* **unfolding** *real-of-int-inject[symmetric]*

**unfolding**  $*$  *floor-real-of-int ..*

**qed**

**lemma** *natfloor-power*:

**assumes**  $x = \text{real } (\text{natfloor } x)$

**shows**  $\text{natfloor } (x \wedge n) = \text{natfloor } x \wedge n$

**proof** –

**from** *assms* **have**  $0 \leq \text{floor } x$  **by** *auto*

**note** *assms[unfolded natfloor-def real-nat-eq-real[OF  $\langle 0 \leq \text{floor } x \rangle$ ]]*

**from** *floor-power[OF this]*

**show** *?thesis* **unfolding** *natfloor-def nat-power-eq[OF  $\langle 0 \leq \text{floor } x \rangle$ , symmetric]*

**by** *simp*

**qed**

**end**

## 68 RealVector: Vector Spaces and Algebras over the Reals

**theory** *RealVector*

**imports** *RComplete*

**begin**

### 68.1 Locale for additive functions

**locale** *additive* =

**fixes**  $f :: 'a::\text{ab-group-add} \Rightarrow 'b::\text{ab-group-add}$

**assumes** *add*:  $f (x + y) = f x + f y$

**begin**

**lemma** *zero*:  $f 0 = 0$

**proof** –

**have**  $f 0 = f (0 + 0)$  **by** *simp*

**also have**  $\dots = f 0 + f 0$  **by** (*rule add*)

**finally show**  $f 0 = 0$  **by** *simp*

**qed**

**lemma** *minus*:  $f (- x) = - f x$

**proof** –

```

  have  $f (- x) + f x = f (- x + x)$  by (rule add [symmetric])
  also have  $\dots = - f x + f x$  by (simp add: zero)
  finally show  $f (- x) = - f x$  by (rule add-right-imp-eq)
qed

```

```

lemma diff:  $f (x - y) = f x - f y$ 
by (simp add: diff-def add minus)

```

```

lemma setsum:  $f (\text{setsum } g \ A) = (\sum x \in A. f (g \ x))$ 
apply (cases finite A)
apply (induct set: finite)
apply (simp add: zero)
apply (simp add: add)
apply (simp add: zero)
done

```

```

end

```

## 68.2 Vector spaces

```

locale vector-space =
  fixes scale :: 'a::field  $\Rightarrow$  'b::ab-group-add  $\Rightarrow$  'b
  assumes scale-right-distrib [algebra-simps]:
     $\text{scale } a (x + y) = \text{scale } a \ x + \text{scale } a \ y$ 
  and scale-left-distrib [algebra-simps]:
     $\text{scale } (a + b) \ x = \text{scale } a \ x + \text{scale } b \ x$ 
  and scale-scale [simp]:  $\text{scale } a (\text{scale } b \ x) = \text{scale } (a * b) \ x$ 
  and scale-one [simp]:  $\text{scale } 1 \ x = x$ 
begin

```

```

lemma scale-left-commute:
   $\text{scale } a (\text{scale } b \ x) = \text{scale } b (\text{scale } a \ x)$ 
by (simp add: mult-commute)

```

```

lemma scale-zero-left [simp]:  $\text{scale } 0 \ x = 0$ 
and scale-minus-left [simp]:  $\text{scale } (- a) \ x = - (\text{scale } a \ x)$ 
and scale-left-diff-distrib [algebra-simps]:
   $\text{scale } (a - b) \ x = \text{scale } a \ x - \text{scale } b \ x$ 

```

```

proof -
  interpret s: additive  $\lambda a. \text{scale } a \ x$ 
  proof qed (rule scale-left-distrib)
  show  $\text{scale } 0 \ x = 0$  by (rule s.zero)
  show  $\text{scale } (- a) \ x = - (\text{scale } a \ x)$  by (rule s.minus)
  show  $\text{scale } (a - b) \ x = \text{scale } a \ x - \text{scale } b \ x$  by (rule s.diff)
qed

```

```

lemma scale-zero-right [simp]:  $\text{scale } a \ 0 = 0$ 
and scale-minus-right [simp]:  $\text{scale } a (- x) = - (\text{scale } a \ x)$ 
and scale-right-diff-distrib [algebra-simps]:

```

$\text{scale } a \ (x - y) = \text{scale } a \ x - \text{scale } a \ y$

**proof** –  
 interpret  $s$ : additive  $\lambda x. \text{scale } a \ x$   
**proof qed** (rule scale-right-distrib)  
 show  $\text{scale } a \ 0 = 0$  **by** (rule s.zero)  
 show  $\text{scale } a \ (-x) = -(\text{scale } a \ x)$  **by** (rule s.minus)  
 show  $\text{scale } a \ (x - y) = \text{scale } a \ x - \text{scale } a \ y$  **by** (rule s.diff)  
**qed**

**lemma** scale-eq-0-iff [simp]:  
 $\text{scale } a \ x = 0 \longleftrightarrow a = 0 \vee x = 0$   
**proof** cases  
 assume  $a = 0$  **thus** ?thesis **by** simp  
**next**  
 assume anz [simp]:  $a \neq 0$   
 { assume  $\text{scale } a \ x = 0$   
   hence  $\text{scale } (\text{inverse } a) \ (\text{scale } a \ x) = 0$  **by** simp  
   hence  $x = 0$  **by** simp }  
**thus** ?thesis **by** force  
**qed**

**lemma** scale-left-imp-eq:  
 $\llbracket a \neq 0; \text{scale } a \ x = \text{scale } a \ y \rrbracket \Longrightarrow x = y$   
**proof** –  
 assume nonzero:  $a \neq 0$   
 assume  $\text{scale } a \ x = \text{scale } a \ y$   
 hence  $\text{scale } a \ (x - y) = 0$   
   **by** (simp add: scale-right-diff-distrib)  
 hence  $x - y = 0$  **by** (simp add: nonzero)  
**thus**  $x = y$  **by** (simp only: right-minus-eq)  
**qed**

**lemma** scale-right-imp-eq:  
 $\llbracket x \neq 0; \text{scale } a \ x = \text{scale } b \ x \rrbracket \Longrightarrow a = b$   
**proof** –  
 assume nonzero:  $x \neq 0$   
 assume  $\text{scale } a \ x = \text{scale } b \ x$   
 hence  $\text{scale } (a - b) \ x = 0$   
   **by** (simp add: scale-left-diff-distrib)  
 hence  $a - b = 0$  **by** (simp add: nonzero)  
**thus**  $a = b$  **by** (simp only: right-minus-eq)  
**qed**

**lemma** scale-cancel-left [simp]:  
 $\text{scale } a \ x = \text{scale } a \ y \longleftrightarrow x = y \vee a = 0$   
**by** (auto intro: scale-left-imp-eq)

**lemma** scale-cancel-right [simp]:  
 $\text{scale } a \ x = \text{scale } b \ x \longleftrightarrow a = b \vee x = 0$

by (auto intro: scale-right-imp-eq)

end

### 68.3 Real vector spaces

```
class scaleR =
  fixes scaleR :: real  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr *R 75)
begin
```

abbreviation

```
divideR :: 'a  $\Rightarrow$  real  $\Rightarrow$  'a (infixl '/R 70)
```

where

```
x /R r == scaleR (inverse r) x
```

end

```
class real-vector = scaleR + ab-group-add +
  assumes scaleR-right-distrib: scaleR a (x + y) = scaleR a x + scaleR a y
  and scaleR-left-distrib: scaleR (a + b) x = scaleR a x + scaleR b x
  and scaleR-scaleR: scaleR a (scaleR b x) = scaleR (a * b) x
  and scaleR-one: scaleR 1 x = x
```

interpretation real-vector:

```
vector-space scaleR :: real  $\Rightarrow$  'a  $\Rightarrow$  'a::real-vector
```

apply unfold-locales

apply (rule scaleR-right-distrib)

apply (rule scaleR-left-distrib)

apply (rule scaleR-scaleR)

apply (rule scaleR-one)

done

Recover original theorem names

```
lemmas scaleR-left-commute = real-vector.scale-left-commute
```

```
lemmas scaleR-zero-left = real-vector.scale-zero-left
```

```
lemmas scaleR-minus-left = real-vector.scale-minus-left
```

```
lemmas scaleR-left-diff-distrib = real-vector.scale-left-diff-distrib
```

```
lemmas scaleR-zero-right = real-vector.scale-zero-right
```

```
lemmas scaleR-minus-right = real-vector.scale-minus-right
```

```
lemmas scaleR-right-diff-distrib = real-vector.scale-right-diff-distrib
```

```
lemmas scaleR-eq-0-iff = real-vector.scale-eq-0-iff
```

```
lemmas scaleR-left-imp-eq = real-vector.scale-left-imp-eq
```

```
lemmas scaleR-right-imp-eq = real-vector.scale-right-imp-eq
```

```
lemmas scaleR-cancel-left = real-vector.scale-cancel-left
```

```
lemmas scaleR-cancel-right = real-vector.scale-cancel-right
```

lemma scaleR-minus1-left [simp]:

```
fixes x :: 'a::real-vector
```

```
shows scaleR (-1) x = - x
```

```

using scaleR-minus-left [of 1 x] by simp

class real-algebra = real-vector + ring +
  assumes mult-scaleR-left [simp]: scaleR a x * y = scaleR a (x * y)
  and mult-scaleR-right [simp]: x * scaleR a y = scaleR a (x * y)

class real-algebra-1 = real-algebra + ring-1

class real-div-algebra = real-algebra-1 + division-ring

class real-field = real-div-algebra + field

instantiation real :: real-field
begin

definition
  real-scaleR-def [simp]: scaleR a x = a * x

instance proof
qed (simp-all add: algebra-simps)

end

interpretation scaleR-left: additive ( $\lambda a. \text{scaleR } a \ x :: 'a :: \text{real-vector}$ )
proof qed (rule scaleR-left-distrib)

interpretation scaleR-right: additive ( $\lambda x. \text{scaleR } a \ x :: 'a :: \text{real-vector}$ )
proof qed (rule scaleR-right-distrib)

lemma nonzero-inverse-scaleR-distrib:
  fixes x :: 'a::real-div-algebra shows
   $\llbracket a \neq 0; x \neq 0 \rrbracket \implies \text{inverse} (\text{scaleR } a \ x) = \text{scaleR} (\text{inverse } a) (\text{inverse } x)$ 
by (rule inverse-unique, simp)

lemma inverse-scaleR-distrib:
  fixes x :: 'a::{real-div-algebra, division-ring-inverse-zero}
  shows  $\text{inverse} (\text{scaleR } a \ x) = \text{scaleR} (\text{inverse } a) (\text{inverse } x)$ 
apply (case-tac a = 0, simp)
apply (case-tac x = 0, simp)
apply (erule (1) nonzero-inverse-scaleR-distrib)
done

```

## 68.4 Embedding of the Reals into any *real-algebra-1*: *of-real*

**definition**  
*of-real* :: *real*  $\Rightarrow$  '*a*::*real-algebra-1* **where**  
*of-real r* = *scaleR r 1*

**lemma** *scaleR-conv-of-real*: *scaleR r x = of-real r \* x*



**by** (*simp add: of-real-def*)

**lemma** *of-real-0* [*simp*]: *of-real 0 = 0*  
**by** (*simp add: of-real-def*)

**lemma** *of-real-1* [*simp*]: *of-real 1 = 1*  
**by** (*simp add: of-real-def*)

**lemma** *of-real-add* [*simp*]: *of-real (x + y) = of-real x + of-real y*  
**by** (*simp add: of-real-def scaleR-left-distrib*)

**lemma** *of-real-minus* [*simp*]: *of-real (- x) = - of-real x*  
**by** (*simp add: of-real-def*)

**lemma** *of-real-diff* [*simp*]: *of-real (x - y) = of-real x - of-real y*  
**by** (*simp add: of-real-def scaleR-left-diff-distrib*)

**lemma** *of-real-mult* [*simp*]: *of-real (x \* y) = of-real x \* of-real y*  
**by** (*simp add: of-real-def mult-commute*)

**lemma** *nonzero-of-real-inverse*:  
 $x \neq 0 \implies \text{of-real (inverse } x) =$   
 $\text{inverse (of-real } x :: 'a::\text{real-div-algebra})}$   
**by** (*simp add: of-real-def nonzero-inverse-scaleR-distrib*)

**lemma** *of-real-inverse* [*simp*]:  
 $\text{of-real (inverse } x) =$   
 $\text{inverse (of-real } x :: 'a::\{\text{real-div-algebra, division-ring-inverse-zero}\})}$   
**by** (*simp add: of-real-def inverse-scaleR-distrib*)

**lemma** *nonzero-of-real-divide*:  
 $y \neq 0 \implies \text{of-real (x / y) =}$   
 $(\text{of-real } x / \text{of-real } y :: 'a::\text{real-field})$   
**by** (*simp add: divide-inverse nonzero-of-real-inverse*)

**lemma** *of-real-divide* [*simp*]:  
 $\text{of-real (x / y) =}$   
 $(\text{of-real } x / \text{of-real } y :: 'a::\{\text{real-field, field-inverse-zero}\})$   
**by** (*simp add: divide-inverse*)

**lemma** *of-real-power* [*simp*]:  
 $\text{of-real (x ^ n) = (of-real } x :: 'a::\{\text{real-algebra-1}\}) ^ n$   
**by** (*induct n*) *simp-all*

**lemma** *of-real-eq-iff* [*simp*]:  $(\text{of-real } x = \text{of-real } y) = (x = y)$   
**by** (*simp add: of-real-def*)

**lemmas** *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff [of - 0, simplified]*

```

lemma of-real-eq-id [simp]: of-real = (id :: real  $\Rightarrow$  real)
proof
  fix r
  show of-real r = id r
    by (simp add: of-real-def)
qed

```

Collapse nested embeddings

```

lemma of-real-of-nat-eq [simp]: of-real (of-nat n) = of-nat n
by (induct n) auto

```

```

lemma of-real-of-int-eq [simp]: of-real (of-int z) = of-int z
by (cases z rule: int-diff-cases, simp)

```

```

lemma of-real-number-of-eq:
  of-real (number-of w) = (number-of w :: 'a::{number-ring, real-algebra-1})
by (simp add: number-of-eq)

```

Every real algebra has characteristic zero

```

instance real-algebra-1 < ring-char-0
proof
  fix m n :: nat
  have (of-real (of-nat m) = (of-real (of-nat n)::'a)) = (m = n)
    by (simp only: of-real-eq-iff of-nat-eq-iff)
  thus (of-nat m = (of-nat n::'a)) = (m = n)
    by (simp only: of-real-of-nat-eq)
qed

```

```

instance real-field < field-char-0 ..

```

## 68.5 The Set of Real Numbers

### definition

*Reals* :: 'a::real-algebra-1 set **where**  
 [code del]: *Reals* = range of-real

### notation (*xsymbols*)

*Reals* ( $\mathbb{R}$ )

```

lemma Reals-of-real [simp]: of-real r  $\in$  Reals
by (simp add: Reals-def)

```

```

lemma Reals-of-int [simp]: of-int z  $\in$  Reals
by (subst of-real-of-int-eq [symmetric], rule Reals-of-real)

```

```

lemma Reals-of-nat [simp]: of-nat n  $\in$  Reals
by (subst of-real-of-nat-eq [symmetric], rule Reals-of-real)

```

```

lemma Reals-number-of [simp]:

```

(number-of w::'a::{number-ring,real-algebra-1})  $\in$  Reals  
 by (subst of-real-number-of-eq [symmetric], rule Reals-of-real)

**lemma** Reals-0 [simp]:  $0 \in$  Reals  
 apply (unfold Reals-def)  
 apply (rule range-eqI)  
 apply (rule of-real-0 [symmetric])  
 done

**lemma** Reals-1 [simp]:  $1 \in$  Reals  
 apply (unfold Reals-def)  
 apply (rule range-eqI)  
 apply (rule of-real-1 [symmetric])  
 done

**lemma** Reals-add [simp]:  $\llbracket a \in \text{Reals}; b \in \text{Reals} \rrbracket \implies a + b \in \text{Reals}$   
 apply (auto simp add: Reals-def)  
 apply (rule range-eqI)  
 apply (rule of-real-add [symmetric])  
 done

**lemma** Reals-minus [simp]:  $a \in \text{Reals} \implies -a \in \text{Reals}$   
 apply (auto simp add: Reals-def)  
 apply (rule range-eqI)  
 apply (rule of-real-minus [symmetric])  
 done

**lemma** Reals-diff [simp]:  $\llbracket a \in \text{Reals}; b \in \text{Reals} \rrbracket \implies a - b \in \text{Reals}$   
 apply (auto simp add: Reals-def)  
 apply (rule range-eqI)  
 apply (rule of-real-diff [symmetric])  
 done

**lemma** Reals-mult [simp]:  $\llbracket a \in \text{Reals}; b \in \text{Reals} \rrbracket \implies a * b \in \text{Reals}$   
 apply (auto simp add: Reals-def)  
 apply (rule range-eqI)  
 apply (rule of-real-mult [symmetric])  
 done

**lemma** nonzero-Reals-inverse:  
 fixes a :: 'a::real-div-algebra  
 shows  $\llbracket a \in \text{Reals}; a \neq 0 \rrbracket \implies \text{inverse } a \in \text{Reals}$   
 apply (auto simp add: Reals-def)  
 apply (rule range-eqI)  
 apply (erule nonzero-of-real-inverse [symmetric])  
 done

**lemma** Reals-inverse [simp]:  
 fixes a :: 'a::{real-div-algebra, division-ring-inverse-zero}

```

  shows  $a \in \text{Reals} \implies \text{inverse } a \in \text{Reals}$ 
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-inverse [symmetric])
done

```

```

lemma nonzero-Reals-divide:
  fixes  $a \ b :: 'a::\text{real-field}$ 
  shows  $\llbracket a \in \text{Reals}; b \in \text{Reals}; b \neq 0 \rrbracket \implies a / b \in \text{Reals}$ 
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (erule nonzero-of-real-divide [symmetric])
done

```

```

lemma Reals-divide [simp]:
  fixes  $a \ b :: 'a::\{\text{real-field}, \text{field-inverse-zero}\}$ 
  shows  $\llbracket a \in \text{Reals}; b \in \text{Reals} \rrbracket \implies a / b \in \text{Reals}$ 
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-divide [symmetric])
done

```

```

lemma Reals-power [simp]:
  fixes  $a :: 'a::\{\text{real-algebra-1}\}$ 
  shows  $a \in \text{Reals} \implies a ^ n \in \text{Reals}$ 
apply (auto simp add: Reals-def)
apply (rule range-eqI)
apply (rule of-real-power [symmetric])
done

```

```

lemma Reals-cases [cases set: Reals]:
  assumes  $q \in \mathbb{R}$ 
  obtains (of-real)  $r$  where  $q = \text{of-real } r$ 
  unfolding Reals-def
proof -
  from  $\langle q \in \mathbb{R} \rangle$  have  $q \in \text{range of-real}$  unfolding Reals-def .
  then obtain  $r$  where  $q = \text{of-real } r$  ..
  then show thesis ..
qed

```

```

lemma Reals-induct [case-names of-real, induct set: Reals]:
   $q \in \mathbb{R} \implies (\bigwedge r. P (\text{of-real } r)) \implies P q$ 
  by (rule Reals-cases) auto

```

## 68.6 Topological spaces

```

class open =
  fixes open :: 'a set  $\Rightarrow$  bool

```

```

class topological-space = open +
  assumes open-UNIV [simp, intro]: open UNIV
  assumes open-Int [intro]: open S  $\implies$  open T  $\implies$  open (S  $\cap$  T)
  assumes open-Union [intro]:  $\forall S \in K. \text{open } S \implies \text{open } (\bigcup K)$ 
begin

definition
  closed :: 'a set  $\implies$  bool where
    closed S  $\longleftrightarrow$  open ( $- S$ )

lemma open-empty [intro, simp]: open {}
  using open-Union [of {}] by simp

lemma open-Un [intro]: open S  $\implies$  open T  $\implies$  open (S  $\cup$  T)
  using open-Union [of {S, T}] by simp

lemma open-UN [intro]:  $\forall x \in A. \text{open } (B\ x) \implies \text{open } (\bigcup_{x \in A} B\ x)$ 
  unfolding UN-eq by (rule open-Union) auto

lemma open-INT [intro]: finite A  $\implies \forall x \in A. \text{open } (B\ x) \implies \text{open } (\bigcap_{x \in A} B\ x)$ 
  by (induct set: finite) auto

lemma open-Inter [intro]: finite S  $\implies \forall T \in S. \text{open } T \implies \text{open } (\bigcap S)$ 
  unfolding Inter-def by (rule open-INT)

lemma closed-empty [intro, simp]: closed {}
  unfolding closed-def by simp

lemma closed-Un [intro]: closed S  $\implies$  closed T  $\implies$  closed (S  $\cup$  T)
  unfolding closed-def by auto

lemma closed-Inter [intro]:  $\forall S \in K. \text{closed } S \implies \text{closed } (\bigcap K)$ 
  unfolding closed-def Inter-def by auto

lemma closed-UNIV [intro, simp]: closed UNIV
  unfolding closed-def by simp

lemma closed-Int [intro]: closed S  $\implies$  closed T  $\implies$  closed (S  $\cap$  T)
  unfolding closed-def by auto

lemma closed-INT [intro]:  $\forall x \in A. \text{closed } (B\ x) \implies \text{closed } (\bigcap_{x \in A} B\ x)$ 
  unfolding closed-def by auto

lemma closed-UN [intro]: finite A  $\implies \forall x \in A. \text{closed } (B\ x) \implies \text{closed } (\bigcup_{x \in A} B\ x)$ 
  by (induct set: finite) auto

lemma closed-Union [intro]: finite S  $\implies \forall T \in S. \text{closed } T \implies \text{closed } (\bigcup S)$ 
  unfolding Union-def by (rule closed-UN)

```

```

lemma open-closed: open  $S \longleftrightarrow$  closed  $(- S)$ 
  unfolding closed-def by simp

lemma closed-open: closed  $S \longleftrightarrow$  open  $(- S)$ 
  unfolding closed-def by simp

lemma open-Diff [intro]: open  $S \implies$  closed  $T \implies$  open  $(S - T)$ 
  unfolding closed-open Diff-eq by (rule open-Int)

lemma closed-Diff [intro]: closed  $S \implies$  open  $T \implies$  closed  $(S - T)$ 
  unfolding open-closed Diff-eq by (rule closed-Int)

lemma open-Compl [intro]: closed  $S \implies$  open  $(- S)$ 
  unfolding closed-open .

lemma closed-Compl [intro]: open  $S \implies$  closed  $(- S)$ 
  unfolding open-closed .

end

```

## 68.7 Metric spaces

```

class dist =
  fixes dist :: 'a  $\Rightarrow$  'a  $\Rightarrow$  real

class open-dist = open + dist +
  assumes open-dist: open  $S \longleftrightarrow (\forall x \in S. \exists e > 0. \forall y. \text{dist } y \ x < e \longrightarrow y \in S)$ 

class metric-space = open-dist +
  assumes dist-eq-0-iff [simp]: dist  $x \ y = 0 \longleftrightarrow x = y$ 
  assumes dist-triangle2: dist  $x \ y \leq$  dist  $x \ z +$  dist  $y \ z$ 
begin

lemma dist-self [simp]: dist  $x \ x = 0$ 
by simp

lemma zero-le-dist [simp]:  $0 \leq$  dist  $x \ y$ 
using dist-triangle2 [of  $x \ x \ y$ ] by simp

lemma zero-less-dist-iff:  $0 < \text{dist } x \ y \longleftrightarrow x \neq y$ 
by (simp add: less-le)

lemma dist-not-less-zero [simp]:  $\neg \text{dist } x \ y < 0$ 
by (simp add: not-less)

lemma dist-le-zero-iff [simp]: dist  $x \ y \leq 0 \longleftrightarrow x = y$ 
by (simp add: le-less)

```

```

lemma dist-commute:  $\text{dist } x \ y = \text{dist } y \ x$ 
proof (rule order-antisym)
  show  $\text{dist } x \ y \leq \text{dist } y \ x$ 
    using dist-triangle2 [of x y x] by simp
  show  $\text{dist } y \ x \leq \text{dist } x \ y$ 
    using dist-triangle2 [of y x y] by simp
qed

lemma dist-triangle:  $\text{dist } x \ z \leq \text{dist } x \ y + \text{dist } y \ z$ 
using dist-triangle2 [of x z y] by (simp add: dist-commute)

lemma dist-triangle3:  $\text{dist } x \ y \leq \text{dist } a \ x + \text{dist } a \ y$ 
using dist-triangle2 [of x y a] by (simp add: dist-commute)

subclass topological-space
proof
  have  $\exists e::\text{real}. \ 0 < e$ 
    by (fast intro: zero-less-one)
  then show open UNIV
    unfolding open-dist by simp
next
  fix S T assume open S open T
  then show open (S ∩ T)
    unfolding open-dist
    apply clarify
    apply (drule (1) bspec)+
    apply (clarify, rename-tac r s)
    apply (rule-tac x=min r s in exI, simp)
    done
next
  fix K assume  $\forall S \in K. \ \text{open } S$  thus open (⋃ K)
    unfolding open-dist by fast
qed

end

```

## 68.8 Real normed vector spaces

```

class norm =
  fixes norm :: 'a  $\Rightarrow$  real

class sgn-div-norm = scaleR + norm + sgn +
  assumes sgn-div-norm:  $\text{sgn } x = x /_R \text{ norm } x$ 

class dist-norm = dist + norm + minus +
  assumes dist-norm:  $\text{dist } x \ y = \text{norm } (x - y)$ 

class real-normed-vector = real-vector + sgn-div-norm + dist-norm + open-dist
+

```

```

assumes norm-ge-zero [simp]:  $0 \leq \text{norm } x$ 
and norm-eq-zero [simp]:  $\text{norm } x = 0 \longleftrightarrow x = 0$ 
and norm-triangle-ineq:  $\text{norm } (x + y) \leq \text{norm } x + \text{norm } y$ 
and norm-scaleR [simp]:  $\text{norm } (\text{scaleR } a \ x) = |a| * \text{norm } x$ 

class real-normed-algebra = real-algebra + real-normed-vector +
  assumes norm-mult-ineq:  $\text{norm } (x * y) \leq \text{norm } x * \text{norm } y$ 

class real-normed-algebra-1 = real-algebra-1 + real-normed-algebra +
  assumes norm-one [simp]:  $\text{norm } 1 = 1$ 

class real-normed-div-algebra = real-div-algebra + real-normed-vector +
  assumes norm-mult:  $\text{norm } (x * y) = \text{norm } x * \text{norm } y$ 

class real-normed-field = real-field + real-normed-div-algebra

instance real-normed-div-algebra < real-normed-algebra-1
proof
  fix x y :: 'a
  show  $\text{norm } (x * y) \leq \text{norm } x * \text{norm } y$ 
    by (simp add: norm-mult)
next
  have  $\text{norm } (1 * 1::'a) = \text{norm } (1::'a) * \text{norm } (1::'a)$ 
    by (rule norm-mult)
  thus  $\text{norm } (1::'a) = 1$  by simp
qed

lemma norm-zero [simp]:  $\text{norm } (0::'a::\text{real-normed-vector}) = 0$ 
by simp

lemma zero-less-norm-iff [simp]:
  fixes x :: 'a::real-normed-vector
  shows  $(0 < \text{norm } x) = (x \neq 0)$ 
by (simp add: order-less-le)

lemma norm-not-less-zero [simp]:
  fixes x :: 'a::real-normed-vector
  shows  $\neg \text{norm } x < 0$ 
by (simp add: linorder-not-less)

lemma norm-le-zero-iff [simp]:
  fixes x :: 'a::real-normed-vector
  shows  $(\text{norm } x \leq 0) = (x = 0)$ 
by (simp add: order-le-less)

lemma norm-minus-cancel [simp]:
  fixes x :: 'a::real-normed-vector
  shows  $\text{norm } (-x) = \text{norm } x$ 
proof –

```



```

have norm (- x) = norm (scaleR (- 1) x)
  by (simp only: scaleR-minus-left scaleR-one)
also have ... = |- 1| * norm x
  by (rule norm-scaleR)
finally show ?thesis by simp
qed

```

```

lemma norm-minus-commute:
  fixes a b :: 'a::real-normed-vector
  shows norm (a - b) = norm (b - a)
proof -
  have norm (- (b - a)) = norm (b - a)
    by (rule norm-minus-cancel)
  thus ?thesis by simp
qed

```

```

lemma norm-triangle-ineq2:
  fixes a b :: 'a::real-normed-vector
  shows norm a - norm b ≤ norm (a - b)
proof -
  have norm (a - b + b) ≤ norm (a - b) + norm b
    by (rule norm-triangle-ineq)
  thus ?thesis by simp
qed

```

```

lemma norm-triangle-ineq3:
  fixes a b :: 'a::real-normed-vector
  shows |norm a - norm b| ≤ norm (a - b)
apply (subst abs-le-iff)
apply auto
apply (rule norm-triangle-ineq2)
apply (subst norm-minus-commute)
apply (rule norm-triangle-ineq2)
done

```

```

lemma norm-triangle-ineq4:
  fixes a b :: 'a::real-normed-vector
  shows norm (a - b) ≤ norm a + norm b
proof -
  have norm (a + - b) ≤ norm a + norm (- b)
    by (rule norm-triangle-ineq)
  thus ?thesis
    by (simp only: diff-minus norm-minus-cancel)
qed

```

```

lemma norm-diff-ineq:
  fixes a b :: 'a::real-normed-vector
  shows norm a - norm b ≤ norm (a + b)
proof -

```

```

have norm a - norm (- b) ≤ norm (a - - b)
  by (rule norm-triangle-ineq2)
thus ?thesis by simp
qed

```

```

lemma norm-diff-triangle-ineq:
  fixes a b c d :: 'a::real-normed-vector
  shows norm ((a + b) - (c + d)) ≤ norm (a - c) + norm (b - d)
proof -
  have norm ((a + b) - (c + d)) = norm ((a - c) + (b - d))
    by (simp add: diff-minus add-ac)
  also have ... ≤ norm (a - c) + norm (b - d)
    by (rule norm-triangle-ineq)
  finally show ?thesis .
qed

```

```

lemma abs-norm-cancel [simp]:
  fixes a :: 'a::real-normed-vector
  shows |norm a| = norm a
by (rule abs-of-nonneg [OF norm-ge-zero])

```

```

lemma norm-add-less:
  fixes x y :: 'a::real-normed-vector
  shows [|norm x < r; norm y < s|] ==> norm (x + y) < r + s
by (rule order-le-less-trans [OF norm-triangle-ineq add-strict-mono])

```

```

lemma norm-mult-less:
  fixes x y :: 'a::real-normed-algebra
  shows [|norm x < r; norm y < s|] ==> norm (x * y) < r * s
apply (rule order-le-less-trans [OF norm-mult-ineq])
apply (simp add: mult-strict-mono')
done

```

```

lemma norm-of-real [simp]:
  norm (of-real r :: 'a::real-normed-algebra-1) = |r|
unfolding of-real-def by simp

```

```

lemma norm-number-of [simp]:
  norm (number-of w :: 'a::{number-ring, real-normed-algebra-1})
    = |number-of w|
by (subst of-real-number-of-eq [symmetric], rule norm-of-real)

```

```

lemma norm-of-int [simp]:
  norm (of-int z :: 'a::real-normed-algebra-1) = |of-int z|
by (subst of-real-of-int-eq [symmetric], rule norm-of-real)

```

```

lemma norm-of-nat [simp]:
  norm (of-nat n :: 'a::real-normed-algebra-1) = of-nat n
apply (subst of-real-of-nat-eq [symmetric])

```

**apply** (*subst norm-of-real, simp*)  
**done**

**lemma** *nonzero-norm-inverse*:  
**fixes**  $a :: 'a::\text{real-normed-div-algebra}$   
**shows**  $a \neq 0 \implies \text{norm } (\text{inverse } a) = \text{inverse } (\text{norm } a)$   
**apply** (*rule inverse-unique [symmetric]*)  
**apply** (*simp add: norm-mult [symmetric]*)  
**done**

**lemma** *norm-inverse*:  
**fixes**  $a :: 'a::\{\text{real-normed-div-algebra}, \text{division-ring-inverse-zero}\}$   
**shows**  $\text{norm } (\text{inverse } a) = \text{inverse } (\text{norm } a)$   
**apply** (*case-tac a = 0, simp*)  
**apply** (*erule nonzero-norm-inverse*)  
**done**

**lemma** *nonzero-norm-divide*:  
**fixes**  $a b :: 'a::\text{real-normed-field}$   
**shows**  $b \neq 0 \implies \text{norm } (a / b) = \text{norm } a / \text{norm } b$   
**by** (*simp add: divide-inverse norm-mult nonzero-norm-inverse*)

**lemma** *norm-divide*:  
**fixes**  $a b :: 'a::\{\text{real-normed-field}, \text{field-inverse-zero}\}$   
**shows**  $\text{norm } (a / b) = \text{norm } a / \text{norm } b$   
**by** (*simp add: divide-inverse norm-mult norm-inverse*)

**lemma** *norm-power-ineq*:  
**fixes**  $x :: 'a::\{\text{real-normed-algebra-1}\}$   
**shows**  $\text{norm } (x ^ n) \leq \text{norm } x ^ n$   
**proof** (*induct n*)  
**case 0** **show**  $\text{norm } (x ^ 0) \leq \text{norm } x ^ 0$  **by** *simp*  
**next**  
**case** (*Suc n*)  
**have**  $\text{norm } (x * x ^ n) \leq \text{norm } x * \text{norm } (x ^ n)$   
**by** (*rule norm-mult-ineq*)  
**also from** *Suc* **have**  $\dots \leq \text{norm } x * \text{norm } x ^ n$   
**using** *norm-ge-zero* **by** (*rule mult-left-mono*)  
**finally show**  $\text{norm } (x ^ \text{Suc } n) \leq \text{norm } x ^ \text{Suc } n$   
**by** *simp*  
**qed**

**lemma** *norm-power*:  
**fixes**  $x :: 'a::\{\text{real-normed-div-algebra}\}$   
**shows**  $\text{norm } (x ^ n) = \text{norm } x ^ n$   
**by** (*induct n*) (*simp-all add: norm-mult*)

Every normed vector space is a metric space.

**instance** *real-normed-vector*  $<$  *metric-space*

```

proof
  fix  $x\ y :: 'a$  show  $\text{dist } x\ y = 0 \iff x = y$ 
    unfolding dist-norm by simp
next
  fix  $x\ y\ z :: 'a$  show  $\text{dist } x\ y \leq \text{dist } x\ z + \text{dist } y\ z$ 
    unfolding dist-norm
    using norm-triangle-ineq4 [of x - z y - z] by simp
qed

```

## 68.9 Class instances for real numbers

```

instantiation real :: real-normed-field
begin

```

```

definition real-norm-def [simp]:
   $\text{norm } r = |r|$ 

```

```

definition dist-real-def:
   $\text{dist } x\ y = |x - y|$ 

```

```

definition open-real-def [code del]:
   $\text{open } (S :: \text{real set}) \iff (\forall x \in S. \exists e > 0. \forall y. \text{dist } y\ x < e \longrightarrow y \in S)$ 

```

```

instance
apply (intro-classes, unfold real-norm-def real-scaleR-def)
apply (rule dist-real-def)
apply (rule open-real-def)
apply (simp add: sgn-real-def)
apply (rule abs-ge-zero)
apply (rule abs-eq-0)
apply (rule abs-triangle-ineq)
apply (rule abs-mult)
apply (rule abs-mult)
done

```

```

end

```

```

lemma open-real-lessThan [simp]:
  fixes  $a :: \text{real}$  shows  $\text{open } \{..<a\}$ 
unfolding open-real-def dist-real-def
proof (clarify)
  fix  $x$  assume  $x < a$ 
  hence  $0 < a - x \wedge (\forall y. |y - x| < a - x \longrightarrow y \in \{..<a\})$  by auto
  thus  $\exists e > 0. \forall y. |y - x| < e \longrightarrow y \in \{..<a\}$  ..
qed

```

```

lemma open-real-greaterThan [simp]:
  fixes  $a :: \text{real}$  shows  $\text{open } \{a<..\}$ 
unfolding open-real-def dist-real-def

```

**proof** (*clarify*)  
**fix**  $x$  **assume**  $a < x$   
**hence**  $0 < x - a \wedge (\forall y. |y - x| < x - a \longrightarrow y \in \{a < ..\})$  **by** *auto*  
**thus**  $\exists e > 0. \forall y. |y - x| < e \longrightarrow y \in \{a < ..\}$  **..**  
**qed**

**lemma** *open-real-greaterThanLessThan* [*simp*]:  
**fixes**  $a\ b :: \text{real}$  **shows** *open*  $\{a < .. < b\}$   
**proof** –  
**have**  $\{a < .. < b\} = \{a < ..\} \cap \{.. < b\}$  **by** *auto*  
**thus** *open*  $\{a < .. < b\}$  **by** (*simp add: open-Int*)  
**qed**

**lemma** *closed-real-atMost* [*simp*]:  
**fixes**  $a :: \text{real}$  **shows** *closed*  $\{..a\}$   
**unfolding** *closed-open* **by** *simp*

**lemma** *closed-real-atLeast* [*simp*]:  
**fixes**  $a :: \text{real}$  **shows** *closed*  $\{a.. \}$   
**unfolding** *closed-open* **by** *simp*

**lemma** *closed-real-atLeastAtMost* [*simp*]:  
**fixes**  $a\ b :: \text{real}$  **shows** *closed*  $\{a..b\}$   
**proof** –  
**have**  $\{a..b\} = \{a.. \} \cap \{..b\}$  **by** *auto*  
**thus** *closed*  $\{a..b\}$  **by** (*simp add: closed-Int*)  
**qed**

## 68.10 Extra type constraints

Only allow *open* in class *topological-space*.

**setup**  $\ll$  *Sign.add-const-constraint*  
 ( $\text{@}\{const\text{-name } open\}, \text{SOME } \text{@}\{typ\ 'a::\text{topological-space set} \Rightarrow \text{bool}\}$ )  $\gg$

Only allow *dist* in class *metric-space*.

**setup**  $\ll$  *Sign.add-const-constraint*  
 ( $\text{@}\{const\text{-name } dist\}, \text{SOME } \text{@}\{typ\ 'a::\text{metric-space} \Rightarrow 'a \Rightarrow \text{real}\}$ )  $\gg$

Only allow *norm* in class *real-normed-vector*.

**setup**  $\ll$  *Sign.add-const-constraint*  
 ( $\text{@}\{const\text{-name } norm\}, \text{SOME } \text{@}\{typ\ 'a::\text{real-normed-vector} \Rightarrow \text{real}\}$ )  $\gg$

## 68.11 Sign function

**lemma** *norm-sgn*:  
 $norm\ (sgn(x::'a::\text{real-normed-vector})) = (if\ x = 0\ then\ 0\ else\ 1)$   
**by** (*simp add: sgn-div-norm*)

**lemma** *sgn-zero* [simp]:  $\text{sgn}(0::'a::\text{real-normed-vector}) = 0$   
**by** (simp add: *sgn-div-norm*)

**lemma** *sgn-zero-iff*:  $(\text{sgn}(x::'a::\text{real-normed-vector}) = 0) = (x = 0)$   
**by** (simp add: *sgn-div-norm*)

**lemma** *sgn-minus*:  $\text{sgn}(-x) = -\text{sgn}(x::'a::\text{real-normed-vector})$   
**by** (simp add: *sgn-div-norm*)

**lemma** *sgn-scaleR*:  
 $\text{sgn}(\text{scaleR } r \ x) = \text{scaleR } r \ (\text{sgn}(x::'a::\text{real-normed-vector}))$   
**by** (simp add: *sgn-div-norm mult-ac*)

**lemma** *sgn-one* [simp]:  $\text{sgn}(1::'a::\text{real-normed-algebra-1}) = 1$   
**by** (simp add: *sgn-div-norm*)

**lemma** *sgn-of-real*:  
 $\text{sgn}(\text{of-real } r::'a::\text{real-normed-algebra-1}) = \text{of-real } (\text{sgn } r)$   
**unfolding** *of-real-def* **by** (simp only: *sgn-scaleR sgn-one*)

**lemma** *sgn-mult*:  
**fixes**  $x \ y :: 'a::\text{real-normed-div-algebra}$   
**shows**  $\text{sgn}(x * y) = \text{sgn } x * \text{sgn } y$   
**by** (simp add: *sgn-div-norm norm-mult mult-commute*)

**lemma** *real-sgn-eq*:  $\text{sgn}(x::\text{real}) = x / |x|$   
**by** (simp add: *sgn-div-norm divide-inverse*)

**lemma** *real-sgn-pos*:  $0 < (x::\text{real}) \implies \text{sgn } x = 1$   
**unfolding** *real-sgn-eq* **by** *simp*

**lemma** *real-sgn-neg*:  $(x::\text{real}) < 0 \implies \text{sgn } x = -1$   
**unfolding** *real-sgn-eq* **by** *simp*

## 68.12 Bounded Linear and Bilinear Operators

**locale** *bounded-linear* = *additive* +  
**constrains**  $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{real-normed-vector}$   
**assumes** *scaleR*:  $f(\text{scaleR } r \ x) = \text{scaleR } r \ (f \ x)$   
**assumes** *bounded*:  $\exists K. \forall x. \text{norm } (f \ x) \leq \text{norm } x * K$   
**begin**

**lemma** *pos-bounded*:  
 $\exists K > 0. \forall x. \text{norm } (f \ x) \leq \text{norm } x * K$

**proof** –

**obtain**  $K$  **where**  $K: \bigwedge x. \text{norm } (f \ x) \leq \text{norm } x * K$   
**using** *bounded* **by** *fast*  
**show** *?thesis*  
**proof** (*intro exI impI conjI allI*)

```

  show  $0 < \max 1 K$ 
  by (rule order-less-le-trans [OF zero-less-one le-maxI1])
next
  fix  $x$ 
  have  $\text{norm } (f x) \leq \text{norm } x * K$  using  $K$  .
  also have  $\dots \leq \text{norm } x * \max 1 K$ 
  by (rule mult-left-mono [OF le-maxI2 norm-ge-zero])
  finally show  $\text{norm } (f x) \leq \text{norm } x * \max 1 K$  .
qed
qed

```

```

lemma nonneg-bounded:
   $\exists K \geq 0. \forall x. \text{norm } (f x) \leq \text{norm } x * K$ 
proof -
  from pos-bounded
  show ?thesis by (auto intro: order-less-imp-le)
qed

end

```

```

locale bounded-bilinear =
  fixes prod :: [ $'a::\text{real-normed-vector}$ ,  $'b::\text{real-normed-vector}$ ]
     $\Rightarrow 'c::\text{real-normed-vector}$ 
  (infixl ** 70)
  assumes add-left:  $\text{prod } (a + a') b = \text{prod } a b + \text{prod } a' b$ 
  assumes add-right:  $\text{prod } a (b + b') = \text{prod } a b + \text{prod } a b'$ 
  assumes scaleR-left:  $\text{prod } (\text{scaleR } r a) b = \text{scaleR } r (\text{prod } a b)$ 
  assumes scaleR-right:  $\text{prod } a (\text{scaleR } r b) = \text{scaleR } r (\text{prod } a b)$ 
  assumes bounded:  $\exists K. \forall a b. \text{norm } (\text{prod } a b) \leq \text{norm } a * \text{norm } b * K$ 
begin

```

```

lemma pos-bounded:
   $\exists K > 0. \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$ 
apply (cut-tac bounded, erule exE)
apply (rule-tac  $x = \max 1 K$  in exI, safe)
apply (rule order-less-le-trans [OF zero-less-one le-maxI1])
apply (drule spec, drule spec, erule order-trans)
apply (rule mult-left-mono [OF le-maxI2])
apply (intro mult-nonneg-nonneg norm-ge-zero)
done

```

```

lemma nonneg-bounded:
   $\exists K \geq 0. \forall a b. \text{norm } (a ** b) \leq \text{norm } a * \text{norm } b * K$ 
proof -
  from pos-bounded
  show ?thesis by (auto intro: order-less-imp-le)
qed

```

```

lemma additive-right: additive ( $\lambda b. \text{prod } a b$ )

```

**by** (*rule additive.intro*, *rule add-right*)

**lemma** *additive-left*: *additive* ( $\lambda a. \text{prod } a \ b$ )  
**by** (*rule additive.intro*, *rule add-left*)

**lemma** *zero-left*: *prod* 0 *b* = 0  
**by** (*rule additive.zero* [*OF additive-left*])

**lemma** *zero-right*: *prod* *a* 0 = 0  
**by** (*rule additive.zero* [*OF additive-right*])

**lemma** *minus-left*: *prod* ( $- \ a$ ) *b* =  $- \ \text{prod } a \ b$   
**by** (*rule additive.minus* [*OF additive-left*])

**lemma** *minus-right*: *prod* *a* ( $- \ b$ ) =  $- \ \text{prod } a \ b$   
**by** (*rule additive.minus* [*OF additive-right*])

**lemma** *diff-left*:  
 $\text{prod } (a - a') \ b = \text{prod } a \ b - \text{prod } a' \ b$   
**by** (*rule additive.diff* [*OF additive-left*])

**lemma** *diff-right*:  
 $\text{prod } a \ (b - b') = \text{prod } a \ b - \text{prod } a \ b'$   
**by** (*rule additive.diff* [*OF additive-right*])

**lemma** *bounded-linear-left*:  
 $\text{bounded-linear } (\lambda a. a \ ** \ b)$   
**apply** (*unfold-locales*)  
**apply** (*rule add-left*)  
**apply** (*rule scaleR-left*)  
**apply** (*cut-tac bounded, safe*)  
**apply** (*rule-tac x=norm b \* K in exI*)  
**apply** (*simp add: mult-ac*)  
**done**

**lemma** *bounded-linear-right*:  
 $\text{bounded-linear } (\lambda b. a \ ** \ b)$   
**apply** (*unfold-locales*)  
**apply** (*rule add-right*)  
**apply** (*rule scaleR-right*)  
**apply** (*cut-tac bounded, safe*)  
**apply** (*rule-tac x=norm a \* K in exI*)  
**apply** (*simp add: mult-ac*)  
**done**

**lemma** *prod-diff-prod*:  
 $(x \ ** \ y - a \ ** \ b) = (x - a) \ ** \ (y - b) + (x - a) \ ** \ b + a \ ** \ (y - b)$   
**by** (*simp add: diff-left diff-right*)



**end**

**interpretation** *mult*:

*bounded-bilinear*  $op * :: 'a \Rightarrow 'a \Rightarrow 'a::real-normed-algebra$   
**apply** (*rule* *bounded-bilinear.intro*)  
**apply** (*rule* *left-distrib*)  
**apply** (*rule* *right-distrib*)  
**apply** (*rule* *mult-scaleR-left*)  
**apply** (*rule* *mult-scaleR-right*)  
**apply** (*rule-tac*  $x=1$  **in** *exI*)  
**apply** (*simp* *add: norm-mult-ineq*)  
**done**

**interpretation** *mult-left*:

*bounded-linear*  $(\lambda x::'a::real-normed-algebra. x * y)$   
**by** (*rule* *mult.bounded-linear-left*)

**interpretation** *mult-right*:

*bounded-linear*  $(\lambda y::'a::real-normed-algebra. x * y)$   
**by** (*rule* *mult.bounded-linear-right*)

**interpretation** *divide*:

*bounded-linear*  $(\lambda x::'a::real-normed-field. x / y)$   
**unfolding** *divide-inverse* **by** (*rule* *mult.bounded-linear-left*)

**interpretation** *scaleR*: *bounded-bilinear scaleR*

**apply** (*rule* *bounded-bilinear.intro*)  
**apply** (*rule* *scaleR-left-distrib*)  
**apply** (*rule* *scaleR-right-distrib*)  
**apply** *simp*  
**apply** (*rule* *scaleR-left-commute*)  
**apply** (*rule-tac*  $x=1$  **in** *exI*, *simp*)  
**done**

**interpretation** *scaleR-left*: *bounded-linear*  $\lambda r. scaleR\ r\ x$   
**by** (*rule* *scaleR.bounded-linear-left*)

**interpretation** *scaleR-right*: *bounded-linear*  $\lambda x. scaleR\ r\ x$   
**by** (*rule* *scaleR.bounded-linear-right*)

**interpretation** *of-real*: *bounded-linear*  $\lambda r. of-real\ r$

**unfolding** *of-real-def* **by** (*rule* *scaleR.bounded-linear-left*)

**end**

**theory** *Real*

**imports** *RComplete RealVector*

**uses** *Tools/SMT/smt-real.ML*

**begin**

```

setup  $\ll$  SMT-Real.setup  $\gg$ 

end

```

## 69 SupInf: Sup and Inf Operators on Sets of Reals.

```

theory SupInf
imports RComplete
begin

instantiation real :: Sup
begin
definition
  Sup-real-def [code del]:  $Sup\ X == (LEAST\ z::real.\ \forall x\in X.\ x\leq z)$ 

instance ..
end

instantiation real :: Inf
begin
definition
  Inf-real-def [code del]:  $Inf\ (X::real\ set) == -\ (Sup\ (uminus\ `X))$ 

instance ..
end

```

### 69.1 Supremum of a set of reals

```

lemma Sup-upper [intro]:
  fixes  $x :: real$ 
  assumes  $x: x \in X$ 
  and  $z: !!x. x \in X \implies x \leq z$ 
  shows  $x \leq Sup\ X$ 
proof (auto simp add: Sup-real-def)
  from reals-complete2
  obtain  $s$  where  $s: (\forall y\in X. y \leq s) \ \&\ (\forall z. ((\forall y\in X. y \leq z) \implies s \leq z))$ 
  by (blast intro: x z)
  hence  $x \leq s$ 
  by (blast intro: x z)
  also with  $s$  have  $... = (LEAST\ z. \forall x\in X. x \leq z)$ 
  by (fast intro: Least-equality [symmetric])
  finally show  $x \leq (LEAST\ z. \forall x\in X. x \leq z)$  .
qed

```

```

lemma Sup-least [intro]:

```

```

fixes  $z :: \text{real}$ 
assumes  $x: X \neq \{\}$ 
  and  $z: \bigwedge x. x \in X \implies x \leq z$ 
shows  $\text{Sup } X \leq z$ 
proof (auto simp add: Sup-real-def)
from reals-complete2  $x$ 
obtain  $s$  where  $s: (\forall y \in X. y \leq s) \ \& \ (\forall z. ((\forall y \in X. y \leq z) \implies s \leq z))$ 
  by (blast intro: z)
hence  $(\text{LEAST } z. \forall x \in X. x \leq z) = s$ 
  by (best intro: Least-equality)
also with  $s \ z$  have  $\dots \leq z$ 
  by blast
finally show  $(\text{LEAST } z. \forall x \in X. x \leq z) \leq z$  .
qed

```

```

lemma less-SupE:
  fixes  $y :: \text{real}$ 
  assumes  $y < \text{Sup } X$   $X \neq \{\}$ 
  obtains  $x$  where  $x \in X$   $y < x$ 
by (metis SupInf.Sup-least assms linorder-not-less that)

```

```

lemma Sup-singleton [simp]:  $\text{Sup } \{x :: \text{real}\} = x$ 
  by (force intro: Least-equality simp add: Sup-real-def)

```

```

lemma Sup-eq-maximum:
  fixes  $z :: \text{real}$ 
  assumes  $X: z \in X$  and  $z: \forall x. x \in X \implies x \leq z$ 
  shows  $\text{Sup } X = z$ 
  by (force intro: Least-equality X z simp add: Sup-real-def)

```

```

lemma Sup-upper2:
  fixes  $x :: \text{real}$ 
  shows  $x \in X \implies y \leq x \implies (\forall x. x \in X \implies x \leq z) \implies y \leq \text{Sup } X$ 
  by (metis Sup-upper order-trans)

```

```

lemma Sup-real-iff :
  fixes  $z :: \text{real}$ 
  shows  $X \sim = \{\} \implies (\forall x. x \in X \implies x \leq z) \implies (\exists x \in X. y < x) <-> y < \text{Sup } X$ 
  by (metis Sup-least Sup-upper linorder-not-le le-less-trans)

```

```

lemma Sup-eq:
  fixes  $a :: \text{real}$ 
  shows  $(\forall x. x \in X \implies x \leq a) \implies (\forall y. (\forall x. x \in X \implies x \leq y) \implies a \leq y) \implies \text{Sup } X = a$ 
  by (metis Sup-least Sup-upper add-le-cancel-left diff-add-cancel insert-absorb insert-not-empty order-antisym)

```

```

lemma Sup-le:

```

fixes  $S :: \text{real set}$   
 shows  $S \neq \{\} \implies S * \leq b \implies \text{Sup } S \leq b$   
 by (metis SupInf.Sup-least settle-def)

lemma Sup-upper-EX:  
 fixes  $x :: \text{real}$   
 shows  $x \in X \implies \exists z. \forall x. x \in X \longrightarrow x \leq z \implies x \leq \text{Sup } X$   
 by blast

lemma Sup-insert-nonempty:  
 fixes  $x :: \text{real}$   
 assumes  $x: x \in X$   
 and  $z: \forall x. x \in X \implies x \leq z$   
 shows  $\text{Sup } (\text{insert } a \ X) = \max a \ (\text{Sup } X)$   
 proof (cases  $\text{Sup } X \leq a$ )  
 case True  
 thus ?thesis  
 apply (simp add: max-def)  
 apply (rule Sup-eq-maximum)  
 apply (rule insertI1)  
 apply (metis Sup-upper [OF - z] insertE linear order-trans)  
 done  
 next  
 case False  
 hence  $1:a < \text{Sup } X$  by simp  
 have  $\text{Sup } X \leq \text{Sup } (\text{insert } a \ X)$   
 apply (rule Sup-least)  
 apply (metis ex-in-conv x)  
 apply (rule Sup-upper-EX)  
 apply blast  
 apply (metis insert-iff linear order-refl order-trans z)  
 done  
 moreover  
 have  $\text{Sup } (\text{insert } a \ X) \leq \text{Sup } X$   
 apply (rule Sup-least)  
 apply blast  
 apply (metis False Sup-upper insertE linear z)  
 done  
 ultimately have  $\text{Sup } (\text{insert } a \ X) = \text{Sup } X$   
 by (blast intro: antisym )  
 thus ?thesis  
 by (metis 1 min-max.le-iff-sup less-le)  
 qed

lemma Sup-insert-if:  
 fixes  $X :: \text{real set}$   
 assumes  $z: \forall x. x \in X \implies x \leq z$   
 shows  $\text{Sup } (\text{insert } a \ X) = (\text{if } X = \{\} \text{ then } a \text{ else } \max a \ (\text{Sup } X))$   
 by auto (metis Sup-insert-nonempty z)

**lemma** *Sup*:  
 fixes  $S :: \text{real set}$   
 shows  $S \neq \{\}$   $\implies (\exists b. S * \leq b) \implies \text{isLub UNIV } S (\text{Sup } S)$   
 by (auto simp add: isLub-def setle-def leastP-def isUb-def intro!: setgeI)

**lemma** *Sup-finite-Max*:  
 fixes  $S :: \text{real set}$   
 assumes  $fS$ : *finite*  $S$  and  $Se$ :  $S \neq \{\}$   
 shows  $\text{Sup } S = \text{Max } S$   
 using  $fS$   $Se$   
 proof –  
 let  $?m = \text{Max } S$   
 from  $\text{Max-ge}[OF\ fS]$  have  $Sm$ :  $\forall x \in S. x \leq ?m$  by metis  
 with  $\text{Sup}[OF\ Se]$  have  $lub$ :  $\text{isLub UNIV } S (\text{Sup } S)$  by (metis setle-def)  
 from  $\text{Max-in}[OF\ fS\ Se]$   $lub$  have  $mrS$ :  $?m \leq \text{Sup } S$   
 by (auto simp add: isLub-def leastP-def setle-def setge-def isUb-def)  
 moreover  
 have  $\text{Sup } S \leq ?m$  using  $Sm$   $lub$   
 by (auto simp add: isLub-def leastP-def isUb-def setle-def setge-def)  
 ultimately show  $?thesis$  by arith  
 qed

**lemma** *Sup-finite-in*:  
 fixes  $S :: \text{real set}$   
 assumes  $fS$ : *finite*  $S$  and  $Se$ :  $S \neq \{\}$   
 shows  $\text{Sup } S \in S$   
 using  $\text{Sup-finite-Max}[OF\ fS\ Se]$   $\text{Max-in}[OF\ fS\ Se]$  by metis

**lemma** *Sup-finite-ge-iff*:  
 fixes  $S :: \text{real set}$   
 assumes  $fS$ : *finite*  $S$  and  $Se$ :  $S \neq \{\}$   
 shows  $a \leq \text{Sup } S \iff (\exists x \in S. a \leq x)$   
 by (metis  $\text{Max-ge } Se$   $\text{Sup-finite-Max } \text{Sup-finite-in } fS$  linorder-not-le less-le-trans)

**lemma** *Sup-finite-le-iff*:  
 fixes  $S :: \text{real set}$   
 assumes  $fS$ : *finite*  $S$  and  $Se$ :  $S \neq \{\}$   
 shows  $a \geq \text{Sup } S \iff (\forall x \in S. a \geq x)$   
 by (metis  $\text{Max-ge } Se$   $\text{Sup-finite-Max } \text{Sup-finite-in } fS$  order-trans)

**lemma** *Sup-finite-gt-iff*:  
 fixes  $S :: \text{real set}$   
 assumes  $fS$ : *finite*  $S$  and  $Se$ :  $S \neq \{\}$   
 shows  $a < \text{Sup } S \iff (\exists x \in S. a < x)$   
 by (metis  $Se$   $\text{Sup-finite-le-iff } fS$  linorder-not-less)

**lemma** *Sup-finite-lt-iff*:  
 fixes  $S :: \text{real set}$

**assumes**  $fS$ : *finite*  $S$  **and**  $Se$ :  $S \neq \{\}$   
**shows**  $a > \text{Sup } S \iff (\forall x \in S. a > x)$   
**by** (*metis*  $Se$  *Sup-finite-ge-iff*  $fS$  *linorder-not-less*)

**lemma** *Sup-unique*:  
**fixes**  $S :: \text{real set}$   
**shows**  $S * \leq b \implies (\forall b' < b. \exists x \in S. b' < x) \implies \text{Sup } S = b$   
**unfolding** *setle-def*  
**apply** (*rule* *Sup-eq*, *auto*)  
**apply** (*metis* *linorder-not-less*)  
**done**

**lemma** *Sup-abs-le*:  
**fixes**  $S :: \text{real set}$   
**shows**  $S \neq \{\} \implies (\forall x \in S. |x| \leq a) \implies |\text{Sup } S| \leq a$   
**by** (*auto simp add: abs-le-interval-iff*) (*metis* *Sup-upper2*)

**lemma** *Sup-bounds*:  
**fixes**  $S :: \text{real set}$   
**assumes**  $Se$ :  $S \neq \{\}$  **and**  $l$ :  $a \leq * S$  **and**  $u$ :  $S \leq * b$   
**shows**  $a \leq \text{Sup } S \wedge \text{Sup } S \leq b$   
**proof**–  
**from**  $\text{Sup}[OF\ Se]\ u$  **have**  $\text{lub}$ : *isLub UNIV*  $S$  ( $\text{Sup } S$ ) **by** *blast*  
**hence**  $b$ :  $\text{Sup } S \leq b$  **using**  $u$   
**by** (*auto simp add: isLub-def leastP-def setle-def setge-def isUb-def*)  
**from**  $Se$  **obtain**  $y$  **where**  $y$ :  $y \in S$  **by** *blast*  
**from**  $\text{lub } l$  **have**  $a \leq \text{Sup } S$   
**by** (*auto simp add: isLub-def leastP-def setle-def setge-def isUb-def*)  
(*metis* *le-iff-sup* *le-sup-iff*  $y$ )  
**with**  $b$  **show** *?thesis* **by** *blast*  
**qed**

**lemma** *Sup-asclose*:  
**fixes**  $S :: \text{real set}$   
**assumes**  $S$ :  $S \neq \{\}$  **and**  $b$ :  $\forall x \in S. |x - l| \leq e$  **shows**  $|\text{Sup } S - l| \leq e$   
**proof**–  
**have**  $th$ :  $\bigwedge (x :: \text{real})\ l\ e. |x - l| \leq e \iff l - e \leq x \wedge x \leq l + e$  **by** *arith*  
**thus** *?thesis* **using**  $S\ b$  *Sup-bounds[of*  $S\ l - e\ l + e]$  **unfolding**  $th$   
**by** (*auto simp add: setge-def setle-def*)  
**qed**

**lemma** *Sup-lessThan[simp]*:  $\text{Sup } \{..<x\} = (x :: \text{real})$   
**by** (*auto intro!: Sup-eq intro: dense-le*)

**lemma** *Sup-greaterThanLessThan[simp]*:  $y < x \implies \text{Sup } \{y <..<x\} = (x :: \text{real})$   
**by** (*auto intro!: Sup-eq intro: dense-le-bounded*)

**lemma** *Sup-atLeastLessThan[simp]*:  $y < x \implies \text{Sup } \{y ..<x\} = (x :: \text{real})$   
**by** (*auto intro!: Sup-eq intro: dense-le-bounded*)

**lemma** *Sup-atMost*[simp]:  $\text{Sup } \{..x\} = (x::\text{real})$   
**by** (*auto intro!*: *Sup-eq-maximum*)

**lemma** *Sup-greaterThanAtMost*[simp]:  $y < x \implies \text{Sup } \{y<..x\} = (x::\text{real})$   
**by** (*auto intro!*: *Sup-eq-maximum*)

**lemma** *Sup-atLeastAtMost*[simp]:  $y \leq x \implies \text{Sup } \{y..x\} = (x::\text{real})$   
**by** (*auto intro!*: *Sup-eq-maximum*)

## 69.2 Infimum of a set of reals

**lemma** *Inf-lower* [intro]:  
**fixes**  $z :: \text{real}$   
**assumes**  $x: x \in X$   
**and**  $z: !!x. x \in X \implies z \leq x$   
**shows**  $\text{Inf } X \leq x$   
**proof** –  
**have**  $-x \leq \text{Sup } (\text{uminus } 'X)$   
**by** (*rule* *Sup-upper* [where  $z = -z$ ]) (*auto simp add: image-iff x z*)  
**thus** ?thesis  
**by** (*auto simp add: Inf-real-def*)  
**qed**

**lemma** *Inf-greatest* [intro]:  
**fixes**  $z :: \text{real}$   
**assumes**  $x: X \neq \{\}$   
**and**  $z: \bigwedge x. x \in X \implies z \leq x$   
**shows**  $z \leq \text{Inf } X$   
**proof** –  
**have**  $\text{Sup } (\text{uminus } 'X) \leq -z$  **using**  $x z$  **by** *force*  
**hence**  $z \leq -\text{Sup } (\text{uminus } 'X)$   
**by** *simp*  
**thus** ?thesis  
**by** (*auto simp add: Inf-real-def*)  
**qed**

**lemma** *Inf-singleton* [simp]:  $\text{Inf } \{x::\text{real}\} = x$   
**by** (*simp add: Inf-real-def*)

**lemma** *Inf-eq-minimum*:  
**fixes**  $z :: \text{real}$   
**assumes**  $x: z \in X$  **and**  $z: !!x. x \in X \implies z \leq x$   
**shows**  $\text{Inf } X = z$   
**proof** –  
**have**  $\text{Sup } (\text{uminus } 'X) = -z$  **using**  $x z$   
**by** (*force intro: Sup-eq-maximum x z*)  
**thus** ?thesis  
**by** (*simp add: Inf-real-def*)

qed

**lemma** *Inf-lower2*:

**fixes**  $x :: \text{real}$

**shows**  $x \in X \implies x \leq y \implies (!x. x \in X \implies z \leq x) \implies \text{Inf } X \leq y$

**by** (*metis Inf-lower order-trans*)

**lemma** *Inf-real-iff*:

**fixes**  $z :: \text{real}$

**shows**  $X \neq \{\} \implies (!x. x \in X \implies z \leq x) \implies (\exists x \in X. x < y) \longleftrightarrow \text{Inf } X < y$

**by** (*metis Inf-greatest Inf-lower less-le-not-le linear order-less-le-trans*)

**lemma** *Inf-eq*:

**fixes**  $a :: \text{real}$

**shows**  $(!x. x \in X \implies a \leq x) \implies (!y. (!x. x \in X \implies y \leq x) \implies y \leq a) \implies$

$\text{Inf } X = a$

**by** (*metis Inf-greatest Inf-lower add-le-cancel-left diff-add-cancel insert-absorb insert-not-empty order-antisym*)

**lemma** *Inf-ge*:

**fixes**  $S :: \text{real set}$

**shows**  $S \neq \{\} \implies b \leq^* S \implies \text{Inf } S \geq b$

**by** (*metis SupInf.Inf-greatest setge-def*)

**lemma** *Inf-lower-EX*:

**fixes**  $x :: \text{real}$

**shows**  $x \in X \implies \exists z. \forall x. x \in X \longrightarrow z \leq x \implies \text{Inf } X \leq x$

**by** *blast*

**lemma** *Inf-insert-nonempty*:

**fixes**  $x :: \text{real}$

**assumes**  $x: x \in X$

**and**  $z: !x. x \in X \implies z \leq x$

**shows**  $\text{Inf } (\text{insert } a \ X) = \min a \ (\text{Inf } X)$

**proof** (*cases*  $a \leq \text{Inf } X$ )

**case** *True*

**thus** *?thesis*

**by** (*simp add: min-def*)

(*blast intro: Inf-eq-minimum order-trans z*)

**next**

**case** *False*

**hence**  $1: \text{Inf } X < a$  **by** *simp*

**have**  $\text{Inf } (\text{insert } a \ X) \leq \text{Inf } X$

**apply** (*rule Inf-greatest*)

**apply** (*metis ex-in-conv x*)

**apply** (*rule Inf-lower-EX*)

**apply** (*erule insertI2*)

**apply** (*metis insert-iff linear order-refl order-trans z*)



```

done
moreover
have  $\text{Inf } X \leq \text{Inf } (\text{insert } a \ X)$ 
  apply (rule Inf-greatest)
  apply blast
  apply (metis False Inf-lower insertE linear z)
done
ultimately have  $\text{Inf } (\text{insert } a \ X) = \text{Inf } X$ 
  by (blast intro: antisym )
thus ?thesis
  by (metis False min-max.inf-absorb2 linear)
qed

```

```

lemma Inf-insert-if:
  fixes  $X :: \text{real set}$ 
  assumes  $z: \forall x. x \in X \implies z \leq x$ 
  shows  $\text{Inf } (\text{insert } a \ X) = (\text{if } X = \{\} \text{ then } a \text{ else } \min a \ (\text{Inf } X))$ 
by auto (metis Inf-insert-nonempty z)

```

```

lemma Inf-greater:
  fixes  $z :: \text{real}$ 
  shows  $X \neq \{\} \implies \text{Inf } X < z \implies \exists x \in X. x < z$ 
  by (metis Inf-real-iff mem-def not-leE)

```

```

lemma Inf-close:
  fixes  $e :: \text{real}$ 
  shows  $X \neq \{\} \implies 0 < e \implies \exists x \in X. x < \text{Inf } X + e$ 
  by (metis add-strict-increasing add-commute Inf-greater linorder-not-le pos-add-strict)

```

```

lemma Inf-finite-Min:
  fixes  $S :: \text{real set}$ 
  shows  $\text{finite } S \implies S \neq \{\} \implies \text{Inf } S = \text{Min } S$ 
by (simp add: Inf-real-def Sup-finite-Max image-image)

```

```

lemma Inf-finite-in:
  fixes  $S :: \text{real set}$ 
  assumes  $fS: \text{finite } S \text{ and } Se: S \neq \{\}$ 
  shows  $\text{Inf } S \in S$ 
  using Inf-finite-Min[OF fS Se] Min-in[OF fS Se] by metis

```

```

lemma Inf-finite-ge-iff:
  fixes  $S :: \text{real set}$ 
  shows  $\text{finite } S \implies S \neq \{\} \implies a \leq \text{Inf } S \longleftrightarrow (\forall x \in S. a \leq x)$ 
by (metis Inf-finite-Min Inf-finite-in Min-le order-trans)

```

```

lemma Inf-finite-le-iff:
  fixes  $S :: \text{real set}$ 
  shows  $\text{finite } S \implies S \neq \{\} \implies a \geq \text{Inf } S \longleftrightarrow (\exists x \in S. a \geq x)$ 
by (metis Inf-finite-Min Inf-finite-ge-iff Inf-finite-in Min-le)

```

*order-antisym linear*)

**lemma** *Inf-finite-gt-iff*:

**fixes**  $S :: \text{real set}$

**shows**  $\text{finite } S \implies S \neq \{\} \implies a < \text{Inf } S \longleftrightarrow (\forall x \in S. a < x)$

**by** (*metis Inf-finite-le-iff linorder-not-less*)

**lemma** *Inf-finite-lt-iff*:

**fixes**  $S :: \text{real set}$

**shows**  $\text{finite } S \implies S \neq \{\} \implies a > \text{Inf } S \longleftrightarrow (\exists x \in S. a > x)$

**by** (*metis Inf-finite-ge-iff linorder-not-less*)

**lemma** *Inf-unique*:

**fixes**  $S :: \text{real set}$

**shows**  $b \leq^* S \implies (\forall b' > b. \exists x \in S. b' > x) \implies \text{Inf } S = b$

**unfolding** *setge-def*

**apply** (*rule Inf-eq, auto*)

**apply** (*metis less-le-not-le linorder-not-less*)

**done**

**lemma** *Inf-abs-ge*:

**fixes**  $S :: \text{real set}$

**shows**  $S \neq \{\} \implies (\forall x \in S. |x| \leq a) \implies |\text{Inf } S| \leq a$

**by** (*simp add: Inf-real-def*) (*rule Sup-abs-le, auto*)

**lemma** *Inf-asclose*:

**fixes**  $S :: \text{real set}$

**assumes**  $S: S \neq \{\}$  **and**  $b: \forall x \in S. |x - l| \leq e$  **shows**  $|\text{Inf } S - l| \leq e$

**proof** –

**have**  $|- \text{Sup } (\text{uminus } ' S) - l| = |\text{Sup } (\text{uminus } ' S) - (-l)|$

**by** *auto*

**also have**  $\dots \leq e$

**apply** (*rule Sup-asclose*)

**apply** (*auto simp add: S*)

**apply** (*metis abs-minus-add-cancel b add-commute diff-def*)

**done**

**finally have**  $|- \text{Sup } (\text{uminus } ' S) - l| \leq e$  .

**thus** *?thesis*

**by** (*simp add: Inf-real-def*)

**qed**

**lemma** *Inf-greaterThanLessThan[simp]*:  $y < x \implies \text{Inf } \{y <..<x\} = (y::\text{real})$

**by** (*simp add: Inf-real-def*)

**lemma** *Inf-atLeastLessThan[simp]*:  $y < x \implies \text{Inf } \{y..<x\} = (y::\text{real})$

**by** (*simp add: Inf-real-def*)

**lemma** *Inf-atLeast[simp]*:  $\text{Inf } \{x.. \} = (x::\text{real})$

**by** (*simp add: Inf-real-def*)

**lemma** *Inf-greaterThanAtMost*[simp]:  $y < x \implies \text{Inf } \{y <..x\} = (y::\text{real})$   
**by** (simp add: Inf-real-def)

**lemma** *Inf-atLeastAtMost*[simp]:  $y \leq x \implies \text{Inf } \{y..x\} = (y::\text{real})$   
**by** (simp add: Inf-real-def)

### 69.3 Relate max and min to Sup and Inf.

**lemma** *real-max-Sup*:  
**fixes**  $x :: \text{real}$   
**shows**  $\max x y = \text{Sup } \{x,y\}$   
**proof** –  
**have**  $f: \text{finite } \{x, y\} \ \{x,y\} \neq \{\}$  **by** simp-all  
**from** *Sup-finite-le-iff*[OF  $f$ , of  $\max x y$ ] **have**  $\text{Sup } \{x,y\} \leq \max x y$  **by** simp  
**moreover**  
**have**  $\max x y \leq \text{Sup } \{x,y\}$  **using** *Sup-finite-ge-iff*[OF  $f$ , of  $\max x y$ ]  
**by** (simp add: linorder-linear)  
**ultimately show** ?thesis **by** arith  
**qed**

**lemma** *real-min-Inf*:  
**fixes**  $x :: \text{real}$   
**shows**  $\min x y = \text{Inf } \{x,y\}$   
**proof** –  
**have**  $f: \text{finite } \{x, y\} \ \{x,y\} \neq \{\}$  **by** simp-all  
**from** *Inf-finite-le-iff*[OF  $f$ , of  $\min x y$ ] **have**  $\text{Inf } \{x,y\} \leq \min x y$   
**by** (simp add: linorder-linear)  
**moreover**  
**have**  $\min x y \leq \text{Inf } \{x,y\}$  **using** *Inf-finite-ge-iff*[OF  $f$ , of  $\min x y$ ]  
**by** simp  
**ultimately show** ?thesis **by** arith  
**qed**

**lemma** *reals-complete-interval*:  
**fixes**  $a::\text{real}$  **and**  $b::\text{real}$   
**assumes**  $a < b$  **and**  $P a$  **and**  $\sim P b$   
**shows**  $\exists c. a \leq c \ \& \ c \leq b \ \& \ (\forall x. a \leq x \ \& \ x < c \implies P x) \ \& \$   
 $(\forall d. (\forall x. a \leq x \ \& \ x < d \implies P x) \implies d \leq c)$   
**proof** (rule exI [where  $x = \text{Sup } \{d. \forall x. a \leq x \ \& \ x < d \implies P x\}$ ], auto)  
**show**  $a \leq \text{Sup } \{d. \forall x. a \leq x \ \& \ x < d \implies P x\}$   
**by** (rule SupInf.Sup-upper [where  $z=b$ ], auto)  
 $(\text{metis } \langle a < b \rangle \langle \neg P b \rangle \text{ linear less-le})$   
**next**  
**show**  $\text{Sup } \{d. \forall x. a \leq x \ \& \ x < d \implies P x\} \leq b$   
**apply** (rule SupInf.Sup-least)  
**apply** (auto simp add: )  
**apply** (metis less-le-not-le)  
**apply** (metis  $\langle a < b \rangle \langle \sim P b \rangle \text{ linear less-le})$

```

    done
next
fix x
  assume x: a ≤ x and lt: x < Sup {d. ∀ c. a ≤ c ∧ c < d → P c}
  show P x
    apply (rule less-SupE [OF lt], auto)
    apply (metis less-le-not-le)
    apply (metis x)
  done
next
fix d
  assume 0: ∀ x. a ≤ x ∧ x < d → P x
  thus d ≤ Sup {d. ∀ c. a ≤ c ∧ c < d → P c}
    by (rule-tac z=b in SupInf.Sup-upper, auto)
    (metis ‹a<b› ‹~ P b› linear less-le)
qed
end

```

## 70 Limits: Filters and Limits

```

theory Limits
imports RealVector
begin

```

### 70.1 Nets

A net is now defined simply as a filter on a set. The definition also allows non-proper filters.

```

locale is-filter =
  fixes net :: ('a ⇒ bool) ⇒ bool
  assumes True: net (λx. True)
  assumes conj: net (λx. P x) ⇒ net (λx. Q x) ⇒ net (λx. P x ∧ Q x)
  assumes mono: ∀ x. P x → Q x ⇒ net (λx. P x) ⇒ net (λx. Q x)

```

```

typedef (open) 'a net =
  {net :: ('a ⇒ bool) ⇒ bool. is-filter net}
proof
  show (λx. True) ∈ ?net by (auto intro: is-filter.intro)
qed

```

```

lemma is-filter-Rep-net: is-filter (Rep-net net)
using Rep-net [of net] by simp

```

```

lemma Abs-net-inverse':
  assumes is-filter net shows Rep-net (Abs-net net) = net
using assms by (simp add: Abs-net-inverse)

```

## 70.2 Eventually

**definition**

*eventually* :: (*'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a* *net*  $\Rightarrow$  *bool* **where**  
 $[code\ del]:\ eventually\ P\ net \longleftrightarrow Rep\_net\ net\ P$

**lemma** *eventually-Abs-net*:

**assumes** *is-filter net* **shows** *eventually P (Abs-net net) = net P*  
**unfolding** *eventually-def* **using** *assms* **by** (*simp add: Abs-net-inverse*)

**lemma** *expand-net-eq*:

**shows** *net = net'*  $\longleftrightarrow (\forall P.\ eventually\ P\ net = eventually\ P\ net')$   
**unfolding** *Rep-net-inject* [*symmetric*] *expand-fun-eq* *eventually-def* ..

**lemma** *eventually-True* [*simp*]: *eventually* ( $\lambda x.\ True$ ) *net*

**unfolding** *eventually-def*

**by** (*rule is-filter.True* [*OF is-filter-Rep-net*])

**lemma** *always-eventually*:  $\forall x.\ P\ x \Longrightarrow eventually\ P\ net$

**proof** –

**assume**  $\forall x.\ P\ x$  **hence**  $P = (\lambda x.\ True)$  **by** (*simp add: ext*)

**thus** *eventually P net* **by** *simp*

**qed**

**lemma** *eventually-mono*:

$(\forall x.\ P\ x \longrightarrow Q\ x) \Longrightarrow eventually\ P\ net \Longrightarrow eventually\ Q\ net$

**unfolding** *eventually-def*

**by** (*rule is-filter.mono* [*OF is-filter-Rep-net*])

**lemma** *eventually-conj*:

**assumes** *P: eventually* ( $\lambda x.\ P\ x$ ) *net*

**assumes** *Q: eventually* ( $\lambda x.\ Q\ x$ ) *net*

**shows** *eventually* ( $\lambda x.\ P\ x \wedge Q\ x$ ) *net*

**using** *assms* **unfolding** *eventually-def*

**by** (*rule is-filter.conj* [*OF is-filter-Rep-net*])

**lemma** *eventually-mp*:

**assumes** *eventually* ( $\lambda x.\ P\ x \longrightarrow Q\ x$ ) *net*

**assumes** *eventually* ( $\lambda x.\ P\ x$ ) *net*

**shows** *eventually* ( $\lambda x.\ Q\ x$ ) *net*

**proof** (*rule eventually-mono*)

**show**  $\forall x.\ (P\ x \longrightarrow Q\ x) \wedge P\ x \longrightarrow Q\ x$  **by** *simp*

**show** *eventually* ( $\lambda x.\ (P\ x \longrightarrow Q\ x) \wedge P\ x$ ) *net*

**using** *assms* **by** (*rule eventually-conj*)

**qed**

**lemma** *eventually-rev-mp*:

**assumes** *eventually* ( $\lambda x.\ P\ x$ ) *net*

**assumes** *eventually* ( $\lambda x.\ P\ x \longrightarrow Q\ x$ ) *net*

**shows** *eventually* ( $\lambda x.\ Q\ x$ ) *net*

**using** *assms*(2) *assms*(1) **by** (*rule eventually-mp*)

**lemma** *eventually-conj-iff*:

*eventually*  $(\lambda x. P\ x \wedge Q\ x)\ net \longleftrightarrow eventually\ P\ net \wedge eventually\ Q\ net$   
**by** (*auto intro: eventually-conj elim: eventually-conv-mp*)

**lemma** *eventually-elim1*:

**assumes** *eventually*  $(\lambda i. P\ i)\ net$   
**assumes**  $\bigwedge i. P\ i \implies Q\ i$   
**shows** *eventually*  $(\lambda i. Q\ i)\ net$   
**using** *assms* **by** (*auto elim!: eventually-conv-mp*)

**lemma** *eventually-elim2*:

**assumes** *eventually*  $(\lambda i. P\ i)\ net$   
**assumes** *eventually*  $(\lambda i. Q\ i)\ net$   
**assumes**  $\bigwedge i. P\ i \implies Q\ i \implies R\ i$   
**shows** *eventually*  $(\lambda i. R\ i)\ net$   
**using** *assms* **by** (*auto elim!: eventually-conv-mp*)

### 70.3 Finer-than relation

$net \leq net'$  means that *net* is finer than *net'*.

**instantiation** *net* :: (*type*) *complete-lattice*  
**begin**

**definition**

*le-net-def* [*code del*]:  
 $net \leq net' \longleftrightarrow (\forall P. eventually\ P\ net' \longrightarrow eventually\ P\ net)$

**definition**

*less-net-def* [*code del*]:  
 $(net :: 'a\ net) < net' \longleftrightarrow net \leq net' \wedge \neg net' \leq net$

**definition**

*top-net-def* [*code del*]:  
 $top = Abs\_net\ (\lambda P. \forall x. P\ x)$

**definition**

*bot-net-def* [*code del*]:  
 $bot = Abs\_net\ (\lambda P. True)$

**definition**

*sup-net-def* [*code del*]:  
 $sup\ net\ net' = Abs\_net\ (\lambda P. eventually\ P\ net \wedge eventually\ P\ net')$

**definition**

*inf-net-def* [*code del*]:  
 $inf\ a\ b = Abs\_net\ (\lambda P. \exists Q\ R. eventually\ Q\ a \wedge eventually\ R\ b \wedge (\forall x. Q\ x \wedge R\ x \longrightarrow P\ x))$

**definition**

*Sup-net-def* [code del]:  

$$\text{Sup } A = \text{Abs-net } (\lambda P. \forall \text{net} \in A. \text{eventually } P \text{ net})$$

**definition**

*Inf-net-def* [code del]:  

$$\text{Inf } A = \text{Sup } \{x :: 'a \text{ net}. \forall y \in A. x \leq y\}$$

**lemma** *eventually-top* [simp]:  $\text{eventually } P \text{ top} \longleftrightarrow (\forall x. P x)$   
**unfolding** *top-net-def*  
**by** (rule *eventually-Abs-net*, rule *is-filter.intro*, auto)

**lemma** *eventually-bot* [simp]:  $\text{eventually } P \text{ bot}$   
**unfolding** *bot-net-def*  
**by** (subst *eventually-Abs-net*, rule *is-filter.intro*, auto)

**lemma** *eventually-sup*:  

$$\text{eventually } P (\text{sup net net}') \longleftrightarrow \text{eventually } P \text{ net} \wedge \text{eventually } P \text{ net}'$$
  
**unfolding** *sup-net-def*  
**by** (rule *eventually-Abs-net*, rule *is-filter.intro*)  
(auto elim!: *eventually-rev-mp*)

**lemma** *eventually-inf*:  

$$\text{eventually } P (\text{inf } a \text{ b}) \longleftrightarrow$$
  

$$(\exists Q R. \text{eventually } Q a \wedge \text{eventually } R b \wedge (\forall x. Q x \wedge R x \longrightarrow P x))$$
  
**unfolding** *inf-net-def*  
**apply** (rule *eventually-Abs-net*, rule *is-filter.intro*)  
**apply** (fast intro: *eventually-True*)  
**apply** *clarify*  
**apply** (intro *exI conjI*)  
**apply** (erule (1) *eventually-conj*)  
**apply** (erule (1) *eventually-conj*)  
**apply** *simp*  
**apply** *auto*  
**done**

**lemma** *eventually-Sup*:  

$$\text{eventually } P (\text{Sup } A) \longleftrightarrow (\forall \text{net} \in A. \text{eventually } P \text{ net})$$
  
**unfolding** *Sup-net-def*  
**apply** (rule *eventually-Abs-net*, rule *is-filter.intro*)  
**apply** (auto intro: *eventually-conj elim!*: *eventually-rev-mp*)  
**done**

**instance proof**

**fix**  $x \ y :: 'a \text{ net}$  **show**  $x < y \longleftrightarrow x \leq y \wedge \neg y \leq x$   
**by** (rule *less-net-def*)  
**next**  
**fix**  $x :: 'a \text{ net}$  **show**  $x \leq x$

```

    unfolding le-net-def by simp
next
  fix x y z :: 'a net assume  $x \leq y$  and  $y \leq z$  thus  $x \leq z$ 
    unfolding le-net-def by simp
next
  fix x y :: 'a net assume  $x \leq y$  and  $y \leq x$  thus  $x = y$ 
    unfolding le-net-def expand-net-eq by fast
next
  fix x :: 'a net show  $x \leq \text{top}$ 
    unfolding le-net-def eventually-top by (simp add: always-eventually)
next
  fix x :: 'a net show  $\text{bot} \leq x$ 
    unfolding le-net-def by simp
next
  fix x y :: 'a net show  $x \leq \text{sup } x y$  and  $y \leq \text{sup } x y$ 
    unfolding le-net-def eventually-sup by simp-all
next
  fix x y z :: 'a net assume  $x \leq z$  and  $y \leq z$  thus  $\text{sup } x y \leq z$ 
    unfolding le-net-def eventually-sup by simp
next
  fix x y :: 'a net show  $\text{inf } x y \leq x$  and  $\text{inf } x y \leq y$ 
    unfolding le-net-def eventually-inf by (auto intro: eventually-True)
next
  fix x y z :: 'a net assume  $x \leq y$  and  $x \leq z$  thus  $x \leq \text{inf } y z$ 
    unfolding le-net-def eventually-inf
    by (auto elim!: eventually-mono intro: eventually-conj)
next
  fix x :: 'a net and A assume  $x \in A$  thus  $x \leq \text{Sup } A$ 
    unfolding le-net-def eventually-Sup by simp
next
  fix A and y :: 'a net assume  $\bigwedge x. x \in A \implies x \leq y$  thus  $\text{Sup } A \leq y$ 
    unfolding le-net-def eventually-Sup by simp
next
  fix z :: 'a net and A assume  $z \in A$  thus  $\text{Inf } A \leq z$ 
    unfolding le-net-def Inf-net-def eventually-Sup Ball-def by simp
next
  fix A and x :: 'a net assume  $\bigwedge y. y \in A \implies x \leq y$  thus  $x \leq \text{Inf } A$ 
    unfolding le-net-def Inf-net-def eventually-Sup Ball-def by simp
qed

end

lemma net-leD:
   $\text{net} \leq \text{net}' \implies \text{eventually } P \text{ net}' \implies \text{eventually } P \text{ net}$ 
  unfolding le-net-def by simp

lemma net-leI:
   $(\bigwedge P. \text{eventually } P \text{ net}' \implies \text{eventually } P \text{ net}) \implies \text{net} \leq \text{net}'$ 
  unfolding le-net-def by simp

```



**lemma** *eventually-False*:  
 $\text{eventually } (\lambda x. \text{False}) \text{ net} \longleftrightarrow \text{net} = \text{bot}$   
**unfolding** *expand-net-eq* **by** (*auto elim*: *eventually-rev-mp*)

## 70.4 Map function for nets

**definition**  
 $\text{netmap} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ net} \Rightarrow 'b \text{ net}$   
**where** [*code del*]:  
 $\text{netmap } f \text{ net} = \text{Abs-net } (\lambda P. \text{eventually } (\lambda x. P (f x)) \text{ net})$

**lemma** *eventually-netmap*:  
 $\text{eventually } P (\text{netmap } f \text{ net}) = \text{eventually } (\lambda x. P (f x)) \text{ net}$   
**unfolding** *netmap-def*  
**apply** (*rule eventually-Abs-net*)  
**apply** (*rule is-filter.intro*)  
**apply** (*auto elim*!: *eventually-rev-mp*)  
**done**

**lemma** *netmap-ident*:  $\text{netmap } (\lambda x. x) \text{ net} = \text{net}$   
**by** (*simp add*: *expand-net-eq eventually-netmap*)

**lemma** *netmap-netmap*:  $\text{netmap } f (\text{netmap } g \text{ net}) = \text{netmap } (\lambda x. f (g x)) \text{ net}$   
**by** (*simp add*: *expand-net-eq eventually-netmap*)

**lemma** *netmap-mono*:  $\text{net} \leq \text{net}' \implies \text{netmap } f \text{ net} \leq \text{netmap } f \text{ net}'$   
**unfolding** *le-net-def eventually-netmap* **by** *simp*

**lemma** *netmap-bot* [*simp*]:  $\text{netmap } f \text{ bot} = \text{bot}$   
**by** (*simp add*: *expand-net-eq eventually-netmap*)

## 70.5 Sequentially

**definition**  
 $\text{sequentially} :: \text{nat net}$   
**where** [*code del*]:  
 $\text{sequentially} = \text{Abs-net } (\lambda P. \exists k. \forall n \geq k. P n)$

**lemma** *eventually-sequentially*:  
 $\text{eventually } P \text{ sequentially} \longleftrightarrow (\exists N. \forall n \geq N. P n)$   
**unfolding** *sequentially-def*  
**proof** (*rule eventually-Abs-net*, *rule is-filter.intro*)  
**fix**  $P Q :: \text{nat} \Rightarrow \text{bool}$   
**assume**  $\exists i. \forall n \geq i. P n$  **and**  $\exists j. \forall n \geq j. Q n$   
**then obtain**  $i j$  **where**  $\forall n \geq i. P n$  **and**  $\forall n \geq j. Q n$  **by** *auto*  
**then have**  $\forall n \geq \max i j. P n \wedge Q n$  **by** *simp*  
**then show**  $\exists k. \forall n \geq k. P n \wedge Q n$  **..**  
**qed** *auto*

**lemma** *sequentially-bot* [simp]: *sequentially*  $\neq$  *bot*  
**unfolding** *expand-net-eq eventually-sequentially* **by** *auto*

**lemma** *eventually-False-sequentially* [simp]:  
 $\neg$  *eventually*  $(\lambda n. \text{False})$  *sequentially*  
**by** (*simp add: eventually-False*)

**lemma** *le-sequentially*:  
 $\text{net} \leq \text{sequentially} \iff (\forall N. \text{eventually } (\lambda n. N \leq n) \text{ net})$   
**unfolding** *le-net-def eventually-sequentially*  
**by** (*safe, fast, drule-tac x=N in spec, auto elim: eventually-rev-mp*)

## 70.6 Standard Nets

### definition

*within* :: 'a net  $\Rightarrow$  'a set  $\Rightarrow$  'a net (**infixr** *within* 70)  
**where** [code del]:  
 $\text{net within } S = \text{Abs-net } (\lambda P. \text{eventually } (\lambda x. x \in S \longrightarrow P x) \text{ net})$

### definition

*nhds* :: 'a::topological-space  $\Rightarrow$  'a net  
**where** [code del]:  
 $\text{nhds } a = \text{Abs-net } (\lambda P. \exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. P x))$

### definition

*at* :: 'a::topological-space  $\Rightarrow$  'a net  
**where** [code del]:  
 $\text{at } a = \text{nhds } a \text{ within } - \{a\}$

### lemma eventually-within:

$\text{eventually } P (\text{net within } S) = \text{eventually } (\lambda x. x \in S \longrightarrow P x) \text{ net}$   
**unfolding** *within-def*  
**by** (*rule eventually-Abs-net, rule is-filter.intro*)  
*(auto elim!: eventually-rev-mp)*

### lemma within-UNIV: net within UNIV = net

**unfolding** *expand-net-eq eventually-within* **by** *simp*

### lemma eventually-nhds:

$\text{eventually } P (\text{nhds } a) \iff (\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. P x))$

### unfolding nhds-def

### proof (rule eventually-Abs-net, rule is-filter.intro)

**have**  $\text{open } \text{UNIV} \wedge a \in \text{UNIV} \wedge (\forall x \in \text{UNIV}. \text{True})$  **by** *simp*

**thus**  $\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. \text{True})$  **by**  $-$  *rule*

**next**

**fix**  $P Q$

**assume**  $\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. P x)$

**and**  $\exists T. \text{open } T \wedge a \in T \wedge (\forall x \in T. Q x)$

**then obtain**  $S T$  **where**

```

  open  $S \wedge a \in S \wedge (\forall x \in S. P\ x)$ 
  open  $T \wedge a \in T \wedge (\forall x \in T. Q\ x)$  by auto
  hence open  $(S \cap T) \wedge a \in S \cap T \wedge (\forall x \in (S \cap T). P\ x \wedge Q\ x)$ 
  by (simp add: open-Int)
  thus  $\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. P\ x \wedge Q\ x)$  by — rule
qed auto

```

**lemma** *eventually-nhds-metric*:

```

  eventually  $P\ (nhds\ a) \longleftrightarrow (\exists d > 0. \forall x. \text{dist } x\ a < d \longrightarrow P\ x)$ 
unfolding eventually-nhds open-dist
apply safe
apply fast
apply (rule-tac  $x = \{x. \text{dist } x\ a < d\}$  in exI, simp)
apply clarsimp
apply (rule-tac  $x = d - \text{dist } x\ a$  in exI, clarsimp)
apply (simp only: less-diff-eq)
apply (erule le-less-trans [OF dist-triangle])
done

```

**lemma** *eventually-at-topological*:

```

  eventually  $P\ (at\ a) \longleftrightarrow (\exists S. \text{open } S \wedge a \in S \wedge (\forall x \in S. x \neq a \longrightarrow P\ x))$ 
unfolding at-def eventually-within eventually-nhds by simp

```

**lemma** *eventually-at*:

```

  fixes  $a :: 'a :: \text{metric-space}$ 
  shows eventually  $P\ (at\ a) \longleftrightarrow (\exists d > 0. \forall x. x \neq a \wedge \text{dist } x\ a < d \longrightarrow P\ x)$ 
unfolding at-def eventually-within eventually-nhds-metric by auto

```

## 70.7 Boundedness

**definition**

```

  Bfun :: ('a  $\Rightarrow$  'b::real-normed-vector)  $\Rightarrow$  'a net  $\Rightarrow$  bool where
  [code del]: Bfun  $f\ net = (\exists K > 0. \text{eventually } (\lambda x. \text{norm } (f\ x) \leq K)\ net)$ 

```

**lemma** *BfunI*:

```

  assumes  $K: \text{eventually } (\lambda x. \text{norm } (f\ x) \leq K)\ net$  shows  $Bfun\ f\ net$ 
unfolding Bfun-def
proof (intro exI conjI allI)
  show  $0 < \max K\ 1$  by simp
next
  show eventually  $(\lambda x. \text{norm } (f\ x) \leq \max K\ 1)\ net$ 
  using  $K$  by (rule eventually-elim1, simp)
qed

```

**lemma** *BfunE*:

```

  assumes  $Bfun\ f\ net$ 
  obtains  $B$  where  $0 < B$  and eventually  $(\lambda x. \text{norm } (f\ x) \leq B)\ net$ 
using assms unfolding Bfun-def by fast

```

## 70.8 Convergence to Zero

**definition**

$Zfun :: ('a \Rightarrow 'b::real-normed-vector) \Rightarrow 'a\ net \Rightarrow bool$  **where**  
 $[code\ del]: Zfun\ f\ net = (\forall r>0. eventually\ (\lambda x. norm\ (f\ x) < r)\ net)$

**lemma** *ZfunI*:

$(\bigwedge r. 0 < r \implies eventually\ (\lambda x. norm\ (f\ x) < r)\ net) \implies Zfun\ f\ net$   
**unfolding** *Zfun-def* **by** *simp*

**lemma** *ZfunD*:

$\llbracket Zfun\ f\ net; 0 < r \rrbracket \implies eventually\ (\lambda x. norm\ (f\ x) < r)\ net$   
**unfolding** *Zfun-def* **by** *simp*

**lemma** *Zfun-ssubst*:

$eventually\ (\lambda x. f\ x = g\ x)\ net \implies Zfun\ g\ net \implies Zfun\ f\ net$   
**unfolding** *Zfun-def* **by** (*auto elim!:: eventually-rev-mp*)

**lemma** *Zfun-zero*:  $Zfun\ (\lambda x. 0)\ net$

**unfolding** *Zfun-def* **by** *simp*

**lemma** *Zfun-norm-iff*:  $Zfun\ (\lambda x. norm\ (f\ x))\ net = Zfun\ (\lambda x. f\ x)\ net$

**unfolding** *Zfun-def* **by** *simp*

**lemma** *Zfun-imp-Zfun*:

**assumes** *f*:  $Zfun\ f\ net$

**assumes** *g*:  $eventually\ (\lambda x. norm\ (g\ x) \leq norm\ (f\ x) * K)\ net$

**shows**  $Zfun\ (\lambda x. g\ x)\ net$

**proof** (*cases*)

**assume** *K*:  $0 < K$

**show** *?thesis*

**proof** (*rule ZfunI*)

**fix** *r*::*real* **assume**  $0 < r$

**hence**  $0 < r / K$

**using** *K* **by** (*rule divide-pos-pos*)

**then have**  $eventually\ (\lambda x. norm\ (f\ x) < r / K)\ net$

**using** *ZfunD* [*OF f*] **by** *fast*

**with** *g* **show**  $eventually\ (\lambda x. norm\ (g\ x) < r)\ net$

**proof** (*rule eventually-elim2*)

**fix** *x*

**assume** *\**:  $norm\ (g\ x) \leq norm\ (f\ x) * K$

**assume**  $norm\ (f\ x) < r / K$

**hence**  $norm\ (f\ x) * K < r$

**by** (*simp add: pos-less-divide-eq K*)

**thus**  $norm\ (g\ x) < r$

**by** (*simp add: order-le-less-trans [OF \*]*)

**qed**

**qed**

**next**

**assume**  $\neg 0 < K$

```

hence  $K: K \leq 0$  by (simp only: not-less)
show ?thesis
proof (rule ZfunI)
  fix  $r :: \text{real}$ 
  assume  $0 < r$ 
  from  $g$  show eventually  $(\lambda x. \text{norm } (g\ x) < r)$  net
  proof (rule eventually-elim1)
    fix  $x$ 
    assume  $\text{norm } (g\ x) \leq \text{norm } (f\ x) * K$ 
    also have  $\dots \leq \text{norm } (f\ x) * 0$ 
      using  $K$  norm-ge-zero by (rule mult-left-mono)
    finally show  $\text{norm } (g\ x) < r$ 
      using  $\langle 0 < r \rangle$  by simp
  qed
qed
qed

lemma Zfun-le:  $\llbracket \text{Zfun } g \text{ net}; \forall x. \text{norm } (f\ x) \leq \text{norm } (g\ x) \rrbracket \implies \text{Zfun } f \text{ net}$ 
by (erule-tac K=1 in Zfun-imp-Zfun, simp)

lemma Zfun-add:
  assumes  $f: \text{Zfun } f \text{ net}$  and  $g: \text{Zfun } g \text{ net}$ 
  shows  $\text{Zfun } (\lambda x. f\ x + g\ x) \text{ net}$ 
proof (rule ZfunI)
  fix  $r :: \text{real}$  assume  $0 < r$ 
  hence  $r: 0 < r / 2$  by simp
  have eventually  $(\lambda x. \text{norm } (f\ x) < r/2)$  net
    using  $f\ r$  by (rule ZfunD)
  moreover
  have eventually  $(\lambda x. \text{norm } (g\ x) < r/2)$  net
    using  $g\ r$  by (rule ZfunD)
  ultimately
  show eventually  $(\lambda x. \text{norm } (f\ x + g\ x) < r)$  net
  proof (rule eventually-elim2)
    fix  $x$ 
    assume *:  $\text{norm } (f\ x) < r/2$   $\text{norm } (g\ x) < r/2$ 
    have  $\text{norm } (f\ x + g\ x) \leq \text{norm } (f\ x) + \text{norm } (g\ x)$ 
      by (rule norm-triangle-ineq)
    also have  $\dots < r/2 + r/2$ 
      using * by (rule add-strict-mono)
    finally show  $\text{norm } (f\ x + g\ x) < r$ 
      by simp
  qed
qed

lemma Zfun-minus:  $\text{Zfun } f \text{ net} \implies \text{Zfun } (\lambda x. - f\ x) \text{ net}$ 
unfolding Zfun-def by simp

lemma Zfun-diff:  $\llbracket \text{Zfun } f \text{ net}; \text{Zfun } g \text{ net} \rrbracket \implies \text{Zfun } (\lambda x. f\ x - g\ x) \text{ net}$ 

```

by (simp only: diff-minus Zfun-add Zfun-minus)

lemma (in bounded-linear) Zfun:

assumes  $g: \text{Zfun } g \text{ net}$

shows  $\text{Zfun } (\lambda x. f (g x)) \text{ net}$

proof –

obtain  $K$  where  $\bigwedge x. \text{norm } (f x) \leq \text{norm } x * K$

using bounded by fast

then have eventually  $(\lambda x. \text{norm } (f (g x)) \leq \text{norm } (g x) * K) \text{ net}$

by simp

with  $g$  show ?thesis

by (rule Zfun-imp-Zfun)

qed

lemma (in bounded-bilinear) Zfun:

assumes  $f: \text{Zfun } f \text{ net}$

assumes  $g: \text{Zfun } g \text{ net}$

shows  $\text{Zfun } (\lambda x. f x ** g x) \text{ net}$

proof (rule ZfunI)

fix  $r::\text{real}$  assume  $r: 0 < r$

obtain  $K$  where  $K: 0 < K$

and norm-le:  $\bigwedge x y. \text{norm } (x ** y) \leq \text{norm } x * \text{norm } y * K$

using pos-bounded by fast

from  $K$  have  $K': 0 < \text{inverse } K$

by (rule positive-imp-inverse-positive)

have eventually  $(\lambda x. \text{norm } (f x) < r) \text{ net}$

using  $f$  by (rule ZfunD)

moreover

have eventually  $(\lambda x. \text{norm } (g x) < \text{inverse } K) \text{ net}$

using  $g$  by (rule ZfunD)

ultimately

show eventually  $(\lambda x. \text{norm } (f x ** g x) < r) \text{ net}$

proof (rule eventually-elim2)

fix  $x$

assume \*:  $\text{norm } (f x) < r \text{ norm } (g x) < \text{inverse } K$

have  $\text{norm } (f x ** g x) \leq \text{norm } (f x) * \text{norm } (g x) * K$

by (rule norm-le)

also have  $\text{norm } (f x) * \text{norm } (g x) * K < r * \text{inverse } K * K$

by (intro mult-strict-right-mono mult-strict-mono' norm-ge-zero \*  $K$ )

also from  $K$  have  $r * \text{inverse } K * K = r$

by simp

finally show  $\text{norm } (f x ** g x) < r$ .

qed

qed

lemma (in bounded-bilinear) Zfun-left:

$\text{Zfun } f \text{ net} \implies \text{Zfun } (\lambda x. f x ** a) \text{ net}$

by (rule bounded-linear-left [THEN bounded-linear.Zfun])

**lemma** (in *bounded-bilinear*) *Zfun-right*:  
 $Zfun\ f\ net \implies Zfun\ (\lambda x. a ** f\ x)\ net$   
**by** (rule *bounded-linear-right* [THEN *bounded-linear.Zfun*])

**lemmas** *Zfun-mult* = *mult.Zfun*  
**lemmas** *Zfun-mult-right* = *mult.Zfun-right*  
**lemmas** *Zfun-mult-left* = *mult.Zfun-left*

## 70.9 Limits

### definition

$tendsto :: ('a \Rightarrow 'b :: topological-space) \Rightarrow 'b \Rightarrow 'a\ net \Rightarrow bool$   
**(infixr**  $---->$  55)  
**where** [code del]:  
 $(f\ ----> l)\ net \iff (\forall S. open\ S \longrightarrow l \in S \longrightarrow eventually\ (\lambda x. f\ x \in S)\ net)$

**ML**  $\ll$   
 $structure\ Tendsto-Intros = Named-Thms$   
 $($   
 $\quad val\ name = tendsto-intros$   
 $\quad val\ description = introduction\ rules\ for\ tendsto$   
 $)$   
 $\gg$

**setup** *Tendsto-Intros.setup*

**lemma** *tendsto-mono*:  $net \leq net' \implies (f\ ----> l)\ net' \implies (f\ ----> l)\ net$   
**unfolding** *tendsto-def le-net-def* **by** *fast*

**lemma** *topological-tendstoI*:  
 $(\bigwedge S. open\ S \implies l \in S \implies eventually\ (\lambda x. f\ x \in S)\ net) \implies (f\ ----> l)\ net$   
**unfolding** *tendsto-def* **by** *auto*

**lemma** *topological-tendstoD*:  
 $(f\ ----> l)\ net \implies open\ S \implies l \in S \implies eventually\ (\lambda x. f\ x \in S)\ net$   
**unfolding** *tendsto-def* **by** *auto*

**lemma** *tendstoI*:  
**assumes**  $\bigwedge e. 0 < e \implies eventually\ (\lambda x. dist\ (f\ x)\ l < e)\ net$   
**shows**  $(f\ ----> l)\ net$   
**apply** (rule *topological-tendstoI*)  
**apply** (simp add: *open-dist*)  
**apply** (drule (1) *bspec, clarify*)  
**apply** (drule *assms*)  
**apply** (erule *eventually-elim1, simp*)  
**done**

**lemma** *tendstoD*:

```

  (f ----> l) net ==> 0 < e ==> eventually (λx. dist (f x) l < e) net
  apply (drule-tac S={x. dist x l < e} in topological-tendstoD)
  apply (clarsimp simp add: open-dist)
  apply (rule-tac x=e - dist x l in exI, clarsimp)
  apply (simp only: less-diff-eq)
  apply (erule le-less-trans [OF dist-triangle])
  apply simp
  apply simp
  done

```

```

lemma tendsto-iff:
  (f ----> l) net <=> (∀ e>0. eventually (λx. dist (f x) l < e) net)
  using tendstoI tendstoD by fast

```

```

lemma tendsto-Zfun-iff: (f ----> a) net = Zfun (λx. f x - a) net
  by (simp only: tendsto-iff Zfun-def dist-norm)

```

```

lemma tendsto-ident-at [tendsto-intros]: ((λx. x) ----> a) (at a)
  unfolding tendsto-def eventually-at-topological by auto

```

```

lemma tendsto-ident-at-within [tendsto-intros]:
  ((λx. x) ----> a) (at a within S)
  unfolding tendsto-def eventually-within eventually-at-topological by auto

```

```

lemma tendsto-const [tendsto-intros]: ((λx. k) ----> k) net
  by (simp add: tendsto-def)

```

```

lemma tendsto-const-iff:
  fixes k l :: 'a::metric-space
  assumes net ≠ bot shows ((λn. k) ----> l) net <=> k = l
  apply (safe intro!: tendsto-const)
  apply (rule ccontr)
  apply (drule-tac e=dist k l in tendstoD)
  apply (simp add: zero-less-dist-iff)
  apply (simp add: eventually-False assms)
  done

```

```

lemma tendsto-dist [tendsto-intros]:
  assumes f: (f ----> l) net and g: (g ----> m) net
  shows ((λx. dist (f x) (g x)) ----> dist l m) net
  proof (rule tendstoI)
    fix e :: real assume 0 < e
    hence e2: 0 < e/2 by simp
    from tendstoD [OF f e2] tendstoD [OF g e2]
    show eventually (λx. dist (dist (f x) (g x)) (dist l m) < e) net
    proof (rule eventually-elim2)
      fix x assume dist (f x) l < e/2 dist (g x) m < e/2
      then show dist (dist (f x) (g x)) (dist l m) < e
        unfolding dist-real-def

```



```

    using dist-triangle2 [of f x g x l]
    using dist-triangle2 [of g x l m]
    using dist-triangle3 [of l m f x]
    using dist-triangle [of f x m g x]
    by arith
  qed
qed

```

**lemma** *norm-conv-dist*:  $\text{norm } x = \text{dist } x \ 0$   
**unfolding** *dist-norm* **by** *simp*

**lemma** *tendsto-norm* [*tendsto-intros*]:  
 $(f \text{ ----> } a) \text{ net} \implies ((\lambda x. \text{norm } (f x)) \text{ ----> } \text{norm } a) \text{ net}$   
**unfolding** *norm-conv-dist* **by** (*intro tendsto-intros*)

**lemma** *tendsto-norm-zero*:  
 $(f \text{ ----> } 0) \text{ net} \implies ((\lambda x. \text{norm } (f x)) \text{ ----> } 0) \text{ net}$   
**by** (*drule tendsto-norm, simp*)

**lemma** *tendsto-norm-zero-cancel*:  
 $((\lambda x. \text{norm } (f x)) \text{ ----> } 0) \text{ net} \implies (f \text{ ----> } 0) \text{ net}$   
**unfolding** *tendsto-iff dist-norm* **by** *simp*

**lemma** *tendsto-norm-zero-iff*:  
 $((\lambda x. \text{norm } (f x)) \text{ ----> } 0) \text{ net} \longleftrightarrow (f \text{ ----> } 0) \text{ net}$   
**unfolding** *tendsto-iff dist-norm* **by** *simp*

**lemma** *add-diff-add*:  
**fixes**  $a \ b \ c \ d :: 'a::\text{ab-group-add}$   
**shows**  $(a + c) - (b + d) = (a - b) + (c - d)$   
**by** *simp*

**lemma** *minus-diff-minus*:  
**fixes**  $a \ b :: 'a::\text{ab-group-add}$   
**shows**  $(- a) - (- b) = - (a - b)$   
**by** *simp*

**lemma** *tendsto-add* [*tendsto-intros*]:  
**fixes**  $a \ b :: 'a::\text{real-normed-vector}$   
**shows**  $[(f \text{ ----> } a) \text{ net}; (g \text{ ----> } b) \text{ net}] \implies ((\lambda x. f x + g x) \text{ ----> } a + b) \text{ net}$   
**by** (*simp only: tendsto-Zfun-iff add-diff-add Zfun-add*)

**lemma** *tendsto-minus* [*tendsto-intros*]:  
**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $(f \text{ ----> } a) \text{ net} \implies ((\lambda x. - f x) \text{ ----> } - a) \text{ net}$   
**by** (*simp only: tendsto-Zfun-iff minus-diff-minus Zfun-minus*)

**lemma** *tendsto-minus-cancel*:

**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $((\lambda x. - f x) \dashrightarrow - a) \text{ net} \implies (f \dashrightarrow a) \text{ net}$   
**by**  $(\text{drule tendsto-minus, simp})$

**lemma** *tendsto-diff* [*tendsto-intros*]:  
**fixes**  $a b :: 'a::\text{real-normed-vector}$   
**shows**  $\llbracket (f \dashrightarrow a) \text{ net}; (g \dashrightarrow b) \text{ net} \rrbracket \implies ((\lambda x. f x - g x) \dashrightarrow a - b) \text{ net}$   
**by**  $(\text{simp add: diff-minus tendsto-add tendsto-minus})$

**lemma** *tendsto-setsum* [*tendsto-intros*]:  
**fixes**  $f :: 'a \Rightarrow 'b \Rightarrow 'c::\text{real-normed-vector}$   
**assumes**  $\bigwedge i. i \in S \implies (f i \dashrightarrow a i) \text{ net}$   
**shows**  $((\lambda x. \sum_{i \in S} f i x) \dashrightarrow (\sum_{i \in S} a i)) \text{ net}$   
**proof**  $(\text{cases finite } S)$   
**assume** *finite S* **thus** ?thesis **using** *assms*  
**proof**  $(\text{induct set: finite})$   
**case** *empty* **show** ?case  
**by**  $(\text{simp add: tendsto-const})$   
**next**  
**case**  $(\text{insert } i F)$  **thus** ?case  
**by**  $(\text{simp add: tendsto-add})$   
**qed**  
**next**  
**assume**  $\neg \text{finite } S$  **thus** ?thesis  
**by**  $(\text{simp add: tendsto-const})$   
**qed**

**lemma**  $(\text{in bounded-linear})$  *tendsto* [*tendsto-intros*]:  
 $(g \dashrightarrow a) \text{ net} \implies ((\lambda x. f (g x)) \dashrightarrow f a) \text{ net}$   
**by**  $(\text{simp only: tendsto-Zfun-iff diff [symmetric] Zfun})$

**lemma**  $(\text{in bounded-bilinear})$  *tendsto* [*tendsto-intros*]:  
 $\llbracket (f \dashrightarrow a) \text{ net}; (g \dashrightarrow b) \text{ net} \rrbracket \implies ((\lambda x. f x ** g x) \dashrightarrow a ** b) \text{ net}$   
**by**  $(\text{simp only: tendsto-Zfun-iff prod-diff-prod Zfun-add Zfun Zfun-left Zfun-right})$

## 70.10 Continuity of Inverse

**lemma**  $(\text{in bounded-bilinear})$  *Zfun-prod-Bfun*:  
**assumes**  $f: \text{Zfun } f \text{ net}$   
**assumes**  $g: \text{Bfun } g \text{ net}$   
**shows**  $\text{Zfun } (\lambda x. f x ** g x) \text{ net}$   
**proof** –  
**obtain**  $K$  **where**  $K: 0 \leq K$   
**and** *norm-le*:  $\bigwedge x y. \text{norm } (x ** y) \leq \text{norm } x * \text{norm } y * K$   
**using** *nonneg-bounded* **by** *fast*  
**obtain**  $B$  **where**  $B: 0 < B$   
**and** *norm-g*: *eventually*  $(\lambda x. \text{norm } (g x) \leq B) \text{ net}$

```

    using g by (rule BfunE)
  have eventually ( $\lambda x. \text{norm } (f x ** g x) \leq \text{norm } (f x) * (B * K)$ ) net
  using norm-g proof (rule eventually-elim1)
  fix x
  assume *:  $\text{norm } (g x) \leq B$ 
  have  $\text{norm } (f x ** g x) \leq \text{norm } (f x) * \text{norm } (g x) * K$ 
    by (rule norm-le)
  also have  $\dots \leq \text{norm } (f x) * B * K$ 
    by (intro mult-mono' order-refl norm-g norm-ge-zero
        mult-nonneg-nonneg K *)
  also have  $\dots = \text{norm } (f x) * (B * K)$ 
    by (rule mult-assoc)
  finally show  $\text{norm } (f x ** g x) \leq \text{norm } (f x) * (B * K)$  .
qed
with f show ?thesis
  by (rule Zfun-imp-Zfun)
qed

```

**lemma** (in bounded-bilinear) flip:

```

  bounded-bilinear ( $\lambda x y. y ** x$ )
  apply default
  apply (rule add-right)
  apply (rule add-left)
  apply (rule scaleR-right)
  apply (rule scaleR-left)
  apply (subst mult-commute)
  using bounded by fast

```

**lemma** (in bounded-bilinear) Bfun-prod-Zfun:

```

  assumes f: Bfun f net
  assumes g: Zfun g net
  shows Zfun ( $\lambda x. f x ** g x$ ) net
  using flip g f by (rule bounded-bilinear.Zfun-prod-Bfun)

```

**lemma** inverse-diff-inverse:

```

   $\llbracket (a :: 'a :: \text{division-ring}) \neq 0; b \neq 0 \rrbracket$ 
   $\implies \text{inverse } a - \text{inverse } b = - (\text{inverse } a * (a - b) * \text{inverse } b)$ 
  by (simp add: algebra-simps)

```

**lemma** Bfun-inverse-lemma:

```

  fixes x :: 'a :: real-normed-div-algebra
  shows  $\llbracket r \leq \text{norm } x; 0 < r \rrbracket \implies \text{norm } (\text{inverse } x) \leq \text{inverse } r$ 
  apply (subst nonzero-norm-inverse, clarsimp)
  apply (erule (1) le-imp-inverse-le)
  done

```

**lemma** Bfun-inverse:

```

  fixes a :: 'a :: real-normed-div-algebra
  assumes f: ( $f \dashrightarrow a$ ) net

```

```

assumes  $a: a \neq 0$ 
shows  $Bfun (\lambda x. inverse (f x)) \text{ net}$ 
proof –
  from  $a$  have  $0 < norm\ a$  by simp
  hence  $\exists r > 0. r < norm\ a$  by (rule dense)
  then obtain  $r$  where  $r1: 0 < r$  and  $r2: r < norm\ a$  by fast
  have eventually  $(\lambda x. dist\ (f\ x)\ a < r)$  net
    using tendstoD [OF f r1] by fast
  hence eventually  $(\lambda x. norm\ (inverse\ (f\ x)) \leq inverse\ (norm\ a - r))$  net
  proof (rule eventually-elim1)
    fix  $x$ 
    assume  $dist\ (f\ x)\ a < r$ 
    hence  $1: norm\ (f\ x - a) < r$ 
      by (simp add: dist-norm)
    hence  $2: f\ x \neq 0$  using  $r2$  by auto
    hence  $norm\ (inverse\ (f\ x)) = inverse\ (norm\ (f\ x))$ 
      by (rule nonzero-norm-inverse)
    also have  $\dots \leq inverse\ (norm\ a - r)$ 
  proof (rule le-imp-inverse-le)
    show  $0 < norm\ a - r$  using  $r2$  by simp
  next
    have  $norm\ a - norm\ (f\ x) \leq norm\ (a - f\ x)$ 
      by (rule norm-triangle-ineq2)
    also have  $\dots = norm\ (f\ x - a)$ 
      by (rule norm-minus-commute)
    also have  $\dots < r$  using  $1$  .
    finally show  $norm\ a - r \leq norm\ (f\ x)$  by simp
  qed
  finally show  $norm\ (inverse\ (f\ x)) \leq inverse\ (norm\ a - r)$  .
qed
thus ?thesis by (rule BfunI)
qed

```

**lemma** *tendsto-inverse-lemma*:

```

fixes  $a :: 'a::real-normed-div-algebra$ 
shows  $\llbracket (f \dashrightarrow a) \text{ net}; a \neq 0; \text{eventually } (\lambda x. f\ x \neq 0) \text{ net} \rrbracket$ 
   $\implies ((\lambda x. inverse\ (f\ x)) \dashrightarrow inverse\ a) \text{ net}$ 
apply (subst tendsto-Zfun-iff)
apply (rule Zfun-ssubst)
apply (erule eventually-elim1)
apply (erule (1) inverse-diff-inverse)
apply (rule Zfun-minus)
apply (rule Zfun-mult-left)
apply (rule mult.Bfun-prod-Zfun)
apply (erule (1) Bfun-inverse)
apply (simp add: tendsto-Zfun-iff)
done

```

**lemma** *tendsto-inverse [tendsto-intros]*:

```

fixes  $a :: 'a::\text{real-normed-div-algebra}$ 
assumes  $f: (f \dashrightarrow a) \text{ net}$ 
assumes  $a: a \neq 0$ 
shows  $((\lambda x. \text{inverse } (f x)) \dashrightarrow \text{inverse } a) \text{ net}$ 
proof –
  from  $a$  have  $0 < \text{norm } a$  by simp
  with  $f$  have eventually  $(\lambda x. \text{dist } (f x) a < \text{norm } a) \text{ net}$ 
    by (rule tendstoD)
  then have eventually  $(\lambda x. f x \neq 0) \text{ net}$ 
    unfolding dist-norm by (auto elim!: eventually-elim1)
  with  $f$  show ?thesis
    by (rule tendsto-inverse-lemma)
qed

lemma tendsto-divide [tendsto-intros]:
  fixes  $a b :: 'a::\text{real-normed-field}$ 
  shows  $\llbracket (f \dashrightarrow a) \text{ net}; (g \dashrightarrow b) \text{ net}; b \neq 0 \rrbracket$ 
     $\implies ((\lambda x. f x / g x) \dashrightarrow a / b) \text{ net}$ 
by (simp add: mult.tendsto tendsto-inverse divide-inverse)

end

```

## 71 SEQ: Sequences and Convergence

```

theory SEQ
imports Limits RComplete
begin

```

### abbreviation

```

LIMSEQ ::  $(\text{nat} \Rightarrow 'a::\text{topological-space}, 'a) \Rightarrow \text{bool}$ 
   $((((-) / \dashrightarrow (-)) [60, 60] 60) \text{ where}$ 
   $X \dashrightarrow L \equiv (X \dashrightarrow L) \text{ sequentially}$ 

```

### definition

```

lim ::  $(\text{nat} \Rightarrow 'a::\text{metric-space}) \Rightarrow 'a$  where
  — Standard definition of limit using choice operator
   $\text{lim } X = (\text{THE } L. X \dashrightarrow L)$ 

```

### definition

```

convergent ::  $(\text{nat} \Rightarrow 'a::\text{metric-space}) \Rightarrow \text{bool}$  where
  — Standard definition of convergence
   $\text{convergent } X = (\exists L. X \dashrightarrow L)$ 

```

### definition

```

Bseq ::  $(\text{nat} \Rightarrow 'a::\text{real-normed-vector}) \Rightarrow \text{bool}$  where
  — Standard definition for bounded sequence
  [code del]:  $\text{Bseq } X = (\exists K > 0. \forall n. \text{norm } (X n) \leq K)$ 

```

**definition**

*monoseq* :: (*nat* => *real*) => *bool* **where**

— Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.

[code del]: *monoseq* *X* = (( $\forall m. \forall n \geq m. X\ m \leq X\ n$ ) | ( $\forall m. \forall n \geq m. X\ n \leq X\ m$ ))

**definition**

*incseq* :: (*nat* => *real*) => *bool* **where**

— Increasing sequence

[code del]: *incseq* *X* = ( $\forall m. \forall n \geq m. X\ m \leq X\ n$ )

**definition**

*decseq* :: (*nat* => *real*) => *bool* **where**

— Increasing sequence

[code del]: *decseq* *X* = ( $\forall m. \forall n \geq m. X\ n \leq X\ m$ )

**definition**

*subseq* :: (*nat* => *nat*) => *bool* **where**

— Definition of subsequence

[code del]: *subseq* *f* = ( $\forall m. \forall n > m. (f\ m) < (f\ n)$ )

**definition**

*Cauchy* :: (*nat* => 'a::metric-space) => *bool* **where**

— Standard definition of the Cauchy condition

[code del]: *Cauchy* *X* = ( $\forall e > 0. \exists M. \forall m \geq M. \forall n \geq M. dist\ (X\ m)\ (X\ n) < e$ )

**71.1 Bounded Sequences**

**lemma** *BseqI'*: **assumes** *K*:  $\bigwedge n. norm\ (X\ n) \leq K$  **shows** *Bseq* *X*

**unfolding** *Bseq-def*

**proof** (*intro exI conjI allI*)

**show**  $0 < max\ K\ 1$  **by** *simp*

**next**

**fix** *n*::*nat*

**have**  $norm\ (X\ n) \leq K$  **by** (*rule* *K*)

**thus**  $norm\ (X\ n) \leq max\ K\ 1$  **by** *simp*

**qed**

**lemma** *BseqE*:  $\llbracket Bseq\ X; \bigwedge K. \llbracket 0 < K; \forall n. norm\ (X\ n) \leq K \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$

**unfolding** *Bseq-def* **by** *auto*

**lemma** *BseqI2'*: **assumes** *K*:  $\forall n \geq N. norm\ (X\ n) \leq K$  **shows** *Bseq* *X*

**proof** (*rule* *BseqI'*)

**let** ?*A* =  $norm\ 'X\ ' \{..N\}$

**have** *1*: *finite* ?*A* **by** *simp*

**fix** *n*::*nat*

```

show  $\text{norm } (X\ n) \leq \max K\ (Max\ ?A)$ 
proof (cases rule: linorder-le-cases)
  assume  $n \geq N$ 
  hence  $\text{norm } (X\ n) \leq K$  using  $K$  by simp
  thus  $\text{norm } (X\ n) \leq \max K\ (Max\ ?A)$  by simp
next
  assume  $n \leq N$ 
  hence  $\text{norm } (X\ n) \in ?A$  by simp
  with 1 have  $\text{norm } (X\ n) \leq Max\ ?A$  by (rule Max-ge)
  thus  $\text{norm } (X\ n) \leq \max K\ (Max\ ?A)$  by simp
qed
qed

```

```

lemma Bseq-ignore-initial-segment:  $Bseq\ X \implies Bseq\ (\lambda n. X\ (n + k))$ 
unfolding Bseq-def by auto

```

```

lemma Bseq-offset:  $Bseq\ (\lambda n. X\ (n + k)) \implies Bseq\ X$ 
apply (erule BseqE)
apply (rule-tac N=k and K=K in BseqI2')
apply clarify
apply (drule-tac x=n - k in spec, simp)
done

```

```

lemma Bseq-conv-Bfun:  $Bseq\ X \longleftrightarrow Bfun\ X\ \text{sequentially}$ 
unfolding Bfun-def eventually-sequentially
apply (rule iffI)
apply (simp add: Bseq-def)
apply (auto intro: BseqI2')
done

```

## 71.2 Limits of Sequences

```

lemma [trans]:  $X=Y \implies Y \dashrightarrow z \implies X \dashrightarrow z$ 
by simp

```

```

lemma LIMSEQ-def:  $X \dashrightarrow L = (\forall r>0. \exists no. \forall n \geq no. \text{dist } (X\ n)\ L < r)$ 
unfolding tendsto-iff eventually-sequentially ..

```

```

lemma LIMSEQ-iff:
  fixes  $L :: 'a::\text{real-normed-vector}$ 
  shows  $(X \dashrightarrow L) = (\forall r>0. \exists no. \forall n \geq no. \text{norm } (X\ n - L) < r)$ 
unfolding LIMSEQ-def dist-norm ..

```

```

lemma LIMSEQ-iff-nz:  $X \dashrightarrow L = (\forall r>0. \exists no>0. \forall n \geq no. \text{dist } (X\ n)\ L < r)$ 
unfolding LIMSEQ-def by (metis Suc-leD zero-less-Suc)

```

```

lemma LIMSEQ-Zfun-iff:  $((\lambda n. X\ n) \dashrightarrow L) = Zfun\ (\lambda n. X\ n - L)\ \text{sequentially}$ 
by (rule tendsto-Zfun-iff)

```

**lemma** *metric-LIMSEQ-I*:

$(\bigwedge r. 0 < r \implies \exists no. \forall n \geq no. \text{dist } (X\ n) \ L < r) \implies X \text{ ----} > L$   
**by** (*simp add: LIMSEQ-def*)

**lemma** *metric-LIMSEQ-D*:

$\llbracket X \text{ ----} > L; 0 < r \rrbracket \implies \exists no. \forall n \geq no. \text{dist } (X\ n) \ L < r$   
**by** (*simp add: LIMSEQ-def*)

**lemma** *LIMSEQ-I*:

**fixes**  $L :: 'a::\text{real-normed-vector}$   
**shows**  $(\bigwedge r. 0 < r \implies \exists no. \forall n \geq no. \text{norm } (X\ n - L) < r) \implies X \text{ ----} > L$   
**by** (*simp add: LIMSEQ-iff*)

**lemma** *LIMSEQ-D*:

**fixes**  $L :: 'a::\text{real-normed-vector}$   
**shows**  $\llbracket X \text{ ----} > L; 0 < r \rrbracket \implies \exists no. \forall n \geq no. \text{norm } (X\ n - L) < r$   
**by** (*simp add: LIMSEQ-iff*)

**lemma** *LIMSEQ-const*:  $(\lambda n. k) \text{ ----} > k$

**by** (*rule tendsto-const*)

**lemma** *LIMSEQ-const-iff*:

**fixes**  $k\ l :: 'a::\text{metric-space}$   
**shows**  $(\lambda n. k) \text{ ----} > l \longleftrightarrow k = l$   
**by** (*rule tendsto-const-iff, rule sequentially-bot*)

**lemma** *LIMSEQ-norm*:

**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $X \text{ ----} > a \implies (\lambda n. \text{norm } (X\ n)) \text{ ----} > \text{norm } a$   
**by** (*rule tendsto-norm*)

**lemma** *LIMSEQ-ignore-initial-segment*:

$f \text{ ----} > a \implies (\lambda n. f\ (n + k)) \text{ ----} > a$   
**apply** (*rule topological-tendstoI*)  
**apply** (*drule (2) topological-tendstoD*)  
**apply** (*simp only: eventually-sequentially*)  
**apply** (*erule exE, rename-tac N*)  
**apply** (*rule-tac x=N in exI*)  
**apply** *simp*  
**done**

**lemma** *LIMSEQ-offset*:

$(\lambda n. f\ (n + k)) \text{ ----} > a \implies f \text{ ----} > a$   
**apply** (*rule topological-tendstoI*)  
**apply** (*drule (2) topological-tendstoD*)  
**apply** (*simp only: eventually-sequentially*)  
**apply** (*erule exE, rename-tac N*)  
**apply** (*rule-tac x=N + k in exI*)



**apply** *clarify*  
**apply** (*drule-tac*  $x=n-k$  **in** *spec*)  
**apply** (*simp add:* *le-diff-conv2*)  
**done**

**lemma** *LIMSEQ-Suc*:  $f \text{ ----> } l \implies (\lambda n. f (Suc\ n)) \text{ ----> } l$   
**by** (*drule-tac*  $k=Suc\ 0$  **in** *LIMSEQ-ignore-initial-segment, simp*)

**lemma** *LIMSEQ-imp-Suc*:  $(\lambda n. f (Suc\ n)) \text{ ----> } l \implies f \text{ ----> } l$   
**by** (*rule-tac*  $k=Suc\ 0$  **in** *LIMSEQ-offset, simp*)

**lemma** *LIMSEQ-Suc-iff*:  $(\lambda n. f (Suc\ n)) \text{ ----> } l = f \text{ ----> } l$   
**by** (*blast intro: LIMSEQ-imp-Suc LIMSEQ-Suc*)

**lemma** *LIMSEQ-linear*:  $\llbracket X \text{ ----> } x ; l > 0 \rrbracket \implies (\lambda n. X\ (n * l)) \text{ ----> } x$   
**unfolding** *tendsto-def eventually-sequentially*  
**by** (*metis div-le-dividend div-mult-self1-is-m le-trans nat-mult-commute*)

**lemma** *LIMSEQ-add*:  
**fixes**  $a\ b :: 'a::\text{real-normed-vector}$   
**shows**  $\llbracket X \text{ ----> } a ; Y \text{ ----> } b \rrbracket \implies (\lambda n. X\ n + Y\ n) \text{ ----> } a + b$   
**by** (*rule tendsto-add*)

**lemma** *LIMSEQ-minus*:  
**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $X \text{ ----> } a \implies (\lambda n. -\ X\ n) \text{ ----> } -\ a$   
**by** (*rule tendsto-minus*)

**lemma** *LIMSEQ-minus-cancel*:  
**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $(\lambda n. -\ X\ n) \text{ ----> } -\ a \implies X \text{ ----> } a$   
**by** (*rule tendsto-minus-cancel*)

**lemma** *LIMSEQ-diff*:  
**fixes**  $a\ b :: 'a::\text{real-normed-vector}$   
**shows**  $\llbracket X \text{ ----> } a ; Y \text{ ----> } b \rrbracket \implies (\lambda n. X\ n - Y\ n) \text{ ----> } a - b$   
**by** (*rule tendsto-diff*)

**lemma** *LIMSEQ-unique*:  
**fixes**  $a\ b :: 'a::\text{metric-space}$   
**shows**  $\llbracket X \text{ ----> } a ; X \text{ ----> } b \rrbracket \implies a = b$   
**apply** (*rule ccontr*)  
**apply** (*drule-tac*  $r=\text{dist } a\ b / 2$  **in** *metric-LIMSEQ-D, simp add: zero-less-dist-iff*)  
**apply** (*drule-tac*  $r=\text{dist } a\ b / 2$  **in** *metric-LIMSEQ-D, simp add: zero-less-dist-iff*)  
**apply** (*clarify, rename-tac*  $M\ N$ )  
**apply** (*subgoal-tac*  $\text{dist } a\ b < \text{dist } a\ b / 2 + \text{dist } a\ b / 2$ , *simp*)  
**apply** (*subgoal-tac*  $\text{dist } a\ b \leq \text{dist } (X\ (\max\ M\ N))\ a + \text{dist } (X\ (\max\ M\ N))\ b$ )  
**apply** (*erule le-less-trans, rule add-strict-mono, simp, simp*)

**apply** (*subst dist-commute, rule dist-triangle*)  
**done**

**lemma** (*in bounded-linear*) *LIMSEQ*:  
 $X \text{ ----> } a \implies (\lambda n. f (X n)) \text{ ----> } f a$   
**by** (*rule tendsto*)

**lemma** (*in bounded-bilinear*) *LIMSEQ*:  
 $\llbracket X \text{ ----> } a; Y \text{ ----> } b \rrbracket \implies (\lambda n. X n ** Y n) \text{ ----> } a ** b$   
**by** (*rule tendsto*)

**lemma** *LIMSEQ-mult*:  
**fixes**  $a b :: 'a::\text{real-normed-algebra}$   
**shows**  $\llbracket X \text{ ----> } a; Y \text{ ----> } b \rrbracket \implies (\%n. X n * Y n) \text{ ----> } a * b$   
**by** (*rule mult.tendsto*)

**lemma** *increasing-LIMSEQ*:  
**fixes**  $f :: \text{nat} \Rightarrow \text{real}$   
**assumes** *inc*:  $!!n. f n \leq f (\text{Suc } n)$   
**and** *bdd*:  $!!n. f n \leq l$   
**and** *en*:  $!!e. 0 < e \implies \exists n. l \leq f n + e$   
**shows**  $f \text{ ----> } l$   
**proof** (*auto simp add: LIMSEQ-def*)  
**fix**  $e :: \text{real}$   
**assume**  $e: 0 < e$   
**then obtain**  $N$  **where**  $l \leq f N + e/2$   
**by** (*metis half-gt-zero e en that*)  
**hence**  $N: l < f N + e$  **using**  $e$   
**by** *simp*  
**{ fix**  $k$   
**have** [*simp*]:  $!!n. |f n - l| = l - f n$   
**by** (*simp add: bdd*)  
**have**  $|f (N+k) - l| < e$   
**proof** (*induct k*)  
**case 0** **show** *?case* **using**  $N$   
**by** *simp*  
**next**  
**case** (*Suc k*) **thus** *?case* **using**  $N$  *inc* [*of N+k*]  
**by** *simp*  
**qed**  
**} note 1 = this**  
**{ fix**  $n$   
**have**  $N \leq n \implies |f n - l| < e$  **using** 1 [*of n-N*]  
**by** *simp*  
**} note [intro] = this**  
**show**  $\exists no. \forall n \geq no. \text{dist } (f n) l < e$   
**by** (*auto simp add: dist-real-def*)  
**qed**

**lemma** *Bseq-inverse-lemma*:

**fixes**  $x :: 'a::\text{real-normed-div-algebra}$   
  **shows**  $\llbracket r \leq \text{norm } x; 0 < r \rrbracket \Longrightarrow \text{norm } (\text{inverse } x) \leq \text{inverse } r$   
**apply** (*subst nonzero-norm-inverse, clarsimp*)  
**apply** (*erule (1) le-imp-inverse-le*)  
**done**

**lemma** *Bseq-inverse*:

**fixes**  $a :: 'a::\text{real-normed-div-algebra}$   
  **shows**  $\llbracket X \text{ ----} > a; a \neq 0 \rrbracket \Longrightarrow Bseq (\lambda n. \text{inverse } (X \ n))$   
**unfolding** *Bseq-conv-Bfun* **by** (*rule Bfun-inverse*)

**lemma** *LIMSEQ-inverse*:

**fixes**  $a :: 'a::\text{real-normed-div-algebra}$   
  **shows**  $\llbracket X \text{ ----} > a; a \neq 0 \rrbracket \Longrightarrow (\lambda n. \text{inverse } (X \ n)) \text{ ----} > \text{inverse } a$   
**by** (*rule tendsto-inverse*)

**lemma** *LIMSEQ-divide*:

**fixes**  $a \ b :: 'a::\text{real-normed-field}$   
  **shows**  $\llbracket X \text{ ----} > a; Y \text{ ----} > b; b \neq 0 \rrbracket \Longrightarrow (\lambda n. X \ n / Y \ n) \text{ ----} > a / b$   
**by** (*rule tendsto-divide*)

**lemma** *LIMSEQ-pow*:

**fixes**  $a :: 'a::\{\text{power, real-normed-algebra}\}$   
  **shows**  $X \text{ ----} > a \Longrightarrow (\lambda n. (X \ n) ^ m) \text{ ----} > a ^ m$   
**by** (*induct m*) (*simp-all add: LIMSEQ-const LIMSEQ-mult*)

**lemma** *LIMSEQ-setsum*:

**fixes**  $L :: 'a \Rightarrow 'b::\text{real-normed-vector}$   
  **assumes**  $n: \bigwedge n. n \in S \Longrightarrow X \ n \text{ ----} > L \ n$   
  **shows**  $(\lambda m. \sum_{n \in S}. X \ n \ m) \text{ ----} > (\sum_{n \in S}. L \ n)$   
**using** *assms* **by** (*rule tendsto-setsum*)

**lemma** *LIMSEQ-setprod*:

**fixes**  $L :: 'a \Rightarrow 'b::\{\text{real-normed-algebra, comm-ring-1}\}$   
  **assumes**  $n: \bigwedge n. n \in S \Longrightarrow X \ n \text{ ----} > L \ n$   
  **shows**  $(\lambda m. \prod_{n \in S}. X \ n \ m) \text{ ----} > (\prod_{n \in S}. L \ n)$   
**proof** (*cases finite S*)  
  **case** *True*  
  **thus** *?thesis* **using** *n*  
  **proof** (*induct*)  
  **case** *empty*  
  **show** *?case*  
  **by** (*simp add: LIMSEQ-const*)  
**next**  
  **case** *insert*  
  **thus** *?case*  
  **by** (*simp add: LIMSEQ-mult*)

```

qed
next
  case False
  thus ?thesis
    by (simp add: setprod-def LIMSEQ-const)
qed

```

```

lemma LIMSEQ-add-const:
  fixes a :: 'a::real-normed-vector
  shows  $f \text{ ----> } a \implies (\%n. (f\ n + b)) \text{ ----> } a + b$ 
  by (intro tendsto-intros)

```

```

lemma LIMSEQ-add-minus:
  fixes a b :: 'a::real-normed-vector
  shows  $[| X \text{ ----> } a; Y \text{ ----> } b |] \implies (\%n. X\ n + -Y\ n) \text{ ----> } a + -b$ 
  by (intro tendsto-intros)

```

```

lemma LIMSEQ-diff-const:
  fixes a b :: 'a::real-normed-vector
  shows  $f \text{ ----> } a \implies (\%n. (f\ n - b)) \text{ ----> } a - b$ 
  by (intro tendsto-intros)

```

```

lemma LIMSEQ-diff-approach-zero:
  fixes L :: 'a::real-normed-vector
  shows  $g \text{ ----> } L \implies (\%x. f\ x - g\ x) \text{ ----> } 0 \implies f \text{ ----> } L$ 
  by (drule (1) LIMSEQ-add, simp)

```

```

lemma LIMSEQ-diff-approach-zero2:
  fixes L :: 'a::real-normed-vector
  shows  $f \text{ ----> } L \implies (\%x. f\ x - g\ x) \text{ ----> } 0 \implies g \text{ ----> } L$ 
  by (drule (1) LIMSEQ-diff, simp)

```

A sequence tends to zero iff its abs does

```

lemma LIMSEQ-norm-zero:
  fixes X :: nat  $\Rightarrow$  'a::real-normed-vector
  shows  $((\lambda n. \text{norm } (X\ n)) \text{ ----> } 0) \longleftrightarrow (X \text{ ----> } 0)$ 
  by (simp add: LIMSEQ-iff)

```

```

lemma LIMSEQ-rabs-zero:  $((\%n. |f\ n|) \text{ ----> } 0) = (f \text{ ----> } (0::real))$ 
  by (simp add: LIMSEQ-iff)

```

```

lemma LIMSEQ-imp-rabs:  $f \text{ ----> } (l::real) \implies (\%n. |f\ n|) \text{ ----> } |l|$ 
  by (drule LIMSEQ-norm, simp)

```

An unbounded sequence's inverse tends to 0

```

lemma LIMSEQ-inverse-zero:
   $\forall r::real. \exists N. \forall n \geq N. r < X\ n \implies (\lambda n. \text{inverse } (X\ n)) \text{ ----> } 0$ 

```

```

apply (rule LIMSEQ-I)
apply (drule-tac x=inverse r in spec, safe)
apply (rule-tac x=N in exI, safe)
apply (drule-tac x=n in spec, safe)
apply (frule positive-imp-inverse-positive)
apply (frule (1) less-imp-inverse-less)
apply (subgoal-tac 0 < X n, simp)
apply (erule (1) order-less-trans)
done

```

The sequence  $(1::'a) / n$  tends to 0 as  $n$  tends to infinity

```

lemma LIMSEQ-inverse-real-of-nat: (%n. inverse(real(Suc n))) ----> 0
apply (rule LIMSEQ-inverse-zero, safe)
apply (cut-tac x = r in reals-Archimedean2)
apply (safe, rule-tac x = n in exI)
apply (auto simp add: real-of-nat-Suc)
done

```

The sequence  $r + (1::'a) / n$  tends to  $r$  as  $n$  tends to infinity is now easily proved

```

lemma LIMSEQ-inverse-real-of-nat-add:
  (%n. r + inverse(real(Suc n))) ----> r
by (cut-tac LIMSEQ-add [OF LIMSEQ-const LIMSEQ-inverse-real-of-nat], auto)

```

```

lemma LIMSEQ-inverse-real-of-nat-add-minus:
  (%n. r + -inverse(real(Suc n))) ----> r
by (cut-tac LIMSEQ-add-minus [OF LIMSEQ-const LIMSEQ-inverse-real-of-nat],
  auto)

```

```

lemma LIMSEQ-inverse-real-of-nat-add-minus-mult:
  (%n. r*(1 + -inverse(real(Suc n)))) ----> r
by (cut-tac b=1 in
  LIMSEQ-mult [OF LIMSEQ-const LIMSEQ-inverse-real-of-nat-add-minus],
  auto)

```

```

lemma LIMSEQ-le-const:
  [X ----> (x::real);  $\exists N. \forall n \geq N. a \leq X n$ ]  $\implies a \leq x$ 
apply (rule ccontr, simp only: linorder-not-le)
apply (drule-tac r=a - x in LIMSEQ-D, simp)
apply clarsimp
apply (drule-tac x=max N no in spec, drule mp, rule le-maxI1)
apply (drule-tac x=max N no in spec, drule mp, rule le-maxI2)
apply simp
done

```

```

lemma LIMSEQ-le-const2:
  [X ----> (x::real);  $\exists N. \forall n \geq N. X n \leq a$ ]  $\implies x \leq a$ 
apply (subgoal-tac - a  $\leq$  - x, simp)
apply (rule LIMSEQ-le-const)

```

```

apply (erule LIMSEQ-minus)
apply simp
done

```

```

lemma LIMSEQ-le:
   $\llbracket X \text{ ----} > x; Y \text{ ----} > y; \exists N. \forall n \geq N. X\ n \leq Y\ n \rrbracket \implies x \leq (y::real)$ 
apply (subgoal-tac  $0 \leq y - x$ , simp)
apply (rule LIMSEQ-le-const)
apply (erule (1) LIMSEQ-diff)
apply (simp add: le-diff-eq)
done

```

### 71.3 Convergence

```

lemma limI:  $X \text{ ----} > L \implies \lim X = L$ 
apply (simp add: lim-def)
apply (blast intro: LIMSEQ-unique)
done

```

```

lemma convergentD:  $\text{convergent } X \implies \exists L. (X \text{ ----} > L)$ 
by (simp add: convergent-def)

```

```

lemma convergentI:  $(X \text{ ----} > L) \implies \text{convergent } X$ 
by (auto simp add: convergent-def)

```

```

lemma convergent-LIMSEQ-iff:  $\text{convergent } X = (X \text{ ----} > \lim X)$ 
by (auto intro: theI LIMSEQ-unique simp add: convergent-def lim-def)

```

```

lemma convergent-const:  $\text{convergent } (\lambda n. c)$ 
by (rule convergentI, rule LIMSEQ-const)

```

```

lemma convergent-add:
  fixes  $X\ Y :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$ 
  assumes  $\text{convergent } (\lambda n. X\ n)$ 
  assumes  $\text{convergent } (\lambda n. Y\ n)$ 
  shows  $\text{convergent } (\lambda n. X\ n + Y\ n)$ 
using assms unfolding convergent-def by (fast intro: LIMSEQ-add)

```

```

lemma convergent-setsum:
  fixes  $X :: 'a \Rightarrow \text{nat} \Rightarrow 'b::\text{real-normed-vector}$ 
  assumes  $\bigwedge i. i \in A \implies \text{convergent } (\lambda n. X\ i\ n)$ 
  shows  $\text{convergent } (\lambda n. \sum_{i \in A} X\ i\ n)$ 
proof (cases finite A)
  case True from this and assms show ?thesis
    by (induct A set: finite) (simp-all add: convergent-const convergent-add)
qed (simp add: convergent-const)

```

```

lemma (in bounded-linear) convergent:
  assumes  $\text{convergent } (\lambda n. X\ n)$ 

```

**shows** *convergent* ( $\lambda n. f (X n)$ )  
**using** *assms unfolding convergent-def* **by** (*fast intro: LIMSEQ*)

**lemma** (*in bounded-bilinear*) *convergent*:  
**assumes** *convergent* ( $\lambda n. X n$ ) **and** *convergent* ( $\lambda n. Y n$ )  
**shows** *convergent* ( $\lambda n. X n ** Y n$ )  
**using** *assms unfolding convergent-def* **by** (*fast intro: LIMSEQ*)

**lemma** *convergent-minus-iff*:  
**fixes**  $X :: nat \Rightarrow 'a::real-normed-vector$   
**shows** *convergent*  $X \longleftrightarrow$  *convergent* ( $\lambda n. - X n$ )  
**apply** (*simp add: convergent-def*)  
**apply** (*auto dest: LIMSEQ-minus*)  
**apply** (*drule LIMSEQ-minus, auto*)  
**done**

**lemma** *lim-le*:  
**fixes**  $x :: real$   
**assumes**  $f: convergent f$  **and**  $fn-le: !!n. f n \leq x$   
**shows**  $lim f \leq x$   
**proof** (*rule classical*)  
**assume**  $\neg lim f \leq x$   
**hence**  $0: 0 < lim f - x$  **by** *arith*  
**have**  $1: f \dashrightarrow lim f$   
**by** (*metis convergent-LIMSEQ-iff f*)  
**thus** *?thesis*  
**proof** (*simp add: LIMSEQ-iff*)  
**assume**  $\forall r > 0. \exists no. \forall n \geq no. |f n - lim f| < r$   
**hence**  $\exists no. \forall n \geq no. |f n - lim f| < lim f - x$   
**by** (*metis 0*)  
**from this obtain no where**  $\forall n \geq no. |f n - lim f| < lim f - x$   
**by** *blast*  
**thus**  $lim f \leq x$   
**by** (*metis add-cancel-end add-minus-cancel diff-def linorder-linear*  
*linorder-not-le minus-diff-eq abs-diff-less-iff fn-le*)  
**qed**  
**qed**

Given a binary function  $f :: nat \Rightarrow 'a \Rightarrow 'a$ , its values are uniquely determined by a function  $g$

**lemma** *nat-function-unique*:  $EX! g. g 0 = e \wedge (\forall n. g (Suc n) = f n (g n))$   
**unfolding** *Ex1-def*  
**apply** (*rule-tac x=nat-rec e f in exI*)  
**apply** (*rule conjI*)  
**apply** (*rule def-nat-rec-0, simp*)  
**apply** (*rule allI, rule def-nat-rec-Suc, simp*)  
**apply** (*rule allI, rule impI, rule ext*)  
**apply** (*erule conjE*)  
**apply** (*induct-tac x*)

```

apply simp
apply (erule-tac  $x=n$  in allE)
apply (simp)
done

```

Subsequence (alternative definition, (e.g. Hoskins))

```

lemma subseq-Suc-iff:  $\text{subseq } f = (\forall n. (f\ n) < (f\ (\text{Suc } n)))$ 
apply (simp add: subseq-def)
apply (auto dest!: less-imp-Suc-add)
apply (induct-tac k)
apply (auto intro: less-trans)
done

```

```

lemma monoseq-Suc:
   $\text{monoseq } X = ((\forall n. X\ n \leq X\ (\text{Suc } n)) \mid (\forall n. X\ (\text{Suc } n) \leq X\ n))$ 
apply (simp add: monoseq-def)
apply (auto dest!: le-imp-less-or-eq)
apply (auto intro!: lessI [THEN less-imp-le] dest!: less-imp-Suc-add)
apply (induct-tac ka)
apply (auto intro: order-trans)
apply (erule contrapos-np)
apply (induct-tac k)
apply (auto intro: order-trans)
done

```

```

lemma monoI1:  $\forall m. \forall n \geq m. X\ m \leq X\ n \implies \text{monoseq } X$ 
by (simp add: monoseq-def)

```

```

lemma monoI2:  $\forall m. \forall n \geq m. X\ n \leq X\ m \implies \text{monoseq } X$ 
by (simp add: monoseq-def)

```

```

lemma mono-SucI1:  $\forall n. X\ n \leq X\ (\text{Suc } n) \implies \text{monoseq } X$ 
by (simp add: monoseq-Suc)

```

```

lemma mono-SucI2:  $\forall n. X\ (\text{Suc } n) \leq X\ n \implies \text{monoseq } X$ 
by (simp add: monoseq-Suc)

```

```

lemma monoseq-minus: assumes monoseq a
  shows  $\text{monoseq } (\lambda n. -\ a\ n)$ 
proof (cases  $\forall m. \forall n \geq m. a\ m \leq a\ n$ )
  case True
    hence  $\forall m. \forall n \geq m. -\ a\ n \leq -\ a\ m$  by auto
    thus ?thesis by (rule monoI2)
  next
    case False
    hence  $\forall m. \forall n \geq m. -\ a\ m \leq -\ a\ n$  using  $\langle \text{monoseq } a \rangle[\text{unfolded monoseq-def}]$ 
  by auto
  thus ?thesis by (rule monoI1)

```



qed

**lemma monoseq-le:** **assumes** *monoseq a* **and**  $a \dashrightarrow x$

**shows**  $((\forall n. a \ n \leq x) \wedge (\forall m. \forall n \geq m. a \ m \leq a \ n)) \vee$   
 $((\forall n. x \leq a \ n) \wedge (\forall m. \forall n \geq m. a \ n \leq a \ m))$

**proof** –

{ **fix**  $x \ n$  **fix**  $a :: \text{nat} \Rightarrow \text{real}$

**assume**  $a \dashrightarrow x$  **and**  $\forall m. \forall n \geq m. a \ m \leq a \ n$

**hence** *monotone*:  $\bigwedge m \ n. m \leq n \implies a \ m \leq a \ n$  **by** *auto*

**have**  $a \ n \leq x$

**proof** (*rule ccontr*)

**assume**  $\neg a \ n \leq x$  **hence**  $x < a \ n$  **by** *auto*

**hence**  $0 < a \ n - x$  **by** *auto*

**from**  $\langle a \dashrightarrow x \rangle$  [*THEN LIMSEQ-D, OF this*]

**obtain** *no* **where**  $\bigwedge n'. no \leq n' \implies \text{norm } (a \ n' - x) < a \ n - x$  **by** *blast*

**hence**  $\text{norm } (a \ (\text{max no } n) - x) < a \ n - x$  **by** *auto*

**moreover**

{ **fix**  $n'$  **have**  $n \leq n' \implies x < a \ n'$  **using** *monotone* [*where m=n and n=n'*]

**and**  $\langle x < a \ n \rangle$  **by** *auto* }

**hence**  $x < a \ (\text{max no } n)$  **by** *auto*

**ultimately**

**have**  $a \ (\text{max no } n) < a \ n$  **by** *auto*

**with** *monotone* [*where m=n and n=max no n*]

**show** *False* **by** (*auto simp:max-def split:split-if-asm*)

qed

} **note** *top-down* = *this*

{ **fix**  $x \ n \ m$  **fix**  $a :: \text{nat} \Rightarrow \text{real}$

**assume**  $a \dashrightarrow x$  **and** *monoseq a* **and**  $a \ m < x$

**have**  $a \ n \leq x \wedge (\forall m. \forall n \geq m. a \ m \leq a \ n)$

**proof** (*cases*  $\forall m. \forall n \geq m. a \ m \leq a \ n$ )

**case** *True* **with** *top-down* **and**  $\langle a \dashrightarrow x \rangle$  **show** *?thesis* **by** *auto*

**next**

**case** *False* **with**  $\langle \text{monoseq } a \rangle$  [*unfolded monoseq-def*] **have**  $\forall m. \forall n \geq m. -$

$a \ m \leq - a \ n$  **by** *auto*

**hence**  $- a \ m \leq - x$  **using** *top-down* [*OF LIMSEQ-minus*] [*OF*  $\langle a \dashrightarrow x \rangle$ ]

**by** *blast*

**hence** *False* **using**  $\langle a \ m < x \rangle$  **by** *auto*

**thus** *?thesis* ..

qed

} **note** *when-decided* = *this*

**show** *?thesis*

**proof** (*cases*  $\exists m. a \ m \neq x$ )

**case** *True* **then** **obtain**  $m$  **where**  $a \ m \neq x$  **by** *auto*

**show** *?thesis*

**proof** (*cases*  $a \ m < x$ )

**case** *True* **with** *when-decided* [*OF*  $\langle a \dashrightarrow x \rangle$ ]  $\langle \text{monoseq } a \rangle$ , **where**  $m2=m$ ]

**show** *?thesis* **by** *blast*

**next**

```

      case False hence  $- a m < - x$  using  $\langle a m \neq x \rangle$  by auto
    with when-decided[OF LIMSEQ-minus[OF  $\langle a \dashrightarrow x \rangle$ ] monoseq-minus[OF
 $\langle \text{monoseq } a \rangle$ ], where  $m2=m$ ]
      show ?thesis by auto
    qed
  qed auto
qed

```

for any sequence, there is a monotonic subsequence

**lemma** seq-monosub:  $\exists f. \text{subseq } f \wedge \text{monoseq } (\lambda n. (s (f n)))$

**proof**–

```

  {assume H:  $\forall n. \exists p > n. \forall m \geq p. s m \leq s p$ 
    let ?P =  $\lambda p n. p > n \wedge (\forall m \geq p. s m \leq s p)$ 
    from nat-function-unique[of SOME p. ?P p 0  $\lambda p n. \text{SOME } p. ?P p n$ ]
    obtain f where f:  $f 0 = (\text{SOME } p. ?P p 0) \wedge \forall n. f (\text{Suc } n) = (\text{SOME } p. ?P$ 
 $p (f n))$  by blast
    have ?P (f 0) 0 unfolding f(1) some-eq-ex[of  $\lambda p. ?P p 0$ ]
      using H apply –
      apply (erule allE[where  $x=0$ ], erule exE, rule-tac  $x=p$  in exI)
      unfolding order-le-less by blast
    hence f0:  $f 0 > 0 \wedge \forall m \geq f 0. s m \leq s (f 0)$  by blast+
    {fix n
      have ?P (f (Suc n)) (f n)
        unfolding f(2)[rule-format, of n] some-eq-ex[of  $\lambda p. ?P p (f n)$ ]
        using H apply –
        apply (erule allE[where  $x=f n$ ], erule exE, rule-tac  $x=p$  in exI)
        unfolding order-le-less by blast
      hence  $f (\text{Suc } n) > f n \wedge \forall m \geq f (\text{Suc } n). s m \leq s (f (\text{Suc } n))$  by blast+}
    note fSuc = this
    {fix p q assume pq:  $p \geq f q$ 
      have  $s p \leq s(f(q))$  using f0(2)[rule-format, of p] pq fSuc
      by (cases q, simp-all) }
    note pqth = this
    {fix q
      have  $f (\text{Suc } q) > f q$  apply (induct q)
      using f0(1) fSuc(1)[of 0] apply simp by (rule fSuc(1))}
    note fss = this
    from fss have th1: subseq f unfolding subseq-Suc-iff ..
    {fix a b
      have  $f a \leq f (a + b)$ 
      proof(induct b)
        case 0 thus ?case by simp
      next
        case (Suc b)
        from fSuc(1)[of a + b] Suc.hyps show ?case by simp
      qed}
    note fmon0 = this
    have monoseq  $(\lambda n. s (f n))$ 
  }
proof–

```

```

{fix n
  have  $s (f n) \geq s (f (Suc n))$ 
  proof(cases n)
    case 0
      assume  $n0: n = 0$ 
      from  $fSuc(1)[of\ 0]$  have  $th0: f\ 0 \leq f (Suc\ 0)$  by simp
      from  $f0(2)[rule-format, OF\ th0]$  show ?thesis using  $n0$  by simp
    next
      case (Suc m)
      assume  $m: n = Suc\ m$ 
      from  $fSuc(1)[of\ n]\ m$  have  $th0: f (Suc\ m) \leq f (Suc (Suc\ m))$  by simp
      from  $m\ fSuc(2)[rule-format, OF\ th0]$  show ?thesis by simp
  qed}
thus monoseq ( $\lambda n. s (f n)$ ) unfolding monoseq-Suc by blast
qed
with th1 have ?thesis by blast}
moreover
{fix N assume  $N: \forall p > N. \exists m \geq p. s\ m > s\ p$ 
  {fix p assume  $p: p \geq Suc\ N$ 
    hence  $pN: p > N$  by arith with N obtain m where  $m: m \geq p\ s\ m > s\ p$ 
  by blast
    have  $m \neq p$  using  $m(2)$  by auto
    with m have  $\exists m > p. s\ p < s\ m$  by - (rule exI[where  $x=m$ ], auto)}
  note th0 = this
  let ?P =  $\lambda m\ x. m > x \wedge s\ x < s\ m$ 
  from nat-function-unique[of SOME x. ?P x (Suc N)  $\lambda m\ x. SOME\ y. ?P\ y\ x]$ 
  obtain f where  $f: f\ 0 = (SOME\ x. ?P\ x (Suc\ N))$ 
     $\forall n. f (Suc\ n) = (SOME\ m. ?P\ m (f\ n))$  by blast
  have ?P (f 0) (Suc N) unfolding f(1) some-eq-ex[of  $\lambda p. ?P\ p (Suc\ N)$ ]
    using N apply -
    apply (erule allE[where  $x=Suc\ N$ ], clarsimp)
    apply (rule-tac  $x=m$  in exI)
    apply auto
    apply (subgoal-tac  $Suc\ N \neq m$ )
    apply simp
    apply (rule ccontr, simp)
  done
  hence  $f0: f\ 0 > Suc\ N\ s (Suc\ N) < s (f\ 0)$  by blast+
  {fix n
    have  $f\ n > N \wedge ?P (f (Suc\ n)) (f\ n)$ 
      unfolding f(2)[rule-format, of n] some-eq-ex[of  $\lambda p. ?P\ p (f\ n)$ ]
    proof (induct n)
      case 0 thus ?case
        using f0 N apply auto
        apply (erule allE[where  $x=f\ 0$ ], clarsimp)
        apply (rule-tac  $x=m$  in exI, simp)
        by (subgoal-tac  $f\ 0 \neq m$ , auto)
      next
        case (Suc n)

```

```

from Suc.hyps have Nfn:  $N < f\ n$  by blast
from Suc.hyps obtain m where  $m: m > f\ n\ s\ (f\ n) < s\ m$  by blast
with Nfn have mN:  $m > N$  by arith
note key = Suc.hyps[unfolded some-eq-ex[of  $\lambda p. ?P\ p\ (f\ n)$ , symmetric]]
f(2)[rule-format, of n, symmetric]]

from key have th0:  $f\ (Suc\ n) > N$  by simp
from N[rule-format, OF th0]
obtain m' where  $m': m' \geq f\ (Suc\ n)\ s\ (f\ (Suc\ n)) < s\ m'$  by blast
have  $m' \neq f\ (Suc\ n)$  apply (rule ccontr) using m'2 by auto
hence  $m' > f\ (Suc\ n)$  using m'1 by simp
with key m'2 show ?case by auto
qed}
note fSuc = this
{fix n
have  $f\ n \geq Suc\ N \wedge f\ (Suc\ n) > f\ n \wedge s\ (f\ n) < s\ (f\ (Suc\ n))$  using fSuc[of
n] by auto
hence  $f\ n \geq Suc\ N\ f\ (Suc\ n) > f\ n\ s\ (f\ n) < s\ (f\ (Suc\ n))$  by blast+}
note thf = this
have sqf: subseq f unfolding subseq-Suc-iff using thf by simp
have monoseq ( $\lambda n. s\ (f\ n)$ ) unfolding monoseq-Suc using thf
apply –
apply (rule disjI1)
apply auto
apply (rule order-less-imp-le)
apply blast
done
then have ?thesis using sqf by blast}
ultimately show ?thesis unfolding linorder-not-less[symmetric] by blast
qed

lemma seq-suble: assumes sf: subseq f shows  $n \leq f\ n$ 
proof(induct n)
case 0 thus ?case by simp
next
case (Suc n)
from sf[unfolded subseq-Suc-iff, rule-format, of n] Suc.hyps
have  $n < f\ (Suc\ n)$  by arith
thus ?case by arith
qed

lemma LIMSEQ-subseq-LIMSEQ:
 $\llbracket X \dashrightarrow L; \text{subseq } f \rrbracket \implies (X \circ f) \dashrightarrow L$ 
apply (rule topological-tendstoI)
apply (drule (2) topological-tendstoD)
apply (simp only: eventually-sequentially)
apply (clarify, rule-tac x=N in exI, clarsimp)
apply (blast intro: seq-suble le-trans dest!:: spec)
done

```

## 71.4 Bounded Monotonic Sequences

Bounded Sequence

**lemma** *BseqD*:  $Bseq\ X \implies \exists K. 0 < K \ \& \ (\forall n. norm\ (X\ n) \leq K)$   
**by** (*simp add: Bseq-def*)

**lemma** *BseqI*:  $[\![\ 0 < K; \forall n. norm\ (X\ n) \leq K\ ]\!] \implies Bseq\ X$   
**by** (*auto simp add: Bseq-def*)

**lemma** *lemma-NBseq-def*:

$(\exists K > 0. \forall n. norm\ (X\ n) \leq K) =$   
 $(\exists N. \forall n. norm\ (X\ n) \leq real(Suc\ N))$

**proof** *auto*

**fix**  $K :: real$

**from** *reals-Archimedean2* **obtain**  $n :: nat$  **where**  $K < real\ n$  ..

**then have**  $K \leq real\ (Suc\ n)$  **by** *auto*

**assume**  $\forall m. norm\ (X\ m) \leq K$

**have**  $\forall m. norm\ (X\ m) \leq real\ (Suc\ n)$

**proof**

**fix**  $m :: 'a$

**from**  $\langle \forall m. norm\ (X\ m) \leq K \rangle$  **have**  $norm\ (X\ m) \leq K$  ..

**with**  $\langle K \leq real\ (Suc\ n) \rangle$  **show**  $norm\ (X\ m) \leq real\ (Suc\ n)$  **by** *auto*

**qed**

**then show**  $\exists N. \forall n. norm\ (X\ n) \leq real\ (Suc\ N)$  ..

**next**

**fix**  $N :: nat$

**have**  $real\ (Suc\ N) > 0$  **by** (*simp add: real-of-nat-Suc*)

**moreover assume**  $\forall n. norm\ (X\ n) \leq real\ (Suc\ N)$

**ultimately show**  $\exists K > 0. \forall n. norm\ (X\ n) \leq K$  **by** *blast*

**qed**

alternative definition for Bseq

**lemma** *Bseq-iff*:  $Bseq\ X = (\exists N. \forall n. norm\ (X\ n) \leq real(Suc\ N))$

**apply** (*simp add: Bseq-def*)

**apply** (*simp (no-asm) add: lemma-NBseq-def*)

**done**

**lemma** *lemma-NBseq-def2*:

$(\exists K > 0. \forall n. norm\ (X\ n) \leq K) = (\exists N. \forall n. norm\ (X\ n) < real(Suc\ N))$

**apply** (*subst lemma-NBseq-def, auto*)

**apply** (*rule-tac x = Suc\ N in exI*)

**apply** (*rule-tac [2] x = N in exI*)

**apply** (*auto simp add: real-of-nat-Suc*)

**prefer 2 apply** (*blast intro: order-less-imp-le*)

**apply** (*drule-tac x = n in spec, simp*)

**done**

**lemma** *Bseq-iff1a*:  $Bseq\ X = (\exists N. \forall n. norm\ (X\ n) < real(Suc\ N))$

by (simp add: Bseq-def lemma-NBseq-def2)

#### 71.4.1 Upper Bounds and Lubs of Bounded Sequences

**lemma** Bseq-isUb:

!!(X::nat=>real). Bseq X ==>  $\exists U. \text{isUb } (UNIV::\text{real set}) \{x. \exists n. X\ n = x\} U$   
 by (auto intro: isUbI settleI simp add: Bseq-def abs-le-iff)

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

**lemma** Bseq-isLub:

!!(X::nat=>real). Bseq X ==>  
 $\exists U. \text{isLub } (UNIV::\text{real set}) \{x. \exists n. X\ n = x\} U$   
 by (blast intro: reals-complete Bseq-isUb)

#### 71.4.2 A Bounded and Monotonic Sequence Converges

**lemma** lemma-converg1:

!!(X::nat=>real). [ $\forall m. \forall n \geq m. X\ m \leq X\ n$ ;  
 $\text{isLub } (UNIV::\text{real set}) \{x. \exists n. X\ n = x\} (X\ ma)$ ]  
 $\implies \forall n \geq ma. X\ n = X\ ma$

apply safe

apply (drule-tac  $y = X\ n$  in isLubD2)

apply (blast dest: order-antisym)+

done

**lemma** Bmonoseq-LIMSEQ:  $\forall n. m \leq n \longrightarrow X\ n = X\ m \implies \exists L. (X \dashrightarrow L)$

unfolding tendsto-def eventually-sequentially

apply (rule-tac  $x = X\ m$  in exI, safe)

apply (rule-tac  $x = m$  in exI, safe)

apply (drule spec, erule impE, auto)

done

**lemma** lemma-converg2:

!!(X::nat=>real).  
 $[\forall m. X\ m \sim U; \text{isLub } UNIV \{x. \exists n. X\ n = x\} U] \implies \forall m. X\ m < U$

apply safe

apply (drule-tac  $y = X\ m$  in isLubD2)

apply (auto dest!: order-le-imp-less-or-eq)

done

**lemma** lemma-converg3:  $!!(X::nat=>real). \forall m. X\ m \leq U \implies \text{isUb } UNIV \{x. \exists n. X\ n = x\} U$

by (rule settleI [THEN isUbI], auto)

FIXME:  $U - T < U$  is redundant

**lemma** lemma-converg4:  $!!(X::nat=>real).$

$[\forall m. X\ m \sim U;$

```

      isLub UNIV {x.  $\exists n. X\ n = x$ } U;
      0 < T;
      U + - T < U
    || ==>  $\exists m. U + -T < X\ m \ \& \ X\ m < U$ 
  apply (drule lemma-converg2, assumption)
  apply (rule ccontr, simp)
  apply (simp add: linorder-not-less)
  apply (drule lemma-converg3)
  apply (drule isLub-le-isUb, assumption)
  apply (auto dest: order-less-le-trans)
done

```

A standard proof of the theorem for monotone increasing sequence

```

lemma Bseq-mono-convergent:
  [| Bseq X;  $\forall m. \forall n \geq m. X\ m \leq X\ n$  |] ==> convergent (X::nat=>real)
  apply (simp add: convergent-def)
  apply (frule Bseq-isLub, safe)
  apply (case-tac  $\exists m. X\ m = U$ , auto)
  apply (blast dest: lemma-converg1 Bmonoseq-LIMSEQ)

  apply (rule-tac  $x = U$  in exI)
  apply (subst LIMSEQ-iff, safe)
  apply (frule lemma-converg2, assumption)
  apply (drule lemma-converg4, auto)
  apply (rule-tac  $x = m$  in exI, safe)
  apply (subgoal-tac  $X\ m \leq X\ n$ )
  prefer 2 apply blast
  apply (drule-tac  $x=n$  and  $P=\%m. X\ m < U$  in spec, arith)
done

```

```

lemma Bseq-minus-iff: Bseq (%n. -(X n)) = Bseq X
by (simp add: Bseq-def)

```

Main monotonicity theorem

```

lemma Bseq-monoseq-convergent: [| Bseq X; monoseq X |] ==> convergent X
  apply (simp add: monoseq-def, safe)
  apply (rule-tac [2] convergent-minus-iff [THEN ssubst])
  apply (drule-tac [2] Bseq-minus-iff [THEN ssubst])
  apply (auto intro!: Bseq-mono-convergent)
done

```

### 71.4.3 Increasing and Decreasing Series

```

lemma incseq-imp-monoseq: incseq X ==> monoseq X
  by (simp add: incseq-def monoseq-def)

```

```

lemma incseq-le: assumes inc: incseq X and lim:  $X \dashrightarrow L$  shows  $X\ n \leq L$ 
  using monoseq-le [OF incseq-imp-monoseq [OF inc] lim]

```

**proof**  
 assume  $(\forall n. X\ n \leq L) \wedge (\forall m\ n. m \leq n \longrightarrow X\ m \leq X\ n)$   
 thus ?thesis by simp  
**next**  
 assume  $(\forall n. L \leq X\ n) \wedge (\forall m\ n. m \leq n \longrightarrow X\ n \leq X\ m)$   
 hence const:  $(!!m\ n. m \leq n \implies X\ n = X\ m)$  using inc  
 by (auto simp add: incseq-def intro: order-antisym)  
 have  $X: !!n. X\ n = X\ 0$   
 by (blast intro: const [of 0])  
 have  $X = (\lambda n. X\ 0)$   
 by (blast intro: ext X)  
 hence  $L = X\ 0$  using LIMSEQ-const [of X 0]  
 by (auto intro: LIMSEQ-unique lim)  
 thus ?thesis  
 by (blast intro: eq-refl X)  
**qed**

**lemma incseq-SucI:**  
 assumes  $\bigwedge n. X\ n \leq X\ (Suc\ n)$   
 shows incseq X unfolding incseq-def  
**proof safe**  
 fix  $m\ n :: nat$   
 { fix  $d\ m :: nat$   
 have  $X\ m \leq X\ (m + d)$   
**proof** (induct d)  
 case (Suc d)  
 also have  $X\ (m + d) \leq X\ (m + Suc\ d)$   
 using assms by simp  
 finally show ?case .  
**qed simp** }  
 note this[of m n - m]  
 moreover assume  $m \leq n$   
 ultimately show  $X\ m \leq X\ n$  by simp  
**qed**

**lemma decseq-imp-monoseq:**  $decseq\ X \implies monoseq\ X$   
 by (simp add: decseq-def monoseq-def)

**lemma decseq-eq-incseq:**  $decseq\ X = incseq\ (\lambda n. -\ X\ n)$   
 by (simp add: decseq-def incseq-def)

**lemma decseq-le:** assumes dec:  $decseq\ X$  and lim:  $X\ ----> L$  shows  $L \leq X$   
 $n$   
**proof** -  
 have inc:  $incseq\ (\lambda n. -\ X\ n)$  using dec  
 by (simp add: decseq-eq-incseq)  
 have  $-\ X\ n \leq -\ L$   
 by (blast intro: incseq-le [OF inc] LIMSEQ-minus lim)



thus ?thesis  
 by simp  
 qed

#### 71.4.4 A Few More Equivalence Theorems for Boundedness

alternative formulation for boundedness

**lemma** *Bseq-iff2*:  $Bseq\ X = (\exists k > 0. \exists x. \forall n. norm\ (X(n) + -x) \leq k)$   
**apply** (*unfold Bseq-def, safe*)  
**apply** (*rule-tac [2] x = k + norm x in exI*)  
**apply** (*rule-tac x = K in exI, simp*)  
**apply** (*rule exI [where x = 0], auto*)  
**apply** (*erule order-less-le-trans, simp*)  
**apply** (*drule-tac x=n in spec, fold diff-def*)  
**apply** (*drule order-trans [OF norm-triangle-ineq2]*)  
**apply** *simp*  
**done**

alternative formulation for boundedness

**lemma** *Bseq-iff3*:  $Bseq\ X = (\exists k > 0. \exists N. \forall n. norm(X(n) + -X(N)) \leq k)$   
**apply** *safe*  
**apply** (*simp add: Bseq-def, safe*)  
**apply** (*rule-tac x = K + norm (X N) in exI*)  
**apply** *auto*  
**apply** (*erule order-less-le-trans, simp*)  
**apply** (*rule-tac x = N in exI, safe*)  
**apply** (*drule-tac x = n in spec*)  
**apply** (*rule order-trans [OF norm-triangle-ineq], simp*)  
**apply** (*auto simp add: Bseq-iff2*)  
**done**

**lemma** *BseqI2*:  $(\forall n. k \leq f\ n \ \& \ f\ n \leq (K::real)) \implies Bseq\ f$   
**apply** (*simp add: Bseq-def*)  
**apply** (*rule-tac x = (|k| + |K|) + 1 in exI, auto*)  
**apply** (*drule-tac x = n in spec, arith*)  
**done**

#### 71.5 Cauchy Sequences

**lemma** *metric-CauchyI*:

$(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. dist\ (X\ m)\ (X\ n) < e) \implies Cauchy\ X$   
**by** (*simp add: Cauchy-def*)

**lemma** *metric-CauchyD*:

$\llbracket Cauchy\ X; 0 < e \rrbracket \implies \exists M. \forall m \geq M. \forall n \geq M. dist\ (X\ m)\ (X\ n) < e$   
**by** (*simp add: Cauchy-def*)

**lemma** *Cauchy-iff*:

**fixes**  $X :: nat \Rightarrow 'a::real-normed-vector$

**shows**  $\text{Cauchy } X \iff (\forall e > 0. \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e)$   
**unfolding** *Cauchy-def dist-norm ..*

**lemma** *Cauchy-iff2*:

*Cauchy*  $X =$   
 $(\forall j. (\exists M. \forall m \geq M. \forall n \geq M. |X\ m - X\ n| < \text{inverse}(\text{real } (\text{Suc } j))))$   
**apply** (*simp add: Cauchy-iff, auto*)  
**apply** (*drule reals-Archimedean, safe*)  
**apply** (*drule-tac x = n in spec, auto*)  
**apply** (*rule-tac x = M in exI, auto*)  
**apply** (*drule-tac x = m in spec, simp*)  
**apply** (*drule-tac x = na in spec, auto*)  
**done**

**lemma** *CauchyI*:

**fixes**  $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$   
**shows**  $(\bigwedge e. 0 < e \implies \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e) \implies$   
*Cauchy*  $X$   
**by** (*simp add: Cauchy-iff*)

**lemma** *CauchyD*:

**fixes**  $X :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$   
**shows**  $\llbracket \text{Cauchy } X; 0 < e \rrbracket \implies \exists M. \forall m \geq M. \forall n \geq M. \text{norm } (X\ m - X\ n) < e$   
**by** (*simp add: Cauchy-iff*)

**lemma** *Cauchy-subseq-Cauchy*:

$\llbracket \text{Cauchy } X; \text{subseq } f \rrbracket \implies \text{Cauchy } (X \circ f)$   
**apply** (*auto simp add: Cauchy-def*)  
**apply** (*drule-tac x=e in spec, clarify*)  
**apply** (*rule-tac x=M in exI, clarify*)  
**apply** (*blast intro: le-trans [OF - seq-suble] dest!: spec*)  
**done**

### 71.5.1 Cauchy Sequences are Bounded

A Cauchy sequence is bounded – this is the standard proof mechanization rather than the nonstandard proof

**lemma** *lemmaCauchy*:  $\forall n \geq M. \text{norm } (X\ M - X\ n) < (1::\text{real})$   
 $\implies \forall n \geq M. \text{norm } (X\ n :: 'a::\text{real-normed-vector}) < 1 + \text{norm } (X\ M)$   
**apply** (*clarify, drule spec, drule (1) mp*)  
**apply** (*simp only: norm-minus-commute*)  
**apply** (*drule order-le-less-trans [OF norm-triangle-ineq2]*)  
**apply** *simp*  
**done**

**lemma** *Cauchy-Bseq*:  $\text{Cauchy } X \implies \text{Bseq } X$

**apply** (*simp add: Cauchy-iff*)  
**apply** (*drule spec, drule mp, rule zero-less-one, safe*)  
**apply** (*drule-tac x=M in spec, simp*)

```

apply (drule lemmaCauchy)
apply (rule-tac k=M in Bseq-offset)
apply (simp add: Bseq-def)
apply (rule-tac x=1 + norm (X M) in exI)
apply (rule conjI, rule order-less-le-trans [OF zero-less-one], simp)
apply (simp add: order-less-imp-le)
done

```

### 71.5.2 Cauchy Sequences are Convergent

```

class complete-space =
  assumes Cauchy-convergent: Cauchy X  $\implies$  convergent X

class banach = real-normed-vector + complete-space

theorem LIMSEQ-imp-Cauchy:
  assumes X: X  $\text{---->}$  a shows Cauchy X
proof (rule metric-CauchyI)
  fix e::real assume 0 < e
  hence 0 < e/2 by simp
  with X have  $\exists N. \forall n \geq N. \text{dist } (X\ n)\ a < e/2$  by (rule metric-LIMSEQ-D)
  then obtain N where N:  $\forall n \geq N. \text{dist } (X\ n)\ a < e/2$  ..
  show  $\exists N. \forall m \geq N. \forall n \geq N. \text{dist } (X\ m)\ (X\ n) < e$ 
  proof (intro exI allI impI)
    fix m assume N  $\leq$  m
    hence m:  $\text{dist } (X\ m)\ a < e/2$  using N by fast
    fix n assume N  $\leq$  n
    hence n:  $\text{dist } (X\ n)\ a < e/2$  using N by fast
    have  $\text{dist } (X\ m)\ (X\ n) \leq \text{dist } (X\ m)\ a + \text{dist } (X\ n)\ a$ 
      by (rule dist-triangle2)
    also from m n have  $\dots < e$  by simp
    finally show  $\text{dist } (X\ m)\ (X\ n) < e$  .
  qed
qed

```

```

lemma convergent-Cauchy: convergent X  $\implies$  Cauchy X
unfolding convergent-def
by (erule exE, erule LIMSEQ-imp-Cauchy)

```

```

lemma Cauchy-convergent-iff:
  fixes X :: nat  $\Rightarrow$  'a::complete-space
  shows Cauchy X = convergent X
by (fast intro: Cauchy-convergent convergent-Cauchy)

```

```

lemma convergent-subseq-convergent:
  fixes X :: nat  $\Rightarrow$  'a::complete-space
  shows  $\llbracket \text{convergent } X; \text{subseq } f \rrbracket \implies \text{convergent } (X \circ f)$ 
  by (simp add: Cauchy-subseq-Cauchy Cauchy-convergent-iff [symmetric])

```

Proof that Cauchy sequences converge based on the one from <http://pirate.shu.edu/~wachsmut/ira/nu>

If sequence  $X$  is Cauchy, then its limit is the lub of  $\{r. \exists N. \forall n \geq N. r < X\ n\}$

**lemma** *isUb-UNIV-I*:  $(\bigwedge y. y \in S \implies y \leq u) \implies \text{isUb UNIV } S\ u$   
**by** (*simp add: isUbI settleI*)

**locale** *real-Cauchy* =  
**fixes**  $X :: \text{nat} \Rightarrow \text{real}$   
**assumes**  $X: \text{Cauchy } X$   
**fixes**  $S :: \text{real set}$   
**defines**  $S\text{-def}$ :  $S \equiv \{x :: \text{real}. \exists N. \forall n \geq N. x < X\ n\}$

**lemma** *real-CauchyI*:  
**assumes**  $\text{Cauchy } X$   
**shows** *real-Cauchy*  $X$   
**proof** **qed** (*fact assms*)

**lemma** (**in** *real-Cauchy*) *mem-S*:  $\forall n \geq N. x < X\ n \implies x \in S$   
**by** (*unfold S-def, auto*)

**lemma** (**in** *real-Cauchy*) *bound-isUb*:  
**assumes**  $N: \forall n \geq N. X\ n < x$   
**shows** *isUb UNIV*  $S\ x$   
**proof** (*rule isUb-UNIV-I*)  
**fix**  $y :: \text{real}$  **assume**  $y \in S$   
**hence**  $\exists M. \forall n \geq M. y < X\ n$   
**by** (*simp add: S-def*)  
**then obtain**  $M$  **where**  $\forall n \geq M. y < X\ n ..$   
**hence**  $y < X\ (\max M N)$  **by** *simp*  
**also have**  $\dots < x$  **using**  $N$  **by** *simp*  
**finally show**  $y \leq x$   
**by** (*rule order-less-imp-le*)  
**qed**

**lemma** (**in** *real-Cauchy*) *isLub-ex*:  $\exists u. \text{isLub UNIV } S\ u$   
**proof** (*rule reals-complete*)  
**obtain**  $N$  **where**  $\forall m \geq N. \forall n \geq N. \text{norm } (X\ m - X\ n) < 1$   
**using** *CauchyD [OF X zero-less-one]* **by** *auto*  
**hence**  $N: \forall n \geq N. \text{norm } (X\ n - X\ N) < 1$  **by** *simp*  
**show**  $\exists x. x \in S$   
**proof**  
**from**  $N$  **have**  $\forall n \geq N. X\ N - 1 < X\ n$   
**by** (*simp add: abs-diff-less-iff*)  
**thus**  $X\ N - 1 \in S$  **by** (*rule mem-S*)  
**qed**  
**show**  $\exists u. \text{isUb UNIV } S\ u$   
**proof**  
**from**  $N$  **have**  $\forall n \geq N. X\ n < X\ N + 1$   
**by** (*simp add: abs-diff-less-iff*)  
**thus** *isUb UNIV*  $S\ (X\ N + 1)$

by (rule bound-isUb)  
 qed  
 qed

lemma (in real-Cauchy) isLub-imp-LIMSEQ:  
 assumes  $x: \text{isLub UNIV } S \ x$   
 shows  $X \dashrightarrow x$   
 proof (rule LIMSEQ-I)  
 fix  $r::\text{real}$  assume  $0 < r$   
 hence  $r: 0 < r/2$  by simp  
 obtain  $N$  where  $\forall n \geq N. \forall m \geq N. \text{norm } (X \ n - X \ m) < r/2$   
 using CauchyD [OF  $X \ r$ ] by auto  
 hence  $\forall n \geq N. \text{norm } (X \ n - X \ N) < r/2$  by simp  
 hence  $N: \forall n \geq N. X \ N - r/2 < X \ n \wedge X \ n < X \ N + r/2$   
 by (simp only: real-norm-def abs-diff-less-iff)

from  $N$  have  $\forall n \geq N. X \ N - r/2 < X \ n$  by fast  
 hence  $X \ N - r/2 \in S$  by (rule mem-S)  
 hence  $1: X \ N - r/2 \leq x$  using  $x \text{ isLub-isUb isUbD}$  by fast

from  $N$  have  $\forall n \geq N. X \ n < X \ N + r/2$  by fast  
 hence  $\text{isUb UNIV } S (X \ N + r/2)$  by (rule bound-isUb)  
 hence  $2: x \leq X \ N + r/2$  using  $x \text{ isLub-le-isUb}$  by fast

show  $\exists N. \forall n \geq N. \text{norm } (X \ n - x) < r$   
 proof (intro exI allI impI)  
 fix  $n$  assume  $n: N \leq n$   
 from  $N \ n$  have  $X \ n < X \ N + r/2$  and  $X \ N - r/2 < X \ n$  by simp+  
 thus  $\text{norm } (X \ n - x) < r$  using 1 2  
 by (simp add: abs-diff-less-iff)  
 qed  
 qed

lemma (in real-Cauchy) LIMSEQ-ex:  $\exists x. X \dashrightarrow x$   
 proof –  
 obtain  $x$  where  $\text{isLub UNIV } S \ x$   
 using isLub-ex by fast  
 hence  $X \dashrightarrow x$   
 by (rule isLub-imp-LIMSEQ)  
 thus ?thesis ..  
 qed

lemma real-Cauchy-convergent:  
 fixes  $X :: \text{nat} \Rightarrow \text{real}$   
 shows  $\text{Cauchy } X \implies \text{convergent } X$   
 unfolding convergent-def  
 by (rule real-Cauchy.LIMSEQ-ex)  
 (rule real-CauchyI)

```
instance real :: banach
by intro-classes (rule real-Cauchy-convergent)
```

## 71.6 Power Sequences

The sequence  $x^n$  tends to 0 if  $(0::'a) \leq x$  and  $x < (1::'a)$ . Proof will use (NS) Cauchy equivalence for convergence and also fact that bounded and monotonic sequence converges.

```
lemma Bseq-realpow: [| 0 ≤ (x::real); x ≤ 1 |] ==> Bseq (%n. x ^ n)
apply (simp add: Bseq-def)
apply (rule-tac x = 1 in exI)
apply (simp add: power-abs)
apply (auto dest: power-mono)
done
```

```
lemma monoseq-realpow: [| 0 ≤ x; x ≤ 1 |] ==> monoseq (%n. x ^ n)
apply (clarify intro!: mono-SucI2)
apply (cut-tac n = n and N = Suc n and a = x in power-decreasing, auto)
done
```

```
lemma convergent-realpow:
  [| 0 ≤ (x::real); x ≤ 1 |] ==> convergent (%n. x ^ n)
by (blast intro!: Bseq-monoseq-convergent Bseq-realpow monoseq-realpow)
```

```
lemma LIMSEQ-inverse-realpow-zero-lemma:
  fixes x :: real
  assumes x: 0 ≤ x
  shows real n * x + 1 ≤ (x + 1) ^ n
apply (induct n)
apply simp
apply simp
apply (rule order-trans)
prefer 2
apply (erule mult-left-mono)
apply (rule add-increasing [OF x], simp)
apply (simp add: real-of-nat-Suc)
apply (simp add: ring-distrib)
apply (simp add: mult-nonneg-nonneg x)
done
```

```
lemma LIMSEQ-inverse-realpow-zero:
  1 < (x::real) ==> (λn. inverse (x ^ n)) ----> 0
proof (rule LIMSEQ-inverse-zero [rule-format])
  fix y :: real
  assume x: 1 < x
  hence 0 < x - 1 by simp
  hence ∀y. ∃N::nat. y < real N * (x - 1)
    by (rule reals-Archimedean3)
  hence ∃N::nat. y < real N * (x - 1) ..
```

then obtain  $N::nat$  where  $y < real\ N * (x - 1) ..$   
 also have  $\dots \leq real\ N * (x - 1) + 1$  by *simp*  
 also have  $\dots \leq (x - 1 + 1) ^ N$   
 by (*rule LIMSEQ-inverse-realpow-zero-lemma, cut-tac x, simp*)  
 also have  $\dots = x ^ N$  by *simp*  
 finally have  $y < x ^ N$  .  
 hence  $\forall n \geq N. y < x ^ n$   
 apply *clarify*  
 apply (*erule order-less-le-trans*)  
 apply (*erule power-increasing*)  
 apply (*rule order-less-imp-le [OF x]*)  
 done  
 thus  $\exists N. \forall n \geq N. y < x ^ n ..$   
 qed

lemma *LIMSEQ-realpow-zero*:

$\llbracket 0 \leq (x::real); x < 1 \rrbracket \implies (\lambda n. x ^ n) \text{ ----> } 0$   
 proof (*cases*)  
 assume  $x = 0$   
 hence  $(\lambda n. x ^ Suc\ n) \text{ ----> } 0$  by (*simp add: LIMSEQ-const*)  
 thus ?thesis by (*rule LIMSEQ-imp-Suc*)  
 next  
 assume  $0 \leq x$  and  $x \neq 0$   
 hence  $x0: 0 < x$  by *simp*  
 assume  $x1: x < 1$   
 from  $x0\ x1$  have  $1 < inverse\ x$   
 by (*rule one-less-inverse*)  
 hence  $(\lambda n. inverse\ (inverse\ x ^ n)) \text{ ----> } 0$   
 by (*rule LIMSEQ-inverse-realpow-zero*)  
 thus ?thesis by (*simp add: power-inverse*)  
 qed

lemma *LIMSEQ-power-zero*:

fixes  $x :: 'a::\{real-normed-algebra-1\}$   
 shows  $norm\ x < 1 \implies (\lambda n. x ^ n) \text{ ----> } 0$   
 apply (*drule LIMSEQ-realpow-zero [OF norm-ge-zero]*)  
 apply (*simp only: LIMSEQ-Zfun-iff, erule Zfun-le*)  
 apply (*simp add: power-abs norm-power-ineq*)  
 done

lemma *LIMSEQ-divide-realpow-zero*:

$1 < (x::real) \implies (\lambda n. a / (x ^ n)) \text{ ----> } 0$   
 apply (*cut-tac a = a and x1 = inverse x in*  
   *LIMSEQ-mult [OF LIMSEQ-const LIMSEQ-realpow-zero]*)  
 apply (*auto simp add: divide-inverse power-inverse*)  
 apply (*simp add: inverse-eq-divide pos-divide-less-eq*)  
 done

Limit of  $c^n$  for  $|c| < (1::'a)$

**lemma** *LIMSEQ-rabs-realpow-zero*:  $|c| < (1::real) \implies (\%n. |c|^n) \dashrightarrow 0$   
**by** (rule *LIMSEQ-realpow-zero* [OF *abs-ge-zero*])

**lemma** *LIMSEQ-rabs-realpow-zero2*:  $|c| < (1::real) \implies (\%n. c^n) \dashrightarrow 0$   
**apply** (rule *LIMSEQ-rabs-zero* [THEN *iffD1*])  
**apply** (auto intro: *LIMSEQ-rabs-realpow-zero simp add: power-abs*)  
**done**

**end**

## 72 Lim: Limits and Continuity

**theory** *Lim*  
**imports** *SEQ*  
**begin**

Standard Definitions

**abbreviation**

*LIM* :: [*a*::*topological-space*  $\Rightarrow$  *b*::*topological-space*, *a*, *b*]  $\Rightarrow$  *bool*  
 $((-)/ \dashrightarrow (-)/ \dashrightarrow (-))$  [60, 0, 60] 60) **where**  
 $f \dashrightarrow a \dashrightarrow L \equiv (f \dashrightarrow L) \text{ (at } a)$

**definition**

*isCont* :: [*a*::*topological-space*  $\Rightarrow$  *b*::*topological-space*, *a*]  $\Rightarrow$  *bool* **where**  
 $isCont\ f\ a = (f \dashrightarrow a \dashrightarrow (f\ a))$

**definition**

*isUCont* :: [*a*::*metric-space*  $\Rightarrow$  *b*::*metric-space*]  $\Rightarrow$  *bool* **where**  
 $[code\ del]: isUCont\ f = (\forall r>0. \exists s>0. \forall x\ y. dist\ x\ y < s \longrightarrow dist\ (f\ x)\ (f\ y) < r)$

### 72.1 Limits of Functions

**lemma** *LIM-def*:  $f \dashrightarrow a \dashrightarrow L =$   
 $(\forall r > 0. \exists s > 0. \forall x. x \neq a \ \& \ dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r)$

**unfolding** *tendsto-iff eventually-at ..*

**lemma** *metric-LIM-I*:

$(\bigwedge r. 0 < r \implies \exists s>0. \forall x. x \neq a \ \& \ dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r)$   
 $\implies f \dashrightarrow a \dashrightarrow L$

**by** (*simp add: LIM-def*)

**lemma** *metric-LIM-D*:

$\llbracket f \dashrightarrow a \dashrightarrow L; 0 < r \rrbracket$   
 $\implies \exists s>0. \forall x. x \neq a \ \& \ dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r$

**by** (*simp add: LIM-def*)



**lemma** *LIM-eq*:

**fixes**  $a :: 'a::\text{real-normed-vector}$  **and**  $L :: 'b::\text{real-normed-vector}$   
**shows**  $f \dashrightarrow a \dashrightarrow L =$   
 $(\forall r > 0. \exists s > 0. \forall x. x \neq a \ \& \ \text{norm } (x - a) < s \dashrightarrow \text{norm } (f\ x - L) < r)$   
**by** (*simp add: LIM-def dist-norm*)

**lemma** *LIM-I*:

**fixes**  $a :: 'a::\text{real-normed-vector}$  **and**  $L :: 'b::\text{real-normed-vector}$   
**shows**  $(!!r. 0 < r \implies \exists s > 0. \forall x. x \neq a \ \& \ \text{norm } (x - a) < s \dashrightarrow \text{norm } (f\ x - L) < r)$   
 $\implies f \dashrightarrow a \dashrightarrow L$   
**by** (*simp add: LIM-eq*)

**lemma** *LIM-D*:

**fixes**  $a :: 'a::\text{real-normed-vector}$  **and**  $L :: 'b::\text{real-normed-vector}$   
**shows**  $[f \dashrightarrow a \dashrightarrow L; 0 < r]$   
 $\implies \exists s > 0. \forall x. x \neq a \ \& \ \text{norm } (x - a) < s \dashrightarrow \text{norm } (f\ x - L) < r$   
**by** (*simp add: LIM-eq*)

**lemma** *LIM-offset*:

**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $f \dashrightarrow a \dashrightarrow L \implies (\lambda x. f\ (x + k)) \dashrightarrow a - k \dashrightarrow L$   
**apply** (*rule topological-tendstoI*)  
**apply** (*drule (2) topological-tendstoD*)  
**apply** (*simp only: eventually-at dist-norm*)  
**apply** (*clarify, rule-tac x=d in exI, safe*)  
**apply** (*drule-tac x=x + k in spec*)  
**apply** (*simp add: algebra-simps*)  
**done**

**lemma** *LIM-offset-zero*:

**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $f \dashrightarrow a \dashrightarrow L \implies (\lambda h. f\ (a + h)) \dashrightarrow 0 \dashrightarrow L$   
**by** (*drule-tac k=a in LIM-offset, simp add: add-commute*)

**lemma** *LIM-offset-zero-cancel*:

**fixes**  $a :: 'a::\text{real-normed-vector}$   
**shows**  $(\lambda h. f\ (a + h)) \dashrightarrow 0 \dashrightarrow L \implies f \dashrightarrow a \dashrightarrow L$   
**by** (*drule-tac k=- a in LIM-offset, simp*)

**lemma** *LIM-const* [*simp*]:  $(\%x. k) \dashrightarrow x \dashrightarrow k$

**by** (*rule tendsto-const*)

**lemma** *LIM-cong-limit*:  $[f \dashrightarrow x \dashrightarrow L; K = L] \implies f \dashrightarrow x \dashrightarrow K$  **by** *simp*

**lemma** *LIM-add*:

**fixes**  $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**assumes**  $f: f \dashrightarrow a \dashrightarrow L$  **and**  $g: g \dashrightarrow a \dashrightarrow M$

**shows**  $(\lambda x. f\ x + g\ x) \dashrightarrow (L + M)$   
**using** *assms* **by** (*rule tendsto-add*)

**lemma** *LIM-add-zero*:

**fixes**  $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket f \dashrightarrow 0; g \dashrightarrow 0 \rrbracket \Longrightarrow (\lambda x. f\ x + g\ x) \dashrightarrow 0$   
**by** (*drule* (1) *LIM-add, simp*)

**lemma** *LIM-minus*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $f \dashrightarrow L \Longrightarrow (\lambda x. -\ f\ x) \dashrightarrow -\ L$   
**by** (*rule tendsto-minus*)

**lemma** *LIM-add-minus*:

**fixes**  $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket f \dashrightarrow l; g \dashrightarrow m \rrbracket \Longrightarrow (\lambda x. f\ x + -g\ x) \dashrightarrow (l + -m)$   
**by** (*intro LIM-add LIM-minus*)

**lemma** *LIM-diff*:

**fixes**  $f\ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket f \dashrightarrow l; g \dashrightarrow m \rrbracket \Longrightarrow (\lambda x. f\ x - g\ x) \dashrightarrow l - m$   
**by** (*rule tendsto-diff*)

**lemma** *LIM-zero*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $f \dashrightarrow l \Longrightarrow (\lambda x. f\ x - l) \dashrightarrow 0$   
**unfolding** *tendsto-iff dist-norm* **by** *simp*

**lemma** *LIM-zero-cancel*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $(\lambda x. f\ x - l) \dashrightarrow 0 \Longrightarrow f \dashrightarrow l$   
**unfolding** *tendsto-iff dist-norm* **by** *simp*

**lemma** *LIM-zero-iff*:

**fixes**  $f :: 'a::\text{metric-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $(\lambda x. f\ x - l) \dashrightarrow 0 = f \dashrightarrow l$   
**unfolding** *tendsto-iff dist-norm* **by** *simp*

**lemma** *metric-LIM-imp-LIM*:

**assumes**  $f: f \dashrightarrow l$   
**assumes**  $le: \bigwedge x. x \neq a \Longrightarrow \text{dist}\ (g\ x)\ m \leq \text{dist}\ (f\ x)\ l$   
**shows**  $g \dashrightarrow m$   
**apply** (*rule tendstoI, drule tendstoD [OF f]*)  
**apply** (*simp add: eventually-at-topological, safe*)  
**apply** (*rule-tac x=S in exI, safe*)  
**apply** (*drule-tac x=x in bspec, safe*)

**apply** (*erule* (1) *order-le-less-trans* [OF *le*])  
**done**

**lemma** *LIM-imp-LIM*:

**fixes** *f* :: 'a::topological-space  $\Rightarrow$  'b::real-normed-vector  
**fixes** *g* :: 'a::topological-space  $\Rightarrow$  'c::real-normed-vector  
**assumes** *f*: *f* -- *a* --> *l*  
**assumes** *le*:  $\bigwedge x. x \neq a \implies \text{norm } (g\ x - m) \leq \text{norm } (f\ x - l)$   
**shows** *g* -- *a* --> *m*  
**apply** (*rule* *metric-LIM-imp-LIM* [OF *f*])  
**apply** (*simp* *add: dist-norm le*)  
**done**

**lemma** *LIM-norm*:

**fixes** *f* :: 'a::topological-space  $\Rightarrow$  'b::real-normed-vector  
**shows** *f* -- *a* --> *l*  $\implies (\lambda x. \text{norm } (f\ x))$  -- *a* --> *norm l*  
**by** (*rule tendsto-norm*)

**lemma** *LIM-norm-zero*:

**fixes** *f* :: 'a::topological-space  $\Rightarrow$  'b::real-normed-vector  
**shows** *f* -- *a* --> 0  $\implies (\lambda x. \text{norm } (f\ x))$  -- *a* --> 0  
**by** (*rule tendsto-norm-zero*)

**lemma** *LIM-norm-zero-cancel*:

**fixes** *f* :: 'a::topological-space  $\Rightarrow$  'b::real-normed-vector  
**shows**  $(\lambda x. \text{norm } (f\ x))$  -- *a* --> 0  $\implies f$  -- *a* --> 0  
**by** (*rule tendsto-norm-zero-cancel*)

**lemma** *LIM-norm-zero-iff*:

**fixes** *f* :: 'a::topological-space  $\Rightarrow$  'b::real-normed-vector  
**shows**  $(\lambda x. \text{norm } (f\ x))$  -- *a* --> 0 = *f* -- *a* --> 0  
**by** (*rule tendsto-norm-zero-iff*)

**lemma** *LIM-rabs*: *f* -- *a* --> (*l*::real)  $\implies (\lambda x. |f\ x|)$  -- *a* --> |*l*|  
**by** (*fold real-norm-def*, *rule LIM-norm*)

**lemma** *LIM-rabs-zero*: *f* -- *a* --> (0::real)  $\implies (\lambda x. |f\ x|)$  -- *a* --> 0  
**by** (*fold real-norm-def*, *rule LIM-norm-zero*)

**lemma** *LIM-rabs-zero-cancel*:  $(\lambda x. |f\ x|)$  -- *a* --> (0::real)  $\implies f$  -- *a* --> 0  
**by** (*fold real-norm-def*, *rule LIM-norm-zero-cancel*)

**lemma** *LIM-rabs-zero-iff*:  $(\lambda x. |f\ x|)$  -- *a* --> (0::real) = *f* -- *a* --> 0  
**by** (*fold real-norm-def*, *rule LIM-norm-zero-iff*)

**lemma** *at-neq-bot*:

**fixes** *a* :: 'a::real-normed-algebra-1  
**shows** *at a*  $\neq \text{bot}$  — TODO: find a more appropriate class

**unfolding** *eventually-False* [*symmetric*]  
**unfolding** *eventually-at dist-norm*  
**by** (*clarsimp*, *rule-tac*  $x=a + \text{of-real } (d/2)$  **in** *exI*, *simp*)

**lemma** *LIM-const-not-eq*:  
**fixes**  $a :: 'a::\text{real-normed-algebra-1}$   
**fixes**  $k\ L :: 'b::\text{metric-space}$   
**shows**  $k \neq L \implies \neg (\lambda x. k) \dashrightarrow a \dashrightarrow L$   
**by** (*simp add: tendsto-const-iff at-neq-bot*)

**lemmas** *LIM-not-zero* = *LIM-const-not-eq* [**where**  $L = 0$ ]

**lemma** *LIM-const-eq*:  
**fixes**  $a :: 'a::\text{real-normed-algebra-1}$   
**fixes**  $k\ L :: 'b::\text{metric-space}$   
**shows**  $(\lambda x. k) \dashrightarrow a \dashrightarrow L \implies k = L$   
**by** (*simp add: tendsto-const-iff at-neq-bot*)

**lemma** *LIM-unique*:  
**fixes**  $a :: 'a::\text{real-normed-algebra-1}$  — TODO: find a more appropriate class  
**fixes**  $L\ M :: 'b::\text{metric-space}$   
**shows**  $\llbracket f \dashrightarrow a \dashrightarrow L; f \dashrightarrow a \dashrightarrow M \rrbracket \implies L = M$   
**by** (*drule* (1) *tendsto-dist*, *simp add: tendsto-const-iff at-neq-bot*)

**lemma** *LIM-ident* [*simp*]:  $(\lambda x. x) \dashrightarrow a \dashrightarrow a$   
**by** (*rule tendsto-ident-at*)

Limits are equal for functions equal except at limit point

**lemma** *LIM-equal*:  
 $\llbracket \forall x. x \neq a \dashrightarrow (f\ x = g\ x) \rrbracket \implies (f \dashrightarrow a \dashrightarrow l) = (g \dashrightarrow a \dashrightarrow l)$   
**unfolding** *tendsto-def eventually-at-topological* **by** *simp*

**lemma** *LIM-cong*:  
 $\llbracket a = b; \bigwedge x. x \neq b \implies f\ x = g\ x; l = m \rrbracket$   
 $\implies ((\lambda x. f\ x) \dashrightarrow a \dashrightarrow l) = ((\lambda x. g\ x) \dashrightarrow b \dashrightarrow m)$   
**by** (*simp add: LIM-equal*)

**lemma** *metric-LIM-equal2*:  
**assumes** 1:  $0 < R$   
**assumes** 2:  $\bigwedge x. \llbracket x \neq a; \text{dist } x\ a < R \rrbracket \implies f\ x = g\ x$   
**shows**  $g \dashrightarrow a \dashrightarrow l \implies f \dashrightarrow a \dashrightarrow l$   
**apply** (*rule topological-tendstoI*)  
**apply** (*drule* (2) *topological-tendstoD*)  
**apply** (*simp add: eventually-at, safe*)  
**apply** (*rule-tac*  $x=\text{min } d\ R$  **in** *exI*, *safe*)  
**apply** (*simp add: 1*)  
**apply** (*simp add: 2*)  
**done**

**lemma** *LIM-equal2*:

**fixes**  $f\ g :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$   
**assumes**  $1: 0 < R$   
**assumes**  $2: \bigwedge x. \llbracket x \neq a; \text{norm } (x - a) < R \rrbracket \implies f\ x = g\ x$   
**shows**  $g \dashrightarrow a \dashrightarrow l \implies f \dashrightarrow a \dashrightarrow l$   
**by** (*rule metric-LIM-equal2 [OF 1 2], simp-all add: dist-norm*)

Two uses in Transcendental.ML

**lemma** *LIM-trans*:

**fixes**  $f\ g :: 'a::\text{metric-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket (\%x. f(x) + -g(x)) \dashrightarrow a \dashrightarrow 0; g \dashrightarrow a \dashrightarrow l \rrbracket \implies f \dashrightarrow a \dashrightarrow l$   
**apply** (*drule LIM-add, assumption*)  
**apply** (*auto simp add: add-assoc*)  
**done**

**lemma** *LIM-compose*:

**assumes**  $g: g \dashrightarrow l \dashrightarrow g\ l$   
**assumes**  $f: f \dashrightarrow a \dashrightarrow l$   
**shows**  $(\lambda x. g\ (f\ x)) \dashrightarrow a \dashrightarrow g\ l$   
**proof** (*rule topological-tendstoI*)  
**fix**  $C$  **assume**  $C: \text{open } C\ g\ l \in C$   
**obtain**  $B$  **where**  $B: \text{open } B\ l \in B$   
**and**  $gC: \bigwedge y. y \in B \implies y \neq l \implies g\ y \in C$   
**using** *topological-tendstoD [OF g C]*  
**unfolding** *eventually-at-topological* **by** *fast*  
**obtain**  $A$  **where**  $A: \text{open } A\ a \in A$   
**and**  $fB: \bigwedge x. x \in A \implies x \neq a \implies f\ x \in B$   
**using** *topological-tendstoD [OF f B]*  
**unfolding** *eventually-at-topological* **by** *fast*  
**show** *eventually*  $(\lambda x. g\ (f\ x) \in C)$  (*at a*)  
**unfolding** *eventually-at-topological*  
**proof** (*intro exI conjI ballI impI*)  
**show** *open A and a ∈ A using A .*  
**next**  
**fix**  $x$  **assume**  $x \in A$  **and**  $x \neq a$   
**then show**  $g\ (f\ x) \in C$   
**by** (*cases f x = l, simp add: C, simp add: gC fB*)  
**qed**  
**qed**

**lemma** *LIM-compose-eventually*:

**assumes**  $f: f \dashrightarrow a \dashrightarrow b$   
**assumes**  $g: g \dashrightarrow b \dashrightarrow c$   
**assumes** *inj: eventually*  $(\lambda x. f\ x \neq b)$  (*at a*)  
**shows**  $(\lambda x. g\ (f\ x)) \dashrightarrow a \dashrightarrow c$   
**proof** (*rule topological-tendstoI*)  
**fix**  $C$  **assume**  $C: \text{open } C\ c \in C$   
**obtain**  $B$  **where**  $B: \text{open } B\ b \in B$

```

    and gC:  $\bigwedge y. y \in B \implies y \neq b \implies g\ y \in C$ 
    using topological-tendstoD [OF g C]
    unfolding eventually-at-topological by fast
  obtain A where A: open A a  $\in$  A
    and fB:  $\bigwedge x. x \in A \implies x \neq a \implies f\ x \in B$ 
    using topological-tendstoD [OF f B]
    unfolding eventually-at-topological by fast
  have eventually ( $\lambda x. f\ x \neq b \longrightarrow g\ (f\ x) \in C$ ) (at a)
  unfolding eventually-at-topological
  proof (intro exI conjI ballI impI)
    show open A and a  $\in$  A using A .
  next
    fix x assume x  $\in$  A and x  $\neq$  a and f x  $\neq$  b
    then show g (f x)  $\in$  C by (simp add: gC fB)
  qed
  with inj show eventually ( $\lambda x. g\ (f\ x) \in C$ ) (at a)
    by (rule eventually-rev-mp)
qed

```

```

lemma metric-LIM-compose2:
  assumes f: f -- a --> b
  assumes g: g -- b --> c
  assumes inj:  $\exists d > 0. \forall x. x \neq a \wedge \text{dist } x\ a < d \longrightarrow f\ x \neq b$ 
  shows ( $\lambda x. g\ (f\ x)$ ) -- a --> c
using f g inj [folded eventually-at]
by (rule LIM-compose-eventually)

```

```

lemma LIM-compose2:
  fixes a :: 'a::real-normed-vector
  assumes f: f -- a --> b
  assumes g: g -- b --> c
  assumes inj:  $\exists d > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \longrightarrow f\ x \neq b$ 
  shows ( $\lambda x. g\ (f\ x)$ ) -- a --> c
by (rule metric-LIM-compose2 [OF f g inj [folded dist-norm]])

```

```

lemma LIM-o:  $\llbracket g -- l --> g\ l; f -- a --> l \rrbracket \implies (g \circ f) -- a --> g\ l$ 
unfolding o-def by (rule LIM-compose)

```

```

lemma real-LIM-sandwich-zero:
  fixes f g :: 'a::topological-space  $\Rightarrow$  real
  assumes f: f -- a --> 0
  assumes 1:  $\bigwedge x. x \neq a \implies 0 \leq g\ x$ 
  assumes 2:  $\bigwedge x. x \neq a \implies g\ x \leq f\ x$ 
  shows g -- a --> 0
proof (rule LIM-imp-LIM [OF f])
  fix x assume x: x  $\neq$  a
  have norm (g x - 0) = g x by (simp add: 1 x)
  also have g x  $\leq$  f x by (rule 2 [OF x])
  also have f x  $\leq$  |f x| by (rule abs-ge-self)

```

also have  $|f x| = \text{norm } (f x - 0)$  by *simp*  
 finally show  $\text{norm } (g x - 0) \leq \text{norm } (f x - 0)$  .  
 qed

### Bounded Linear Operators

**lemma** (in *bounded-linear*) *LIM*:  
 $g \text{ -- } a \text{ -->} l \implies (\lambda x. f (g x)) \text{ -- } a \text{ -->} f l$   
 by (rule *tendsto*)

**lemma** (in *bounded-linear*) *cont*:  $f \text{ -- } a \text{ -->} f a$   
 by (rule *LIM* [OF *LIM-ident*])

**lemma** (in *bounded-linear*) *LIM-zero*:  
 $g \text{ -- } a \text{ -->} 0 \implies (\lambda x. f (g x)) \text{ -- } a \text{ -->} 0$   
 by (drule *LIM*, *simp* only: *zero*)

### Bounded Bilinear Operators

**lemma** (in *bounded-bilinear*) *LIM*:  
 $\llbracket f \text{ -- } a \text{ -->} L; g \text{ -- } a \text{ -->} M \rrbracket \implies (\lambda x. f x ** g x) \text{ -- } a \text{ -->} L ** M$   
 by (rule *tendsto*)

**lemma** (in *bounded-bilinear*) *LIM-prod-zero*:  
 fixes  $a :: 'd::\text{metric-space}$   
 assumes  $f: f \text{ -- } a \text{ -->} 0$   
 assumes  $g: g \text{ -- } a \text{ -->} 0$   
 shows  $(\lambda x. f x ** g x) \text{ -- } a \text{ -->} 0$   
 using *LIM* [OF *f g*] by (*simp* add: *zero-left*)

**lemma** (in *bounded-bilinear*) *LIM-left-zero*:  
 $f \text{ -- } a \text{ -->} 0 \implies (\lambda x. f x ** c) \text{ -- } a \text{ -->} 0$   
 by (rule *bounded-linear.LIM-zero* [OF *bounded-linear-left*])

**lemma** (in *bounded-bilinear*) *LIM-right-zero*:  
 $f \text{ -- } a \text{ -->} 0 \implies (\lambda x. c ** f x) \text{ -- } a \text{ -->} 0$   
 by (rule *bounded-linear.LIM-zero* [OF *bounded-linear-right*])

**lemmas** *LIM-mult* = *mult.LIM*

**lemmas** *LIM-mult-zero* = *mult.LIM-prod-zero*

**lemmas** *LIM-mult-left-zero* = *mult.LIM-left-zero*

**lemmas** *LIM-mult-right-zero* = *mult.LIM-right-zero*

**lemmas** *LIM-scaleR* = *scaleR.LIM*

**lemmas** *LIM-of-real* = *of-real.LIM*

**lemma** *LIM-power*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\{\text{power}, \text{real-normed-algebra}\}$   
**assumes**  $f: f \dashv\dashv a \dashv\dashv l$   
**shows**  $(\lambda x. f\ x \wedge n) \dashv\dashv a \dashv\dashv l \wedge n$   
**by** (*induct*  $n$ , *simp*, *simp add: LIM-mult*  $f$ )

### 72.1.1 Derived theorems about LIM

**lemma** *LIM-inverse*:

**fixes**  $L :: 'a::\text{real-normed-div-algebra}$   
**shows**  $\llbracket f \dashv\dashv a \dashv\dashv L; L \neq 0 \rrbracket \Longrightarrow (\lambda x. \text{inverse } (f\ x)) \dashv\dashv a \dashv\dashv \text{inverse } L$   
**by** (*rule tendsto-inverse*)

**lemma** *LIM-inverse-fun*:

**assumes**  $a: a \neq 0 :: 'a::\text{real-normed-div-algebra}$   
**shows**  $\text{inverse} \dashv\dashv a \dashv\dashv \text{inverse } a$   
**by** (*rule LIM-inverse [OF LIM-ident*  $a]$ )

**lemma** *LIM-sgn*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket f \dashv\dashv a \dashv\dashv l; l \neq 0 \rrbracket \Longrightarrow (\lambda x. \text{sgn } (f\ x)) \dashv\dashv a \dashv\dashv \text{sgn } l$   
**unfolding** *sgn-div-norm*  
**by** (*simp add: LIM-scaleR LIM-inverse LIM-norm*)

## 72.2 Continuity

**lemma** *LIM-isCont-iff*:

**fixes**  $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$   
**shows**  $(f \dashv\dashv a \dashv\dashv f\ a) = ((\lambda h. f\ (a + h)) \dashv\dashv 0 \dashv\dashv f\ a)$   
**by** (*rule iffI [OF LIM-offset-zero LIM-offset-zero-cancel]*)

**lemma** *isCont-iff*:

**fixes**  $f :: 'a::\text{real-normed-vector} \Rightarrow 'b::\text{topological-space}$   
**shows**  $\text{isCont } f\ x = (\lambda h. f\ (x + h)) \dashv\dashv 0 \dashv\dashv f\ x$   
**by** (*simp add: isCont-def LIM-isCont-iff*)

**lemma** *isCont-ident* [*simp*]:  $\text{isCont } (\lambda x. x)\ a$

**unfolding** *isCont-def* **by** (*rule LIM-ident*)

**lemma** *isCont-const* [*simp*]:  $\text{isCont } (\lambda x. k)\ a$

**unfolding** *isCont-def* **by** (*rule LIM-const*)

**lemma** *isCont-norm*:

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\text{isCont } f\ a \Longrightarrow \text{isCont } (\lambda x. \text{norm } (f\ x))\ a$   
**unfolding** *isCont-def* **by** (*rule LIM-norm*)

**lemma** *isCont-rabs*:  $\text{isCont } f\ a \Longrightarrow \text{isCont } (\lambda x. |f\ x|)\ a$

**unfolding** *isCont-def* **by** (*rule LIM-rabs*)

**lemma** *isCont-add*:



**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket \text{isCont } f \ a; \text{isCont } g \ a \rrbracket \Longrightarrow \text{isCont } (\lambda x. f \ x + g \ x) \ a$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-add}$ )

**lemma**  $\text{isCont-minus}$ :

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\text{isCont } f \ a \Longrightarrow \text{isCont } (\lambda x. - f \ x) \ a$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-minus}$ )

**lemma**  $\text{isCont-diff}$ :

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-vector}$   
**shows**  $\llbracket \text{isCont } f \ a; \text{isCont } g \ a \rrbracket \Longrightarrow \text{isCont } (\lambda x. f \ x - g \ x) \ a$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-diff}$ )

**lemma**  $\text{isCont-mult}$ :

**fixes**  $f \ g :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-algebra}$   
**shows**  $\llbracket \text{isCont } f \ a; \text{isCont } g \ a \rrbracket \Longrightarrow \text{isCont } (\lambda x. f \ x * g \ x) \ a$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-mult}$ )

**lemma**  $\text{isCont-inverse}$ :

**fixes**  $f :: 'a::\text{topological-space} \Rightarrow 'b::\text{real-normed-div-algebra}$   
**shows**  $\llbracket \text{isCont } f \ a; f \ a \neq 0 \rrbracket \Longrightarrow \text{isCont } (\lambda x. \text{inverse } (f \ x)) \ a$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-inverse}$ )

**lemma**  $\text{isCont-LIM-compose}$ :

$\llbracket \text{isCont } g \ l; f \ \dashv\dashv \ a \dashv\dashv \ l \rrbracket \Longrightarrow (\lambda x. g \ (f \ x)) \ \dashv\dashv \ a \dashv\dashv \ g \ l$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-compose}$ )

**lemma**  $\text{metric-isCont-LIM-compose2}$ :

**assumes**  $f \ [\text{unfolded } \text{isCont-def}]: \text{isCont } f \ a$   
**assumes**  $g: g \ \dashv\dashv \ f \ a \dashv\dashv \ l$   
**assumes**  $\text{inj}: \exists d > 0. \forall x. x \neq a \wedge \text{dist } x \ a < d \longrightarrow f \ x \neq f \ a$   
**shows**  $(\lambda x. g \ (f \ x)) \ \dashv\dashv \ a \dashv\dashv \ l$   
**by** (rule  $\text{metric-LIM-compose2}$  [OF  $f \ g \ \text{inj}$ ])

**lemma**  $\text{isCont-LIM-compose2}$ :

**fixes**  $a :: 'a::\text{real-normed-vector}$   
**assumes**  $f \ [\text{unfolded } \text{isCont-def}]: \text{isCont } f \ a$   
**assumes**  $g: g \ \dashv\dashv \ f \ a \dashv\dashv \ l$   
**assumes**  $\text{inj}: \exists d > 0. \forall x. x \neq a \wedge \text{norm } (x - a) < d \longrightarrow f \ x \neq f \ a$   
**shows**  $(\lambda x. g \ (f \ x)) \ \dashv\dashv \ a \dashv\dashv \ l$   
**by** (rule  $\text{LIM-compose2}$  [OF  $f \ g \ \text{inj}$ ])

**lemma**  $\text{isCont-o2}$ :  $\llbracket \text{isCont } f \ a; \text{isCont } g \ (f \ a) \rrbracket \Longrightarrow \text{isCont } (\lambda x. g \ (f \ x)) \ a$   
**unfolding**  $\text{isCont-def}$  **by** (rule  $\text{LIM-compose}$ )

**lemma**  $\text{isCont-o}$ :  $\llbracket \text{isCont } f \ a; \text{isCont } g \ (f \ a) \rrbracket \Longrightarrow \text{isCont } (g \ o \ f) \ a$   
**unfolding**  $\text{o-def}$  **by** (rule  $\text{isCont-o2}$ )

**lemma** (in *bounded-linear*) *isCont*: *isCont* *f* *a*  
**unfolding** *isCont-def* **by** (rule *cont*)

**lemma** (in *bounded-bilinear*) *isCont*:  
 $\llbracket \text{isCont } f \text{ } a; \text{isCont } g \text{ } a \rrbracket \implies \text{isCont } (\lambda x. f \text{ } x ** g \text{ } x) \text{ } a$   
**unfolding** *isCont-def* **by** (rule *LIM*)

**lemmas** *isCont-scaleR* = *scaleR.isCont*

**lemma** *isCont-of-real*:  
 $\text{isCont } f \text{ } a \implies \text{isCont } (\lambda x. \text{of-real } (f \text{ } x)::'b::\text{real-normed-algebra-1}) \text{ } a$   
**unfolding** *isCont-def* **by** (rule *LIM-of-real*)

**lemma** *isCont-power*:  
**fixes** *f* :: '*a*::*topological-space*  $\Rightarrow$  '*b*::{*power,real-normed-algebra*}  
**shows** *isCont* *f* *a*  $\implies \text{isCont } (\lambda x. f \text{ } x \wedge n) \text{ } a$   
**unfolding** *isCont-def* **by** (rule *LIM-power*)

**lemma** *isCont-sgn*:  
**fixes** *f* :: '*a*::*topological-space*  $\Rightarrow$  '*b*::*real-normed-vector*  
**shows**  $\llbracket \text{isCont } f \text{ } a; f \text{ } a \neq 0 \rrbracket \implies \text{isCont } (\lambda x. \text{sgn } (f \text{ } x)) \text{ } a$   
**unfolding** *isCont-def* **by** (rule *LIM-sgn*)

**lemma** *isCont-abs* [*simp*]: *isCont* *abs* (*a*::*real*)  
**by** (rule *isCont-rabs* [*OF isCont-ident*])

**lemma** *isCont-setsum*:  
**fixes** *f* :: '*a*  $\Rightarrow$  '*b*::*topological-space*  $\Rightarrow$  '*c*::*real-normed-vector*  
**fixes** *A* :: '*a* **set** **assumes** *finite A*  
**shows**  $\forall i \in A. \text{isCont } (f \text{ } i) \text{ } x \implies \text{isCont } (\lambda x. \sum i \in A. f \text{ } i \text{ } x) \text{ } x$   
**using**  $\langle \text{finite } A \rangle$   
**proof** *induct*  
**case** (*insert a F*) **show** *isCont*  $(\lambda x. \sum i \in (\text{insert } a \text{ } F). f \text{ } i \text{ } x) \text{ } x$   
**unfolding** *setsum-insert* [*OF*  $\langle \text{finite } F \rangle \langle a \notin F \rangle$ ] **by** (rule *isCont-add*, *auto simp*  
*add: insert*)  
**qed** *auto*

**lemma** *LIM-less-bound*: **fixes** *f* :: *real*  $\Rightarrow$  *real* **assumes** *b* < *x*  
**and** *all-le*:  $\forall x' \in \{ b <..< x \}. 0 \leq f \text{ } x'$  **and** *isCont*: *isCont* *f* *x*  
**shows**  $0 \leq f \text{ } x$   
**proof** (rule *ccontr*)  
**assume**  $\neg 0 \leq f \text{ } x$  **hence** *f* *x* < 0 **by** *auto*  
**hence**  $0 < -f \text{ } x / 2$  **by** *auto*  
**from** *isCont* [*unfolded isCont-def*, *THEN LIM-D*, *OF this*]  
**obtain** *s* **where** *s* > 0 **and** *s-D*:  $\bigwedge x'. \llbracket x' \neq x ; |x' - x| < s \rrbracket \implies |f \text{ } x' - f \text{ } x| < -f \text{ } x / 2$  **by** *auto*

**let** *?x* = *x* - *min* (*s* / 2) ((*x* - *b*) / 2)  
**have** *?x* < *x* **and**  $|?x - x| < s$

```

    using ⟨b < x⟩ and ⟨0 < s⟩ by auto
  have b < ?x
proof (cases s < x - b)
  case True thus ?thesis using ⟨0 < s⟩ by auto
next
  case False hence s / 2 ≥ (x - b) / 2 by auto
  hence ?x = (x + b) / 2 by (simp add: field-simps min-max.inf-absorb2)
  thus ?thesis using ⟨b < x⟩ by auto
qed
hence 0 ≤ f ?x using all-le ⟨?x < x⟩ by auto
moreover have |f ?x - f x| < - f x / 2
  using s-D[OF - ⟨| ?x - x | < s⟩] ⟨?x < x⟩ by auto
hence f ?x - f x < - f x / 2 by auto
hence f ?x < f x / 2 by auto
hence f ?x < 0 using ⟨f x < 0⟩ by auto
thus False using ⟨0 ≤ f ?x⟩ by auto
qed

```

### 72.3 Uniform Continuity

**lemma** *isUCont-isCont*: *isUCont f ==> isCont f x*  
**by** (simp add: *isUCont-def isCont-def LIM-def*, force)

**lemma** *isUCont-Cauchy*:  
 $\llbracket \text{isUCont } f; \text{Cauchy } X \rrbracket \implies \text{Cauchy } (\lambda n. f (X n))$   
**unfolding** *isUCont-def*  
**apply** (rule *metric-CauchyI*)  
**apply** (drule-tac *x=e* in *spec*, *safe*)  
**apply** (drule-tac *e=s* in *metric-CauchyD*, *safe*)  
**apply** (rule-tac *x=M* in *exI*, *simp*)  
**done**

**lemma** (in *bounded-linear*) *isUCont*: *isUCont f*  
**unfolding** *isUCont-def dist-norm*  
**proof** (intro *allI impI*)  
 fix *r::real* assume *r*:  $0 < r$   
 obtain *K* where *K*:  $0 < K$  and *norm-le*:  $\bigwedge x. \text{norm } (f x) \leq \text{norm } x * K$   
 using *pos-bounded* by *fast*  
 show  $\exists s > 0. \forall x y. \text{norm } (x - y) < s \longrightarrow \text{norm } (f x - f y) < r$   
**proof** (rule *exI*, *safe*)  
 from *r K* show  $0 < r / K$  by (rule *divide-pos-pos*)  
**next**  
 fix *x y* :: 'a  
 assume *xy*:  $\text{norm } (x - y) < r / K$   
 have  $\text{norm } (f x - f y) = \text{norm } (f (x - y))$  by (simp only: *diff*)  
 also have  $\dots \leq \text{norm } (x - y) * K$  by (rule *norm-le*)  
 also from *K xy* have  $\dots < r$  by (simp only: *pos-less-divide-eq*)  
 finally show  $\text{norm } (f x - f y) < r$ .  
**qed**

qed

**lemma** (in *bounded-linear*) *Cauchy*:  $\text{Cauchy } X \implies \text{Cauchy } (\lambda n. f (X n))$   
**by** (rule *isUCont* [THEN *isUCont-Cauchy*])

## 72.4 Relation of LIM and LIMSEQ

**lemma** *LIMSEQ-SEQ-conv1*:

**fixes**  $a :: 'a::\text{metric-space}$  **and**  $L :: 'b::\text{metric-space}$   
**assumes**  $X: X \dashrightarrow a \dashrightarrow L$   
**shows**  $\forall S. (\forall n. S n \neq a) \wedge S \dashrightarrow a \longrightarrow (\lambda n. X (S n)) \dashrightarrow L$   
**proof** (safe intro!: *metric-LIMSEQ-I*)  
**fix**  $S :: \text{nat} \Rightarrow 'a$   
**fix**  $r :: \text{real}$   
**assume**  $\text{rgz}: 0 < r$   
**assume**  $\text{as}: \forall n. S n \neq a$   
**assume**  $S: S \dashrightarrow a$   
**from** *metric-LIM-D* [OF  $X \text{ rgz}$ ] **obtain**  $s$   
**where**  $\text{sgz}: 0 < s$   
**and**  $\text{aux}: \bigwedge x. \llbracket x \neq a; \text{dist } x a < s \rrbracket \implies \text{dist } (X x) L < r$   
**by** *fast*  
**from** *metric-LIMSEQ-D* [OF  $S \text{ sgz}$ ]  
**obtain**  $\text{no}$  **where**  $\forall n \geq \text{no}. \text{dist } (S n) a < s$  **by** *blast*  
**hence**  $\forall n \geq \text{no}. \text{dist } (X (S n)) L < r$  **by** (*simp add: aux as*)  
**thus**  $\exists \text{no}. \forall n \geq \text{no}. \text{dist } (X (S n)) L < r$  ..  
 qed

**lemma** *LIMSEQ-SEQ-conv2*:

**fixes**  $a :: \text{real}$  **and**  $L :: 'a::\text{metric-space}$   
**assumes**  $\forall S. (\forall n. S n \neq a) \wedge S \dashrightarrow a \longrightarrow (\lambda n. X (S n)) \dashrightarrow L$   
**shows**  $X \dashrightarrow a \dashrightarrow L$   
**proof** (rule *ccontr*)  
**assume**  $\neg (X \dashrightarrow a \dashrightarrow L)$   
**hence**  $\neg (\forall r > 0. \exists s > 0. \forall x. x \neq a \ \& \ \text{norm } (x - a) < s \dashrightarrow \text{dist } (X x) L < r)$   
**unfolding** *LIM-def dist-norm* .  
**hence**  $\exists r > 0. \forall s > 0. \exists x. \neg (x \neq a \wedge |x - a| < s \dashrightarrow \text{dist } (X x) L < r)$   
**by** *simp*  
**hence**  $\exists r > 0. \forall s > 0. \exists x. (x \neq a \wedge |x - a| < s \wedge \text{dist } (X x) L \geq r)$  **by** (*simp add: not-less*)  
**then obtain**  $r$  **where**  $\text{rdef}: r > 0 \wedge (\forall s > 0. \exists x. (x \neq a \wedge |x - a| < s \wedge \text{dist } (X x) L \geq r))$  **by** *auto*

**let**  $?F = \lambda n::\text{nat}. \text{SOME } x. x \neq a \wedge |x - a| < \text{inverse } (\text{real } (\text{Suc } n)) \wedge \text{dist } (X x) L \geq r$   
**have**  $\bigwedge n. \exists x. x \neq a \wedge |x - a| < \text{inverse } (\text{real } (\text{Suc } n)) \wedge \text{dist } (X x) L \geq r$   
**using** *rdef* **by** *simp*  
**hence**  $F: \bigwedge n. ?F n \neq a \wedge |?F n - a| < \text{inverse } (\text{real } (\text{Suc } n)) \wedge \text{dist } (X (?F$

```

n))  $L \geq r$ 
  by (rule someI-ex)
hence  $F1: \bigwedge n. ?F\ n \neq a$ 
  and  $F2: \bigwedge n. |?F\ n - a| < \text{inverse}(\text{real}(\text{Suc}\ n))$ 
  and  $F3: \bigwedge n. \text{dist}(X(?F\ n))\ L \geq r$ 
  by fast+

have  $?F\ ----> a$ 
proof (rule LIMSEQ-I, unfold real-norm-def)
  fix  $e::\text{real}$ 
  assume  $0 < e$ 

  then have  $\exists no. \text{inverse}(\text{real}(\text{Suc}\ no)) < e$  by (rule reals-Archimedean)
  then obtain  $m$  where  $nodef: \text{inverse}(\text{real}(\text{Suc}\ m)) < e$  by auto
  show  $\exists no. \forall n. no \leq n \implies |?F\ n - a| < e$ 
  proof (intro exI allI impI)
    fix  $n$ 
    assume  $mle n: m \leq n$ 
    have  $|?F\ n - a| < \text{inverse}(\text{real}(\text{Suc}\ n))$ 
      by (rule F2)
    also have  $\text{inverse}(\text{real}(\text{Suc}\ n)) \leq \text{inverse}(\text{real}(\text{Suc}\ m))$ 
      using  $mle n$  by auto
    also from  $nodef$  have
       $\text{inverse}(\text{real}(\text{Suc}\ m)) < e$  .
    finally show  $|?F\ n - a| < e$  .
  qed
qed

moreover have  $\forall n. ?F\ n \neq a$ 
  by (rule allI) (rule F1)

moreover from prems have  $\forall S. (\forall n. S\ n \neq a) \wedge S\ ----> a \implies (\lambda n. X(S\ n))\ ----> L$  by simp
ultimately have  $(\lambda n. X(?F\ n))\ ----> L$  by simp

moreover have  $\neg ((\lambda n. X(?F\ n))\ ----> L)$ 
proof -
{
  fix  $no::\text{nat}$ 
  obtain  $n$  where  $n = no + 1$  by simp
  then have  $nolen: no \leq n$  by simp

  have  $\text{dist}(X(?F\ n))\ L \geq r$ 
    by (rule F3)
  with  $nolen$  have  $\exists n. no \leq n \wedge \text{dist}(X(?F\ n))\ L \geq r$  by fast
}
then have  $(\forall no. \exists n. no \leq n \wedge \text{dist}(X(?F\ n))\ L \geq r)$  by simp
with  $rdef$  have  $\exists e>0. (\forall no. \exists n. no \leq n \wedge \text{dist}(X(?F\ n))\ L \geq e)$  by auto
thus  $?thesis$  by (unfold LIMSEQ-def, auto simp add: not-less)

```

qed  
 ultimately show *False* by *simp*  
 qed

**lemma** *LIMSEQ-SEQ-conv*:

$(\forall S. (\forall n. S\ n \neq a) \wedge S \text{ ----> } (a::real) \longrightarrow (\lambda n. X\ (S\ n)) \text{ ----> } L) =$   
 $(X \text{ -- } a \text{ --> } (L::'a::metric-space))$

**proof**

**assume**  $\forall S. (\forall n. S\ n \neq a) \wedge S \text{ ----> } a \longrightarrow (\lambda n. X\ (S\ n)) \text{ ----> } L$

**thus**  $X \text{ -- } a \text{ --> } L$  **by** (*rule LIMSEQ-SEQ-conv2*)

**next**

**assume**  $(X \text{ -- } a \text{ --> } L)$

**thus**  $\forall S. (\forall n. S\ n \neq a) \wedge S \text{ ----> } a \longrightarrow (\lambda n. X\ (S\ n)) \text{ ----> } L$  **by** (*rule LIMSEQ-SEQ-conv1*)

qed

end

## 73 Deriv: Differentiation

**theory** *Deriv*

**imports** *Lim*

**begin**

Standard Definitions

**definition**

*deriv* ::  $'a::real\text{-normed-field} \Rightarrow 'a, 'a, 'a] \Rightarrow bool$

— Differentiation: D is derivative of function f at x

$((DERIV\ (-)/\ (-)/\ :>\ (-))\ [1000, 1000, 60]\ 60)$  **where**

$DERIV\ f\ x\ :>\ D = ((\%h. (f(x + h) - f\ x) / h) \text{ -- } 0 \text{ --> } D)$

**primrec**

*Bolzano-bisect* ::  $(real \times real \Rightarrow bool) \Rightarrow real \Rightarrow real \Rightarrow nat \Rightarrow real \times real$  **where**

*Bolzano-bisect*  $P\ a\ b\ 0 = (a, b)$

| *Bolzano-bisect*  $P\ a\ b\ (Suc\ n) =$

$(let\ (x, y) = Bolzano-bisect\ P\ a\ b\ n$

$in\ if\ P\ (x, (x+y) / 2) \text{ then } ((x+y)/2, y)$

$else\ (x, (x+y)/2))$

### 73.1 Derivatives

**lemma** *DERIV-iff*:  $(DERIV\ f\ x\ :>\ D) = ((\%h. (f(x + h) - f(x))/h) \text{ -- } 0 \text{ --> } D)$

**by** (*simp add: deriv-def*)

**lemma** *DERIV-D*:  $DERIV\ f\ x\ :>\ D ==> (\%h. (f(x + h) - f(x))/h) \text{ -- } 0 \text{ --> } D$

**by** (*simp add: deriv-def*)

**lemma** *DERIV-const* [*simp*]:  $DERIV (\lambda x. k) x :> 0$   
**by** (*simp add: deriv-def*)

**lemma** *DERIV-ident* [*simp*]:  $DERIV (\lambda x. x) x :> 1$   
**by** (*simp add: deriv-def cong: LIM-cong*)

**lemma** *add-diff-add*:  
**fixes**  $a\ b\ c\ d :: 'a::ab-group-add$   
**shows**  $(a + c) - (b + d) = (a - b) + (c - d)$   
**by** *simp*

**lemma** *DERIV-add*:  
 $\llbracket DERIV\ f\ x :> D; DERIV\ g\ x :> E \rrbracket \Longrightarrow DERIV (\lambda x. f\ x + g\ x) x :> D + E$   
**by** (*simp only: deriv-def add-diff-add add-divide-distrib LIM-add*)

**lemma** *DERIV-minus*:  
 $DERIV\ f\ x :> D \Longrightarrow DERIV (\lambda x. - f\ x) x :> - D$   
**by** (*simp only: deriv-def minus-diff-minus divide-minus-left LIM-minus*)

**lemma** *DERIV-diff*:  
 $\llbracket DERIV\ f\ x :> D; DERIV\ g\ x :> E \rrbracket \Longrightarrow DERIV (\lambda x. f\ x - g\ x) x :> D - E$   
**by** (*simp only: diff-def DERIV-add DERIV-minus*)

**lemma** *DERIV-add-minus*:  
 $\llbracket DERIV\ f\ x :> D; DERIV\ g\ x :> E \rrbracket \Longrightarrow DERIV (\lambda x. f\ x + - g\ x) x :> D + - E$   
**by** (*simp only: DERIV-add DERIV-minus*)

**lemma** *DERIV-isCont*:  $DERIV\ f\ x :> D \Longrightarrow isCont\ f\ x$

**proof** (*unfold isCont-iff*)  
**assume**  $DERIV\ f\ x :> D$   
**hence**  $(\lambda h. (f(x+h) - f(x)) / h) \dashrightarrow 0 \dashrightarrow D$   
**by** (*rule DERIV-D*)  
**hence**  $(\lambda h. (f(x+h) - f(x)) / h * h) \dashrightarrow 0 \dashrightarrow D * 0$   
**by** (*intro LIM-mult LIM-ident*)  
**hence**  $(\lambda h. (f(x+h) - f(x)) * (h / h)) \dashrightarrow 0 \dashrightarrow 0$   
**by** *simp*  
**hence**  $(\lambda h. f(x+h) - f(x)) \dashrightarrow 0 \dashrightarrow 0$   
**by** (*simp cong: LIM-cong*)  
**thus**  $(\lambda h. f(x+h)) \dashrightarrow 0 \dashrightarrow f(x)$   
**by** (*simp add: LIM-def dist-norm*)  
**qed**

**lemma** *DERIV-mult-lemma*:  
**fixes**  $a\ b\ c\ d :: 'a::real-field$   
**shows**  $(a * b - c * d) / h = a * ((b - d) / h) + ((a - c) / h) * d$   
**by** (*simp add: diff-minus add-divide-distrib [symmetric] ring-distrib*)

```

lemma DERIV-mult':
  assumes  $f: \text{DERIV } f \, x :> D$ 
  assumes  $g: \text{DERIV } g \, x :> E$ 
  shows  $\text{DERIV } (\lambda x. f \, x * g \, x) \, x :> f \, x * E + D * g \, x$ 
proof (unfold deriv-def)
  from  $f$  have isCont  $f \, x$ 
    by (rule DERIV-isCont)
  hence  $(\lambda h. f(x+h)) \dashrightarrow 0 \dashrightarrow f \, x$ 
    by (simp only: isCont-iff)
  hence  $(\lambda h. f(x+h) * ((g(x+h) - g \, x) / h) +$ 
     $((f(x+h) - f \, x) / h) * g \, x)$ 
     $\dashrightarrow 0 \dashrightarrow f \, x * E + D * g \, x$ 
    by (intro LIM-add LIM-mult LIM-const DERIV-D f g)
  thus  $(\lambda h. (f(x+h) * g(x+h) - f \, x * g \, x) / h)$ 
     $\dashrightarrow 0 \dashrightarrow f \, x * E + D * g \, x$ 
    by (simp only: DERIV-mult-lemma)
qed

```

```

lemma DERIV-mult:
   $[[ \text{DERIV } f \, x :> D a; \text{DERIV } g \, x :> D b ]]$ 
   $\implies \text{DERIV } (\%x. f \, x * g \, x) \, x :> (D a * g(x)) + (D b * f(x))$ 
by (drule (1) DERIV-mult', simp only: mult-commute add-commute)

```

```

lemma DERIV-unique:
   $[[ \text{DERIV } f \, x :> D; \text{DERIV } f \, x :> E ]] \implies D = E$ 
apply (simp add: deriv-def)
apply (blast intro: LIM-unique)
done

```

Differentiation of finite sum

```

lemma DERIV-setsum:
  assumes finite  $S$ 
  and  $\bigwedge n. n \in S \implies \text{DERIV } (\%x. f \, x \, n) \, x :> (f' \, x \, n)$ 
  shows  $\text{DERIV } (\%x. \text{setsum } (f \, x) \, S) \, x :> \text{setsum } (f' \, x) \, S$ 
  using assms by induct (auto intro!: DERIV-add)

```

```

lemma DERIV-sumr [rule-format (no-asm)]:
   $(\forall r. m \leq r \ \& \ r < (m + n) \dashrightarrow \text{DERIV } (\%x. f \, r \, x) \, x :> (f' \, r \, x))$ 
   $\dashrightarrow \text{DERIV } (\%x. \sum_{n=m..<n::\text{nat}. f \, n \, x :: \text{real}}) \, x :> (\sum_{r=m..<n. f' \, r \, x})$ 
  by (auto intro: DERIV-setsum)

```

Alternative definition for differentiability

```

lemma DERIV-LIM-iff:
  fixes  $f :: 'a :: \{\text{real-normed-vector, inverse}\} \Rightarrow 'a$  shows
     $((\%h. (f(a + h) - f(a)) / h) \dashrightarrow 0 \dashrightarrow D) =$ 
     $((\%x. (f(x) - f(a)) / (x - a)) \dashrightarrow a \dashrightarrow D)$ 
apply (rule iffI)
apply (drule-tac k=- a in LIM-offset)

```



```

apply (simp add: diff-minus)
apply (drule-tac k=a in LIM-offset)
apply (simp add: add-commute)
done

```

```

lemma DERIV-iff2: (DERIV f x :> D) = ((%z. (f(z) - f(x)) / (z-x)) -- x
--> D)
by (simp add: deriv-def diff-minus [symmetric] DERIV-LIM-iff)

```

```

lemma inverse-diff-inverse:
  [(a::'a::division-ring) ≠ 0; b ≠ 0]
  ⇒ inverse a - inverse b = - (inverse a * (a - b) * inverse b)
by (simp add: algebra-simps)

```

```

lemma DERIV-inverse-lemma:
  [a ≠ 0; b ≠ (0::'a::real-normed-field)]
  ⇒ (inverse a - inverse b) / h
  = - (inverse a * ((a - b) / h) * inverse b)
by (simp add: inverse-diff-inverse)

```

```

lemma DERIV-inverse':
  assumes der: DERIV f x :> D
  assumes neq: f x ≠ 0
  shows DERIV (λx. inverse (f x)) x :> - (inverse (f x) * D * inverse (f x))
    (is DERIV - - :> ?E)
proof (unfold DERIV-iff2)
  from der have lim-f: f -- x --> f x
    by (rule DERIV-isCont [unfolded isCont-def])

```

```

from neq have 0 < norm (f x) by simp
with LIM-D [OF lim-f] obtain s
  where s: 0 < s
  and less-fx: ∧z. [z ≠ x; norm (z - x) < s]
    ⇒ norm (f z - f x) < norm (f x)
by fast

```

```

show (λz. (inverse (f z) - inverse (f x)) / (z - x)) -- x --> ?E
proof (rule LIM-equal2 [OF s])
  fix z
  assume z ≠ x norm (z - x) < s
  hence norm (f z - f x) < norm (f x) by (rule less-fx)
  hence f z ≠ 0 by auto
  thus (inverse (f z) - inverse (f x)) / (z - x) =
    - (inverse (f z) * ((f z - f x) / (z - x)) * inverse (f x))
    using neq by (rule DERIV-inverse-lemma)
  next
  from der have (λz. (f z - f x) / (z - x)) -- x --> D
    by (unfold DERIV-iff2)
  thus (λz. - (inverse (f z) * ((f z - f x) / (z - x)) * inverse (f x)))

```

```

-- x --> ?E
by (intro LIM-mult LIM-inverse LIM-minus LIM-const lim-f neq)
qed
qed

```

**lemma** *DERIV-divide*:

```

[[DERIV f x :> D; DERIV g x :> E; g x ≠ 0]]
  => DERIV (λx. f x / g x) x :> (D * g x - f x * E) / (g x * g x)
apply (subgoal-tac f x * - (inverse (g x) * E * inverse (g x)) +
      D * inverse (g x) = (D * g x - f x * E) / (g x * g x))
apply (erule subst)
apply (unfold divide-inverse)
apply (erule DERIV-mult')
apply (erule (1) DERIV-inverse')
apply (simp add: ring-distrib nonzero-inverse-mult-distrib)
apply (simp add: mult-ac)
done

```

**lemma** *DERIV-power-Suc*:

```

fixes f :: 'a ⇒ 'a::{real-normed-field}
assumes f: DERIV f x :> D
shows DERIV (λx. f x ^ Suc n) x :> (1 + of-nat n) * (D * f x ^ n)
proof (induct n)
case 0
  show ?case by (simp add: f)
case (Suc k)
  from DERIV-mult' [OF f Suc] show ?case
  apply (simp only: of-nat-Suc ring-distrib mult-1-left)
  apply (simp only: power-Suc algebra-simps)
  done
qed

```

**lemma** *DERIV-power*:

```

fixes f :: 'a ⇒ 'a::{real-normed-field}
assumes f: DERIV f x :> D
shows DERIV (λx. f x ^ n) x :> of-nat n * (D * f x ^ (n - Suc 0))
by (cases n, simp, simp add: DERIV-power-Suc f del: power-Suc)

```

Caratheodory formulation of derivative at a point

**lemma** *CARAT-DERIV*:

```

(DERIV f x :> l) =
  (∃ g. (∀ z. f z - f x = g z * (z - x)) & isCont g x & g x = l)
  (is ?lhs = ?rhs)
proof
  assume der: DERIV f x :> l
  show ∃ g. (∀ z. f z - f x = g z * (z - x)) ∧ isCont g x ∧ g x = l
  proof (intro exI conjI)
    let ?g = (%z. if z = x then l else (f z - f x) / (z - x))
    show ∀ z. f z - f x = ?g z * (z - x) by simp
  end
end

```

```

show isCont ?g x using der
  by (simp add: isCont-iff DERIV-iff diff-minus
      cong: LIM-equal [rule-format])
show ?g x = l by simp
qed
next
assume ?rhs
then obtain g where
  ( $\forall z. f\ z - f\ x = g\ z * (z - x)$ ) and isCont g x and g x = l by blast
thus (DERIV f x :> l)
  by (auto simp add: isCont-iff DERIV-iff cong: LIM-cong)
qed

```

```

lemma DERIV-chain':
  assumes f: DERIV f x :> D
  assumes g: DERIV g (f x) :> E
  shows DERIV ( $\lambda x. g\ (f\ x)$ ) x :> E * D
proof (unfold DERIV-iff2)
  obtain d where d:  $\forall y. g\ y - g\ (f\ x) = d\ y * (y - f\ x)$ 
    and cont-d: isCont d (f x) and dfx: d (f x) = E
    using CARAT-DERIV [THEN iffD1, OF g] by fast
  from f have f -- x --> f x
    by (rule DERIV-isCont [unfolded isCont-def])
  with cont-d have ( $\lambda z. d\ (f\ z)$ ) -- x --> d (f x)
    by (rule isCont-LIM-compose)
  hence ( $\lambda z. d\ (f\ z) * ((f\ z - f\ x) / (z - x))$ )
    -- x --> d (f x) * D
    by (rule LIM-mult [OF - f [unfolded DERIV-iff2]])
  thus ( $\lambda z. (g\ (f\ z) - g\ (f\ x)) / (z - x)$ ) -- x --> E * D
    by (simp add: d dfx)
qed

```

Let's do the standard proof, though theorem *LIM-mult2* follows from a NS proof

```

lemma DERIV-cmult:
  DERIV f x :> D ==> DERIV ( $\%x. c * f\ x$ ) x :> c*D
by (drule DERIV-mult' [OF DERIV-const], simp)

```

```

lemma DERIV-cdivide: DERIV f x :> D ==> DERIV ( $\%x. f\ x / c$ ) x :> D / c
apply (subgoal-tac DERIV ( $\%x. (1 / c) * f\ x$ ) x :> (1 / c) * D, force)
apply (erule DERIV-cmult)
done

```

Standard version

```

lemma DERIV-chain: [DERIV f (g x) :> Da; DERIV g x :> Db] ==> DERIV
(f o g) x :> Da * Db
by (drule (1) DERIV-chain', simp add: o-def mult-commute)

```

**lemma** *DERIV-chain2*:  $[[ \text{DERIV } f \ (g \ x) :> Da; \text{DERIV } g \ x :> Db ]]$   $\implies$   
 $\text{DERIV } (\%x. f \ (g \ x)) \ x :> Da * Db$   
**by** (*auto dest: DERIV-chain simp add: o-def*)

Derivative of linear multiplication

**lemma** *DERIV-cmult-Id* [*simp*]:  $\text{DERIV } (op * c) \ x :> c$   
**by** (*cut-tac c = c and x = x in DERIV-ident [THEN DERIV-cmult], simp*)

**lemma** *DERIV-pow*:  $\text{DERIV } (\%x. x ^ n) \ x :> \text{real } n * (x ^ (n - \text{Suc } 0))$   
**apply** (*cut-tac DERIV-power [OF DERIV-ident]*)  
**apply** (*simp add: real-of-nat-def*)  
**done**

Power of  $-1$

**lemma** *DERIV-inverse*:  
**fixes**  $x :: 'a::\{\text{real-normed-field}\}$   
**shows**  $x \neq 0 \implies \text{DERIV } (\%x. \text{inverse}(x)) \ x :> -(\text{inverse } x ^ \text{Suc } (\text{Suc } 0))$   
**by** (*drule DERIV-inverse' [OF DERIV-ident] simp*)

Derivative of inverse

**lemma** *DERIV-inverse-fun*:  
**fixes**  $x :: 'a::\{\text{real-normed-field}\}$   
**shows**  $[[ \text{DERIV } f \ x :> d; f(x) \neq 0 ]]$   
 $\implies \text{DERIV } (\%x. \text{inverse}(f \ x)) \ x :> -(d * \text{inverse}(f(x) ^ \text{Suc } (\text{Suc } 0)))$   
**by** (*drule (1) DERIV-inverse' (simp add: mult-ac nonzero-inverse-mult-distrib)*)

Derivative of quotient

**lemma** *DERIV-quotient*:  
**fixes**  $x :: 'a::\{\text{real-normed-field}\}$   
**shows**  $[[ \text{DERIV } f \ x :> d; \text{DERIV } g \ x :> e; g(x) \neq 0 ]]$   
 $\implies \text{DERIV } (\%y. f(y) / (g \ y)) \ x :> (d * g(x) - (e * f(x))) / (g(x) ^ \text{Suc } (\text{Suc } 0))$   
**by** (*drule (2) DERIV-divide (simp add: mult-commute)*)

**lemma** *lemma-DERIV-subst*:  $[[ \text{DERIV } f \ x :> D; D = E ]]$   $\implies \text{DERIV } f \ x :> E$   
**by** *auto*

*DERIV-intros*

**ML**  $\langle\langle$   
*structure Deriv-Intros = Named-Thms*  
*(*  
*val name = DERIV-intros*  
*val description = DERIV introduction rules*  
*)*  
 $\rangle\rangle$

**setup** *Deriv-Intros.setup*

**lemma** *DERIV-cong*:  $\llbracket \text{DERIV } f \ x \ :> \ X \ ; \ X = Y \rrbracket \implies \text{DERIV } f \ x \ :> \ Y$   
**by** *simp*

**declare**

*DERIV-const*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-ident*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-add*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-minus*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-mult*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-diff*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-inverse'*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-divide*[*THEN DERIV-cong, DERIV-intros*]  
*DERIV-power*[**where** *'a=real, THEN DERIV-cong,*  
*unfolded real-of-nat-def[symmetric], DERIV-intros*]  
*DERIV-setsum*[*THEN DERIV-cong, DERIV-intros*]

### 73.2 Differentiability predicate

**definition**

*differentiable* :: [*'a::real-normed-field*  $\Rightarrow$  *'a, 'a*]  $\Rightarrow$  *bool*  
**(infixl** *differentiable* 60) **where**  
*f differentiable* *x* = ( $\exists D. \text{DERIV } f \ x \ :> \ D$ )

**lemma** *differentiableE* [*elim?*]:

**assumes** *f differentiable* *x*  
**obtains** *df* **where** *DERIV f x :> df*  
**using** *prems* **unfolding** *differentiable-def* ..

**lemma** *differentiableD*: *f differentiable* *x*  $\implies \exists D. \text{DERIV } f \ x \ :> \ D$   
**by** (*simp add: differentiable-def*)

**lemma** *differentiableI*: *DERIV f x :> D*  $\implies f$  *differentiable* *x*  
**by** (*force simp add: differentiable-def*)

**lemma** *differentiable-ident* [*simp*]:  $(\lambda x. x)$  *differentiable* *x*  
**by** (*rule DERIV-ident [THEN differentiableI]*)

**lemma** *differentiable-const* [*simp*]:  $(\lambda z. a)$  *differentiable* *x*  
**by** (*rule DERIV-const [THEN differentiableI]*)

**lemma** *differentiable-compose*:

**assumes** *f: f differentiable (g x)*  
**assumes** *g: g differentiable x*  
**shows**  $(\lambda x. f \ (g \ x))$  *differentiable* *x*

**proof** –

**from**  $\langle f \text{ differentiable } (g \ x) \rangle$  **obtain** *df* **where** *DERIV f (g x) :> df* ..  
**moreover**  
**from**  $\langle g \text{ differentiable } x \rangle$  **obtain** *dg* **where** *DERIV g x :> dg* ..

ultimately  
 have  $DERIV (\lambda x. f (g x)) x :> df * dg$  by (rule *DERIV-chain2*)  
 thus ?thesis by (rule *differentiableI*)  
 qed

lemma *differentiable-sum* [simp]:  
 assumes  $f$  differentiable  $x$   
 and  $g$  differentiable  $x$   
 shows  $(\lambda x. f x + g x)$  differentiable  $x$   
 proof –  
 from  $\langle f \text{ differentiable } x \rangle$  obtain  $df$  where  $DERIV f x :> df$  ..  
 moreover  
 from  $\langle g \text{ differentiable } x \rangle$  obtain  $dg$  where  $DERIV g x :> dg$  ..  
 ultimately  
 have  $DERIV (\lambda x. f x + g x) x :> df + dg$  by (rule *DERIV-add*)  
 thus ?thesis by (rule *differentiableI*)  
 qed

lemma *differentiable-minus* [simp]:  
 assumes  $f$  differentiable  $x$   
 shows  $(\lambda x. - f x)$  differentiable  $x$   
 proof –  
 from  $\langle f \text{ differentiable } x \rangle$  obtain  $df$  where  $DERIV f x :> df$  ..  
 hence  $DERIV (\lambda x. - f x) x :> - df$  by (rule *DERIV-minus*)  
 thus ?thesis by (rule *differentiableI*)  
 qed

lemma *differentiable-diff* [simp]:  
 assumes  $f$  differentiable  $x$   
 assumes  $g$  differentiable  $x$   
 shows  $(\lambda x. f x - g x)$  differentiable  $x$   
 unfolding *diff-minus* using *prems* by *simp*

lemma *differentiable-mult* [simp]:  
 assumes  $f$  differentiable  $x$   
 assumes  $g$  differentiable  $x$   
 shows  $(\lambda x. f x * g x)$  differentiable  $x$   
 proof –  
 from  $\langle f \text{ differentiable } x \rangle$  obtain  $df$  where  $DERIV f x :> df$  ..  
 moreover  
 from  $\langle g \text{ differentiable } x \rangle$  obtain  $dg$  where  $DERIV g x :> dg$  ..  
 ultimately  
 have  $DERIV (\lambda x. f x * g x) x :> df * g x + dg * f x$  by (rule *DERIV-mult*)  
 thus ?thesis by (rule *differentiableI*)  
 qed

lemma *differentiable-inverse* [simp]:  
 assumes  $f$  differentiable  $x$  and  $f x \neq 0$   
 shows  $(\lambda x. \text{inverse } (f x))$  differentiable  $x$

**proof** –

**from**  $\langle f \text{ differentiable } x \rangle$  **obtain**  $df$  **where**  $DERIV\ f\ x\ :\>\ df\ ..$   
**hence**  $DERIV\ (\lambda x. \text{inverse}\ (f\ x))\ x\ :\>\ -\ (\text{inverse}\ (f\ x) * df * \text{inverse}\ (f\ x))$   
**using**  $\langle f\ x \neq 0 \rangle$  **by** (rule  $DERIV\text{-inverse}'$ )  
**thus**  $?thesis$  **by** (rule  $\text{differentiableI}$ )  
**qed**

**lemma**  $\text{differentiable-divide}$  [simp]:

**assumes**  $f \text{ differentiable } x$   
**assumes**  $g \text{ differentiable } x$  **and**  $g\ x \neq 0$   
**shows**  $(\lambda x. f\ x / g\ x) \text{ differentiable } x$   
**unfolding**  $\text{divide-inverse}$  **using**  $\text{prems}$  **by**  $\text{simp}$

**lemma**  $\text{differentiable-power}$  [simp]:

**fixes**  $f :: 'a :: \{\text{real-normed-field}\} \Rightarrow 'a$   
**assumes**  $f \text{ differentiable } x$   
**shows**  $(\lambda x. f\ x ^ n) \text{ differentiable } x$   
**by** (induct  $n$ ,  $\text{simp}$ ,  $\text{simp add: prems}$ )

### 73.3 Nested Intervals and Bisection

Lemmas about nested intervals and proof by bisection (cf.Harrison). All considerably tidied by lcp.

**lemma**  $\text{lemma-f-mono-add}$  [rule-format (no-asm)]:  $(\forall n. (f :: \text{nat} \Rightarrow \text{real})\ n \leq f\ (\text{Suc}\ n)) \longrightarrow f\ m \leq f\ (m + n)$   
**apply** (induct  $n$ )  
**apply** (auto intro: order-trans)  
**done**

**lemma**  $f\text{-inc-g-dec-Beq-f}$ :  $[| \forall n. f(n) \leq f(\text{Suc}\ n);$   
 $\forall n. g(\text{Suc}\ n) \leq g(n);$   
 $\forall n. f(n) \leq g(n) |]$   
 $\implies \text{Bseq}\ (f :: \text{nat} \Rightarrow \text{real})$   
**apply** (rule-tac  $k = f\ 0$  **and**  $K = g\ 0$  **in**  $\text{BseqI2}$ , rule  $\text{allI}$ )  
**apply** (induct-tac  $n$ )  
**apply** (auto intro: order-trans)  
**apply** (rule-tac  $y = g\ (\text{Suc}\ na)$  **in** order-trans)  
**apply** (induct-tac [2]  $na$ )  
**apply** (auto intro: order-trans)  
**done**

**lemma**  $f\text{-inc-g-dec-Beq-g}$ :  $[| \forall n. f(n) \leq f(\text{Suc}\ n);$   
 $\forall n. g(\text{Suc}\ n) \leq g(n);$   
 $\forall n. f(n) \leq g(n) |]$   
 $\implies \text{Bseq}\ (g :: \text{nat} \Rightarrow \text{real})$   
**apply** (subst  $\text{Bseq-minus-iff}$  [symmetric])  
**apply** (rule-tac  $g = \%x. -\ (f\ x)$  **in**  $f\text{-inc-g-dec-Beq-f}$ )  
**apply** auto  
**done**

**lemma** *f-inc-imp-le-lim*:

fixes  $f :: \text{nat} \Rightarrow \text{real}$   
 shows  $\llbracket \forall n. f\ n \leq f\ (\text{Suc}\ n); \text{convergent}\ f \rrbracket \Longrightarrow f\ n \leq \lim\ f$   
 apply (rule linorder-not-less [THEN iffD1])  
 apply (auto simp add: convergent-LIMSEQ-iff LIMSEQ-iff monoseq-Suc)  
 apply (drule-tac  $x = f\ n - \lim\ f$  in spec, clarsimp)  
 apply (drule-tac  $P = \%na. no \leq na \longrightarrow ?Q\ na$  and  $x = no + n$  in spec, auto)  
 apply (subgoal-tac  $\lim\ f \leq f\ (no + n)$ )  
 apply (drule-tac  $no=no$  and  $m=n$  in lemma-f-mono-add)  
 apply (auto simp add: add-commute)  
 apply (induct-tac no)  
 apply simp  
 apply (auto intro: order-trans simp add: diff-minus abs-if)  
 done

**lemma** *lim-uminus*:

fixes  $g :: \text{nat} \Rightarrow 'a::\text{real-normed-vector}$   
 shows  $\text{convergent}\ g \Longrightarrow \lim\ (\%x. -\ g\ x) = -\ (\lim\ g)$   
 apply (rule LIMSEQ-minus [THEN limI])  
 apply (simp add: convergent-LIMSEQ-iff)  
 done

**lemma** *g-dec-imp-lim-le*:

fixes  $g :: \text{nat} \Rightarrow \text{real}$   
 shows  $\llbracket \forall n. g\ (\text{Suc}\ n) \leq g\ (n); \text{convergent}\ g \rrbracket \Longrightarrow \lim\ g \leq g\ n$   
 apply (subgoal-tac  $-(g\ n) \leq -(\lim\ g)$ )  
 apply (cut-tac [2]  $f = \%x. -(g\ x)$  in f-inc-imp-le-lim)  
 apply (auto simp add: lim-uminus convergent-minus-iff [symmetric])  
 done

**lemma** *lemma-nest*:  $\llbracket \forall n. f\ (n) \leq f\ (\text{Suc}\ n);$

$\forall n. g\ (\text{Suc}\ n) \leq g\ (n);$

$\forall n. f\ (n) \leq g\ (n) \rrbracket$

$\Longrightarrow \exists l\ m :: \text{real}. l \leq m \ \& \ ((\forall n. f\ (n) \leq l) \ \& \ f \dashrightarrow l) \ \& \ ((\forall n. m \leq g\ (n)) \ \& \ g \dashrightarrow m)$

apply (subgoal-tac monoseq  $f$  & monoseq  $g$ )

prefer 2 apply (force simp add: LIMSEQ-iff monoseq-Suc)

apply (subgoal-tac Bseq  $f$  & Bseq  $g$ )

prefer 2 apply (blast intro: f-inc-g-dec-Beq-f f-inc-g-dec-Beq-g)

apply (auto dest!: Bseq-monoseq-convergent simp add: convergent-LIMSEQ-iff)

apply (rule-tac  $x = \lim\ f$  in exI)

apply (rule-tac  $x = \lim\ g$  in exI)

apply (auto intro: LIMSEQ-le)

apply (auto simp add: f-inc-imp-le-lim g-dec-imp-lim-le convergent-LIMSEQ-iff)

done

**lemma** *lemma-nest-unique*:  $\llbracket \forall n. f\ (n) \leq f\ (\text{Suc}\ n);$

$\forall n. g\ (\text{Suc}\ n) \leq g\ (n);$



$$\begin{aligned} & \forall n. f(n) \leq g(n); \\ & (\%n. f(n) - g(n)) \text{ ----> } 0 \text{ []} \\ \implies & \exists l::\text{real}. ((\forall n. f(n) \leq l) \ \& \ f \text{ ----> } l) \ \& \\ & ((\forall n. l \leq g(n)) \ \& \ g \text{ ----> } l) \end{aligned}$$

```

apply (drule lemma-nest, auto)
apply (subgoal-tac l = m)
apply (drule-tac [2] X = f in LIMSEQ-diff)
apply (auto intro: LIMSEQ-unique)
done

```

The universal quantifiers below are required for the declaration of *Bolzano-nest-unique* below.

```

lemma Bolzano-bisect-le:
  a ≤ b ==> ∀ n. fst (Bolzano-bisect P a b n) ≤ snd (Bolzano-bisect P a b n)
apply (rule allI)
apply (induct-tac n)
apply (auto simp add: Let-def split-def)
done

```

```

lemma Bolzano-bisect-fst-le-Suc: a ≤ b ==>
  ∀ n. fst(Bolzano-bisect P a b n) ≤ fst (Bolzano-bisect P a b (Suc n))
apply (rule allI)
apply (induct-tac n)
apply (auto simp add: Bolzano-bisect-le Let-def split-def)
done

```

```

lemma Bolzano-bisect-Suc-le-snd: a ≤ b ==>
  ∀ n. snd(Bolzano-bisect P a b (Suc n)) ≤ snd (Bolzano-bisect P a b n)
apply (rule allI)
apply (induct-tac n)
apply (auto simp add: Bolzano-bisect-le Let-def split-def)
done

```

```

lemma eq-divide-2-times-iff: ((x::real) = y / (2 * z)) = (2 * x = y/z)
apply (auto)
apply (drule-tac f = %u. (1/2) * u in arg-cong)
apply (simp)
done

```

```

lemma Bolzano-bisect-diff:
  a ≤ b ==>
  snd(Bolzano-bisect P a b n) - fst(Bolzano-bisect P a b n) =
  (b-a) / (2 ^ n)
apply (induct n)
apply (auto simp add: eq-divide-2-times-iff add-divide-distrib Let-def split-def)
done

```

```

lemmas Bolzano-nest-unique =
  lemma-nest-unique

```

[OF Bolzano-bisect-fst-le-Suc Bolzano-bisect-Suc-le-snd Bolzano-bisect-le]

**lemma** *not-P-Bolzano-bisect*:  
**assumes**  $P$ :  $\llbracket \forall a\ b\ c. \llbracket P(a,b); P(b,c); a \leq b; b \leq c \rrbracket \implies P(a,c)$   
**and**  $\text{not}P$ :  $\sim P(a,b)$   
**and**  $\text{le}$ :  $a \leq b$   
**shows**  $\sim P(\text{fst}(\text{Bolzano-bisect } P\ a\ b\ n), \text{snd}(\text{Bolzano-bisect } P\ a\ b\ n))$   
**proof** (*induct*  $n$ )  
**case** 0 **show** ?case **using**  $\text{not}P$  **by** *simp*  
**next**  
**case** (*Suc*  $n$ )  
**thus** ?case  
**by** (*auto simp del: surjective-pairing [symmetric]*  
*simp add: Let-def split-def Bolzano-bisect-le [OF le]*  
 $P$  [*of fst (Bolzano-bisect*  $P\ a\ b\ n$ ) - *snd (Bolzano-bisect*  $P\ a\ b\ n$ )])  
**qed**

**lemma** *not-P-Bolzano-bisect'*:  
 $\llbracket \forall a\ b\ c. P(a,b) \ \&\ P(b,c) \ \&\ a \leq b \ \&\ b \leq c \implies P(a,c);$   
 $\sim P(a,b); a \leq b \rrbracket \implies$   
 $\forall n. \sim P(\text{fst}(\text{Bolzano-bisect } P\ a\ b\ n), \text{snd}(\text{Bolzano-bisect } P\ a\ b\ n))$   
**by** (*blast elim!: not-P-Bolzano-bisect [THEN [2] rev-notE]*)

**lemma** *lemma-BOLZANO*:  
 $\llbracket \forall a\ b\ c. P(a,b) \ \&\ P(b,c) \ \&\ a \leq b \ \&\ b \leq c \implies P(a,c);$   
 $\forall x. \exists d::\text{real}. 0 < d \ \&$   
 $(\forall a\ b. a \leq x \ \&\ x \leq b \ \&\ (b-a) < d \implies P(a,b));$   
 $a \leq b \rrbracket$   
 $\implies P(a,b)$   
**apply** (*rule Bolzano-nest-unique [where*  $P1=P$ , *THEN*  $\text{exE}$ ], *assumption+*)  
**apply** (*rule LIMSEQ-minus-cancel*)  
**apply** (*simp (no-asm-simp) add: Bolzano-bisect-diff LIMSEQ-divide-realpow-zero*)  
**apply** (*rule ccontr*)  
**apply** (*drule not-P-Bolzano-bisect', assumption+*)  
**apply** (*rename-tac*  $l$ )  
**apply** (*drule-tac*  $x = l$  **in** *spec*, *clarify*)  
**apply** (*simp add: LIMSEQ-iff*)  
**apply** (*drule-tac*  $P = \%r. 0 < r \implies ?Q\ r$  **and**  $x = d/2$  **in** *spec*)  
**apply** (*drule-tac*  $P = \%r. 0 < r \implies ?Q\ r$  **and**  $x = d/2$  **in** *spec*)  
**apply** (*drule real-less-half-sum, auto*)  
**apply** (*drule-tac*  $x = \text{fst}(\text{Bolzano-bisect } P\ a\ b\ (\text{no} + \text{noa}))$  **in** *spec*)  
**apply** (*drule-tac*  $x = \text{snd}(\text{Bolzano-bisect } P\ a\ b\ (\text{no} + \text{noa}))$  **in** *spec*)  
**apply** *safe*  
**apply** (*simp-all (no-asm-simp)*)  
**apply** (*rule-tac*  $y = \text{abs}(\text{fst}(\text{Bolzano-bisect } P\ a\ b\ (\text{no} + \text{noa})) - l) + \text{abs}(\text{snd}$

```

(Bolzano-bisect  $P$   $a$   $b$  ( $no + noa$ ))  $- l$ ) in order-le-less-trans)
apply (simp (no-asm-simp) add: abs-if)
apply (rule real-sum-of-halves [THEN subst])
apply (rule add-strict-mono)
apply (simp-all add: diff-minus [symmetric])
done

```

```

lemma lemma-BOLZANO2: (( $\forall a\ b\ c. (a \leq b \ \& \ b \leq c \ \& \ P(a,b) \ \& \ P(b,c)) \dashrightarrow$ 
 $P(a,c)$ ) &
  ( $\forall x. \exists d::real. 0 < d \ \&$ 
    ( $\forall a\ b. a \leq x \ \& \ x \leq b \ \& \ (b-a) < d \dashrightarrow P(a,b)$ )))
   $\dashrightarrow (\forall a\ b. a \leq b \dashrightarrow P(a,b))$ 
apply clarify
apply (blast intro: lemma-BOLZANO)
done

```

### 73.4 Intermediate Value Theorem

Prove Contrapositive by Bisection

```

lemma IVT: [|  $f(a::real) \leq (y::real); y \leq f(b);$ 
   $a \leq b;$ 
  ( $\forall x. a \leq x \ \& \ x \leq b \dashrightarrow isCont\ f\ x$ ) |]
   $\implies \exists x. a \leq x \ \& \ x \leq b \ \& \ f(x) = y$ 
apply (rule contrapos-pp, assumption)
apply (cut-tac  $P = \% (u,v) . a \leq u \ \& \ u \leq v \ \& \ v \leq b \dashrightarrow \sim (f\ (u) \leq y \ \& \ y \leq$ 
 $f\ (v))$  in lemma-BOLZANO2)
apply safe
apply simp-all
apply (simp add: isCont-iff LIM-eq)
apply (rule ccontr)
apply (subgoal-tac  $a \leq x \ \& \ x \leq b$ )
prefer 2
apply simp
apply (drule-tac  $P = \% d. 0 < d \dashrightarrow ?P\ d$  and  $x = 1$  in spec, arith)
apply (drule-tac  $x = x$  in spec) +
apply simp
apply (drule-tac  $P = \% r. ?P\ r \dashrightarrow (\exists s > 0. ?Q\ r\ s)$  and  $x = |y - f\ x|$  in spec)
apply safe
apply simp
apply (drule-tac  $x = s$  in spec, clarify)
apply (cut-tac  $x = f\ x$  and  $y = y$  in linorder-less-linear, safe)
apply (drule-tac  $x = ba - x$  in spec)
apply (simp-all add: abs-if)
apply (drule-tac  $x = aa - x$  in spec)
apply (case-tac  $x \leq aa$ , simp-all)
done

```

```

lemma IVT2: [|  $f(b::real) \leq (y::real); y \leq f(a);$ 

```

```

      a ≤ b;
      (∀ x. a ≤ x & x ≤ b --> isCont f x)
    || ==> ∃ x. a ≤ x & x ≤ b & f(x) = y
  apply (subgoal-tac - f a ≤ -y & -y ≤ - f b, clarify)
  apply (drule IVT [where f = %x. - f x], assumption)
  apply (auto intro: isCont-minus)
done

```

```

lemma IVT-objl: (f(a::real) ≤ (y::real) & y ≤ f(b) & a ≤ b &
  (∀ x. a ≤ x & x ≤ b --> isCont f x))
  --> (∃ x. a ≤ x & x ≤ b & f(x) = y)
apply (blast intro: IVT)
done

```

```

lemma IVT2-objl: (f(b::real) ≤ (y::real) & y ≤ f(a) & a ≤ b &
  (∀ x. a ≤ x & x ≤ b --> isCont f x))
  --> (∃ x. a ≤ x & x ≤ b & f(x) = y)
apply (blast intro: IVT2)
done

```

### 73.5 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

```

lemma isCont-bounded:
  || a ≤ b; ∀ x. a ≤ x & x ≤ b --> isCont f x ||
  ==> ∃ M::real. ∀ x::real. a ≤ x & x ≤ b --> f(x) ≤ M
apply (cut-tac P = % (u,v) . a ≤ u & u ≤ v & v ≤ b --> (∃ M. ∀ x. u ≤ x &
  x ≤ v --> f x ≤ M) in lemma-BOLZANO2)
apply safe
apply simp-all
apply (rename-tac x xa ya M Ma)
apply (metis linorder-not-less order-le-less order-trans)
apply (case-tac a ≤ x & x ≤ b)
prefer 2
  apply (rule-tac x = 1 in exI, force)
apply (simp add: LIM-eq isCont-iff)
apply (drule-tac x = x in spec, auto)
apply (erule-tac V = ∀ M. ∃ x. a ≤ x & x ≤ b & ~ f x ≤ M in thin-rl)
apply (drule-tac x = 1 in spec, auto)
apply (rule-tac x = s in exI, clarify)
apply (rule-tac x = |f x| + 1 in exI, clarify)
apply (drule-tac x = xa - x in spec)
apply (auto simp add: abs-ge-self)
done

```

Refine the above to existence of least upper bound

```

lemma lemma-reals-complete: ((∃ x. x ∈ S) & (∃ y. isUb UNIV S (y::real))) -->
  (∃ t. isLub UNIV S t)

```

by (blast intro: reals-complete)

```

lemma isCont-has-Ub: [| a ≤ b; ∀ x. a ≤ x & x ≤ b --> isCont f x |]
  ==> ∃ M::real. (∀ x::real. a ≤ x & x ≤ b --> f(x) ≤ M) &
    (∀ N. N < M --> (∃ x. a ≤ x & x ≤ b & N < f(x)))
apply (cut-tac S = Collect (%y. ∃ x. a ≤ x & x ≤ b & y = f x)
  in lemma-reals-complete)
apply auto
apply (drule isCont-bounded, assumption)
apply (auto simp add: isUb-def leastP-def isLub-def setge-def settle-def)
apply (rule exI, auto)
apply (auto dest!: spec simp add: linorder-not-less)
done

```

Now show that it attains its upper bound

```

lemma isCont-eq-Ub:
  assumes le: a ≤ b
    and con: ∀ x::real. a ≤ x & x ≤ b --> isCont f x
  shows ∃ M::real. (∀ x. a ≤ x & x ≤ b --> f(x) ≤ M) &
    (∃ x. a ≤ x & x ≤ b & f(x) = M)
proof -
  from isCont-has-Ub [OF le con]
  obtain M where M1: ∀ x. a ≤ x & x ≤ b --> f x ≤ M
    and M2: !!N. N < M ==> ∃ x. a ≤ x & x ≤ b & N < f x by blast
  show ?thesis
  proof (intro exI, intro conjI)
    show ∀ x. a ≤ x & x ≤ b --> f x ≤ M by (rule M1)
    show ∃ x. a ≤ x & x ≤ b & f x = M
    proof (rule ccontr)
      assume ¬ (∃ x. a ≤ x & x ≤ b & f x = M)
      with M1 have M3: ∀ x. a ≤ x & x ≤ b --> f x < M
        by (fastsimp simp add: linorder-not-le [symmetric])
      hence ∀ x. a ≤ x & x ≤ b --> isCont (%x. inverse (M - f x)) x
        by (auto simp add: isCont-inverse isCont-diff con)
      from isCont-bounded [OF le this]
      obtain k where k: !!x. a ≤ x & x ≤ b --> inverse (M - f x) ≤ k by auto
      have Minv: !!x. a ≤ x & x ≤ b --> 0 < inverse (M - f x))
        by (simp add: M3 algebra-simps)
      have !!x. a ≤ x & x ≤ b --> inverse (M - f x) < k+1 using k
        by (auto intro: order-le-less-trans [of - k])
      with Minv
      have !!x. a ≤ x & x ≤ b --> inverse(k+1) < inverse(inverse(M - f x))
        by (intro strip less-imp-inverse-less, simp-all)
      hence invlt: !!x. a ≤ x & x ≤ b --> inverse(k+1) < M - f x
        by simp
      have M - inverse (k+1) < M using k [of a] Minv [of a] le
        by (simp, arith)
      from M2 [OF this]
      obtain x where ax: a ≤ x & x ≤ b & M - inverse(k+1) < f x ..

```

```

      thus False using invlt [of x] by force
    qed
  qed
qed

```

Same theorem for lower bound

```

lemma isCont-eq-Lb: [| a ≤ b; ∀ x. a ≤ x & x ≤ b --> isCont f x |]
  ==> ∃ M::real. (∀ x::real. a ≤ x & x ≤ b --> M ≤ f(x)) &
    (∃ x. a ≤ x & x ≤ b & f(x) = M)
apply (subgoal-tac ∀ x. a ≤ x & x ≤ b --> isCont (%x. - (f x)) x)
prefer 2 apply (blast intro: isCont-minus)
apply (drule-tac f = (%x. - (f x)) in isCont-eq-Ub)
apply safe
apply auto
done

```

Another version.

```

lemma isCont-Lb-Ub: [| a ≤ b; ∀ x. a ≤ x & x ≤ b --> isCont f x |]
  ==> ∃ L M::real. (∀ x::real. a ≤ x & x ≤ b --> L ≤ f(x) & f(x) ≤ M) &
    (∀ y. L ≤ y & y ≤ M --> (∃ x. a ≤ x & x ≤ b & (f(x) = y)))
apply (frule isCont-eq-Lb)
apply (frule-tac [2] isCont-eq-Ub)
apply (assumption+, safe)
apply (rule-tac x = f x in exI)
apply (rule-tac x = f xa in exI, simp, safe)
apply (cut-tac x = x and y = xa in linorder-linear, safe)
apply (cut-tac f = f and a = x and b = xa and y = y in IVT-objl)
apply (cut-tac [2] f = f and a = xa and b = x and y = y in IVT2-objl, safe)
apply (rule-tac [2] x = xb in exI)
apply (rule-tac [4] x = xb in exI, simp-all)
done

```

### 73.6 Local extrema

If  $(0::'a) < f' x$  then  $x$  is Locally Strictly Increasing At The Right

```

lemma DERIV-pos-inc-right:
  fixes f :: real => real
  assumes der: DERIV f x :> l
  and l: 0 < l
  shows ∃ d > 0. ∀ h > 0. h < d --> f(x) < f(x + h)
proof -
  from l der [THEN DERIV-D, THEN LIM-D [where r = l]]
  have ∃ s > 0. (∀ z. z ≠ 0 ∧ |z| < s ⟶ |(f(x+z) - f x) / z - l| < l)
    by (simp add: diff-minus)
  then obtain s
    where s: 0 < s
    and all: !!z. z ≠ 0 ∧ |z| < s ⟶ |(f(x+z) - f x) / z - l| < l
  by auto

```

```

thus ?thesis
proof (intro exI conjI strip)
  show  $0 < s$  using  $s$  .
  fix  $h :: \text{real}$ 
  assume  $0 < h$   $h < s$ 
  with all [of  $h$ ] show  $f\ x < f\ (x+h)$ 
  proof (simp add: abs-if pos-less-divide-eq diff-minus [symmetric]
    split add: split-if-asm)
    assume  $\sim (f\ (x+h) - f\ x) / h < l$  and  $h: 0 < h$ 
    with  $l$ 
    have  $0 < (f\ (x+h) - f\ x) / h$  by arith
    thus  $f\ x < f\ (x+h)$ 
  by (simp add: pos-less-divide-eq  $h$ )
  qed
qed
qed

lemma DERIV-neg-dec-left:
  fixes  $f :: \text{real} \Rightarrow \text{real}$ 
  assumes  $\text{der}: \text{DERIV}\ f\ x :> l$ 
  and  $l: l < 0$ 
  shows  $\exists d > 0. \forall h > 0. h < d \longrightarrow f(x) < f(x-h)$ 
proof -
  from  $l\ \text{der}$  [THEN DERIV-D, THEN LIM-D [where  $r = -l$ ]]
  have  $\exists s > 0. (\forall z. z \neq 0 \wedge |z| < s \longrightarrow |(f(x+z) - f(x)) / z - l| < -l)$ 
  by (simp add: diff-minus)
  then obtain  $s$ 
  where  $s: 0 < s$ 
  and all:  $\forall z. z \neq 0 \wedge |z| < s \longrightarrow |(f(x+z) - f(x)) / z - l| < -l$ 
  by auto
  thus ?thesis
proof (intro exI conjI strip)
  show  $0 < s$  using  $s$  .
  fix  $h :: \text{real}$ 
  assume  $0 < h$   $h < s$ 
  with all [of  $-h$ ] show  $f\ x < f\ (x-h)$ 
  proof (simp add: abs-if pos-less-divide-eq diff-minus [symmetric]
    split add: split-if-asm)
    assume  $\sim ((f\ (x-h) - f\ x) / h) < l$  and  $h: 0 < h$ 
    with  $l$ 
    have  $0 < (f\ (x-h) - f\ x) / h$  by arith
    thus  $f\ x < f\ (x-h)$ 
  by (simp add: pos-less-divide-eq  $h$ )
  qed
qed
qed

```

**lemma** DERIV-pos-inc-left:

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
shows  $\text{DERIV } f \, x :> l \Longrightarrow 0 < l \Longrightarrow \exists d > 0. \forall h > 0. h < d \longrightarrow f(x - h) < f(x)$ 
apply (rule DERIV-neg-dec-left [of  $\%x. - f \, x \, x - l$ , simplified])
apply (auto simp add: DERIV-minus)
done

```

**lemma** *DERIV-neg-dec-right*:

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
shows  $\text{DERIV } f \, x :> l \Longrightarrow l < 0 \Longrightarrow \exists d > 0. \forall h > 0. h < d \longrightarrow f(x) > f(x + h)$ 
apply (rule DERIV-pos-inc-right [of  $\%x. - f \, x \, x - l$ , simplified])
apply (auto simp add: DERIV-minus)
done

```

**lemma** *DERIV-local-max*:

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
assumes  $\text{der}: \text{DERIV } f \, x :> l$ 
and  $d: 0 < d$ 
and  $le: \forall y. |x-y| < d \longrightarrow f(y) \leq f(x)$ 
shows  $l = 0$ 
proof (cases rule: linorder-cases [of  $l \, 0$ ])
case equal thus ?thesis .
next
case less
from DERIV-neg-dec-left [OF der less]
obtain  $d'$  where  $0 < d'$ 
and  $lt: \forall h > 0. h < d' \longrightarrow f \, x < f \, (x - h)$  by blast
from real-lbound-gt-zero [OF  $d \, d'$ ]
obtain  $e$  where  $0 < e \wedge e < d \wedge e < d' ..$ 
with  $lt \, le$  [THEN spec [where  $x=x-e$ ]]
show ?thesis by (auto simp add: abs-if)
next
case greater
from DERIV-pos-inc-right [OF der greater]
obtain  $d'$  where  $0 < d'$ 
and  $lt: \forall h > 0. h < d' \longrightarrow f \, x < f \, (x + h)$  by blast
from real-lbound-gt-zero [OF  $d \, d'$ ]
obtain  $e$  where  $0 < e \wedge e < d \wedge e < d' ..$ 
with  $lt \, le$  [THEN spec [where  $x=x+e$ ]]
show ?thesis by (auto simp add: abs-if)
qed

```

Similar theorem for a local minimum

**lemma** *DERIV-local-min*:

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
shows  $[\text{DERIV } f \, x :> l; 0 < d; \forall y. |x-y| < d \longrightarrow f(x) \leq f(y)] \Longrightarrow l = 0$ 
by (drule DERIV-minus [THEN DERIV-local-max], auto)

```



In particular, if a function is locally flat

**lemma** *DERIV-local-const*:

**fixes**  $f :: \text{real} \Rightarrow \text{real}$   
**shows**  $[| \text{DERIV } f \ x :> l; 0 < d; \forall y. |x-y| < d \longrightarrow f(x) = f(y) |] \Longrightarrow l = 0$   
**by** (*auto dest!*: *DERIV-local-max*)

### 73.7 Rolle’s Theorem

Lemma about introducing open ball in open interval

**lemma** *lemma-interval-lt*:

$[| a < x; x < b |]$   
 $\Longrightarrow \exists d::\text{real}. 0 < d \ \& \ (\forall y. |x-y| < d \longrightarrow a < y \ \& \ y < b)$

**apply** (*simp add: abs-less-iff*)  
**apply** (*insert linorder-linear [of x-a b-x], safe*)  
**apply** (*rule-tac x = x-a in exI*)  
**apply** (*rule-tac [2] x = b-x in exI, auto*)  
**done**

**lemma** *lemma-interval*:  $[| a < x; x < b |] \Longrightarrow$   
 $\exists d::\text{real}. 0 < d \ \& \ (\forall y. |x-y| < d \longrightarrow a \leq y \ \& \ y \leq b)$   
**apply** (*drule lemma-interval-lt, auto*)  
**apply** (*auto intro!: exI*)  
**done**

Rolle’s Theorem. If  $f$  is defined and continuous on the closed interval  $[a,b]$  and differentiable on the open interval  $(a,b)$ , and  $f a = f b$ , then there exists  $x0 \in (a,b)$  such that  $f' x0 = (0::'a)$

**theorem** *Rolle*:

**assumes** *lt*:  $a < b$   
**and** *eq*:  $f(a) = f(b)$   
**and** *con*:  $\forall x. a \leq x \ \& \ x \leq b \longrightarrow \text{isCont } f \ x$   
**and** *dif* [*rule-format*]:  $\forall x. a < x \ \& \ x < b \longrightarrow f \text{ differentiable } x$   
**shows**  $\exists z::\text{real}. a < z \ \& \ z < b \ \& \ \text{DERIV } f \ z :> 0$   
**proof** –  
**have** *le*:  $a \leq b$  **using** *lt* **by** *simp*  
**from** *isCont-eq-Ub* [*OF le con*]  
**obtain** *x* **where** *x-max*:  $\forall z. a \leq z \ \& \ z \leq b \longrightarrow f \ z \leq f \ x$   
**and** *alex*:  $a \leq x$  **and** *xleb*:  $x \leq b$   
**by** *blast*  
**from** *isCont-eq-Lb* [*OF le con*]  
**obtain** *x'* **where** *x'-min*:  $\forall z. a \leq z \ \& \ z \leq b \longrightarrow f \ x' \leq f \ z$   
**and** *alex'*:  $a \leq x'$  **and** *x'leb*:  $x' \leq b$   
**by** *blast*  
**show** *?thesis*  
**proof** *cases*  
**assume** *axb*:  $a < x \ \& \ x < b$

—  $f$  attains its maximum within the interval  
**hence**  $ax: a < x$  **and**  $xb: x < b$  **by** *arith* +  
**from** *lemma-interval* [*OF ax xb*]  
**obtain**  $d$  **where**  $d: 0 < d$  **and**  $bound: \forall y. |x - y| < d \longrightarrow a \leq y \wedge y \leq b$   
**by** *blast*  
**hence**  $bound': \forall y. |x - y| < d \longrightarrow f y \leq f x$  **using** *x-max*  
**by** *blast*  
**from** *differentiableD* [*OF dif* [*OF axb*]]  
**obtain**  $l$  **where**  $der: DERIV f x :> l$  ..  
**have**  $l=0$  **by** (*rule DERIV-local-max* [*OF der d bound'*])  
 — the derivative at a local maximum is zero  
**thus** *?thesis* **using**  $ax\ xb\ der$  **by** *auto*  
**next**  
**assume**  $notaxb: \sim (a < x \ \& \ x < b)$   
**hence**  $xeqab: x=a \mid x=b$  **using**  $alex\ xleb$  **by** *arith*  
**hence**  $fb\text{-}eq\text{-}fx: f\ b = f\ x$  **by** (*auto simp add: eq*)  
**show** *?thesis*  
**proof** *cases*  
**assume**  $ax'b: a < x' \ \& \ x' < b$   
 —  $f$  attains its minimum within the interval  
**hence**  $ax': a < x'$  **and**  $x'b: x' < b$  **by** *arith* +  
**from** *lemma-interval* [*OF ax' x'b*]  
**obtain**  $d$  **where**  $d: 0 < d$  **and**  $bound: \forall y. |x' - y| < d \longrightarrow a \leq y \wedge y \leq b$   
**by** *blast*  
**hence**  $bound': \forall y. |x' - y| < d \longrightarrow f\ x' \leq f\ y$  **using** *x'-min*  
**by** *blast*  
**from** *differentiableD* [*OF dif* [*OF ax'b*]]  
**obtain**  $l$  **where**  $der: DERIV f x' :> l$  ..  
**have**  $l=0$  **by** (*rule DERIV-local-min* [*OF der d bound'*])  
 — the derivative at a local minimum is zero  
**thus** *?thesis* **using**  $ax'\ x'b\ der$  **by** *auto*  
**next**  
**assume**  $notax'b: \sim (a < x' \ \& \ x' < b)$   
 —  $f$  is constant throughout the interval  
**hence**  $x'eqab: x'=a \mid x'=b$  **using**  $alex'\ x'leb$  **by** *arith*  
**hence**  $fb\text{-}eq\text{-}fx': f\ b = f\ x'$  **by** (*auto simp add: eq*)  
**from** *dense* [*OF lt*]  
**obtain**  $r$  **where**  $ar: a < r$  **and**  $rb: r < b$  **by** *blast*  
**from** *lemma-interval* [*OF ar rb*]  
**obtain**  $d$  **where**  $d: 0 < d$  **and**  $bound: \forall y. |r - y| < d \longrightarrow a \leq y \wedge y \leq b$   
**by** *blast*  
**have**  $eq\text{-}fb: \forall z. a \leq z \longrightarrow z \leq b \longrightarrow f\ z = f\ b$   
**proof** (*clarify*)  
**fix**  $z::real$   
**assume**  $az: a \leq z$  **and**  $zb: z \leq b$   
**show**  $f\ z = f\ b$   
**proof** (*rule order-antisym*)  
**show**  $f\ z \leq f\ b$  **by** (*simp add: fb-eq-fx x-max az zb*)  
**show**  $f\ b \leq f\ z$  **by** (*simp add: fb-eq-fx' x'-min az zb*)

```

    qed
  qed
  have bound':  $\forall y. |r-y| < d \longrightarrow f\ r = f\ y$ 
  proof (intro strip)
    fix y::real
    assume lt:  $|r-y| < d$ 
    hence  $f\ y = f\ b$  by (simp add: eq-fb bound)
    thus  $f\ r = f\ y$  by (simp add: eq-fb ar rb order-less-imp-le)
  qed
  from differentiableD [OF dif [OF conjI [OF ar rb]]]
  obtain l where der: DERIV f r :> l ..
  have l=0 by (rule DERIV-local-const [OF der d bound'])
    — the derivative of a constant function is zero
  thus ?thesis using ar rb der by auto
  qed
  qed
  qed

```

### 73.8 Mean Value Theorem

lemma lemma-MVT:

```

   $f\ a - (f\ b - f\ a)/(b-a) * a = f\ b - (f\ b - f\ a)/(b-a) * (b::real)$ 
  proof cases
    assume a=b thus ?thesis by simp
  next
    assume a≠b
    hence ba:  $b-a \neq 0$  by arith
    show ?thesis
      by (rule real-mult-left-cancel [OF ba, THEN iffD1],
          simp add: right-diff-distrib,
          simp add: left-diff-distrib)
  qed

```

theorem MVT:

```

  assumes lt:  $a < b$ 
    and con:  $\forall x. a \leq x \ \& \ x \leq b \longrightarrow \text{isCont } f\ x$ 
    and dif [rule-format]:  $\forall x. a < x \ \& \ x < b \longrightarrow f \text{ differentiable } x$ 
  shows  $\exists l\ z::real. a < z \ \& \ z < b \ \& \ \text{DERIV } f\ z :> l \ \& \ (f(b) - f(a) = (b-a) * l)$ 
  proof -
    let ?F =  $\lambda x. f\ x - ((f\ b - f\ a) / (b-a)) * x$ 
    have contF:  $\forall x. a \leq x \ \wedge \ x \leq b \longrightarrow \text{isCont } ?F\ x$  using con
      by (fast intro: isCont-diff isCont-const isCont-mult isCont-ident)
    have difF:  $\forall x. a < x \ \wedge \ x < b \longrightarrow ?F \text{ differentiable } x$ 
    proof (clarify)
      fix x::real
      assume ax:  $a < x$  and xb:  $x < b$ 
      from differentiableD [OF dif [OF conjI [OF ax xb]]]
      obtain l where der: DERIV f x :> l ..
    qed
  qed

```

```

show ?F differentiable x
  by (rule differentiableI [where D = l - (f b - f a)/(b-a)],
      blast intro: DERIV-diff DERIV-cmult-Id der)
qed
from Rolle [where f = ?F, OF lt lemma-MVT contF difF]
obtain z where az: a < z and zb: z < b and der: DERIV ?F z :> 0
  by blast
have DERIV (%x. ((f b - f a)/(b-a)) * x) z :> (f b - f a)/(b-a)
  by (rule DERIV-cmult-Id)
hence derF: DERIV (λx. ?F x + (f b - f a) / (b - a) * x) z
      :> 0 + (f b - f a) / (b - a)
  by (rule DERIV-add [OF der])
show ?thesis
proof (intro exI conjI)
  show a < z using az .
  show z < b using zb .
  show f b - f a = (b - a) * ((f b - f a)/(b-a)) by (simp)
  show DERIV f z :> ((f b - f a)/(b-a)) using derF by simp
qed
qed

```

**lemma** MVT2:

```

  [| a < b; ∀ x. a ≤ x & x ≤ b --> DERIV f x :> f'(x) |]
  ==> ∃ z::real. a < z & z < b & (f b - f a = (b - a) * f'(z))
apply (drule MVT)
apply (blast intro: DERIV-isCont)
apply (force dest: order-less-imp-le simp add: differentiable-def)
apply (blast dest: DERIV-unique order-less-imp-le)
done

```

A function is constant if its derivative is 0 over an interval.

**lemma** DERIV-isconst-end:

```

  fixes f :: real => real
  shows [| a < b;
    ∀ x. a ≤ x & x ≤ b --> isCont f x;
    ∀ x. a < x & x < b --> DERIV f x :> 0 |]
    ==> f b = f a
apply (drule MVT, assumption)
apply (blast intro: differentiableI)
apply (auto dest!: DERIV-unique simp add: diff-eq-eq)
done

```

**lemma** DERIV-isconst1:

```

  fixes f :: real => real
  shows [| a < b;
    ∀ x. a ≤ x & x ≤ b --> isCont f x;
    ∀ x. a < x & x < b --> DERIV f x :> 0 |]
    ==> ∀ x. a ≤ x & x ≤ b --> f x = f a
apply safe

```

```

apply (drule-tac  $x = a$  in order-le-imp-less-or-eq, safe)
apply (drule-tac  $b = x$  in DERIV-isconst-end, auto)
done

```

```

lemma DERIV-isconst2:
  fixes  $f :: \text{real} \Rightarrow \text{real}$ 
  shows  $\llbracket a < b;$ 
     $\forall x. a \leq x \ \& \ x \leq b \longrightarrow \text{isCont } f \ x;$ 
     $\forall x. a < x \ \& \ x < b \longrightarrow \text{DERIV } f \ x :> 0;$ 
     $a \leq x; x \leq b \rrbracket$ 
     $\implies f \ x = f \ a$ 
apply (blast dest: DERIV-isconst1)
done

```

```

lemma DERIV-isconst3: fixes  $a \ b \ x \ y :: \text{real}$ 
  assumes  $a < b$  and  $x \in \{a <..< b\}$  and  $y \in \{a <..< b\}$ 
  assumes derivable:  $\bigwedge x. x \in \{a <..< b\} \implies \text{DERIV } f \ x :> 0$ 
  shows  $f \ x = f \ y$ 
proof (cases  $x = y$ )
  case False
    let  $?a = \min \ x \ y$ 
    let  $?b = \max \ x \ y$ 

    have  $\forall z. ?a \leq z \wedge z \leq ?b \longrightarrow \text{DERIV } f \ z :> 0$ 
    proof (rule allI, rule impI)
      fix  $z :: \text{real}$  assume  $?a \leq z \wedge z \leq ?b$ 
      hence  $a < z$  and  $z < b$  using  $\langle x \in \{a <..< b\} \rangle$  and  $\langle y \in \{a <..< b\} \rangle$  by
        auto
      hence  $z \in \{a <..< b\}$  by auto
      thus  $\text{DERIV } f \ z :> 0$  by (rule derivable)
    qed
    hence isCont:  $\forall z. ?a \leq z \wedge z \leq ?b \longrightarrow \text{isCont } f \ z$ 
    and DERIV:  $\forall z. ?a < z \wedge z < ?b \longrightarrow \text{DERIV } f \ z :> 0$  using DERIV-isCont
  by auto

  have  $?a < ?b$  using  $\langle x \neq y \rangle$  by auto
  from DERIV-isconst2[OF this isCont DERIV, of  $x$ ] and DERIV-isconst2[OF
    this isCont DERIV, of  $y$ ]
  show ?thesis by auto
qed auto

```

```

lemma DERIV-isconst-all:
  fixes  $f :: \text{real} \Rightarrow \text{real}$ 
  shows  $\forall x. \text{DERIV } f \ x :> 0 \implies f(x) = f(y)$ 
apply (rule linorder-cases [of  $x \ y$ ])
apply (blast intro: sym DERIV-isCont DERIV-isconst-end)+
done

```

```

lemma DERIV-const-ratio-const:

```

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
shows  $[|a \neq b; \forall x. \text{DERIV } f \ x :> k|] \Rightarrow (f(b) - f(a)) = (b-a) * k$ 
apply (rule linorder-cases [of a b], auto)
apply (drule-tac [|]  $f = f$  in MVT)
apply (auto dest: DERIV-isCont DERIV-unique simp add: differentiable-def)
apply (auto dest: DERIV-unique simp add: ring-distrib diff-minus)
done

```

**lemma** DERIV-const-ratio-const2:

```

fixes  $f :: \text{real} \Rightarrow \text{real}$ 
shows  $[|a \neq b; \forall x. \text{DERIV } f \ x :> k|] \Rightarrow (f(b) - f(a))/(b-a) = k$ 
apply (rule-tac  $c1 = b-a$  in real-mult-right-cancel [THEN iffD1])
apply (auto dest!: DERIV-const-ratio-const simp add: mult-assoc)
done

```

**lemma** real-average-minus-first [simp]:  $((a + b) / 2 - a) = (b-a)/(2::\text{real})$   
**by** (simp)

**lemma** real-average-minus-second [simp]:  $((b + a) / 2 - a) = (b-a)/(2::\text{real})$   
**by** (simp)

Gallileo’s ”trick”: average velocity = av. of end velocities

**lemma** DERIV-const-average:

```

fixes  $v :: \text{real} \Rightarrow \text{real}$ 
assumes  $\text{neq: } a \neq (b::\text{real})$ 
and  $\text{der: } \forall x. \text{DERIV } v \ x :> k$ 
shows  $v ((a + b) / 2) = (v a + v b) / 2$ 
proof (cases rule: linorder-cases [of a b])
case equal with  $\text{neq}$  show ?thesis by simp
next
case less
have  $(v b - v a) / (b - a) = k$ 
by (rule DERIV-const-ratio-const2 [OF neq der])
hence  $(b-a) * ((v b - v a) / (b-a)) = (b-a) * k$  by simp
moreover have  $(v ((a + b) / 2) - v a) / ((a + b) / 2 - a) = k$ 
by (rule DERIV-const-ratio-const2 [OF - der], simp add: neq)
ultimately show ?thesis using  $\text{neq}$  by force
next
case greater
have  $(v b - v a) / (b - a) = k$ 
by (rule DERIV-const-ratio-const2 [OF neq der])
hence  $(b-a) * ((v b - v a) / (b-a)) = (b-a) * k$  by simp
moreover have  $(v ((b + a) / 2) - v a) / ((b + a) / 2 - a) = k$ 
by (rule DERIV-const-ratio-const2 [OF - der], simp add: neq)
ultimately show ?thesis using  $\text{neq}$  by (force simp add: add-commute)
qed

```

**lemma** DERIV-pos-imp-increasing:

```

fixes  $a::\text{real}$  and  $b::\text{real}$  and  $f::\text{real} \Rightarrow \text{real}$ 
assumes  $a < b$  and  $\forall x. a \leq x \ \& \ x \leq b \longrightarrow (EX \ y. \text{DERIV } f \ x :> y \ \& \ y >$ 
0)
shows  $f \ a < f \ b$ 
proof (rule ccontr)
assume  $\sim f \ a < f \ b$ 
have  $EX \ l \ z. a < z \ \& \ z < b \ \& \ \text{DERIV } f \ z :> l$ 
 $\ \& \ f \ b - f \ a = (b - a) * l$ 
apply (rule MVT)
using assms
apply auto
apply (metis DERIV-isCont)
apply (metis differentiableI less-le)
done
then obtain  $l \ z$  where  $a < z$  and  $z < b$  and  $\text{DERIV } f \ z :> l$ 
 $\ \text{and } f \ b - f \ a = (b - a) * l$ 
by auto

from prems have  $\sim(l > 0)$ 
by (metis linorder-not-le mult-le-0-iff diff-le-0-iff-le)
with prems show False
by (metis DERIV-unique less-le)
qed

```

**lemma** *DERIV-nonneg-imp-nonincreasing*:

```

fixes  $a::\text{real}$  and  $b::\text{real}$  and  $f::\text{real} \Rightarrow \text{real}$ 
assumes  $a \leq b$  and
 $\forall x. a \leq x \ \& \ x \leq b \longrightarrow (\exists y. \text{DERIV } f \ x :> y \ \& \ y \geq 0)$ 
shows  $f \ a \leq f \ b$ 
proof (rule ccontr, cases  $a = b$ )
assume  $\sim f \ a \leq f \ b$ 
assume  $a = b$ 
with prems show False by auto
next assume  $\sim f \ a \leq f \ b$ 
assume  $a \sim = b$ 
with assms have  $EX \ l \ z. a < z \ \& \ z < b \ \& \ \text{DERIV } f \ z :> l$ 
 $\ \& \ f \ b - f \ a = (b - a) * l$ 
apply –
apply (rule MVT)
apply auto
apply (metis DERIV-isCont)
apply (metis differentiableI less-le)
done
then obtain  $l \ z$  where  $a < z$  and  $z < b$  and  $\text{DERIV } f \ z :> l$ 
 $\ \text{and } f \ b - f \ a = (b - a) * l$ 
by auto
from prems have  $\sim(l >= 0)$ 
by (metis diff-self le-eqI le-iff-diff-le-0 order-antisym linear
split-mult-pos-le)

```

```

with prems show False
  by (metis DERIV-unique order-less-imp-le)
qed

```

```

lemma DERIV-neg-imp-decreasing:
  fixes a::real and b::real and f::real => real
  assumes a < b and
     $\forall x. a \leq x \ \& \ x \leq b \ \longrightarrow (\exists y. \text{DERIV } f \ x \ :\> \ y \ \& \ y < 0)$ 
  shows f a > f b
proof -
  have  $(\%x. -f \ x) \ a < (\%x. -f \ x) \ b$ 
  apply (rule DERIV-pos-imp-increasing [of a b %x. -f x])
  using assms
  apply auto
  apply (metis DERIV-minus neg-0-less-iff-less)
  done
  thus ?thesis
  by simp
qed

```

```

lemma DERIV-nonpos-imp-nonincreasing:
  fixes a::real and b::real and f::real => real
  assumes a ≤ b and
     $\forall x. a \leq x \ \& \ x \leq b \ \longrightarrow (\exists y. \text{DERIV } f \ x \ :\> \ y \ \& \ y \leq 0)$ 
  shows f a ≥ f b
proof -
  have  $(\%x. -f \ x) \ a \leq (\%x. -f \ x) \ b$ 
  apply (rule DERIV-nonneg-imp-nonincreasing [of a b %x. -f x])
  using assms
  apply auto
  apply (metis DERIV-minus neg-0-le-iff-le)
  done
  thus ?thesis
  by simp
qed

```

### 73.9 Continuous injective functions

Dull lemma: an continuous injection on an interval must have a strict maximum at an end point, not in the middle.

```

lemma lemma-isCont-inj:
  fixes f :: real ⇒ real
  assumes d: 0 < d
    and inj [rule-format]:  $\forall z. |z-x| \leq d \ \longrightarrow \ g(f \ z) = z$ 
    and cont:  $\forall z. |z-x| \leq d \ \longrightarrow \ \text{isCont } f \ z$ 
  shows  $\exists z. |z-x| \leq d \ \& \ f \ x < f \ z$ 
proof (rule ccontr)
  assume  $\sim (\exists z. |z-x| \leq d \ \& \ f \ x < f \ z)$ 
  hence all [rule-format]:  $\forall z. |z-x| \leq d \ \longrightarrow \ f \ z \leq f \ x$  by auto

```



```

show False
proof (cases rule: linorder-le-cases [of  $f(x-d)$   $f(x+d)$ ])
  case le
    from d cont all [of  $x+d$ ]
    have flef:  $f(x+d) \leq f\ x$ 
    and xlex:  $x - d \leq x$ 
    and cont':  $\forall z. x - d \leq z \wedge z \leq x \longrightarrow \text{isCont } f\ z$ 
    by (auto simp add: abs-if)
    from IVT [OF le flef xlex cont']
    obtain  $x'$  where  $x-d \leq x' \leq x$   $f\ x' = f(x+d)$  by blast
    moreover
    hence  $g(f\ x') = g(f(x+d))$  by simp
    ultimately show False using d inj [of  $x'$ ] inj [of  $x+d$ ]
    by (simp add: abs-le-iff)
  next
    case ge
    from d cont all [of  $x-d$ ]
    have flef:  $f(x-d) \leq f\ x$ 
    and xlex:  $x \leq x+d$ 
    and cont':  $\forall z. x \leq z \wedge z \leq x+d \longrightarrow \text{isCont } f\ z$ 
    by (auto simp add: abs-if)
    from IVT2 [OF ge flef xlex cont']
    obtain  $x'$  where  $x \leq x' \leq x+d$   $f\ x' = f(x-d)$  by blast
    moreover
    hence  $g(f\ x') = g(f(x-d))$  by simp
    ultimately show False using d inj [of  $x'$ ] inj [of  $x-d$ ]
    by (simp add: abs-le-iff)
  qed
qed

```

Similar version for lower bound.

```

lemma lemma-isCont-inj2:
  fixes  $f\ g :: \text{real} \Rightarrow \text{real}$ 
  shows [ $0 < d; \forall z. |z-x| \leq d \longrightarrow g(f\ z) = z;$ 
     $\forall z. |z-x| \leq d \longrightarrow \text{isCont } f\ z$ ]
     $\implies \exists z. |z-x| \leq d \ \& \ f\ z < f\ x$ 
apply (insert lemma-isCont-inj
  [where  $f = \%x. -\ f\ x$  and  $g = \%y. g(-y)$  and  $x = x$  and  $d = d$ ])
apply (simp add: isCont-minus linorder-not-le)
done

```

Show there's an interval surrounding  $f\ x$  in  $f[[x - d, x + d]]$ .

```

lemma isCont-inj-range:
  fixes  $f :: \text{real} \Rightarrow \text{real}$ 
  assumes  $d: 0 < d$ 
    and inj:  $\forall z. |z-x| \leq d \longrightarrow g(f\ z) = z$ 
    and cont:  $\forall z. |z-x| \leq d \longrightarrow \text{isCont } f\ z$ 
  shows  $\exists e > 0. \forall y. |y - f\ x| \leq e \longrightarrow (\exists z. |z-x| \leq d \ \& \ f\ z = y)$ 
proof -

```

```

have  $x-d \leq x+d \ \forall z. \ x-d \leq z \wedge z \leq x+d \longrightarrow isCont \ f \ z$  using cont d
  by (auto simp add: abs-le-iff)
from isCont-Lb-Ub [OF this]
obtain  $L \ M$ 
where all1 [rule-format]:  $\forall z. \ x-d \leq z \wedge z \leq x+d \longrightarrow L \leq f \ z \wedge f \ z \leq M$ 
  and all2 [rule-format]:
     $\forall y. \ L \leq y \wedge y \leq M \longrightarrow (\exists z. \ x-d \leq z \wedge z \leq x+d \wedge f \ z = y)$ 
  by auto
with  $d$  have  $L \leq f \ x \ \& \ f \ x \leq M$  by simp
moreover have  $L \neq f \ x$ 
proof –
  from lemma-isCont-inj2 [OF d inj cont]
  obtain  $u$  where  $|u - x| \leq d \wedge f \ u < f \ x$  by auto
  thus ?thesis using all1 [of u] by arith
qed
moreover have  $f \ x \neq M$ 
proof –
  from lemma-isCont-inj [OF d inj cont]
  obtain  $u$  where  $|u - x| \leq d \wedge f \ x < f \ u$  by auto
  thus ?thesis using all1 [of u] by arith
qed
ultimately have  $L < f \ x \ \& \ f \ x < M$  by arith
hence  $0 < f \ x - L \ \& \ 0 < M - f \ x$  by arith+
from real-lbound-gt-zero [OF this]
obtain  $e$  where  $e: 0 < e \wedge e < f \ x - L \ \& \ e < M - f \ x$  by auto
thus ?thesis
proof (intro exI conjI)
  show  $0 < e$  using  $e(1)$  .
  show  $\forall y. \ |y - f \ x| \leq e \longrightarrow (\exists z. \ |z - x| \leq d \wedge f \ z = y)$ 
  proof (intro strip)
    fix  $y::real$ 
    assume  $|y - f \ x| \leq e$ 
    with  $e$  have  $L \leq y \wedge y \leq M$  by arith
    from all2 [OF this]
    obtain  $z$  where  $x - d \leq z \wedge z \leq x + d \wedge f \ z = y$  by blast
    thus  $\exists z. \ |z - x| \leq d \wedge f \ z = y$ 
      by (force simp add: abs-le-iff)
  qed
qed
qed

```

Continuity of inverse function

**lemma** *isCont-inverse-function*:

**fixes**  $f \ g :: real \Rightarrow real$

**assumes**  $d: 0 < d$

**and** *inj*:  $\forall z. \ |z-x| \leq d \longrightarrow g(f \ z) = z$

**and** *cont*:  $\forall z. \ |z-x| \leq d \longrightarrow isCont \ f \ z$

**shows** *isCont*  $g \ (f \ x)$

**proof** (*simp add: isCont-iff LIM-eq*)

```

show  $\forall r. 0 < r \longrightarrow$ 
   $(\exists s > 0. \forall z. z \neq 0 \wedge |z| < s \longrightarrow |g(f\ x + z) - g(f\ x)| < r)$ 
proof (intro strip)
  fix  $r :: \text{real}$ 
  assume  $r: 0 < r$ 
  from real-lbound-gt-zero [OF  $r\ d$ ]
  obtain  $e$  where  $e: 0 < e$  and  $e\text{-lt}: e < r \wedge e < d$  by blast
  with inj cont
  have  $e\text{-simps}: \forall z. |z - x| \leq e \longrightarrow g\ (f\ z) = z$ 
     $\forall z. |z - x| \leq e \longrightarrow \text{isCont}\ f\ z$  by auto
  from isCont-inj-range [OF  $e\ \text{this}$ ]
  obtain  $e'$  where  $e': 0 < e'$ 
    and  $\text{all}: \forall y. |y - f\ x| \leq e' \longrightarrow (\exists z. |z - x| \leq e \wedge f\ z = y)$ 
    by blast
  show  $\exists s > 0. \forall z. z \neq 0 \wedge |z| < s \longrightarrow |g(f\ x + z) - g(f\ x)| < r$ 
proof (intro exI conjI)
  show  $0 < e'$  using  $e'$ .
  show  $\forall z. z \neq 0 \wedge |z| < e' \longrightarrow |g\ (f\ x + z) - g\ (f\ x)| < r$ 
proof (intro strip)
  fix  $z :: \text{real}$ 
  assume  $z: z \neq 0 \wedge |z| < e'$ 
  with  $e\ e\text{-lt}\ e\text{-simps}\ \text{all}$  [rule-format, of  $f\ x + z$ ]
  show  $|g\ (f\ x + z) - g\ (f\ x)| < r$  by force
  qed
qed
qed
qed

```

Derivative of inverse function

**lemma** *DERIV-inverse-function:*

```

fixes  $f\ g :: \text{real} \Rightarrow \text{real}$ 
assumes  $\text{der}: \text{DERIV}\ f\ (g\ x) :> D$ 
assumes  $\text{neq}: D \neq 0$ 
assumes  $a: a < x$  and  $b: x < b$ 
assumes  $\text{inj}: \forall y. a < y \wedge y < b \longrightarrow f\ (g\ y) = y$ 
assumes  $\text{cont}: \text{isCont}\ g\ x$ 
shows  $\text{DERIV}\ g\ x :> \text{inverse}\ D$ 
unfolding DERIV-iff2
proof (rule LIM-equal2)
  show  $0 < \min\ (x - a)\ (b - x)$ 
    using  $a\ b$  by arith
next
  fix  $y$ 
  assume  $\text{norm}\ (y - x) < \min\ (x - a)\ (b - x)$ 
  hence  $a < y$  and  $y < b$ 
    by (simp-all add: abs-less-iff)
  thus  $(g\ y - g\ x) / (y - x) =$ 
     $\text{inverse}\ ((f\ (g\ y) - x) / (g\ y - g\ x))$ 
    by (simp add: inj)

```

```

next
  have (λz. (f z - f (g x)) / (z - g x)) -- g x --> D
    by (rule der [unfolded DERIV-iff2])
  hence 1: (λz. (f z - x) / (z - g x)) -- g x --> D
    using inj a b by simp
  have 2: ∃ d > 0. ∀ y. y ≠ x ∧ norm (y - x) < d ⟶ g y ≠ g x
  proof (safe intro!: exI)
    show 0 < min (x - a) (b - x)
      using a b by simp
  next
    fix y
    assume norm (y - x) < min (x - a) (b - x)
    hence y: a < y y < b
      by (simp-all add: abs-less-iff)
    assume g y = g x
    hence f (g y) = f (g x) by simp
    hence y = x using inj y a b by simp
    also assume y ≠ x
    finally show False by simp
  qed
  have (λy. (f (g y) - x) / (g y - g x)) -- x --> D
    using cont 1 2 by (rule isCont-LIM-compose2)
  thus (λy. inverse ((f (g y) - x) / (g y - g x)))
    -- x --> inverse D
    using neq by (rule LIM-inverse)
qed

```

### 73.10 Generalized Mean Value Theorem

**theorem GMVT:**

```

fixes a b :: real
assumes alb: a < b
and fc: ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont f x
and fd: ∀ x. a < x ∧ x < b ⟶ f differentiable x
and gc: ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont g x
and gd: ∀ x. a < x ∧ x < b ⟶ g differentiable x
shows ∃ g'c f'c c. DERIV g c :> g'c ∧ DERIV f c :> f'c ∧ a < c ∧ c < b ∧
((f b - f a) * g'c) = ((g b - g a) * f'c)
proof -
  let ?h = λx. (f b - f a)*(g x) - (g b - g a)*(f x)
  from prems have a < b by simp
  moreover have ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont ?h x
  proof -
    have ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont (λx. f b - f a) x by simp
    with gc have ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont (λx. (f b - f a) * g x) x
      by (auto intro: isCont-mult)
    moreover
    have ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont (λx. g b - g a) x by simp
    with fc have ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont (λx. (g b - g a) * f x) x

```

```

    by (auto intro: isCont-mult)
    ultimately show ?thesis
    by (fastsimp intro: isCont-diff)
qed
moreover
have  $\forall x. a < x \wedge x < b \longrightarrow ?h \text{ differentiable } x$ 
proof -
  have  $\forall x. a < x \wedge x < b \longrightarrow (\lambda x. f \, b - f \, a) \text{ differentiable } x$  by simp
  with gd have  $\forall x. a < x \wedge x < b \longrightarrow (\lambda x. (f \, b - f \, a) * g \, x) \text{ differentiable } x$ 
by simp
  moreover
  have  $\forall x. a < x \wedge x < b \longrightarrow (\lambda x. g \, b - g \, a) \text{ differentiable } x$  by simp
  with fd have  $\forall x. a < x \wedge x < b \longrightarrow (\lambda x. (g \, b - g \, a) * f \, x) \text{ differentiable } x$ 
by simp
  ultimately show ?thesis by simp
qed
ultimately have  $\exists l \, z. a < z \wedge z < b \wedge \text{DERIV } ?h \, z :> l \wedge ?h \, b - ?h \, a = (b - a) * l$  by (rule MVT)
then obtain l where ldef:  $\exists z. a < z \wedge z < b \wedge \text{DERIV } ?h \, z :> l \wedge ?h \, b - ?h \, a = (b - a) * l$  ..
then obtain c where cdef:  $a < c \wedge c < b \wedge \text{DERIV } ?h \, c :> l \wedge ?h \, b - ?h \, a = (b - a) * l$  ..

from cdef have cint:  $a < c \wedge c < b$  by auto
with gd have g differentiable c by simp
hence  $\exists D. \text{DERIV } g \, c :> D$  by (rule differentiableD)
then obtain g'c where g'cdef:  $\text{DERIV } g \, c :> g'c$  ..

from cdef have a < c & c < b by auto
with fd have f differentiable c by simp
hence  $\exists D. \text{DERIV } f \, c :> D$  by (rule differentiableD)
then obtain f'c where f'cdef:  $\text{DERIV } f \, c :> f'c$  ..

from cdef have  $\text{DERIV } ?h \, c :> l$  by auto
moreover
{
  have  $\text{DERIV } (\lambda x. (f \, b - f \, a) * g \, x) \, c :> g'c * (f \, b - f \, a)$ 
  apply (insert DERIV-const [where k=f b - f a])
  apply (drule meta-spec [of - c])
  apply (drule DERIV-mult [OF - g'cdef])
  by simp
  moreover have  $\text{DERIV } (\lambda x. (g \, b - g \, a) * f \, x) \, c :> f'c * (g \, b - g \, a)$ 
  apply (insert DERIV-const [where k=g b - g a])
  apply (drule meta-spec [of - c])
  apply (drule DERIV-mult [OF - f'cdef])
  by simp
  ultimately have  $\text{DERIV } ?h \, c :> g'c * (f \, b - f \, a) - f'c * (g \, b - g \, a)$ 
  by (simp add: DERIV-diff)
}

```

**ultimately have**  $leq$ :  $l = g'c * (f b - f a) - f'c * (g b - g a)$  **by** (rule *DERIV-unique*)

```
{
  from cdef have ?h b - ?h a = (b - a) * l by auto
  also with leq have ... = (b - a) * (g'c * (f b - f a) - f'c * (g b - g a)) by
simp
  finally have ?h b - ?h a = (b - a) * (g'c * (f b - f a) - f'c * (g b - g a))
by simp
}
moreover
{
  have ?h b - ?h a =
    ((f b)*(g b) - (f a)*(g b) - (g b)*(f b) + (g a)*(f b)) -
    ((f b)*(g a) - (f a)*(g a) - (g b)*(f a) + (g a)*(f a))
    by (simp add: algebra-simps)
  hence ?h b - ?h a = 0 by auto
}
ultimately have (b - a) * (g'c * (f b - f a) - f'c * (g b - g a)) = 0 by auto
with alb have g'c * (f b - f a) - f'c * (g b - g a) = 0 by simp
hence g'c * (f b - f a) = f'c * (g b - g a) by simp
hence (f b - f a) * g'c = (g b - g a) * f'c by (simp add: mult-ac)
```

**with**  $g'cdef f'cdef cint$  **show**  $?thesis$  **by** auto  
qed

### 73.11 Theorems about Limits

**lemma** *isCont-inv-fun*:

```
fixes f g :: real  $\Rightarrow$  real
shows [| 0 < d;  $\forall z. |z - x| \leq d \longrightarrow g(f(z)) = z$ ;
 $\forall z. |z - x| \leq d \longrightarrow isCont f z$  |]
 $\implies isCont g (f x)$ 
by (rule isCont-inverse-function)
```

**lemma** *isCont-inv-fun-inv*:

```
fixes f g :: real  $\Rightarrow$  real
shows [| 0 < d;
 $\forall z. |z - x| \leq d \longrightarrow g(f(z)) = z$ ;
 $\forall z. |z - x| \leq d \longrightarrow isCont f z$  |]
 $\implies \exists e. 0 < e \ \& \$ 
 $(\forall y. 0 < |y - f(x)| \ \& \ |y - f(x)| < e \longrightarrow f(g(y)) = y)$ 
apply (drule isCont-inj-range)
prefer 2 apply (assumption, assumption, auto)
apply (rule-tac x = e in exI, auto)
apply (rotate-tac 2)
apply (drule-tac x = y in spec, auto)
done
```

**lemma** *LIM-fun-gt-zero*:

```

  [| f -- c --> (l::real); 0 < l |]
    ==> ∃ r. 0 < r & (∀ x::real. x ≠ c & |c - x| < r --> 0 < f x)
apply (auto simp add: LIM-eq)
apply (drule-tac x = l/2 in spec, safe, force)
apply (rule-tac x = s in exI)
apply (auto simp only: abs-less-iff)
done

```

**lemma** *LIM-fun-less-zero*:

```

  [| f -- c --> (l::real); l < 0 |]
    ==> ∃ r. 0 < r & (∀ x::real. x ≠ c & |c - x| < r --> f x < 0)
apply (auto simp add: LIM-eq)
apply (drule-tac x = -l/2 in spec, safe, force)
apply (rule-tac x = s in exI)
apply (auto simp only: abs-less-iff)
done

```

**lemma** *LIM-fun-not-zero*:

```

  [| f -- c --> (l::real); l ≠ 0 |]
    ==> ∃ r. 0 < r & (∀ x::real. x ≠ c & |c - x| < r --> f x ≠ 0)
apply (cut-tac x = l and y = 0 in linorder-less-linear, auto)
apply (drule LIM-fun-less-zero)
apply (drule-tac [3] LIM-fun-gt-zero)
apply force+
done

```

**end**

## 74 Series: Finite Summation and Infinite Series

**theory** *Series*

**imports** *SEQ Deriv*

**begin**

**definition**

```

  sums :: (nat ⇒ 'a::real-normed-vector) ⇒ 'a ⇒ bool
    (infixr sums 80) where
    f sums s = (%n. setsum f {0.. $n$ }) ----> s

```

**definition**

```

  summable :: (nat ⇒ 'a::real-normed-vector) ⇒ bool where
    summable f = (∃ s. f sums s)

```

**definition**

```

  suminf :: (nat ⇒ 'a::real-normed-vector) ⇒ 'a where
    suminf f = (THE s. f sums s)

```

**syntax**

$\text{-suminf} :: \text{idt} \Rightarrow 'a \Rightarrow 'a \ (\sum \cdot. - [0, 10] \ 10)$

**translations**

$\sum i. b == \text{CONST suminf } (\%i. b)$

**lemma**  $[\text{trans}]: f=g ==> g \text{ sums } z ==> f \text{ sums } z$   
**by** *simp*

**lemma** *sumr-diff-mult-const*:

$\text{setsum } f \ \{0..<n\} - (\text{real } n * r) = \text{setsum } (\%i. f \ i - r) \ \{0..<n::\text{nat}\}$   
**by** (*simp add: diff-minus setsum-addf real-of-nat-def*)

**lemma** *real-setsum-nat-ivl-bounded*:

$(!!p. p < n \implies f(p) \leq K)$   
 $\implies \text{setsum } f \ \{0..<n::\text{nat}\} \leq \text{real } n * K$   
**using** *setsum-bounded* [**where**  $A = \{0..<n\}$ ]  
**by** (*auto simp: real-of-nat-def*)

**lemma** *sumr-minus-one-realpov-zero* [*simp*]:

$(\sum i=0..<2*n. (-1) ^ \text{Suc } i) = (0::\text{real})$   
**by** (*induct n, auto*)

**lemma** *sumr-one-lb-realpov-zero* [*simp*]:

$(\sum n=\text{Suc } 0..<n. f(n) * (0::\text{real}) ^ n) = 0$   
**by** (*rule setsum-0', simp*)

**lemma** *sumr-group*:

$(\sum m=0..<n::\text{nat}. \text{setsum } f \ \{m * k ..< m*k + k\}) = \text{setsum } f \ \{0 ..< n * k\}$   
**apply** (*subgoal-tac k = 0 | 0 < k, auto*)  
**apply** (*induct n*)  
**apply** (*simp-all add: setsum-add-nat-ivl add-commute*)  
**done**

**lemma** *sumr-offset3*:

$\text{setsum } f \ \{0::\text{nat}..<n+k\} = (\sum m=0..<n. f(m+k)) + \text{setsum } f \ \{0..<k\}$   
**apply** (*subst setsum-shift-bounds-nat-ivl [symmetric]*)  
**apply** (*simp add: setsum-add-nat-ivl add-commute*)  
**done**

**lemma** *sumr-offset*:

**fixes**  $f :: \text{nat} \Rightarrow 'a::\text{ab-group-add}$   
**shows**  $(\sum m=0..<n. f(m+k)) = \text{setsum } f \ \{0..<n+k\} - \text{setsum } f \ \{0..<k\}$   
**by** (*simp add: sumr-offset3*)

**lemma** *sumr-offset2*:



$\forall f. (\sum m=0..<n::nat. f(m+k)::real) = \text{setsum } f \{0..<n+k\} - \text{setsum } f \{0..<k\}$   
**by** (*simp add: sumr-offset*)

**lemma** *sumr-offset4*:

$\forall n f. \text{setsum } f \{0::nat..<n+k\} = (\sum m=0..<n. f(m+k)::real) + \text{setsum } f \{0..<k\}$   
**by** (*clarify, rule sumr-offset3*)

## 74.1 Infinite Sums, by the Properties of Limits

**lemma** *sums-summable*:  $f \text{ sums } l \implies \text{summable } f$   
**by** (*simp add: sums-def summable-def, blast*)

**lemma** *summable-sums*:  $\text{summable } f \implies f \text{ sums } (\text{suminf } f)$   
**apply** (*simp add: summable-def suminf-def sums-def*)  
**apply** (*fast intro: theI LIMSEQ-unique*)  
**done**

**lemma** *summable-sumr-LIMSEQ-suminf*:

$\text{summable } f \implies (\%n. \text{setsum } f \{0..<n\}) \text{ ----> } (\text{suminf } f)$   
**by** (*rule summable-sums [unfolded sums-def]*)

**lemma** *suminf-eq-lim*:  $\text{suminf } f = \lim (\%n. \text{setsum } f \{0..<n\})$   
**by** (*simp add: suminf-def sums-def lim-def*)

**lemma** *sums-unique*:  $f \text{ sums } s \implies (s = \text{suminf } f)$   
**apply** (*frule sums-summable [THEN summable-sums]*)  
**apply** (*auto intro!: LIMSEQ-unique simp add: sums-def*)  
**done**

**lemma** *sums-iff*:  $f \text{ sums } x \longleftrightarrow \text{summable } f \wedge (\text{suminf } f = x)$   
**by** (*metis summable-sums sums-summable sums-unique*)

**lemma** *sums-split-initial-segment*:  $f \text{ sums } s \implies$   
 $(\%n. f(n+k)) \text{ sums } (s - (\text{SUM } i = 0..<k. f i))$   
**apply** (*unfold sums-def*)  
**apply** (*simp add: sumr-offset*)  
**apply** (*rule LIMSEQ-diff-const*)  
**apply** (*rule LIMSEQ-ignore-initial-segment*)  
**apply** *assumption*  
**done**

**lemma** *summable-ignore-initial-segment*:  $\text{summable } f \implies$   
 $\text{summable } (\%n. f(n+k))$   
**apply** (*unfold summable-def*)  
**apply** (*auto intro: sums-split-initial-segment*)  
**done**

**lemma** *suminf-minus-initial-segment*: *summable*  $f \implies$   
 $\text{suminf } f = s \implies \text{suminf } (\%n. f(n + k)) = s - (\text{SUM } i = 0..< k. f i)$   
**apply** (*frule* *summable-ignore-initial-segment*)  
**apply** (*rule* *sums-unique* [*THEN sym*])  
**apply** (*frule* *summable-sums*)  
**apply** (*rule* *sums-split-initial-segment*)  
**apply** *auto*  
**done**

**lemma** *suminf-split-initial-segment*: *summable*  $f \implies$   
 $\text{suminf } f = (\text{SUM } i = 0..< k. f i) + \text{suminf } (\%n. f(n + k))$   
**by** (*auto simp add: suminf-minus-initial-segment*)

**lemma** *suminf-exist-split*: **fixes**  $r :: \text{real}$  **assumes**  $0 < r$  **and** *summable*  $a$   
**shows**  $\exists N. \forall n \geq N. |\sum i. a(i + n)| < r$   
**proof** –  
**from** *LIMSEQ-D*[*OF* *summable-sumr-LIMSEQ-suminf*[*OF*  $\langle \text{summable } a \rangle$ ]  $\langle 0 < r \rangle$ ]  
**obtain**  $N :: \text{nat}$  **where**  $\forall n \geq N. \text{norm } (\text{setsum } a \{0..<n\} - \text{suminf } a) < r$   
**by** *auto*  
**thus** *?thesis* **unfolding** *suminf-minus-initial-segment*[*OF*  $\langle \text{summable } a \rangle$ ] *refl*]  
*abs-minus-commute real-norm-def*  
**by** *auto*  
**qed**

**lemma** *sums-Suc*: **assumes** *sumSuc*:  $(\lambda n. f(\text{Suc } n)) \text{ sums } l$  **shows**  $f \text{ sums } (l + f 0)$   
**proof** –  
**from** *sumSuc*[*unfolded* *sums-def*]  
**have**  $(\lambda i. \sum n = \text{Suc } 0..<\text{Suc } i. f n) \dashrightarrow l$  **unfolding** *setsum-reindex*[*OF* *inj-Suc*] *image-Suc-atLeastLessThan*[*symmetric*] *comp-def* .  
**from** *LIMSEQ-add-const*[*OF* *this*, **where**  $b=f 0$ ]  
**have**  $(\lambda i. \sum n = 0..<\text{Suc } i. f n) \dashrightarrow l + f 0$  **unfolding** *add-commute* *setsum-head-upt-Suc*[*OF* *zero-less-Suc*] .  
**thus** *?thesis* **unfolding** *sums-def* **by** (*rule* *LIMSEQ-imp-Suc*)  
**qed**

**lemma** *series-zero*:  
 $(\forall m. n \leq m \dashrightarrow f(m) = 0) \implies f \text{ sums } (\text{setsum } f \{0..<n\})$   
**apply** (*simp add: sums-def LIMSEQ-iff diff-minus*[*symmetric*], *safe*)  
**apply** (*rule-tac*  $x = n$  **in** *exI*)  
**apply** (*clarsimp simp add: setsum-diff*[*symmetric*] *cong: setsum-ivl-cong*)  
**done**

**lemma** *sums-zero*:  $(\lambda n. 0) \text{ sums } 0$   
**unfolding** *sums-def* **by** (*simp add: LIMSEQ-const*)

**lemma** *summable-zero*: *summable*  $(\lambda n. 0)$   
**by** (*rule* *sums-zero* [*THEN* *sums-summable*])

**lemma** *suminf-zero*:  $\text{suminf } (\lambda n. 0) = 0$   
**by** (rule *sums-zero* [THEN *sums-unique*, *symmetric*])

**lemma** (in *bounded-linear*) *sums*:  
 $(\lambda n. X\ n) \text{ sums } a \implies (\lambda n. f\ (X\ n)) \text{ sums } (f\ a)$   
**unfolding** *sums-def* **by** (drule *LIMSEQ*, *simp only: setsum*)

**lemma** (in *bounded-linear*) *summable*:  
 $\text{summable } (\lambda n. X\ n) \implies \text{summable } (\lambda n. f\ (X\ n))$   
**unfolding** *summable-def* **by** (auto intro: *sums*)

**lemma** (in *bounded-linear*) *suminf*:  
 $\text{summable } (\lambda n. X\ n) \implies f\ (\sum n. X\ n) = (\sum n. f\ (X\ n))$   
**by** (intro *sums-unique sums summable-sums*)

**lemma** *sums-mult*:  
**fixes**  $c :: 'a::\text{real-normed-algebra}$   
**shows**  $f \text{ sums } a \implies (\lambda n. c * f\ n) \text{ sums } (c * a)$   
**by** (rule *mult-right.sums*)

**lemma** *summable-mult*:  
**fixes**  $c :: 'a::\text{real-normed-algebra}$   
**shows**  $\text{summable } f \implies \text{summable } (\%n. c * f\ n)$   
**by** (rule *mult-right.summable*)

**lemma** *suminf-mult*:  
**fixes**  $c :: 'a::\text{real-normed-algebra}$   
**shows**  $\text{summable } f \implies \text{suminf } (\lambda n. c * f\ n) = c * \text{suminf } f$   
**by** (rule *mult-right.suminf [symmetric]*)

**lemma** *sums-mult2*:  
**fixes**  $c :: 'a::\text{real-normed-algebra}$   
**shows**  $f \text{ sums } a \implies (\lambda n. f\ n * c) \text{ sums } (a * c)$   
**by** (rule *mult-left.sums*)

**lemma** *summable-mult2*:  
**fixes**  $c :: 'a::\text{real-normed-algebra}$   
**shows**  $\text{summable } f \implies \text{summable } (\lambda n. f\ n * c)$   
**by** (rule *mult-left.summable*)

**lemma** *suminf-mult2*:  
**fixes**  $c :: 'a::\text{real-normed-algebra}$   
**shows**  $\text{summable } f \implies \text{suminf } f * c = (\sum n. f\ n * c)$   
**by** (rule *mult-left.suminf*)

**lemma** *sums-divide*:  
**fixes**  $c :: 'a::\text{real-normed-field}$   
**shows**  $f \text{ sums } a \implies (\lambda n. f\ n / c) \text{ sums } (a / c)$

**by** (*rule divide.sums*)

**lemma** *summable-divide*:

**fixes**  $c :: 'a::\text{real-normed-field}$

**shows**  $\text{summable } f \implies \text{summable } (\lambda n. f\ n / c)$

**by** (*rule divide.summable*)

**lemma** *suminf-divide*:

**fixes**  $c :: 'a::\text{real-normed-field}$

**shows**  $\text{summable } f \implies \text{suminf } (\lambda n. f\ n / c) = \text{suminf } f / c$

**by** (*rule divide.suminf [symmetric]*)

**lemma** *sums-add*:  $\llbracket X \text{ sums } a; Y \text{ sums } b \rrbracket \implies (\lambda n. X\ n + Y\ n) \text{ sums } (a + b)$

**unfolding** *sums-def* **by** (*simp add: setsum-addf LIMSEQ-add*)

**lemma** *summable-add*:  $\llbracket \text{summable } X; \text{summable } Y \rrbracket \implies \text{summable } (\lambda n. X\ n + Y\ n)$

**unfolding** *summable-def* **by** (*auto intro: sums-add*)

**lemma** *suminf-add*:

$\llbracket \text{summable } X; \text{summable } Y \rrbracket \implies \text{suminf } X + \text{suminf } Y = (\sum n. X\ n + Y\ n)$

**by** (*intro sums-unique sums-add summable-sums*)

**lemma** *sums-diff*:  $\llbracket X \text{ sums } a; Y \text{ sums } b \rrbracket \implies (\lambda n. X\ n - Y\ n) \text{ sums } (a - b)$

**unfolding** *sums-def* **by** (*simp add: setsum-subtractf LIMSEQ-diff*)

**lemma** *summable-diff*:  $\llbracket \text{summable } X; \text{summable } Y \rrbracket \implies \text{summable } (\lambda n. X\ n - Y\ n)$

**unfolding** *summable-def* **by** (*auto intro: sums-diff*)

**lemma** *suminf-diff*:

$\llbracket \text{summable } X; \text{summable } Y \rrbracket \implies \text{suminf } X - \text{suminf } Y = (\sum n. X\ n - Y\ n)$

**by** (*intro sums-unique sums-diff summable-sums*)

**lemma** *sums-minus*:  $X \text{ sums } a \implies (\lambda n. - X\ n) \text{ sums } (- a)$

**unfolding** *sums-def* **by** (*simp add: setsum-negf LIMSEQ-minus*)

**lemma** *summable-minus*:  $\text{summable } X \implies \text{summable } (\lambda n. - X\ n)$

**unfolding** *summable-def* **by** (*auto intro: sums-minus*)

**lemma** *suminf-minus*:  $\text{summable } X \implies (\sum n. - X\ n) = - (\sum n. X\ n)$

**by** (*intro sums-unique [symmetric] sums-minus summable-sums*)

**lemma** *sums-group*:

$\llbracket \text{summable } f; 0 < k \rrbracket \implies (\%n. \text{setsum } f \{n*k..<n*k+k\}) \text{ sums } (\text{suminf } f)$

**apply** (*drule summable-sums*)

**apply** (*simp only: sums-def sumr-group*)

**apply** (*unfold LIMSEQ-iff, safe*)

**apply** (*drule-tac x=r in spec, safe*)

```

apply (rule-tac x=no in exI, safe)
apply (drule-tac x=n*k in spec)
apply (erule mp)
apply (erule order-trans)
apply simp
done

```

A summable series of positive terms has limit that is at least as great as any partial sum.

```

lemma pos-summable:
  fixes f :: nat  $\Rightarrow$  real
  assumes pos:  $\forall n. 0 \leq f\ n$  and le:  $\forall n. \text{setsum } f \ \{0..<n\} \leq x$ 
  shows summable f
proof –
  have convergent ( $\lambda n. \text{setsum } f \ \{0..<n\}$ )
  proof (rule Bseq-mono-convergent)
    show Bseq ( $\lambda n. \text{setsum } f \ \{0..<n\}$ )
    by (rule f-inc-g-dec-Beq-f [of ( $\lambda n. \text{setsum } f \ \{0..<n\}$ )  $\lambda n. x$ ])
      (auto simp add: le pos)
  next
    show  $\forall m\ n. m \leq n \longrightarrow \text{setsum } f \ \{0..<m\} \leq \text{setsum } f \ \{0..<n\}$ 
    by (auto intro: setsum-mono2 pos)
  qed
  then obtain L where ( $\forall n. \text{setsum } f \ \{0..<n\} \dashrightarrow L$ )
  by (blast dest: convergentD)
  thus ?thesis
  by (force simp add: summable-def sums-def)
qed

```

```

lemma series-pos-le:
  fixes f :: nat  $\Rightarrow$  real
  shows  $\llbracket \text{summable } f; \forall m \geq n. 0 \leq f\ m \rrbracket \Longrightarrow \text{setsum } f \ \{0..<n\} \leq \text{suminf } f$ 
apply (drule summable-sums)
apply (simp add: sums-def)
apply (cut-tac k = setsum f {0..<n} in LIMSEQ-const)
apply (erule LIMSEQ-le, blast)
apply (rule-tac x=n in exI, clarify)
apply (rule setsum-mono2)
apply auto
done

```

```

lemma series-pos-less:
  fixes f :: nat  $\Rightarrow$  real
  shows  $\llbracket \text{summable } f; \forall m \geq n. 0 < f\ m \rrbracket \Longrightarrow \text{setsum } f \ \{0..<n\} < \text{suminf } f$ 
apply (rule-tac y=setsum f {0..<Suc n} in order-less-le-trans)
apply simp
apply (erule series-pos-le)
apply (simp add: order-less-imp-le)
done

```

**lemma** *suminf-gt-zero*:

**fixes**  $f :: \text{nat} \Rightarrow \text{real}$

**shows**  $\llbracket \text{summable } f; \forall n. 0 < f\ n \rrbracket \Longrightarrow 0 < \text{suminf } f$

**by** (*drule-tac*  $n=0$  **in** *series-pos-less*, *simp-all*)

**lemma** *suminf-ge-zero*:

**fixes**  $f :: \text{nat} \Rightarrow \text{real}$

**shows**  $\llbracket \text{summable } f; \forall n. 0 \leq f\ n \rrbracket \Longrightarrow 0 \leq \text{suminf } f$

**by** (*drule-tac*  $n=0$  **in** *series-pos-le*, *simp-all*)

**lemma** *sumr-pos-lt-pair*:

**fixes**  $f :: \text{nat} \Rightarrow \text{real}$

**shows**  $\llbracket \text{summable } f;$

$\forall d. 0 < f\ (k + (\text{Suc}(\text{Suc } 0) * d)) + f\ (k + ((\text{Suc}(\text{Suc } 0) * d) + 1)) \rrbracket$

$\Longrightarrow \text{setsum } f\ \{0..<k\} < \text{suminf } f$

**unfolding** *One-nat-def*

**apply** (*subst* *suminf-split-initial-segment* [**where**  $k=k$ ])

**apply** *assumption*

**apply** *simp*

**apply** (*drule-tac*  $k=k$  **in** *summable-ignore-initial-segment*)

**apply** (*drule-tac*  $k=\text{Suc } (\text{Suc } 0)$  **in** *sums-group*, *simp*)

**apply** *simp*

**apply** (*frule* *sums-unique*)

**apply** (*drule* *sums-summable*)

**apply** *simp*

**apply** (*erule* *suminf-gt-zero*)

**apply** (*simp* *add*: *add-ac*)

**done**

Sum of a geometric progression.

**lemmas** *sumr-geometric* = *geometric-sum* [**where**  $'a = \text{real}$ ]

**lemma** *geometric-sums*:

**fixes**  $x :: 'a::\{\text{real-normed-field}\}$

**shows**  $\text{norm } x < 1 \Longrightarrow (\lambda n. x ^ n) \text{ sums } (1 / (1 - x))$

**proof** –

**assume** *less-1*:  $\text{norm } x < 1$

**hence** *neq-1*:  $x \neq 1$  **by** *auto*

**hence** *neq-0*:  $x - 1 \neq 0$  **by** *simp*

**from** *less-1* **have** *lim-0*:  $(\lambda n. x ^ n) \text{ ----} > 0$

**by** (*rule* *LIMSEQ-power-zero*)

**hence**  $(\lambda n. x ^ n / (x - 1) - 1 / (x - 1)) \text{ ----} > 0 / (x - 1) - 1 / (x - 1)$

**using** *neq-0* **by** (*intro* *LIMSEQ-divide* *LIMSEQ-diff* *LIMSEQ-const*)

**hence**  $(\lambda n. (x ^ n - 1) / (x - 1)) \text{ ----} > 1 / (1 - x)$

**by** (*simp* *add*: *nonzero-minus-divide-right* [*OF* *neq-0*] *diff-divide-distrib*)

**thus**  $(\lambda n. x ^ n) \text{ sums } (1 / (1 - x))$

**by** (*simp* *add*: *sums-def* *geometric-sum* *neq-1*)

qed

**lemma** *summable-geometric*:

fixes  $x :: 'a::\{\text{real-normed-field}\}$   
 shows  $\text{norm } x < 1 \implies \text{summable } (\lambda n. x ^ n)$   
 by (rule *geometric-sums* [THEN *sums-summable*])

**lemma** *half*:  $0 < 1 / (2::'a::\{\text{number-ring, linordered-field-inverse-zero}\})$   
 by *arith*

**lemma** *power-half-series*:  $(\lambda n. (1/2::\text{real}) ^ \text{Suc } n) \text{ sums } 1$

**proof** –

have  $2: (\lambda n. (1/2::\text{real}) ^ n) \text{ sums } 2$  **using** *geometric-sums* [of  $1/2::\text{real}$ ]  
 by *auto*  
 have  $(\lambda n. (1/2::\text{real}) ^ \text{Suc } n) = (\lambda n. (1 / 2) ^ n / 2)$   
 by *simp*  
 thus ?thesis **using** *divide.sums* [OF 2, of 2]  
 by *simp*

qed

Cauchy-type criterion for convergence of series (c.f. Harrison)

**lemma** *summable-convergent-sumr-iff*:

$\text{summable } f = \text{convergent } (\%n. \text{setsum } f \{0..<n\})$   
 by (*simp add: summable-def sums-def convergent-def*)

**lemma** *summable-LIMSEQ-zero*:  $\text{summable } f \implies f \text{ ----} > 0$

**apply** (*drule summable-convergent-sumr-iff* [THEN *iffD1*])  
**apply** (*drule convergent-Cauchy*)  
**apply** (*simp only: Cauchy-iff LIMSEQ-iff, safe*)  
**apply** (*drule-tac x=r in spec, safe*)  
**apply** (*rule-tac x=M in exI, safe*)  
**apply** (*drule-tac x=Suc n in spec, simp*)  
**apply** (*drule-tac x=n in spec, simp*)  
**done**

**lemma** *suminf-le*:

fixes  $x :: \text{real}$   
 shows  $\text{summable } f \implies (!n. \text{setsum } f \{0..<n\} \leq x) \implies \text{suminf } f \leq x$   
 by (*simp add: summable-convergent-sumr-iff suminf-eq-lim lim-le*)

**lemma** *summable-Cauchy*:

$\text{summable } (f::\text{nat} \Rightarrow 'a::\text{banach}) =$   
 $(\forall e > 0. \exists N. \forall m \geq N. \forall n. \text{norm } (\text{setsum } f \{m..<n\}) < e)$   
**apply** (*simp only: summable-convergent-sumr-iff Cauchy-convergent-iff* [*symmetric*]  
*Cauchy-iff, safe*)  
**apply** (*drule spec, drule (1) mp*)  
**apply** (*erule exE, rule-tac x=M in exI, clarify*)  
**apply** (*rule-tac x=m and y=n in linorder-le-cases*)  
**apply** (*frule (1) order-trans*)

```

apply (drule-tac x=n in spec, drule (1) mp)
apply (drule-tac x=m in spec, drule (1) mp)
apply (simp add: setsum-diff [symmetric])
apply simp
apply (drule spec, drule (1) mp)
apply (erule exE, rule-tac x=N in exI, clarify)
apply (rule-tac x=m and y=n in linorder-le-cases)
apply (subst norm-minus-commute)
apply (simp add: setsum-diff [symmetric])
apply (simp add: setsum-diff [symmetric])
done

```

Comparison test

```

lemma norm-setsum:
  fixes f :: 'a  $\Rightarrow$  'b::real-normed-vector
  shows norm (setsum f A)  $\leq$  ( $\sum i \in A.$  norm (f i))
apply (case-tac finite A)
apply (erule finite-induct)
apply simp
apply simp
apply (erule order-trans [OF norm-triangle-ineq add-left-mono])
apply simp
done

```

```

lemma summable-comparison-test:
  fixes f :: nat  $\Rightarrow$  'a::banach
  shows  $\llbracket \exists N. \forall n \geq N. \text{norm } (f\ n) \leq g\ n; \text{summable } g \rrbracket \implies \text{summable } f$ 
apply (simp add: summable-Cauchy, safe)
apply (drule-tac x=e in spec, safe)
apply (rule-tac x = N + Na in exI, safe)
apply (rotate-tac 2)
apply (drule-tac x = m in spec)
apply (auto, rotate-tac 2, drule-tac x = n in spec)
apply (rule-tac y =  $\sum k=m..<n. \text{norm } (f\ k)$  in order-le-less-trans)
apply (rule norm-setsum)
apply (rule-tac y = setsum g {m.. $<n$ } in order-le-less-trans)
apply (auto intro: setsum-mono simp add: abs-less-iff)
done

```

```

lemma summable-norm-comparison-test:
  fixes f :: nat  $\Rightarrow$  'a::banach
  shows  $\llbracket \exists N. \forall n \geq N. \text{norm } (f\ n) \leq g\ n; \text{summable } g \rrbracket$ 
     $\implies \text{summable } (\lambda n. \text{norm } (f\ n))$ 
apply (rule summable-comparison-test)
apply (auto)
done

```

```

lemma summable-rabs-comparison-test:
  fixes f :: nat  $\Rightarrow$  real

```



```

  shows  $\llbracket \exists N. \forall n \geq N. |f\ n| \leq g\ n; \text{summable } g \rrbracket \implies \text{summable } (\lambda n. |f\ n|)$ 
  apply (rule summable-comparison-test)
  apply (auto)
  done

```

Summability of geometric series for real algebras

```

lemma complete-algebra-summable-geometric:
  fixes  $x :: 'a::\{\text{real-normed-algebra-1}, \text{banach}\}$ 
  shows  $\text{norm } x < 1 \implies \text{summable } (\lambda n. x \wedge n)$ 
proof (rule summable-comparison-test)
  show  $\exists N. \forall n \geq N. \text{norm } (x \wedge n) \leq \text{norm } x \wedge n$ 
    by (simp add: norm-power-ineq)
  show  $\text{norm } x < 1 \implies \text{summable } (\lambda n. \text{norm } x \wedge n)$ 
    by (simp add: summable-geometric)
qed

```

Limit comparison property for series (c.f. jrh)

```

lemma summable-le:
  fixes  $f\ g :: \text{nat} \Rightarrow \text{real}$ 
  shows  $\llbracket \forall n. f\ n \leq g\ n; \text{summable } f; \text{summable } g \rrbracket \implies \text{suminf } f \leq \text{suminf } g$ 
  apply (drule summable-sums)+
  apply (simp only: sums-def, erule (1) LIMSEQ-le)
  apply (rule exI)
  apply (auto intro!: setsum-mono)
  done

```

```

lemma summable-le2:
  fixes  $f\ g :: \text{nat} \Rightarrow \text{real}$ 
  shows  $\llbracket \forall n. |f\ n| \leq g\ n; \text{summable } g \rrbracket \implies \text{summable } f \wedge \text{suminf } f \leq \text{suminf } g$ 
  apply (subgoal-tac summable f)
  apply (auto intro!: summable-le)
  apply (simp add: abs-le-iff)
  apply (rule-tac  $g=g$  in summable-comparison-test, simp-all)
  done

```

```

lemma suminf-0-le:
  fixes  $f :: \text{nat} \Rightarrow \text{real}$ 
  assumes  $gt0: \forall n. 0 \leq f\ n$  and  $sm: \text{summable } f$ 
  shows  $0 \leq \text{suminf } f$ 
proof –
  let  $?g = (\lambda n. (0 :: \text{real}))$ 
  from  $gt0$  have  $\forall n. ?g\ n \leq f\ n$  by simp
  moreover have  $\text{summable } ?g$  by (rule summable-zero)
  moreover from  $sm$  have  $\text{summable } f$  .
  ultimately have  $\text{suminf } ?g \leq \text{suminf } f$  by (rule summable-le)
  then show  $0 \leq \text{suminf } f$  by (simp add: suminf-zero)
qed

```

Absolute convergence implies normal convergence

```

lemma summable-norm-cancel:
  fixes  $f :: \text{nat} \Rightarrow 'a::\text{banach}$ 
  shows  $\text{summable } (\lambda n. \text{norm } (f\ n)) \implies \text{summable } f$ 
apply (simp only: summable-Cauchy, safe)
apply (drule-tac x=e in spec, safe)
apply (rule-tac x=N in exI, safe)
apply (drule-tac x=m in spec, safe)
apply (rule order-le-less-trans [OF norm-setsum])
apply (rule order-le-less-trans [OF abs-ge-self])
apply simp
done

```

```

lemma summable-rabs-cancel:
  fixes  $f :: \text{nat} \Rightarrow \text{real}$ 
  shows  $\text{summable } (\lambda n. |f\ n|) \implies \text{summable } f$ 
by (rule summable-norm-cancel, simp)

```

Absolute convergence of series

```

lemma summable-norm:
  fixes  $f :: \text{nat} \Rightarrow 'a::\text{banach}$ 
  shows  $\text{summable } (\lambda n. \text{norm } (f\ n)) \implies \text{norm } (\text{suminf } f) \leq (\sum n. \text{norm } (f\ n))$ 
by (auto intro: LIMSEQ-le LIMSEQ-norm summable-norm-cancel
      summable-sumr-LIMSEQ-suminf norm-setsum)

```

```

lemma summable-rabs:
  fixes  $f :: \text{nat} \Rightarrow \text{real}$ 
  shows  $\text{summable } (\lambda n. |f\ n|) \implies |\text{suminf } f| \leq (\sum n. |f\ n|)$ 
by (fold real-norm-def, rule summable-norm)

```

## 74.2 The Ratio Test

```

lemma norm-ratiotest-lemma:
  fixes  $x\ y :: 'a::\text{real-normed-vector}$ 
  shows  $\llbracket c \leq 0; \text{norm } x \leq c * \text{norm } y \rrbracket \implies x = 0$ 
apply (subgoal-tac norm x ≤ 0, simp)
apply (erule order-trans)
apply (simp add: mult-le-0-iff)
done

```

```

lemma rabs-ratiotest-lemma:  $\llbracket c \leq 0; \text{abs } x \leq c * \text{abs } y \rrbracket \implies x = (0::\text{real})$ 
by (erule norm-ratiotest-lemma, simp)

```

```

lemma le-Suc-ex:  $(k::\text{nat}) \leq l \implies (\exists n. l = k + n)$ 
apply (drule le-imp-less-or-eq)
apply (auto dest: less-imp-Suc-add)
done

```

```

lemma le-Suc-ex-iff:  $((k::\text{nat}) \leq l) = (\exists n. l = k + n)$ 
by (auto simp add: le-Suc-ex)

```

```

lemma ratio-test-lemma2:
  fixes f :: nat  $\Rightarrow$  'a::banach
  shows  $\llbracket \forall n \geq N. \text{norm } (f \text{ (Suc } n)) \leq c * \text{norm } (f \text{ } n) \rrbracket \Longrightarrow 0 < c \vee \text{summable } f$ 
  apply (simp (no-asm) add: linorder-not-le [symmetric])
  apply (simp add: summable-Cauchy)
  apply (safe, subgoal-tac  $\forall n. N < n \longrightarrow f \text{ } (n) = 0$ )
  prefer 2
  apply clarify
  apply (erule-tac  $x = n - \text{Suc } 0$  in allE)
  apply (simp add: diff-Suc split: nat.splits)
  apply (blast intro: norm-ratiotest-lemma)
  apply (rule-tac  $x = \text{Suc } N$  in exI, clarify)
  apply (simp cong: setsum-ivl-cong)
done

```

```

lemma ratio-test:
  fixes f :: nat  $\Rightarrow$  'a::banach
  shows  $\llbracket c < 1; \forall n \geq N. \text{norm } (f \text{ (Suc } n)) \leq c * \text{norm } (f \text{ } n) \rrbracket \Longrightarrow \text{summable } f$ 
  apply (frule ratio-test-lemma2, auto)
  apply (rule-tac  $g = \%n. (\text{norm } (f \text{ } N) / (c \wedge N)) * c \wedge n$ 
    in summable-comparison-test)
  apply (rule-tac  $x = N$  in exI, safe)
  apply (drule le-Suc-ex-iff [THEN iffD1])
  apply (auto simp add: power-add field-power-not-zero)
  apply (induct-tac na, auto)
  apply (rule-tac  $y = c * \text{norm } (f \text{ } (N + n))$  in order-trans)
  apply (auto intro: mult-right-mono simp add: summable-def)
  apply (rule-tac  $x = \text{norm } (f \text{ } N) * (1 / (1 - c)) / (c \wedge N)$  in exI)
  apply (rule sums-divide)
  apply (rule sums-mult)
  apply (auto intro!: geometric-sums)
done

```

### 74.3 Cauchy Product Formula

```

lemma setsum-triangle-reindex:
  fixes n :: nat
  shows  $(\sum (i,j) \in \{(i,j). i+j < n\}. f \text{ } i \text{ } j) = (\sum k=0..<n. \sum i=0..k. f \text{ } i \text{ } (k-i))$ 
  proof -
    have  $(\sum (i,j) \in \{(i,j). i+j < n\}. f \text{ } i \text{ } j) =$ 
       $(\sum (k,i) \in (\text{SIGMA } k:\{0..<n\}. \{0..k\}). f \text{ } i \text{ } (k-i))$ 
    proof (rule setsum-reindex-cong)
      show inj-on  $(\lambda(k,i). (i, k-i)) (\text{SIGMA } k:\{0..<n\}. \{0..k\})$ 
        by (rule inj-on-inverseI [where  $g = \lambda(i,j). (i+j, i)$ ], auto)
      show  $\{(i,j). i+j < n\} = (\lambda(k,i). (i, k-i)) \text{ ` } (\text{SIGMA } k:\{0..<n\}. \{0..k\})$ 
        by (safe, rule-tac  $x = (a+b, a)$  in image-eqI, auto)
      show  $\bigwedge a. (\lambda(k,i). f \text{ } i \text{ } (k-i)) a = \text{split } f ((\lambda(k,i). (i, k-i)) a)$ 

```

```

    by clarify
  qed
  thus ?thesis by (simp add: setsum-Sigma)
qed

lemma Cauchy-product-sums:
  fixes a b :: nat  $\Rightarrow$  'a::{real-normed-algebra,banach}
  assumes a: summable ( $\lambda k. \text{norm } (a\ k)$ )
  assumes b: summable ( $\lambda k. \text{norm } (b\ k)$ )
  shows ( $\lambda k. \sum_{i=0..k} a\ i * b\ (k - i)$ ) sums (( $\sum k. a\ k$ ) * ( $\sum k. b\ k$ ))
proof -
  let ?S1 =  $\lambda n::nat. \{0..<n\} \times \{0..<n\}$ 
  let ?S2 =  $\lambda n::nat. \{(i,j). i + j < n\}$ 
  have S1-mono:  $\bigwedge m\ n. m \leq n \implies ?S1\ m \subseteq ?S1\ n$  by auto
  have S2-le-S1:  $\bigwedge n. ?S2\ n \subseteq ?S1\ n$  by auto
  have S1-le-S2:  $\bigwedge n. ?S1\ (n\ \text{div}\ 2) \subseteq ?S2\ n$  by auto
  have finite-S1:  $\bigwedge n. \text{finite } (?S1\ n)$  by simp
  with S2-le-S1 have finite-S2:  $\bigwedge n. \text{finite } (?S2\ n)$  by (rule finite-subset)

  let ?g =  $\lambda(i,j). a\ i * b\ j$ 
  let ?f =  $\lambda(i,j). \text{norm } (a\ i) * \text{norm } (b\ j)$ 
  have f-nonneg:  $\bigwedge x. 0 \leq ?f\ x$ 
    by (auto simp add: mult-nonneg-nonneg)
  hence norm-setsum-f:  $\bigwedge A. \text{norm } (\text{setsum } ?f\ A) = \text{setsum } ?f\ A$ 
    unfolding real-norm-def
    by (simp only: abs-of-nonneg setsum-nonneg [rule-format])

  have ( $\lambda n. (\sum_{k=0..<n} a\ k) * (\sum_{k=0..<n} b\ k)$ )
    -----> ( $\sum k. a\ k$ ) * ( $\sum k. b\ k$ )
    by (intro LIMSEQ-mult summable-sumr-LIMSEQ-suminf
        summable-norm-cancel [OF a] summable-norm-cancel [OF b])
  hence 1: ( $\lambda n. \text{setsum } ?g\ (?S1\ n)$ ) -----> ( $\sum k. a\ k$ ) * ( $\sum k. b\ k$ )
    by (simp only: setsum-product setsum-Sigma [rule-format]
        finite-atLeastLessThan)

  have ( $\lambda n. (\sum_{k=0..<n} \text{norm } (a\ k)) * (\sum_{k=0..<n} \text{norm } (b\ k))$ )
    -----> ( $\sum k. \text{norm } (a\ k)$ ) * ( $\sum k. \text{norm } (b\ k)$ )
    using a b by (intro LIMSEQ-mult summable-sumr-LIMSEQ-suminf)
  hence ( $\lambda n. \text{setsum } ?f\ (?S1\ n)$ ) -----> ( $\sum k. \text{norm } (a\ k)$ ) * ( $\sum k. \text{norm } (b\ k)$ )
    by (simp only: setsum-product setsum-Sigma [rule-format]
        finite-atLeastLessThan)
  hence convergent ( $\lambda n. \text{setsum } ?f\ (?S1\ n)$ )
    by (rule convergentI)
  hence Cauchy: Cauchy ( $\lambda n. \text{setsum } ?f\ (?S1\ n)$ )
    by (rule convergent-Cauchy)
  have Zfun ( $\lambda n. \text{setsum } ?f\ (?S1\ n - ?S2\ n)$ ) sequentially
  proof (rule ZfunI, simp only: eventually-sequentially norm-setsum-f)
    fix r :: real
    assume r:  $0 < r$ 

```

```

from CauchyD [OF Cauchy r] obtain N
where  $\forall m \geq N. \forall n \geq N. \text{norm} (\text{setsum } ?f (?S1\ m) - \text{setsum } ?f (?S1\ n)) < r$ 
..
hence  $\bigwedge m\ n. \llbracket N \leq n; n \leq m \rrbracket \implies \text{norm} (\text{setsum } ?f (?S1\ m - ?S1\ n)) < r$ 
  by (simp only: setsum-diff finite-S1 S1-mono)
hence  $N: \bigwedge m\ n. \llbracket N \leq n; n \leq m \rrbracket \implies \text{setsum } ?f (?S1\ m - ?S1\ n) < r$ 
  by (simp only: norm-setsum-f)
show  $\exists N. \forall n \geq N. \text{setsum } ?f (?S1\ n - ?S2\ n) < r$ 
proof (intro exI allI impI)
  fix n assume  $2 * N \leq n$ 
  hence  $n: N \leq n \text{ div } 2$  by simp
  have  $\text{setsum } ?f (?S1\ n - ?S2\ n) \leq \text{setsum } ?f (?S1\ n - ?S1\ (n \text{ div } 2))$ 
    by (intro setsum-mono2 finite-Diff finite-S1 f-nonneg
      Diff-mono subset-refl S1-le-S2)
  also have  $\dots < r$ 
    using  $n \text{ div-le-dividend}$  by (rule N)
  finally show  $\text{setsum } ?f (?S1\ n - ?S2\ n) < r$  .
qed
qed
hence  $Zfun (\lambda n. \text{setsum } ?g (?S1\ n - ?S2\ n)) \text{ sequentially}$ 
  apply (rule Zfun-le [rule-format])
  apply (simp only: norm-setsum-f)
  apply (rule order-trans [OF norm-setsum setsum-mono])
  apply (auto simp add: norm-mult-ineq)
  done
hence  $2: (\lambda n. \text{setsum } ?g (?S1\ n) - \text{setsum } ?g (?S2\ n)) \dashrightarrow 0$ 
  unfolding tendsto-Zfun-iff diff-0-right
  by (simp only: setsum-diff finite-S1 S2-le-S1)

with 1 have  $(\lambda n. \text{setsum } ?g (?S2\ n)) \dashrightarrow (\sum k. a\ k) * (\sum k. b\ k)$ 
  by (rule LIMSEQ-diff-approach-zero2)
thus thesis by (simp only: sums-def setsum-triangle-reindex)
qed

lemma Cauchy-product:
  fixes  $a\ b :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra}, \text{banach}\}$ 
  assumes  $a: \text{summable } (\lambda k. \text{norm } (a\ k))$ 
  assumes  $b: \text{summable } (\lambda k. \text{norm } (b\ k))$ 
  shows  $(\sum k. a\ k) * (\sum k. b\ k) = (\sum k. \sum i=0..k. a\ i * b\ (k - i))$ 
using  $a\ b$ 
by (rule Cauchy-product-sums [THEN sums-unique])

end

```

## 75 NthRoot: Nth Roots of Real Numbers

```

theory NthRoot
imports Parity Deriv

```

begin

### 75.1 Existence of Nth Root

Existence follows from the Intermediate Value Theorem

**lemma** *realpow-pos-nth*:

**assumes**  $n: 0 < n$

**assumes**  $a: 0 < a$

**shows**  $\exists r > 0. r \wedge n = (a::real)$

**proof** –

**have**  $\exists r \geq 0. r \leq (\max 1 a) \wedge r \wedge n = a$

**proof** (*rule IVT*)

**show**  $0 \wedge n \leq a$  **using**  $n a$  **by** (*simp add: power-0-left*)

**show**  $0 \leq \max 1 a$  **by** *simp*

**from**  $n$  **have**  $n1: 1 \leq n$  **by** *simp*

**have**  $a \leq \max 1 a \wedge 1$  **by** *simp*

**also have**  $\max 1 a \wedge 1 \leq \max 1 a \wedge n$

**using**  $n1$  **by** (*rule power-increasing, simp*)

**finally show**  $a \leq \max 1 a \wedge n$ .

**show**  $\forall r. 0 \leq r \wedge r \leq \max 1 a \longrightarrow \text{isCont } (\lambda x. x \wedge n) r$

**by** (*simp add: isCont-power*)

**qed**

**then obtain**  $r$  **where**  $r: 0 \leq r \wedge r \wedge n = a$  **by** *fast*

**with**  $n a$  **have**  $r \neq 0$  **by** (*auto simp add: power-0-left*)

**with**  $r$  **have**  $0 < r \wedge r \wedge n = a$  **by** *simp*

**thus** *?thesis* ..

**qed**

**lemma** *realpow-pos-nth2*:  $(0::real) < a \implies \exists r > 0. r \wedge \text{Suc } n = a$

**by** (*blast intro: realpow-pos-nth*)

Uniqueness of nth positive root

**lemma** *realpow-pos-nth-unique*:

$\llbracket 0 < n; 0 < a \rrbracket \implies \exists! r. 0 < r \wedge r \wedge n = (a::real)$

**apply** (*auto intro!: realpow-pos-nth*)

**apply** (*rule-tac n=n in power-eq-imp-eq-base, simp-all*)

**done**

### 75.2 Nth Root

We define roots of negative reals such that  $\text{root } n \ (-x) = - \text{root } n \ x$ . This allows us to omit side conditions from many theorems.

**definition**

$\text{root} :: [nat, real] \Rightarrow real$  **where**

$\text{root } n \ x = (\text{if } 0 < x \text{ then } (\text{THE } u. 0 < u \wedge u \wedge n = x) \text{ else}$

$\text{if } x < 0 \text{ then } - (\text{THE } u. 0 < u \wedge u \wedge n = -x) \text{ else } 0)$

**lemma** *real-root-zero* [*simp*]:  $\text{root } n \ 0 = 0$   
**unfolding** *root-def* **by** *simp*

**lemma** *real-root-minus*:  $0 < n \implies \text{root } n \ (-x) = - \text{root } n \ x$   
**unfolding** *root-def* **by** *simp*

**lemma** *real-root-gt-zero*:  $\llbracket 0 < n; 0 < x \rrbracket \implies 0 < \text{root } n \ x$   
**apply** (*simp add: root-def*)  
**apply** (*drule (1) realpow-pos-nth-unique*)  
**apply** (*erule theI' [THEN conjunct1]*)  
**done**

**lemma** *real-root-pow-pos*:  
 $\llbracket 0 < n; 0 < x \rrbracket \implies \text{root } n \ x \wedge n = x$   
**apply** (*simp add: root-def*)  
**apply** (*drule (1) realpow-pos-nth-unique*)  
**apply** (*erule theI' [THEN conjunct2]*)  
**done**

**lemma** *real-root-pow-pos2* [*simp*]:  
 $\llbracket 0 < n; 0 \leq x \rrbracket \implies \text{root } n \ x \wedge n = x$   
**by** (*auto simp add: order-le-less real-root-pow-pos*)

**lemma** *odd-real-root-pow*:  $\text{odd } n \implies \text{root } n \ x \wedge n = x$   
**apply** (*rule-tac x=0 and y=x in linorder-le-cases*)  
**apply** (*erule (1) real-root-pow-pos2 [OF odd-pos]*)  
**apply** (*subgoal-tac root n (-x) ^ n = -x*)  
**apply** (*simp add: real-root-minus odd-pos*)  
**apply** (*simp add: odd-pos*)  
**done**

**lemma** *real-root-ge-zero*:  $\llbracket 0 < n; 0 \leq x \rrbracket \implies 0 \leq \text{root } n \ x$   
**by** (*auto simp add: order-le-less real-root-gt-zero*)

**lemma** *real-root-power-cancel*:  $\llbracket 0 < n; 0 \leq x \rrbracket \implies \text{root } n \ (x \wedge n) = x$   
**apply** (*subgoal-tac 0 ≤ x ^ n*)  
**apply** (*subgoal-tac 0 ≤ root n (x ^ n)*)  
**apply** (*subgoal-tac root n (x ^ n) ^ n = x ^ n*)  
**apply** (*erule (3) power-eq-imp-eq-base*)  
**apply** (*erule (1) real-root-pow-pos2*)  
**apply** (*erule (1) real-root-ge-zero*)  
**apply** (*erule zero-le-power*)  
**done**

**lemma** *odd-real-root-power-cancel*:  $\text{odd } n \implies \text{root } n \ (x \wedge n) = x$   
**apply** (*rule-tac x=0 and y=x in linorder-le-cases*)  
**apply** (*erule (1) real-root-power-cancel [OF odd-pos]*)  
**apply** (*subgoal-tac root n ((-x) ^ n) = -x*)  
**apply** (*simp add: real-root-minus odd-pos*)

**apply** (*erule* *real-root-power-cancel* [*OF odd-pos*], *simp*)  
**done**

**lemma** *real-root-pos-unique*:  
 $\llbracket 0 < n; 0 \leq y; y \wedge n = x \rrbracket \implies \text{root } n \ x = y$   
**by** (*erule* *subst*, *rule* *real-root-power-cancel*)

**lemma** *odd-real-root-unique*:  
 $\llbracket \text{odd } n; y \wedge n = x \rrbracket \implies \text{root } n \ x = y$   
**by** (*erule* *subst*, *rule* *odd-real-root-power-cancel*)

**lemma** *real-root-one* [*simp*]:  $0 < n \implies \text{root } n \ 1 = 1$   
**by** (*simp* *add*: *real-root-pos-unique*)

Root function is strictly monotonic, hence injective

**lemma** *real-root-less-mono-lemma*:  
 $\llbracket 0 < n; 0 \leq x; x < y \rrbracket \implies \text{root } n \ x < \text{root } n \ y$   
**apply** (*subgoal-tac*  $0 \leq y$ )  
**apply** (*subgoal-tac*  $\text{root } n \ x \wedge n < \text{root } n \ y \wedge n$ )  
**apply** (*erule* *power-less-imp-less-base*)  
**apply** (*erule* (1) *real-root-ge-zero*)  
**apply** *simp*  
**apply** *simp*  
**done**

**lemma** *real-root-less-mono*:  $\llbracket 0 < n; x < y \rrbracket \implies \text{root } n \ x < \text{root } n \ y$   
**apply** (*cases*  $0 \leq x$ )  
**apply** (*erule* (2) *real-root-less-mono-lemma*)  
**apply** (*cases*  $0 \leq y$ )  
**apply** (*rule-tac*  $y=0$  **in** *order-less-le-trans*)  
**apply** (*subgoal-tac*  $0 < \text{root } n \ (- \ x)$ )  
**apply** (*simp* *add*: *real-root-minus*)  
**apply** (*simp* *add*: *real-root-gt-zero*)  
**apply** (*simp* *add*: *real-root-ge-zero*)  
**apply** (*subgoal-tac*  $\text{root } n \ (- \ y) < \text{root } n \ (- \ x)$ )  
**apply** (*simp* *add*: *real-root-minus*)  
**apply** (*simp* *add*: *real-root-less-mono-lemma*)  
**done**

**lemma** *real-root-le-mono*:  $\llbracket 0 < n; x \leq y \rrbracket \implies \text{root } n \ x \leq \text{root } n \ y$   
**by** (*auto* *simp* *add*: *order-le-less* *real-root-less-mono*)

**lemma** *real-root-less-iff* [*simp*]:  
 $0 < n \implies (\text{root } n \ x < \text{root } n \ y) = (x < y)$   
**apply** (*cases*  $x < y$ )  
**apply** (*simp* *add*: *real-root-less-mono*)  
**apply** (*simp* *add*: *linorder-not-less* *real-root-le-mono*)  
**done**



**lemma** *real-root-le-iff* [simp]:  
 $0 < n \implies (\text{root } n \ x \leq \text{root } n \ y) = (x \leq y)$   
**apply** (cases  $x \leq y$ )  
**apply** (simp add: *real-root-le-mono*)  
**apply** (simp add: *linorder-not-le real-root-less-mono*)  
**done**

**lemma** *real-root-eq-iff* [simp]:  
 $0 < n \implies (\text{root } n \ x = \text{root } n \ y) = (x = y)$   
**by** (simp add: *order-eq-iff*)

**lemmas** *real-root-gt-0-iff* [simp] = *real-root-less-iff* [where  $x=0$ , *simplified*]  
**lemmas** *real-root-lt-0-iff* [simp] = *real-root-less-iff* [where  $y=0$ , *simplified*]  
**lemmas** *real-root-ge-0-iff* [simp] = *real-root-le-iff* [where  $x=0$ , *simplified*]  
**lemmas** *real-root-le-0-iff* [simp] = *real-root-le-iff* [where  $y=0$ , *simplified*]  
**lemmas** *real-root-eq-0-iff* [simp] = *real-root-eq-iff* [where  $y=0$ , *simplified*]

**lemma** *real-root-gt-1-iff* [simp]:  $0 < n \implies (1 < \text{root } n \ y) = (1 < y)$   
**by** (insert *real-root-less-iff* [where  $x=1$ ], *simp*)

**lemma** *real-root-lt-1-iff* [simp]:  $0 < n \implies (\text{root } n \ x < 1) = (x < 1)$   
**by** (insert *real-root-less-iff* [where  $y=1$ ], *simp*)

**lemma** *real-root-ge-1-iff* [simp]:  $0 < n \implies (1 \leq \text{root } n \ y) = (1 \leq y)$   
**by** (insert *real-root-le-iff* [where  $x=1$ ], *simp*)

**lemma** *real-root-le-1-iff* [simp]:  $0 < n \implies (\text{root } n \ x \leq 1) = (x \leq 1)$   
**by** (insert *real-root-le-iff* [where  $y=1$ ], *simp*)

**lemma** *real-root-eq-1-iff* [simp]:  $0 < n \implies (\text{root } n \ x = 1) = (x = 1)$   
**by** (insert *real-root-eq-iff* [where  $y=1$ ], *simp*)

Roots of roots

**lemma** *real-root-Suc-0* [simp]:  $\text{root } (\text{Suc } 0) \ x = x$   
**by** (simp add: *odd-real-root-unique*)

**lemma** *real-root-pos-mult-exp*:  
 $\llbracket 0 < m; 0 < n; 0 < x \rrbracket \implies \text{root } (m * n) \ x = \text{root } m \ (\text{root } n \ x)$   
**by** (rule *real-root-pos-unique*, *simp-all* add: *power-mult*)

**lemma** *real-root-mult-exp*:  
 $\llbracket 0 < m; 0 < n \rrbracket \implies \text{root } (m * n) \ x = \text{root } m \ (\text{root } n \ x)$   
**apply** (rule *linorder-cases* [where  $x=x$  and  $y=0$ ])  
**apply** (subgoal-tac  $\text{root } (m * n) \ (-x) = \text{root } m \ (\text{root } n \ (-x))$ )  
**apply** (simp add: *real-root-minus*)  
**apply** (simp-all add: *real-root-pos-mult-exp*)  
**done**

**lemma** *real-root-commute*:

$\llbracket 0 < m; 0 < n \rrbracket \implies \text{root } m (\text{root } n x) = \text{root } n (\text{root } m x)$   
**by** (*simp add: real-root-mult-exp [symmetric] mult-commute*)

Monotonicity in first argument

**lemma** *real-root-strict-decreasing*:

$\llbracket 0 < n; n < N; 1 < x \rrbracket \implies \text{root } N x < \text{root } n x$   
**apply** (*subgoal-tac root n (root N x) ^ n < root N (root n x) ^ N, simp*)  
**apply** (*simp add: real-root-commute power-strict-increasing*  
*del: real-root-pow-pos2*)

**done**

**lemma** *real-root-strict-increasing*:

$\llbracket 0 < n; n < N; 0 < x; x < 1 \rrbracket \implies \text{root } n x < \text{root } N x$   
**apply** (*subgoal-tac root N (root n x) ^ N < root n (root N x) ^ n, simp*)  
**apply** (*simp add: real-root-commute power-strict-decreasing*  
*del: real-root-pow-pos2*)

**done**

**lemma** *real-root-decreasing*:

$\llbracket 0 < n; n < N; 1 \leq x \rrbracket \implies \text{root } N x \leq \text{root } n x$   
**by** (*auto simp add: order-le-less real-root-strict-decreasing*)

**lemma** *real-root-increasing*:

$\llbracket 0 < n; n < N; 0 \leq x; x \leq 1 \rrbracket \implies \text{root } n x \leq \text{root } N x$   
**by** (*auto simp add: order-le-less real-root-strict-increasing*)

Roots of multiplication and division

**lemma** *real-root-mult-lemma*:

$\llbracket 0 < n; 0 \leq x; 0 \leq y \rrbracket \implies \text{root } n (x * y) = \text{root } n x * \text{root } n y$   
**by** (*simp add: real-root-pos-unique mult-nonneg-nonneg power-mult-distrib*)

**lemma** *real-root-inverse-lemma*:

$\llbracket 0 < n; 0 \leq x \rrbracket \implies \text{root } n (\text{inverse } x) = \text{inverse } (\text{root } n x)$   
**by** (*simp add: real-root-pos-unique power-inverse [symmetric]*)

**lemma** *real-root-mult*:

**assumes** *n: 0 < n*  
**shows** *root n (x \* y) = root n x \* root n y*  
**proof** (*rule linorder-le-cases, rule-tac [!] linorder-le-cases*)  
**assume** *0 ≤ x and 0 ≤ y*  
**thus** *?thesis* **by** (*rule real-root-mult-lemma [OF n]*)  
**next**  
**assume** *0 ≤ x and y ≤ 0*  
**hence** *0 ≤ x and 0 ≤ - y* **by** *simp-all*  
**hence** *root n (x \* - y) = root n x \* root n (- y)*  
**by** (*rule real-root-mult-lemma [OF n]*)  
**thus** *?thesis* **by** (*simp add: real-root-minus [OF n]*)  
**next**  
**assume** *x ≤ 0 and 0 ≤ y*

hence  $0 \leq -x$  and  $0 \leq y$  by *simp-all*  
 hence  $\text{root } n \ (-x * y) = \text{root } n \ (-x) * \text{root } n \ y$   
 by (rule *real-root-mult-lemma* [OF  $n$ ])  
 thus ?thesis by (simp add: *real-root-minus* [OF  $n$ ])  
 next  
 assume  $x \leq 0$  and  $y \leq 0$   
 hence  $0 \leq -x$  and  $0 \leq -y$  by *simp-all*  
 hence  $\text{root } n \ (-x * -y) = \text{root } n \ (-x) * \text{root } n \ (-y)$   
 by (rule *real-root-mult-lemma* [OF  $n$ ])  
 thus ?thesis by (simp add: *real-root-minus* [OF  $n$ ])  
 qed

**lemma** *real-root-inverse*:  
 assumes  $n: 0 < n$   
 shows  $\text{root } n \ (\text{inverse } x) = \text{inverse } (\text{root } n \ x)$   
**proof** (rule *linorder-le-cases*)  
 assume  $0 \leq x$   
 thus ?thesis by (rule *real-root-inverse-lemma* [OF  $n$ ])  
 next  
 assume  $x \leq 0$   
 hence  $0 \leq -x$  by *simp*  
 hence  $\text{root } n \ (\text{inverse } (-x)) = \text{inverse } (\text{root } n \ (-x))$   
 by (rule *real-root-inverse-lemma* [OF  $n$ ])  
 thus ?thesis by (simp add: *real-root-minus* [OF  $n$ ])  
 qed

**lemma** *real-root-divide*:  
 $0 < n \implies \text{root } n \ (x / y) = \text{root } n \ x / \text{root } n \ y$   
 by (simp add: *divide-inverse* *real-root-mult* *real-root-inverse*)

**lemma** *real-root-power*:  
 $0 < n \implies \text{root } n \ (x ^ k) = \text{root } n \ x ^ k$   
 by (induct  $k$ , *simp-all* add: *real-root-mult*)

**lemma** *real-root-abs*:  $0 < n \implies \text{root } n \ |x| = |\text{root } n \ x|$   
 by (simp add: *abs-if* *real-root-minus*)

Continuity and derivatives

**lemma** *isCont-root-pos*:  
 assumes  $n: 0 < n$   
 assumes  $x: 0 < x$   
 shows *isCont* ( $\text{root } n$ )  $x$   
**proof** –  
 have *isCont* ( $\text{root } n$ ) ( $\text{root } n \ x ^ n$ )  
**proof** (rule *isCont-inverse-function* [where  $f = \lambda a. a ^ n$ ])  
 show  $0 < \text{root } n \ x$  using  $n \ x$  by *simp*  
 show  $\forall z. |z - \text{root } n \ x| \leq \text{root } n \ x \implies \text{root } n \ (z ^ n) = z$   
 by (simp add: *abs-le-iff* *real-root-power-cancel*  $n$ )  
 show  $\forall z. |z - \text{root } n \ x| \leq \text{root } n \ x \implies \text{isCont } (\lambda a. a ^ n) \ z$

```

    by (simp add: isCont-power)
  qed
  thus ?thesis using n x by simp
qed

```

```

lemma isCont-root-neg:
   $\llbracket 0 < n; x < 0 \rrbracket \implies \text{isCont } (\text{root } n) x$ 
  apply (subgoal-tac isCont ( $\lambda x. - \text{root } n (-x)$ ) x)
  apply (simp add: real-root-minus)
  apply (rule isCont-o2 [OF isCont-minus [OF isCont-ident]])
  apply (simp add: isCont-minus isCont-root-pos)
done

```

```

lemma isCont-root-zero:
   $0 < n \implies \text{isCont } (\text{root } n) 0$ 
  unfolding isCont-def
  apply (rule LIM-I)
  apply (rule-tac  $x=r \wedge n$  in exI, safe)
  apply (simp)
  apply (simp add: real-root-abs [symmetric])
  apply (rule-tac  $n=n$  in power-less-imp-less-base, simp-all)
done

```

```

lemma isCont-real-root:  $0 < n \implies \text{isCont } (\text{root } n) x$ 
  apply (rule-tac  $x=x$  and  $y=0$  in linorder-cases)
  apply (simp-all add: isCont-root-pos isCont-root-neg isCont-root-zero)
done

```

```

lemma DERIV-real-root:
  assumes n:  $0 < n$ 
  assumes x:  $0 < x$ 
  shows DERIV ( $\text{root } n$ ) x :> inverse (real n * root n x ^ (n - Suc 0))
proof (rule DERIV-inverse-function)
  show  $0 < x$  using x .
  show  $x < x + 1$  by simp
  show  $\forall y. 0 < y \wedge y < x + 1 \longrightarrow \text{root } n y \wedge n = y$ 
    using n by simp
  show DERIV ( $\lambda x. x \wedge n$ ) (root n x) :> real n * root n x ^ (n - Suc 0)
    by (rule DERIV-pow)
  show real n * root n x ^ (n - Suc 0)  $\neq 0$ 
    using n x by simp
  show isCont ( $\text{root } n$ ) x
    using n by (rule isCont-real-root)
qed

```

```

lemma DERIV-odd-real-root:
  assumes n: odd n
  assumes x:  $x \neq 0$ 
  shows DERIV ( $\text{root } n$ ) x :> inverse (real n * root n x ^ (n - Suc 0))

```

```

proof (rule DERIV-inverse-function)
  show  $x - 1 < x$  by simp
  show  $x < x + 1$  by simp
  show  $\forall y. x - 1 < y \wedge y < x + 1 \longrightarrow \text{root } n \ y \ ^n = y$ 
    using  $n$  by (simp add: odd-real-root-pow)
  show DERIV  $(\lambda x. x \ ^n) (\text{root } n \ x) :> \text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)}$ 
    by (rule DERIV-pow)
  show  $\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)} \neq 0$ 
    using odd-pos [OF n]  $x$  by simp
  show isCont  $(\text{root } n) \ x$ 
    using odd-pos [OF n] by (rule isCont-real-root)
qed

lemma DERIV-even-real-root:
  assumes  $n: 0 < n$  and even n
  assumes  $x: x < 0$ 
  shows DERIV  $(\text{root } n) \ x :> \text{inverse } (- \text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
proof (rule DERIV-inverse-function)
  show  $x - 1 < x$  by simp
  show  $x < 0$  using  $x$  .
next
  show  $\forall y. x - 1 < y \wedge y < 0 \longrightarrow - (\text{root } n \ y \ ^n) = y$ 
  proof (rule allI, rule impI, erule conjE)
    fix  $y$  assume  $x - 1 < y$  and  $y < 0$ 
    hence  $\text{root } n \ (-y) \ ^n = -y$  using  $\langle 0 < n \rangle$  by simp
    with real-root-minus[OF <0 < n>] and  $\langle \text{even } n \rangle$ 
    show  $- (\text{root } n \ y \ ^n) = y$  by simp
  qed
next
  show DERIV  $(\lambda x. - (x \ ^n)) (\text{root } n \ x) :> - \text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)}$ 
    by (auto intro!: DERIV-intros)
  show  $- \text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)} \neq 0$ 
    using  $n \ x$  by simp
  show isCont  $(\text{root } n) \ x$ 
    using  $n$  by (rule isCont-real-root)
qed

lemma DERIV-real-root-generic:
  assumes  $0 < n$  and  $x \neq 0$ 
  and even:  $\llbracket \text{even } n ; 0 < x \rrbracket \Longrightarrow D = \text{inverse } (\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
  and even:  $\llbracket \text{even } n ; x < 0 \rrbracket \Longrightarrow D = - \text{inverse } (\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
  and odd:  $\text{odd } n \Longrightarrow D = \text{inverse } (\text{real } n * \text{root } n \ x \ ^{(n - \text{Suc } 0)})$ 
  shows DERIV  $(\text{root } n) \ x :> D$ 
using assms by (cases even n, cases 0 < x,
  auto intro: DERIV-real-root[THEN DERIV-cong]
  DERIV-odd-real-root[THEN DERIV-cong]
  DERIV-even-real-root[THEN DERIV-cong])

```

### 75.3 Square Root

**definition**

$\text{sqrt} :: \text{real} \Rightarrow \text{real}$  **where**  
 $\text{sqrt} = \text{root } 2$

**lemma** *pos2*:  $0 < (2::\text{nat})$  **by** *simp*

**lemma** *real-sqrt-unique*:  $\llbracket y^2 = x; 0 \leq y \rrbracket \implies \text{sqrt } x = y$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-pos-unique* [*OF pos2*])

**lemma** *real-sqrt-abs* [*simp*]:  $\text{sqrt } (x^2) = |x|$   
**apply** (*rule* *real-sqrt-unique*)  
**apply** (*rule* *power2-abs*)  
**apply** (*rule* *abs-ge-zero*)  
**done**

**lemma** *real-sqrt-pow2* [*simp*]:  $0 \leq x \implies (\text{sqrt } x)^2 = x$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-pow-pos2* [*OF pos2*])

**lemma** *real-sqrt-pow2-iff* [*simp*]:  $((\text{sqrt } x)^2 = x) = (0 \leq x)$   
**apply** (*rule* *iffI*)  
**apply** (*erule* *subst*)  
**apply** (*rule* *zero-le-power2*)  
**apply** (*erule* *real-sqrt-pow2*)  
**done**

**lemma** *real-sqrt-zero* [*simp*]:  $\text{sqrt } 0 = 0$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-zero*)

**lemma** *real-sqrt-one* [*simp*]:  $\text{sqrt } 1 = 1$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-one* [*OF pos2*])

**lemma** *real-sqrt-minus*:  $\text{sqrt } (-x) = -\text{sqrt } x$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-minus* [*OF pos2*])

**lemma** *real-sqrt-mult*:  $\text{sqrt } (x * y) = \text{sqrt } x * \text{sqrt } y$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-mult* [*OF pos2*])

**lemma** *real-sqrt-inverse*:  $\text{sqrt } (\text{inverse } x) = \text{inverse } (\text{sqrt } x)$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-inverse* [*OF pos2*])

**lemma** *real-sqrt-divide*:  $\text{sqrt } (x / y) = \text{sqrt } x / \text{sqrt } y$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-divide* [*OF pos2*])

**lemma** *real-sqrt-power*:  $\text{sqrt } (x ^ k) = \text{sqrt } x ^ k$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-power* [*OF pos2*])

**lemma** *real-sqrt-gt-zero*:  $0 < x \implies 0 < \text{sqrt } x$   
**unfolding** *sqrt-def* **by** (*rule* *real-root-gt-zero* [*OF pos2*])

**lemma** *real-sqrt-ge-zero*:  $0 \leq x \implies 0 \leq \text{sqrt } x$   
**unfolding** *sqrt-def* **by** (rule *real-root-ge-zero* [OF pos2])

**lemma** *real-sqrt-less-mono*:  $x < y \implies \text{sqrt } x < \text{sqrt } y$   
**unfolding** *sqrt-def* **by** (rule *real-root-less-mono* [OF pos2])

**lemma** *real-sqrt-le-mono*:  $x \leq y \implies \text{sqrt } x \leq \text{sqrt } y$   
**unfolding** *sqrt-def* **by** (rule *real-root-le-mono* [OF pos2])

**lemma** *real-sqrt-less-iff* [simp]:  $(\text{sqrt } x < \text{sqrt } y) = (x < y)$   
**unfolding** *sqrt-def* **by** (rule *real-root-less-iff* [OF pos2])

**lemma** *real-sqrt-le-iff* [simp]:  $(\text{sqrt } x \leq \text{sqrt } y) = (x \leq y)$   
**unfolding** *sqrt-def* **by** (rule *real-root-le-iff* [OF pos2])

**lemma** *real-sqrt-eq-iff* [simp]:  $(\text{sqrt } x = \text{sqrt } y) = (x = y)$   
**unfolding** *sqrt-def* **by** (rule *real-root-eq-iff* [OF pos2])

**lemmas** *real-sqrt-gt-0-iff* [simp] = *real-sqrt-less-iff* [where  $x=0$ , simplified]  
**lemmas** *real-sqrt-lt-0-iff* [simp] = *real-sqrt-less-iff* [where  $y=0$ , simplified]  
**lemmas** *real-sqrt-ge-0-iff* [simp] = *real-sqrt-le-iff* [where  $x=0$ , simplified]  
**lemmas** *real-sqrt-le-0-iff* [simp] = *real-sqrt-le-iff* [where  $y=0$ , simplified]  
**lemmas** *real-sqrt-eq-0-iff* [simp] = *real-sqrt-eq-iff* [where  $y=0$ , simplified]

**lemmas** *real-sqrt-gt-1-iff* [simp] = *real-sqrt-less-iff* [where  $x=1$ , simplified]  
**lemmas** *real-sqrt-lt-1-iff* [simp] = *real-sqrt-less-iff* [where  $y=1$ , simplified]  
**lemmas** *real-sqrt-ge-1-iff* [simp] = *real-sqrt-le-iff* [where  $x=1$ , simplified]  
**lemmas** *real-sqrt-le-1-iff* [simp] = *real-sqrt-le-iff* [where  $y=1$ , simplified]  
**lemmas** *real-sqrt-eq-1-iff* [simp] = *real-sqrt-eq-iff* [where  $y=1$ , simplified]

**lemma** *isCont-real-sqrt*: *isCont* *sqrt*  $x$   
**unfolding** *sqrt-def* **by** (rule *isCont-real-root* [OF pos2])

**lemma** *DERIV-real-sqrt-generic*:  
**assumes**  $x \neq 0$   
**assumes**  $x > 0 \implies D = \text{inverse } (\text{sqrt } x) / 2$   
**assumes**  $x < 0 \implies D = - \text{inverse } (\text{sqrt } x) / 2$   
**shows** *DERIV* *sqrt*  $x$   $:>$   $D$   
**using** *assms* **unfolding** *sqrt-def*  
**by** (auto intro!: *DERIV-real-root-generic*)

**lemma** *DERIV-real-sqrt*:  
 $0 < x \implies \text{DERIV } \text{sqrt } x$   $:>$   $\text{inverse } (\text{sqrt } x) / 2$   
**using** *DERIV-real-sqrt-generic* **by** *simp*

**declare**  
*DERIV-real-sqrt-generic* [THEN *DERIV-chain2*, THEN *DERIV-cong*, *DERIV-intros*]  
*DERIV-real-root-generic* [THEN *DERIV-chain2*, THEN *DERIV-cong*, *DERIV-intros*]

```

lemma not-real-square-gt-zero [simp]: ( $\sim (0::real) < x*x$ ) = ( $x = 0$ )
apply auto
apply (cut-tac  $x = x$  and  $y = 0$  in linorder-less-linear)
apply (simp add: zero-less-mult-iff)
done

```

```

lemma real-sqrt-abs2 [simp]:  $\text{sqrt}(x*x) = |x|$ 
apply (subst power2-eq-square [symmetric])
apply (rule real-sqrt-abs)
done

```

```

lemma real-sqrt-pow2-gt-zero:  $0 < x \implies 0 < (\text{sqrt } x)^2$ 
by simp

```

```

lemma real-sqrt-not-eq-zero:  $0 < x \implies \text{sqrt } x \neq 0$ 
by simp

```

```

lemma real-inv-sqrt-pow2:  $0 < x \implies \text{inverse } (\text{sqrt}(x)) ^ 2 = \text{inverse } x$ 
by (simp add: power-inverse [symmetric])

```

```

lemma real-sqrt-eq-zero-cancel: [ $0 \leq x; \text{sqrt}(x) = 0$ ]  $\implies x = 0$ 
by simp

```

```

lemma real-sqrt-ge-one:  $1 \leq x \implies 1 \leq \text{sqrt } x$ 
by simp

```

```

lemma real-sqrt-two-gt-zero [simp]:  $0 < \text{sqrt } 2$ 
by simp

```

```

lemma real-sqrt-two-ge-zero [simp]:  $0 \leq \text{sqrt } 2$ 
by simp

```

```

lemma real-sqrt-two-gt-one [simp]:  $1 < \text{sqrt } 2$ 
by simp

```

```

lemma sqrt-divide-self-eq:
  assumes nneg:  $0 \leq x$ 
  shows  $\text{sqrt } x / x = \text{inverse } (\text{sqrt } x)$ 
proof cases
  assume  $x=0$  thus ?thesis by simp
next
  assume nz:  $x \neq 0$ 
  hence pos:  $0 < x$  using nneg by arith
  show ?thesis
  proof (rule right-inverse-eq [THEN iffD1, THEN sym])
    show  $\text{sqrt } x / x \neq 0$  by (simp add: divide-inverse nneg nz)
    show  $\text{inverse } (\text{sqrt } x) / (\text{sqrt } x / x) = 1$ 
      by (simp add: divide-inverse mult-assoc [symmetric])
  qed

```



```

      power2-eq-square [symmetric] real-inv-sqrt-pow2 pos nz)
qed
qed

```

```

lemma real-divide-square-eq [simp]: (((r::real) * a) / (r * r)) = a / r
apply (simp add: divide-inverse)
apply (case-tac r=0)
apply (auto simp add: mult-ac)
done

```

```

lemma lemma-real-divide-sqrt-less: 0 < u ==> u / sqrt 2 < u
by (simp add: divide-less-eq)

```

```

lemma four-x-squared:
  fixes x::real
  shows 4 * x2 = (2 * x)2
by (simp add: power2-eq-square)

```

## 75.4 Square Root of Sum of Squares

```

lemma real-sqrt-mult-self-sum-ge-zero [simp]: 0 ≤ sqrt(x*x + y*y)
by (rule real-sqrt-ge-zero [OF sum-squares-ge-zero])

```

```

lemma real-sqrt-sum-squares-ge-zero [simp]: 0 ≤ sqrt (x2 + y2)
by simp

```

```

declare real-sqrt-sum-squares-ge-zero [THEN abs-of-nonneg, simp]

```

```

lemma real-sqrt-sum-squares-mult-ge-zero [simp]:
  0 ≤ sqrt ((x2 + y2)*(xa2 + ya2))
by (auto intro!: real-sqrt-ge-zero simp add: zero-le-mult-iff)

```

```

lemma real-sqrt-sum-squares-mult-squared-eq [simp]:
  sqrt ((x2 + y2) * (xa2 + ya2)) ^ 2 = (x2 + y2) * (xa2 + ya2)
by (auto simp add: zero-le-mult-iff)

```

```

lemma real-sqrt-sum-squares-eq-cancel: sqrt (x2 + y2) = x ==> y = 0
by (drule-tac f = %x. x2 in arg-cong, simp)

```

```

lemma real-sqrt-sum-squares-eq-cancel2: sqrt (x2 + y2) = y ==> x = 0
by (drule-tac f = %x. x2 in arg-cong, simp)

```

```

lemma real-sqrt-sum-squares-ge1 [simp]: x ≤ sqrt (x2 + y2)
by (rule power2-le-imp-le, simp-all)

```

```

lemma real-sqrt-sum-squares-ge2 [simp]: y ≤ sqrt (x2 + y2)
by (rule power2-le-imp-le, simp-all)

```

```

lemma real-sqrt-ge-abs1 [simp]: |x| ≤ sqrt (x2 + y2)

```

**by** (rule power2-le-imp-le, simp-all)

**lemma** real-sqrt-ge-abs2 [simp]:  $|y| \leq \text{sqrt } (x^2 + y^2)$   
**by** (rule power2-le-imp-le, simp-all)

**lemma** le-real-sqrt-sumsq [simp]:  $x \leq \text{sqrt } (x * x + y * y)$   
**by** (simp add: power2-eq-square [symmetric])

**lemma** real-sqrt-sum-squares-triangle-ineq:  
 $\text{sqrt } ((a + c)^2 + (b + d)^2) \leq \text{sqrt } (a^2 + b^2) + \text{sqrt } (c^2 + d^2)$   
**apply** (rule power2-le-imp-le, simp)  
**apply** (simp add: power2-sum)  
**apply** (simp only: mult-assoc right-distrib [symmetric])  
**apply** (rule mult-left-mono)  
**apply** (rule power2-le-imp-le)  
**apply** (simp add: power2-sum power-mult-distrib)  
**apply** (simp add: ring-distrib)  
**apply** (subgoal-tac  $0 \leq b^2 * c^2 + a^2 * d^2 - 2 * (a * c) * (b * d)$ , simp)  
**apply** (rule-tac  $b = (a * d - b * c)^2$  in ord-le-eq-trans)  
**apply** (rule zero-le-power2)  
**apply** (simp add: power2-diff power-mult-distrib)  
**apply** (simp add: mult-nonneg-nonneg)  
**apply** simp  
**apply** (simp add: add-increasing)  
**done**

**lemma** real-sqrt-sum-squares-less:  
 $\llbracket |x| < u / \text{sqrt } 2; |y| < u / \text{sqrt } 2 \rrbracket \implies \text{sqrt } (x^2 + y^2) < u$   
**apply** (rule power2-less-imp-less, simp)  
**apply** (drule power-strict-mono [OF - abs-ge-zero pos2])  
**apply** (drule power-strict-mono [OF - abs-ge-zero pos2])  
**apply** (simp add: power-divide)  
**apply** (drule order-le-less-trans [OF abs-ge-zero])  
**apply** (simp add: zero-less-divide-iff)  
**done**

Needed for the infinitely close relation over the nonstandard complex numbers

**lemma** lemma-sqrt-hcomplex-capprox:  
 $\llbracket 0 < u; x < u/2; y < u/2; 0 \leq x; 0 \leq y \rrbracket \implies \text{sqrt } (x^2 + y^2) < u$   
**apply** (rule-tac  $y = u/\text{sqrt } 2$  in order-le-less-trans)  
**apply** (erule-tac [2] lemma-real-divide-sqrt-less)  
**apply** (rule power2-le-imp-le)  
**apply** (auto simp add: real-0-le-divide-iff power-divide)  
**apply** (rule-tac  $t = u^2$  in real-sum-of-halves [THEN subst])  
**apply** (rule add-mono)  
**apply** (auto simp add: four-x-squared intro: power-mono)  
**done**

Legacy theorem names:

```

lemmas real-root-pos2 = real-root-power-cancel
lemmas real-root-pos-pos = real-root-gt-zero [THEN order-less-imp-le]
lemmas real-root-pos-pos-le = real-root-ge-zero
lemmas real-sqrt-mult-distrib = real-sqrt-mult
lemmas real-sqrt-mult-distrib2 = real-sqrt-mult
lemmas real-sqrt-eq-zero-cancel-iff = real-sqrt-eq-0-iff

```

```

lemma real-root-pos:  $0 < x \implies \text{root } (\text{Suc } n) (x \wedge (\text{Suc } n)) = x$ 
by (rule real-root-power-cancel [OF zero-less-Suc order-less-imp-le])

```

**end**

## 76 Transcendental: Power Series, Transcendental Functions etc.

```

theory Transcendental
imports Fact Series Deriv NthRoot
begin

```

### 76.1 Properties of Power Series

```

lemma lemma-realpow-diff:
  fixes  $y :: 'a::\text{monoid-mult}$ 
  shows  $p \leq n \implies y \wedge (\text{Suc } n - p) = (y \wedge (n - p)) * y$ 
proof –
  assume  $p \leq n$ 
  hence  $\text{Suc } n - p = \text{Suc } (n - p)$  by (rule Suc-diff-le)
  thus ?thesis by (simp add: power-commutes)
qed

```

```

lemma lemma-realpow-diff-sumr:
  fixes  $y :: 'a::\{\text{comm-semiring-0}, \text{monoid-mult}\}$  shows
     $(\sum_{p=0..<\text{Suc } n} (x \wedge p) * y \wedge (\text{Suc } n - p)) =$ 
     $y * (\sum_{p=0..<\text{Suc } n} (x \wedge p) * y \wedge (n - p))$ 
by (simp add: setsum-right-distrib lemma-realpow-diff mult-ac
    del: setsum-op-ivl-Suc)

```

```

lemma lemma-realpow-diff-sumr2:
  fixes  $y :: 'a::\{\text{comm-ring}, \text{monoid-mult}\}$  shows
     $x \wedge (\text{Suc } n) - y \wedge (\text{Suc } n) =$ 
     $(x - y) * (\sum_{p=0..<\text{Suc } n} (x \wedge p) * y \wedge (n - p))$ 
apply (induct n, simp)
apply (simp del: setsum-op-ivl-Suc)
apply (subst setsum-op-ivl-Suc)
apply (subst lemma-realpow-diff-sumr)

```

```

apply (simp add: right-distrib del: setsum-op-ivl-Suc)
apply (subst mult-left-commute [of x - y])
apply (erule subst)
apply (simp add: algebra-simps)
done

```

```

lemma lemma-realpow-rev-sumr:
  
$$\left(\sum_{p=0..<Suc\ n} (x \wedge p) * (y \wedge (n - p))\right) =$$


$$\left(\sum_{p=0..<Suc\ n} (x \wedge (n - p)) * (y \wedge p)\right)$$

apply (rule setsum-reindex-cong [where f= $\lambda i. n - i$ ])
apply (rule inj-onI, simp)
apply auto
apply (rule-tac x= $n - x$  in image-eqI, simp, simp)
done

```

Power series has a ‘circle’ of convergence, i.e. if it sums for  $x$ , then it sums absolutely for  $z$  with  $|z| < |x|$ .

```

lemma powser-insidea:
  fixes x z :: 'a::{real-normed-field,banach}
  assumes 1: summable ( $\lambda n. f\ n * x \wedge n$ )
  assumes 2: norm z < norm x
  shows summable ( $\lambda n. norm (f\ n * z \wedge n)$ )
proof -
  from 2 have x-neq-0:  $x \neq 0$  by clarsimp
  from 1 have ( $\lambda n. f\ n * x \wedge n$ ) ----> 0
    by (rule summable-LIMSEQ-zero)
  hence convergent ( $\lambda n. f\ n * x \wedge n$ )
    by (rule convergentI)
  hence Cauchy ( $\lambda n. f\ n * x \wedge n$ )
    by (simp add: Cauchy-convergent-iff)
  hence Bseq ( $\lambda n. f\ n * x \wedge n$ )
    by (rule Cauchy-Bseq)
  then obtain K where 3:  $0 < K$  and 4:  $\forall n. norm (f\ n * x \wedge n) \leq K$ 
    by (simp add: Bseq-def, safe)
  have  $\exists N. \forall n \geq N. norm (norm (f\ n * z \wedge n)) \leq$ 
     $K * norm (z \wedge n) * inverse (norm (x \wedge n))$ 
proof (intro exI allI impI)
  fix n::nat assume 0  $\leq n$ 
  have  $norm (norm (f\ n * z \wedge n)) * norm (x \wedge n) =$ 
     $norm (f\ n * x \wedge n) * norm (z \wedge n)$ 
    by (simp add: norm-mult abs-mult)
  also have  $\dots \leq K * norm (z \wedge n)$ 
    by (simp only: mult-right-mono 4 norm-ge-zero)
  also have  $\dots = K * norm (z \wedge n) * (inverse (norm (x \wedge n)) * norm (x \wedge n))$ 
    by (simp add: x-neq-0)
  also have  $\dots = K * norm (z \wedge n) * inverse (norm (x \wedge n)) * norm (x \wedge n)$ 
    by (simp only: mult-assoc)
  finally show  $norm (norm (f\ n * z \wedge n)) \leq$ 
     $K * norm (z \wedge n) * inverse (norm (x \wedge n))$ 

```

```

    by (simp add: mult-le-cancel-right x-neq-0)
  qed
  moreover have summable ( $\lambda n. K * \text{norm } (z ^ n) * \text{inverse } (\text{norm } (x ^ n))$ )
  proof -
    from 2 have norm ( $\text{norm } (z * \text{inverse } x)$ ) < 1
    using x-neq-0
    by (simp add: nonzero-norm-divide divide-inverse [symmetric])
    hence summable ( $\lambda n. \text{norm } (z * \text{inverse } x) ^ n$ )
    by (rule summable-geometric)
    hence summable ( $\lambda n. K * \text{norm } (z * \text{inverse } x) ^ n$ )
    by (rule summable-mult)
    thus summable ( $\lambda n. K * \text{norm } (z ^ n) * \text{inverse } (\text{norm } (x ^ n))$ )
    using x-neq-0
    by (simp add: norm-mult nonzero-norm-inverse power-mult-distrib
      power-inverse norm-power mult-assoc)
  qed
  ultimately show summable ( $\lambda n. \text{norm } (f n * z ^ n)$ )
  by (rule summable-comparison-test)
qed

```

**lemma powser-inside:**

```

  fixes f :: nat  $\Rightarrow$  'a::{real-normed-field,banach} shows
    [| summable ( $\%n. f(n) * (x ^ n)$ ); norm z < norm x |]
    ==> summable ( $\%n. f(n) * (z ^ n)$ )
  by (rule powser-insidea [THEN summable-norm-cancel])

```

**lemma sum-split-even-odd:** fixes f :: nat  $\Rightarrow$  real shows

```

  ( $\sum i = 0 ..< 2 * n. \text{if even } i \text{ then } f i \text{ else } g i$ ) =
  ( $\sum i = 0 ..< n. f (2 * i)$ ) + ( $\sum i = 0 ..< n. g (2 * i + 1)$ )
proof (induct n)
  case (Suc n)
  have ( $\sum i = 0 ..< 2 * \text{Suc } n. \text{if even } i \text{ then } f i \text{ else } g i$ ) =
    ( $\sum i = 0 ..< n. f (2 * i)$ ) + ( $\sum i = 0 ..< n. g (2 * i + 1)$ ) + (f (2 *
n) + g (2 * n + 1))
  using Suc.hyps unfolding One-nat-def by auto
  also have ... = ( $\sum i = 0 ..< \text{Suc } n. f (2 * i)$ ) + ( $\sum i = 0 ..< \text{Suc } n. g (2 *
i + 1)$ ) by auto
  finally show ?case .
qed auto

```

**lemma sums-if':** fixes g :: nat  $\Rightarrow$  real assumes g sums x

```

  shows ( $\lambda n. \text{if even } n \text{ then } 0 \text{ else } g ((n - 1) \text{ div } 2)$ ) sums x
  unfolding sums-def
proof (rule LIMSEQ-I)
  fix r :: real assume 0 < r
  from (g sums x)[unfolded sums-def, THEN LIMSEQ-D, OF this]
  obtain no where no-eq:  $\bigwedge n. n \geq \text{no} \implies (\text{norm } (\text{setsum } g \{ 0..<n \} - x) <
r)$  by blast

```

```

let ?SUM = λ m. ∑ i = 0 ..< m. if even i then 0 else g ((i - 1) div 2)
{ fix m assume m ≥ 2 * no hence m div 2 ≥ no by auto
  have sum-eq: ?SUM (2 * (m div 2)) = setsum g { 0 ..< m div 2 }
    using sum-split-even-odd by auto
  hence (norm (?SUM (2 * (m div 2)) - x) < r) using no-eq unfolding sum-eq
using ⟨m div 2 ≥ no⟩ by auto
  moreover
  have ?SUM (2 * (m div 2)) = ?SUM m
  proof (cases even m)
    case True show ?thesis unfolding even-nat-div-two-times-two[OF True,
unfolded numeral-2-eq-2[symmetric]] ..
  next
    case False hence even (Suc m) by auto
    from even-nat-div-two-times-two[OF this, unfolded numeral-2-eq-2[symmetric]]
odd-nat-plus-one-div-two[OF False, unfolded numeral-2-eq-2[symmetric]]
    have eq: Suc (2 * (m div 2)) = m by auto
    hence even (2 * (m div 2)) using ⟨odd m⟩ by auto
    have ?SUM m = ?SUM (Suc (2 * (m div 2))) unfolding eq ..
    also have ... = ?SUM (2 * (m div 2)) using ⟨even (2 * (m div 2))⟩ by auto
    finally show ?thesis by auto
  qed
  ultimately have (norm (?SUM m - x) < r) by auto
}
thus ∃ no. ∀ m ≥ no. norm (?SUM m - x) < r by blast
qed

```

```

lemma sums-if: fixes g :: nat ⇒ real assumes g sums x and f sums y
shows (λ n. if even n then f (n div 2) else g ((n - 1) div 2)) sums (x + y)
proof -
  let ?s = λ n. if even n then 0 else f ((n - 1) div 2)
  { fix B T E have (if B then (0 :: real) else E) + (if B then T else 0) = (if B
then T else E)
    by (cases B) auto } note if-sum = this
  have g-sums: (λ n. if even n then 0 else g ((n - 1) div 2)) sums x using
sums-if'[OF ⟨g sums x⟩] .
  {
    have ?s 0 = 0 by auto
    have Suc-m1: ∧ n. Suc n - 1 = n by auto
    { fix B T E have (if ¬ B then T else E) = (if B then E else T) by auto }
  }
note if-eq = this

```

```

  have ?s sums y using sums-if'[OF ⟨f sums y⟩] .
  from this[unfolded sums-def, THEN LIMSEQ-Suc]
  have (λ n. if even n then f (n div 2) else 0) sums y
    unfolding sums-def setsum-shift-lb-Suc0-0-upt[where f=?s, OF ⟨?s 0 = 0⟩,
symmetric]
    image-Suc-atLeastLessThan[symmetric] setsum-reindex[OF inj-Suc,
unfolded comp-def]
    even-Suc Suc-m1 if-eq .

```

```

} from sums-add[OF g-sums this]
show ?thesis unfolding if-sum .
qed

```

## 76.2 Alternating series test / Leibniz formula

**lemma** *sums-alternating-upper-lower*:

```

fixes a :: nat ⇒ real
assumes mono:  $\bigwedge n. a (Suc\ n) \leq a\ n$  and a-pos:  $\bigwedge n. 0 \leq a\ n$  and a ---->
0
shows  $\exists l. ((\forall n. (\sum_{i=0..2*n} -1^{i*a\ i} \leq l) \wedge (\lambda n. \sum_{i=0..2*n} -1^{i*a\ i}
i) ----> l) \wedge$ 
 $(\forall n. l \leq (\sum_{i=0..2*n+1} -1^{i*a\ i})) \wedge (\lambda n. \sum_{i=0..2*n+1}
-1^{i*a\ i} ----> l)$ 
(is  $\exists l. ((\forall n. ?f\ n \leq l) \wedge -) \wedge ((\forall n. l \leq ?g\ n) \wedge -)$ )
proof -
have fg-diff:  $\bigwedge n. ?f\ n - ?g\ n = - a\ (2 * n)$  unfolding One-nat-def by auto

have  $\forall n. ?f\ n \leq ?f\ (Suc\ n)$ 
proof fix n show  $?f\ n \leq ?f\ (Suc\ n)$  using mono[of 2*n] by auto qed
moreover
have  $\forall n. ?g\ (Suc\ n) \leq ?g\ n$ 
proof fix n show  $?g\ (Suc\ n) \leq ?g\ n$  using mono[of Suc (2*n)]
unfolding One-nat-def by auto qed
moreover
have  $\forall n. ?f\ n \leq ?g\ n$ 
proof fix n show  $?f\ n \leq ?g\ n$  using fg-diff a-pos
unfolding One-nat-def by auto qed
moreover
have  $(\lambda n. ?f\ n - ?g\ n) ----> 0$  unfolding fg-diff
proof (rule LIMSEQ-I)
fix r :: real assume  $0 < r$ 
with  $\langle a ----> 0 \rangle [THEN\ LIMSEQ-D]$ 
obtain N where  $\bigwedge n. n \geq N \implies norm\ (a\ n - 0) < r$  by auto
hence  $\forall n \geq N. norm\ (- a\ (2 * n) - 0) < r$  by auto
thus  $\exists N. \forall n \geq N. norm\ (- a\ (2 * n) - 0) < r$  by auto
qed
ultimately
show ?thesis by (rule lemma-nest-unique)
qed

```

**lemma** *summable-Leibniz'*: fixes a :: nat ⇒ real

```

assumes a-zero: a ----> 0 and a-pos:  $\bigwedge n. 0 \leq a\ n$ 
and a-monotone:  $\bigwedge n. a (Suc\ n) \leq a\ n$ 
shows summable: summable  $(\lambda n. (-1)^{n*a\ n})$ 
and  $\bigwedge n. (\sum_{i=0..2*n} (-1)^{i*a\ i} \leq (\sum i. (-1)^{i*a\ i})$ 
and  $(\lambda n. \sum_{i=0..2*n} (-1)^{i*a\ i} ----> (\sum i. (-1)^{i*a\ i})$ 
and  $\bigwedge n. (\sum i. (-1)^{i*a\ i} \leq (\sum_{i=0..2*n+1} (-1)^{i*a\ i})$ 
and  $(\lambda n. \sum_{i=0..2*n+1} (-1)^{i*a\ i} ----> (\sum i. (-1)^{i*a\ i})$ 

```

**proof** –

**let**  $?S\ n = (-1)^n * a\ n$   
**let**  $?P\ n = \sum_{i=0..<n} ?S\ i$   
**let**  $?f\ n = ?P\ (2 * n)$   
**let**  $?g\ n = ?P\ (2 * n + 1)$   
**obtain**  $l :: \text{real}$  **where** *below-l*:  $\forall\ n. ?f\ n \leq l$  **and** *above-l*:  $\forall\ n. l \leq ?g\ n$  **and**  $?g\ n \rightarrow l$   
**using** *sums-alternating-upper-lower*[*OF a-monotone a-pos a-zero*] **by** *blast*

**let**  $?Sa = \lambda\ m. \sum_{n=0..<m} ?S\ n$   
**have**  $?Sa \rightarrow l$

**proof** (*rule LIMSEQ-I*)

**fix**  $r :: \text{real}$  **assume**  $0 < r$

**with**  $\langle ?f \rightarrow l \rangle$  [*THEN LIMSEQ-D*]

**obtain** *f-no* **where**  $f: \bigwedge n. n \geq \text{f-no} \implies \text{norm } (?f\ n - l) < r$  **by** *auto*

**from**  $\langle 0 < r \rangle$   $\langle ?g \rightarrow l \rangle$  [*THEN LIMSEQ-D*]

**obtain** *g-no* **where**  $g: \bigwedge n. n \geq \text{g-no} \implies \text{norm } (?g\ n - l) < r$  **by** *auto*

{ **fix**  $n :: \text{nat}$

**assume**  $n \geq (\max (2 * \text{f-no}) (2 * \text{g-no}))$  **hence**  $n \geq 2 * \text{f-no}$  **and**  $n \geq 2 * \text{g-no}$

**by** *auto*

**have**  $\text{norm } (?Sa\ n - l) < r$

**proof** (*cases even n*)

**case** *True* **from** *even-nat-div-two-times-two*[*OF this*]

**have** *n-eq*:  $2 * (n \text{ div } 2) = n$  **unfolding** *numeral-2-eq-2*[*symmetric*] **by** *auto*

**with**  $\langle n \geq 2 * \text{f-no} \rangle$  **have**  $n \text{ div } 2 \geq \text{f-no}$  **by** *auto*

**from** *f*[*OF this*]

**show** *?thesis* **unfolding** *n-eq atLeastLessThanSuc-atLeastAtMost* .

**next**

**case** *False* **hence** *even (n - 1)* **by** *simp*

**from** *even-nat-div-two-times-two*[*OF this*]

**have** *n-eq*:  $2 * ((n - 1) \text{ div } 2) = n - 1$  **unfolding** *numeral-2-eq-2*[*symmetric*]

**by** *auto*

**hence** *range-eq*:  $n - 1 + 1 = n$  **using** *odd-pos*[*OF False*] **by** *auto*

**from** *n-eq*  $\langle n \geq 2 * \text{g-no} \rangle$  **have**  $(n - 1) \text{ div } 2 \geq \text{g-no}$  **by** *auto*

**from** *g*[*OF this*]

**show** *?thesis* **unfolding** *n-eq atLeastLessThanSuc-atLeastAtMost range-eq* .

**qed**

}

**thus**  $\exists\ no. \forall\ n \geq no. \text{norm } (?Sa\ n - l) < r$  **by** *blast*

**qed**

**hence** *sums-l*:  $(\lambda i. (-1)^i * a\ i)$  *sums l* **unfolding** *sums-def atLeastLessThanSuc-atLeastAtMost*[*symmetric*]

.

**thus** *summable ?S* **using** *summable-def* **by** *auto*

**have**  $l = \text{suminf } ?S$  **using** *sums-unique*[*OF sums-l*] .



```

{ fix n show suminf ?S ≤ ?g n unfolding sums-unique[OF sums-l, symmetric]
using above-l by auto }
{ fix n show ?f n ≤ suminf ?S unfolding sums-unique[OF sums-l, symmetric]
using below-l by auto }
show ?g -----> suminf ?S using ⟨?g -----> l⟩ ⟨l = suminf ?S⟩ by auto
show ?f -----> suminf ?S using ⟨?f -----> l⟩ ⟨l = suminf ?S⟩ by auto
qed

```

```

theorem summable-Leibniz: fixes a :: nat ⇒ real
assumes a-zero: a -----> 0 and monoseq a
shows summable (λ n. (-1)^n * a n) (is ?summable)
and 0 < a 0 ⟶ (∀ n. (∑ i. -1^i * a i) ∈ { ∑ i=0..<2*n. -1^i * a i ..
∑ i=0..<2*n+1. -1^i * a i }) (is ?pos)
and a 0 < 0 ⟶ (∀ n. (∑ i. -1^i * a i) ∈ { ∑ i=0..<2*n+1. -1^i * a i ..
∑ i=0..<2*n. -1^i * a i }) (is ?neg)
and (λ n. ∑ i=0..<2*n. -1^i * a i) -----> (∑ i. -1^i * a i) (is ?f)
and (λ n. ∑ i=0..<2*n+1. -1^i * a i) -----> (∑ i. -1^i * a i) (is ?g)
proof -
have ?summable ∧ ?pos ∧ ?neg ∧ ?f ∧ ?g
proof (cases (∀ n. 0 ≤ a n) ∧ (∀ m. ∀ n ≥ m. a n ≤ a m))
case True
hence ord: ⋀ n m. m ≤ n ⟹ a n ≤ a m and ge0: ⋀ n. 0 ≤ a n by auto
{ fix n have a (Suc n) ≤ a n using ord[where n=Suc n and m=n] by auto
}
note leibniz = summable-Leibniz'[OF ⟨a -----> 0⟩ ge0] and mono = this
from leibniz[OF mono]
show ?thesis using ⟨0 ≤ a 0⟩ by auto
next
let ?a = λ n. - a n
case False
with monoseq-le[OF ⟨monoseq a⟩ ⟨a -----> 0⟩]
have (∀ n. a n ≤ 0) ∧ (∀ m. ∀ n ≥ m. a m ≤ a n) by auto
hence ord: ⋀ n m. m ≤ n ⟹ ?a n ≤ ?a m and ge0: ⋀ n. 0 ≤ ?a n by auto
{ fix n have ?a (Suc n) ≤ ?a n using ord[where n=Suc n and m=n] by
auto }
note monotone = this
note leibniz = summable-Leibniz'[OF - ge0, of λ x. x, OF LIMSEQ-minus[OF
⟨a -----> 0⟩, unfolded minus-zero] monotone]
have summable (λ n. (-1)^n * ?a n) using leibniz(1) by auto
then obtain l where (λ n. (-1)^n * ?a n) sums l unfolding summable-def
by auto
from this[THEN sums-minus]
have (λ n. (-1)^n * a n) sums -l by auto
hence ?summable unfolding summable-def by auto
moreover
have ⋀ a b :: real. | - a - - b | = | a - b | unfolding minus-diff-minus by
auto

```

```

from suminf-minus[OF leibniz(1), unfolded mult-minus-right minus-minus]
have move-minus:  $(\sum n. - (-1 ^ n * a n)) = - (\sum n. -1 ^ n * a n)$  by auto

  have ?pos using <0 ≤ ?a 0> by auto
  moreover have ?neg using leibniz(2,4) unfolding mult-minus-right setsum-negf
  move-minus neg-le-iff-le by auto
  moreover have ?f and ?g using leibniz(3,5)[unfolded mult-minus-right setsum-negf
  move-minus, THEN LIMSEQ-minus-cancel] by auto
  ultimately show ?thesis by auto
qed
from this[THEN conjunct1] this[THEN conjunct2, THEN conjunct1] this[THEN
  conjunct2, THEN conjunct2, THEN conjunct1] this[THEN conjunct2, THEN con-
  junct2, THEN conjunct2, THEN conjunct1]
  this[THEN conjunct2, THEN conjunct2, THEN conjunct2, THEN conjunct2]
  show ?summable and ?pos and ?neg and ?f and ?g .
qed

```

### 76.3 Term-by-Term Differentiability of Power Series

**definition**

```

diffs :: (nat => 'a::ring-1) => nat => 'a where
diffs c = (%n. of-nat (Suc n) * c(Suc n))

```

Lemma about distributing negation over it

```

lemma diffs-minus:  $\text{diffs } (\%n. - c n) = (\%n. - \text{diffs } c n)$ 
by (simp add: diffs-def)

```

**lemma** sums-Suc-imp:

```

  assumes f:  $f 0 = 0$ 
  shows  $(\lambda n. f (Suc n)) \text{ sums } s \implies (\lambda n. f n) \text{ sums } s$ 
unfolding sums-def
apply (rule LIMSEQ-imp-Suc)
apply (subst setsum-shift-lb-Suc0-0-upt [where f=f, OF f, symmetric])
apply (simp only: setsum-shift-bounds-Suc-ivl)
done

```

**lemma** diffs-equiv:

```

  summable (%n. (diffs c)(n) * (x ^ n)) ==>
  (%n. of-nat n * c(n) * (x ^ (n - Suc 0))) sums
  ( $\sum n. (\text{diffs } c)(n) * (x ^ n)$ )
unfolding diffs-def
apply (drule summable-sums)
apply (rule sums-Suc-imp, simp-all)
done

```

**lemma** lemma-termdiff1:

```

fixes z :: 'a :: {monoid-mult, comm-ring} shows
 $(\sum p=0..<m. (((z + h) ^ (m - p)) * (z ^ p)) - (z ^ m)) =$ 
 $(\sum p=0..<m. (z ^ p) * (((z + h) ^ (m - p)) - (z ^ (m - p))))$ 

```

**by**(*auto simp add: algebra-simps power-add [symmetric] cong: strong-setsum-cong*)

**lemma** *sumr-diff-mult-const2*:

*setsum f {0..*n*} - of-nat *n* \* (r::'a::ring-1) = ( $\sum i = 0..*n*. f i - r)$*   
**by** (*simp add: setsum-subtractf*)

**lemma** *lemma-termdiff2*:

**fixes** *h* :: 'a :: {field}  
**assumes** *h*: *h* ≠ 0 **shows**  
 $((z + h) ^ n - z ^ n) / h - \text{of-nat } n * z ^ (n - \text{Suc } 0) =$   
 $h * (\sum p=0..*n* - \text{Suc } 0. \sum q=0..*n* - \text{Suc } 0 - p.$   
 $(z + h) ^ q * z ^ (n - 2 - q))$  (**is** ?*lhs* = ?*rhs*)  
**apply** (*subgoal-tac h \* ?lhs = h \* ?rhs, simp add: h*)  
**apply** (*simp add: right-diff-distrib diff-divide-distrib h*)  
**apply** (*simp add: mult-assoc [symmetric]*)  
**apply** (*cases n, simp*)  
**apply** (*simp add: lemma-realpow-diff-sumr2 h*  
 $\text{right-diff-distrib [symmetric] mult-assoc}$   
 $\text{del: power-Suc setsum-op-ivl-Suc of-nat-Suc}$ )  
**apply** (*subst lemma-realpow-rev-sumr*)  
**apply** (*subst sumr-diff-mult-const2*)  
**apply** *simp*  
**apply** (*simp only: lemma-termdiff1 setsum-right-distrib*)  
**apply** (*rule setsum-cong [OF refl]*)  
**apply** (*simp add: diff-minus [symmetric] less-iff-Suc-add*)  
**apply** (*clarify*)  
**apply** (*simp add: setsum-right-distrib lemma-realpow-diff-sumr2 mult-ac*  
 $\text{del: setsum-op-ivl-Suc power-Suc}$ )  
**apply** (*subst mult-assoc [symmetric], subst power-add [symmetric]*)  
**apply** (*simp add: mult-ac*)  
**done**

**lemma** *real-setsum-nat-ivl-bounded2*:

**fixes** *K* :: 'a::linordered-semidom  
**assumes** *f*:  $\bigwedge p::\text{nat}. p < n \implies f p \leq K$   
**assumes** *K*:  $0 \leq K$   
**shows**  $\text{setsum } f \{0..*n-k\} \leq \text{of-nat } n * K*$   
**apply** (*rule order-trans [OF setsum-mono]*)  
**apply** (*rule f, simp*)  
**apply** (*simp add: mult-right-mono K*)  
**done**

**lemma** *lemma-termdiff3*:

**fixes** *h z* :: 'a::{real-normed-field}  
**assumes** 1: *h* ≠ 0  
**assumes** 2:  $\text{norm } z \leq K$   
**assumes** 3:  $\text{norm } (z + h) \leq K$   
**shows**  $\text{norm } (((z + h) ^ n - z ^ n) / h - \text{of-nat } n * z ^ (n - \text{Suc } 0))$   
 $\leq \text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * K ^ (n - 2) * \text{norm } h$

**proof** –

**have**  $\text{norm } ((z + h) \wedge^n - z \wedge^n) / h - \text{of-nat } n * z \wedge^{(n - \text{Suc } 0)}) =$   
 $\text{norm } (\sum p = 0..<n - \text{Suc } 0. \sum q = 0..<n - \text{Suc } 0 - p.$   
 $(z + h) \wedge^q * z \wedge^{(n - 2 - q)}) * \text{norm } h$   
**apply** (*subst lemma-termdiff2 [OF 1]*)  
**apply** (*subst norm-mult*)  
**apply** (*rule mult-commute*)  
**done**  
**also have**  $\dots \leq \text{of-nat } n * (\text{of-nat } (n - \text{Suc } 0) * K \wedge^{(n - 2)}) * \text{norm } h$   
**proof** (*rule mult-right-mono [OF - norm-ge-zero]*)  
**from** *norm-ge-zero 2* **have**  $K: 0 \leq K$  **by** (*rule order-trans*)  
**have** *le-Kn*:  $\bigwedge i j n. i + j = n \implies \text{norm } ((z + h) \wedge^i * z \wedge^j) \leq K \wedge^n$   
**apply** (*erule subst*)  
**apply** (*simp only: norm-mult norm-power power-add*)  
**apply** (*intro mult-mono power-mono 2 3 norm-ge-zero zero-le-power K*)  
**done**  
**show**  $\text{norm } (\sum p = 0..<n - \text{Suc } 0. \sum q = 0..<n - \text{Suc } 0 - p.$   
 $(z + h) \wedge^q * z \wedge^{(n - 2 - q)})$   
 $\leq \text{of-nat } n * (\text{of-nat } (n - \text{Suc } 0) * K \wedge^{(n - 2)})$   
**apply** (*intro*  
*order-trans [OF norm-setsum]*  
*real-setsum-nat-ivl-bounded2*  
*mult-nonneg-nonneg*  
*zero-le-imp-of-nat*  
*zero-le-power K*)  
**apply** (*rule le-Kn, simp*)  
**done**  
**qed**  
**also have**  $\dots = \text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * K \wedge^{(n - 2)} * \text{norm } h$   
**by** (*simp only: mult-assoc*)  
**finally show** *?thesis* .

**qed**

**lemma** *lemma-termdiff4*:

**fixes**  $f :: 'a::\{\text{real-normed-field}\} \Rightarrow$   
 $'b::\text{real-normed-vector}$   
**assumes**  $k: 0 < (k::\text{real})$   
**assumes**  $le: \bigwedge h. \llbracket h \neq 0; \text{norm } h < k \rrbracket \implies \text{norm } (f h) \leq K * \text{norm } h$   
**shows**  $f -- 0 --> 0$

**unfolding** *LIM-eq diff-0-right*

**proof** (*safe*)

**let**  $?h = \text{of-real } (k / 2)::'a$   
**have**  $?h \neq 0$  **and**  $\text{norm } ?h < k$  **using**  $k$  **by** *simp-all*  
**hence**  $\text{norm } (f ?h) \leq K * \text{norm } ?h$  **by** (*rule le*)  
**hence**  $0 \leq K * \text{norm } ?h$  **by** (*rule order-trans [OF norm-ge-zero]*)  
**hence**  $\text{zero-le-K}: 0 \leq K$  **using**  $k$  **by** (*simp add: zero-le-mult-iff*)

**fix**  $r::\text{real}$  **assume**  $r: 0 < r$

**show**  $\exists s. 0 < s \wedge (\forall x. x \neq 0 \wedge \text{norm } x < s \longrightarrow \text{norm } (f x) < r)$

```

proof (cases)
  assume  $K = 0$ 
  with  $k\ r\ le$  have  $0 < k \wedge (\forall x. x \neq 0 \wedge \text{norm } x < k \longrightarrow \text{norm } (f\ x) < r)$ 
    by simp
  thus  $\exists s. 0 < s \wedge (\forall x. x \neq 0 \wedge \text{norm } x < s \longrightarrow \text{norm } (f\ x) < r)$  ..
next
  assume  $K\text{-neq-zero}: K \neq 0$ 
  with  $\text{zero-le-}K$  have  $K: 0 < K$  by simp
  show  $\exists s. 0 < s \wedge (\forall x. x \neq 0 \wedge \text{norm } x < s \longrightarrow \text{norm } (f\ x) < r)$ 
    proof (rule exI, safe)
      from  $k\ r\ K$  show  $0 < \min k\ (r * \text{inverse } K / 2)$ 
        by (simp add: mult-pos-pos positive-imp-inverse-positive)
    next
      fix  $x::'a$ 
      assume  $x1: x \neq 0$  and  $x2: \text{norm } x < \min k\ (r * \text{inverse } K / 2)$ 
      from  $x2$  have  $x3: \text{norm } x < k$  and  $x4: \text{norm } x < r * \text{inverse } K / 2$ 
        by simp-all
      from  $x1\ x3\ le$  have  $\text{norm } (f\ x) \leq K * \text{norm } x$  by simp
      also from  $x4\ K$  have  $K * \text{norm } x < K * (r * \text{inverse } K / 2)$ 
        by (rule mult-strict-left-mono)
      also have  $\dots = r / 2$ 
        using  $K\text{-neq-zero}$  by simp
      also have  $r / 2 < r$ 
        using  $r$  by simp
      finally show  $\text{norm } (f\ x) < r$  .
    qed
  qed
qed

lemma lemma-termdiff5:
  fixes  $g :: 'a::\{\text{real-normed-field}\} \Rightarrow$ 
     $\text{nat} \Rightarrow 'b::\text{banach}$ 
  assumes  $k: 0 < (k::\text{real})$ 
  assumes  $f: \text{summable } f$ 
  assumes  $le: \bigwedge h\ n. \llbracket h \neq 0; \text{norm } h < k \rrbracket \Longrightarrow \text{norm } (g\ h\ n) \leq f\ n * \text{norm } h$ 
  shows  $(\lambda h. \text{suminf } (g\ h)) \dashv\dashv 0 \dashv\dashv> 0$ 
proof (rule lemma-termdiff4 [OF k])
  fix  $h::'a$  assume  $h \neq 0$  and  $\text{norm } h < k$ 
  hence  $A: \forall n. \text{norm } (g\ h\ n) \leq f\ n * \text{norm } h$ 
    by (simp add: le)
  hence  $\exists N. \forall n \geq N. \text{norm } (\text{norm } (g\ h\ n)) \leq f\ n * \text{norm } h$ 
    by simp
  moreover from  $f$  have  $B: \text{summable } (\lambda n. f\ n * \text{norm } h)$ 
    by (rule summable-mult2)
  ultimately have  $C: \text{summable } (\lambda n. \text{norm } (g\ h\ n))$ 
    by (rule summable-comparison-test)
  hence  $\text{norm } (\text{suminf } (g\ h)) \leq (\sum n. \text{norm } (g\ h\ n))$ 
    by (rule summable-norm)
  also from  $A\ C\ B$  have  $(\sum n. \text{norm } (g\ h\ n)) \leq (\sum n. f\ n * \text{norm } h)$ 

```

by (rule summable-le)  
 also from  $f$  have  $(\sum n. f\ n * \text{norm } h) = \text{suminf } f * \text{norm } h$   
 by (rule suminf-mult2 [symmetric])  
 finally show  $\text{norm } (\text{suminf } (g\ h)) \leq \text{suminf } f * \text{norm } h$  .  
 qed

FIXME: Long proofs

lemma termdiffs-aux:

fixes  $x :: 'a :: \{\text{real-normed-field}, \text{banach}\}$   
 assumes 1: summable  $(\lambda n. \text{diffs } (\text{diffs } c)\ n * K ^ n)$   
 assumes 2:  $\text{norm } x < \text{norm } K$   
 shows  $(\lambda h. \sum n. c\ n * (((x + h) ^ n - x ^ n) / h - \text{of-nat } n * x ^ (n - \text{Suc } 0))) \rightarrow 0 \rightarrow 0$   
 proof -  
 from dense [OF 2]  
 obtain  $r$  where  $r1: \text{norm } x < r$  and  $r2: r < \text{norm } K$  by fast  
 from norm-ge-zero  $r1$  have  $r: 0 < r$   
 by (rule order-le-less-trans)  
 hence  $r\text{-neq-0}: r \neq 0$  by simp  
 show ?thesis  
 proof (rule lemma-termdiff5)  
 show  $0 < r - \text{norm } x$  using  $r1$  by simp  
 next  
 from  $r\ r2$  have  $\text{norm } (\text{of-real } r :: 'a) < \text{norm } K$   
 by simp  
 with 1 have summable  $(\lambda n. \text{norm } (\text{diffs } (\text{diffs } c)\ n * (\text{of-real } r ^ n)))$   
 by (rule powser-insidea)  
 hence summable  $(\lambda n. \text{diffs } (\text{diffs } (\lambda n. \text{norm } (c\ n)))\ n * r ^ n)$   
 using  $r$   
 by (simp add: diffs-def norm-mult norm-power del: of-nat-Suc)  
 hence summable  $(\lambda n. \text{of-nat } n * \text{diffs } (\lambda n. \text{norm } (c\ n))\ n * r ^ (n - \text{Suc } 0))$   
 by (rule diffs-equiv [THEN sums-summable])  
 also have  $(\lambda n. \text{of-nat } n * \text{diffs } (\lambda n. \text{norm } (c\ n))\ n * r ^ (n - \text{Suc } 0))$   
 =  $(\lambda n. \text{diffs } (\%m. \text{of-nat } (m - \text{Suc } 0) * \text{norm } (c\ m) * \text{inverse } r)\ n * (r ^ n))$   
 apply (rule ext)  
 apply (simp add: diffs-def)  
 apply (case-tac  $n$ , simp-all add:  $r\text{-neq-0}$ )  
 done  
 finally have summable  
 $(\lambda n. \text{of-nat } n * (\text{of-nat } (n - \text{Suc } 0) * \text{norm } (c\ n) * \text{inverse } r) * r ^ (n - \text{Suc } 0))$   
 by (rule diffs-equiv [THEN sums-summable])  
 also have  
 $(\lambda n. \text{of-nat } n * (\text{of-nat } (n - \text{Suc } 0) * \text{norm } (c\ n) * \text{inverse } r) * r ^ (n - \text{Suc } 0)) =$   
 $(\lambda n. \text{norm } (c\ n) * \text{of-nat } n * \text{of-nat } (n - \text{Suc } 0) * r ^ (n - 2))$   
 apply (rule ext)  
 apply (case-tac  $n$ , simp)

```

    apply (case-tac nat, simp)
    apply (simp add: r-neq-0)
    done
  finally show
    summable (λn. norm (c n) * of-nat n * of-nat (n - Suc 0) * r ^ (n - 2)) .
next
  fix h::'a and n::nat
  assume h: h ≠ 0
  assume norm h < r - norm x
  hence norm x + norm h < r by simp
  with norm-triangle-ineq have xh: norm (x + h) < r
    by (rule order-le-less-trans)
  show norm (c n * ((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0)))
    ≤ norm (c n) * of-nat n * of-nat (n - Suc 0) * r ^ (n - 2) * norm h
    apply (simp only: norm-mult mult-assoc)
    apply (rule mult-left-mono [OF - norm-ge-zero])
    apply (simp (no-asm) add: mult-assoc [symmetric])
    apply (rule lemma-termdiff3)
    apply (rule h)
    apply (rule r1 [THEN order-less-imp-le])
    apply (rule xh [THEN order-less-imp-le])
    done
qed
qed

lemma termdiffs:
  fixes K x :: 'a::{real-normed-field,banach}
  assumes 1: summable (λn. c n * K ^ n)
  assumes 2: summable (λn. (diffs c) n * K ^ n)
  assumes 3: summable (λn. (diffs (diffs c)) n * K ^ n)
  assumes 4: norm x < norm K
  shows DERIV (λx. ∑ n. c n * x ^ n) x :> (∑ n. (diffs c) n * x ^ n)
unfolding deriv-def
proof (rule LIM-zero-cancel)
  show (λh. (suminf (λn. c n * (x + h) ^ n) - suminf (λn. c n * x ^ n)) / h
    - suminf (λn. diffs c n * x ^ n)) -- 0 --> 0
proof (rule LIM-equal2)
  show 0 < norm K - norm x using 4 by (simp add: less-diff-eq)
next
  fix h :: 'a
  assume h ≠ 0
  assume norm (h - 0) < norm K - norm x
  hence norm x + norm h < norm K by simp
  hence 5: norm (x + h) < norm K
    by (rule norm-triangle-ineq [THEN order-le-less-trans])
  have A: summable (λn. c n * x ^ n)
    by (rule powser-inside [OF 1 4])
  have B: summable (λn. c n * (x + h) ^ n)
    by (rule powser-inside [OF 1 5])

```

```

have C: summable (λn. diffs c n * x ^ n)
  by (rule powser-inside [OF 2 4])
show ((∑ n. c n * (x + h) ^ n) - (∑ n. c n * x ^ n)) / h
  - (∑ n. diffs c n * x ^ n) =
  (∑ n. c n * (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0)))
  apply (subst sums-unique [OF diffs-equiv [OF C]])
  apply (subst suminf-diff [OF B A])
  apply (subst suminf-divide [symmetric])
  apply (rule summable-diff [OF B A])
  apply (subst suminf-diff)
  apply (rule summable-divide)
  apply (rule summable-diff [OF B A])
  apply (rule sums-summable [OF diffs-equiv [OF C]])
  apply (rule arg-cong [where f=suminf], rule ext)
  apply (simp add: algebra-simps)
done
next
show (λh. ∑ n. c n * (((x + h) ^ n - x ^ n) / h -
  of-nat n * x ^ (n - Suc 0))) -- 0 --> 0
  by (rule termdiffs-aux [OF 3 4])
qed
qed

```

## 76.4 Some properties of factorials

**lemma** *real-of-nat-fact-not-zero* [simp]: *real (fact (n::nat)) ≠ 0*  
**by** *auto*

**lemma** *real-of-nat-fact-gt-zero* [simp]: *0 < real(fact (n::nat))*  
**by** *auto*

**lemma** *real-of-nat-fact-ge-zero* [simp]: *0 ≤ real(fact (n::nat))*  
**by** *simp*

**lemma** *inv-real-of-nat-fact-gt-zero* [simp]: *0 < inverse (real (fact (n::nat)))*  
**by** (*auto simp add: positive-imp-inverse-positive*)

**lemma** *inv-real-of-nat-fact-ge-zero* [simp]: *0 ≤ inverse (real (fact (n::nat)))*  
**by** (*auto intro: order-less-imp-le*)

## 76.5 Derivability of power series

**lemma** *DERIV-series'*: **fixes** *f :: real ⇒ nat ⇒ real*  
**assumes** *DERIV-f*:  $\bigwedge n. \text{DERIV } (\lambda x. f x n) x0 :> (f' x0 n)$   
**and** *allf-summable*:  $\bigwedge x. x \in \{a <..< b\} \implies \text{summable } (f x)$  **and** *x0-in-I*:  $x0 \in \{a <..< b\}$   
**and** *summable (f' x0)*  
**and** *summable L and L-def*:  $\bigwedge n x y. \llbracket x \in \{a <..< b\}; y \in \{a <..< b\} \rrbracket$   
 $\implies |f x n - f y n| \leq L n * |x - y|$   
**shows** *DERIV*  $(\lambda x. \text{suminf } (f x)) x0 :> (\text{suminf } (f' x0))$



**unfolding** *deriv-def*  
**proof** (*rule LIM-I*)  
**fix**  $r :: \text{real}$  **assume**  $0 < r$  **hence**  $0 < r/3$  **by** *auto*  
  
**obtain**  $N-L$  **where**  $N-L: \bigwedge n. N-L \leq n \implies |\sum i. L(i+n)| < r/3$   
**using** *suminf-exist-split*[*OF*  $\langle 0 < r/3 \rangle$  *summable*  $L$ ] **by** *auto*  
  
**obtain**  $N-f'$  **where**  $N-f': \bigwedge n. N-f' \leq n \implies |\sum i. f' x0(i+n)| < r/3$   
**using** *suminf-exist-split*[*OF*  $\langle 0 < r/3 \rangle$  *summable*  $(f' x0)$ ] **by** *auto*  
  
**let**  $?N = \text{Suc}(\max N-L N-f')$   
**have**  $|\sum i. f' x0(i + ?N)| < r/3$  (**is**  $?f'\text{-part} < r/3$ ) **and**  
 $L\text{-estimate}: |\sum i. L(i + ?N)| < r/3$  **using**  $N-L[\text{of } ?N]$  **and**  $N-f'[\text{of } ?N]$  **by**  
*auto*  
  
**let**  $?diff\ i\ x = (f(x0 + x)\ i - f\ x0\ i) / x$   
  
**let**  $?r = r / (3 * \text{real } ?N)$   
**have**  $0 < 3 * \text{real } ?N$  **by** *auto*  
**from** *divide-pos-pos*[*OF*  $\langle 0 < r \rangle$  *this*]  
**have**  $0 < ?r$  .  
  
**let**  $?s\ n = \text{SOME } s. 0 < s \wedge (\forall x. x \neq 0 \wedge |x| < s \longrightarrow |?diff\ n\ x - f' x0\ n| < ?r)$   
**def**  $S' \equiv \text{Min} (?s ' \{ 0 ..< ?N \})$   
  
**have**  $0 < S'$  **unfolding**  $S'\text{-def}$   
**proof** (*rule iffD2*[*OF*  $\text{Min-gr-iff}$ ])  
**show**  $\forall x \in (?s ' \{ 0 ..< ?N \}). 0 < x$   
**proof** (*rule ballI*)  
**fix**  $x$  **assume**  $x \in ?s ' \{ 0 ..< ?N \}$   
**then obtain**  $n$  **where**  $x = ?s\ n$  **and**  $n \in \{ 0 ..< ?N \}$  **using** *image-iff*[*THEN* *iffD1*] **by** *blast*  
**from** *DERIV-D*[*OF* *DERIV-f*[**where**  $n=n$ ], *THEN* *LIM-D*, *OF*  $\langle 0 < ?r \rangle$ , *unfolded real-norm-def*]  
**obtain**  $s$  **where**  $s\text{-bound}: 0 < s \wedge (\forall x. x \neq 0 \wedge |x| < s \longrightarrow |?diff\ n\ x - f' x0\ n| < ?r)$  **by** *auto*  
**have**  $0 < ?s\ n$  **by** (*rule someI2*[**where**  $a=s$ ], *auto simp add: s-bound*)  
**thus**  $0 < x$  **unfolding**  $(x = ?s\ n)$  .  
**qed**  
**qed** *auto*  
  
**def**  $S \equiv \min(\min(x0 - a)(b - x0))\ S'$   
**hence**  $0 < S$  **and**  $S\text{-a}: S \leq x0 - a$  **and**  $S\text{-b}: S \leq b - x0$  **and**  $S \leq S'$  **using**  
 $x0\text{-in-I}$  **and**  $\langle 0 < S' \rangle$   
**by** *auto*  
  
**{ fix**  $x$  **assume**  $x \neq 0$  **and**  $|x| < S$   
**hence**  $x\text{-in-I}: x0 + x \in \{ a <..< b \}$  **using**  $S\text{-a}$   $S\text{-b}$  **by** *auto*

**note**  $\text{diff-smb1} = \text{summable-diff}[OF \text{ allf-summable}[OF x\text{-in-}I] \text{ allf-summable}[OF x0\text{-in-}I]]$   
**note**  $\text{div-smb1} = \text{summable-divide}[OF \text{ diff-smb1}]$   
**note**  $\text{all-smb1} = \text{summable-diff}[OF \text{ div-smb1 } \langle \text{summable } (f' x0) \rangle]$   
**note**  $\text{ign} = \text{summable-ignore-initial-segment}[\text{where } k=?N]$   
**note**  $\text{diff-shft-smb1} = \text{summable-diff}[OF \text{ ign}[OF \text{ allf-summable}[OF x\text{-in-}I]] \text{ ign}[OF \text{ allf-summable}[OF x0\text{-in-}I]]]$   
**note**  $\text{div-shft-smb1} = \text{summable-divide}[OF \text{ diff-shft-smb1}]$   
**note**  $\text{all-shft-smb1} = \text{summable-diff}[OF \text{ div-smb1 } \text{ign}[OF \langle \text{summable } (f' x0) \rangle]]$

**{ fix }  $n$**   
**have**  $| ?\text{diff } (n + ?N) x | \leq L (n + ?N) * | (x0 + x) - x0 | / | x |$   
**using**  $\text{divide-right-mono}[OF L\text{-def}[OF x\text{-in-}I x0\text{-in-}I] \text{ abs-ge-zero}]$  **unfolding**  $\text{abs-divide}$  .  
**hence**  $| ( | ?\text{diff } (n + ?N) x | ) | \leq L (n + ?N)$  **using**  $\langle x \neq 0 \rangle$  **by**  $\text{auto}$   
**} note**  $L\text{-ge} = \text{summable-le2}[OF \text{ allI}[OF \text{ this}] \text{ ign}[OF \langle \text{summable } L \rangle]]$   
**from**  $\text{order-trans}[OF \text{ summable-rabs}[OF \text{ conjunct1}[OF L\text{-ge}]] L\text{-ge}[THEN \text{ conjunct2}]]$   
**have**  $| \sum i. ?\text{diff } (i + ?N) x | \leq (\sum i. L (i + ?N))$  .  
**hence**  $| \sum i. ?\text{diff } (i + ?N) x | \leq r / 3$  **(is**  $?L\text{-part} \leq r/3$ ) **using**  $L\text{-estimate}$  **by**  $\text{auto}$

**have**  $| \sum n \in \{ 0 ..< ?N \}. ?\text{diff } n x - f' x0 n | \leq (\sum n \in \{ 0 ..< ?N \}. | ?\text{diff } n x - f' x0 n |)$  ..  
**also have**  $\dots < (\sum n \in \{ 0 ..< ?N \}. ?r)$   
**proof**  $(\text{rule setsum-strict-mono})$   
**fix**  $n$  **assume**  $n \in \{ 0 ..< ?N \}$   
**have**  $| x | < S$  **using**  $\langle | x | < S \rangle$  .  
**also have**  $S \leq S'$  **using**  $\langle S \leq S' \rangle$  .  
**also have**  $S' \leq ?s n$  **unfolding**  $S'\text{-def}$   
**proof**  $(\text{rule Min-le-iff}[THEN \text{ iffD2}])$   
**have**  $?s n \in (?s ' \{ 0..<?N \}) \wedge ?s n \leq ?s n$  **using**  $\langle n \in \{ 0 ..< ?N \} \rangle$  **by**  $\text{auto}$   
**thus**  $\exists a \in (?s ' \{ 0..<?N \}). a \leq ?s n$  **by**  $\text{blast}$   
**qed**  $\text{auto}$   
**finally have**  $| x | < ?s n$  .

**from**  $\text{DERIV-D}[OF \text{ DERIV-f}[\text{where } n=n], \text{ THEN LIM-D}, OF \langle 0 < ?r \rangle,$   
 $\text{unfolded real-norm-def diff-0-right}, \text{ unfolded some-eq-ex[symmetric]}, \text{ THEN conjunct2}]$   
**have**  $\forall x. x \neq 0 \wedge |x| < ?s n \longrightarrow | ?\text{diff } n x - f' x0 n | < ?r$  .  
**with**  $\langle x \neq 0 \rangle$  **and**  $\langle |x| < ?s n \rangle$   
**show**  $| ?\text{diff } n x - f' x0 n | < ?r$  **by**  $\text{blast}$   
**qed**  $\text{auto}$   
**also have**  $\dots = \text{of-nat } (\text{card } \{ 0 ..< ?N \}) * ?r$  **by**  $(\text{rule setsum-constant})$   
**also have**  $\dots = \text{real } ?N * ?r$  **unfolding**  $\text{real-eq-of-nat}$  **by**  $\text{auto}$   
**also have**  $\dots = r/3$  **by**  $\text{auto}$   
**finally have**  $| \sum n \in \{ 0 ..< ?N \}. ?\text{diff } n x - f' x0 n | < r / 3$  **(is**  $?diff\text{-part}$

$< r / 3$ ).

**from** *suminf-diff*[*OF allf-summable*[*OF x-in-I*] *allf-summable*[*OF x0-in-I*]]  
**have**  $| (\text{suminf } (f \ (x0 + x)) - (\text{suminf } (f \ x0))) / x - \text{suminf } (f' \ x0) | =$   
 $| \sum n. ?diff \ n \ x - f' \ x0 \ n |$  **unfolding** *suminf-diff*[*OF div-smb1*  
 $\langle \text{summable } (f' \ x0) \rangle, \text{symmetric}$ ] **using** *suminf-divide*[*OF diff-smb1, symmetric*] **by**  
*auto*  
**also have**  $\dots \leq ?diff\text{-part} + | (\sum n. ?diff \ (n + ?N) \ x) - (\sum n. f' \ x0 \ (n + ?N))$   
 $|$  **unfolding** *suminf-split-initial-segment*[*OF all-smb1, where k=?N*] **unfolding**  
*suminf-diff*[*OF div-shft-smb1 ign*[*OF*  $\langle \text{summable } (f' \ x0) \rangle$ ]] **by** (*rule abs-triangle-ineq*)  
**also have**  $\dots \leq ?diff\text{-part} + ?L\text{-part} + ?f'\text{-part}$  **using** *abs-triangle-ineq4* **by**  
*auto*  
**also have**  $\dots < r / 3 + r / 3 + r / 3$   
**using**  $\langle ?diff\text{-part} < r / 3 \rangle \langle ?L\text{-part} \leq r / 3 \rangle$  **and**  $\langle ?f'\text{-part} < r / 3 \rangle$   
**by** (*rule add-strict-mono* [*OF add-less-le-mono*])  
**finally have**  $| (\text{suminf } (f \ (x0 + x)) - (\text{suminf } (f \ x0))) / x - \text{suminf } (f' \ x0)$   
 $| < r$   
**by** *auto*  
**} thus**  $\exists s > 0. \forall x. x \neq 0 \wedge \text{norm } (x - 0) < s \longrightarrow$   
 $\text{norm } (((\sum n. f \ (x0 + x) \ n) - (\sum n. f \ x0 \ n)) / x - (\sum n. f' \ x0 \ n)) < r$   
**using**  $\langle 0 < S \rangle$   
**unfolding** *real-norm-def diff-0-right* **by** *blast*  
**qed**

**lemma** *DERIV-power-series'*: **fixes**  $f :: \text{nat} \Rightarrow \text{real}$

**assumes** *converges*:  $\bigwedge x. x \in \{-R <..< R\} \implies \text{summable } (\lambda n. f \ n * \text{real } (\text{Suc } n) * x^n)$

**and** *x0-in-I*:  $x0 \in \{-R <..< R\}$  **and**  $0 < R$

**shows** *DERIV*  $(\lambda x. (\sum n. f \ n * x^n (\text{Suc } n))) \ x0 :> (\sum n. f \ n * \text{real } (\text{Suc } n) * x0^n)$

(**is** *DERIV*  $(\lambda x. (\text{suminf } (?f \ x))) \ x0 :> (\text{suminf } (?f' \ x0)))$ )

**proof** –

**{ fix**  $R'$  **assume**  $0 < R'$  **and**  $R' < R$  **and**  $-R' < x0$  **and**  $x0 < R'$

**hence**  $x0 \in \{-R' <..< R'\}$  **and**  $R' \in \{-R <..< R\}$  **and**  $x0 \in \{-R <..< R\}$

**by** *auto*

**have** *DERIV*  $(\lambda x. (\text{suminf } (?f \ x))) \ x0 :> (\text{suminf } (?f' \ x0))$

**proof** (*rule DERIV-series'*)

**show** *summable*  $(\lambda n. |f \ n * \text{real } (\text{Suc } n) * R'^n|)$

**proof** –

**have**  $(R' + R) / 2 < R$  **and**  $0 < (R' + R) / 2$  **using**  $\langle 0 < R' \rangle \langle 0 < R \rangle$   
 $\langle R' < R \rangle$  **by** *auto*

**hence** *in-Rball*:  $(R' + R) / 2 \in \{-R <..< R\}$  **using**  $\langle R' < R \rangle$  **by** *auto*

**have**  $\text{norm } R' < \text{norm } ((R' + R) / 2)$  **using**  $\langle 0 < R' \rangle \langle 0 < R \rangle \langle R' < R \rangle$

**by** *auto*

**from** *powser-insidea*[*OF converges*[*OF in-Rball*] *this*] **show** *?thesis* **by** *auto*

**qed**

**{ fix**  $n \ x \ y$  **assume**  $x \in \{-R' <..< R'\}$  **and**  $y \in \{-R' <..< R'\}$

**show**  $|?f \ x \ n - ?f \ y \ n| \leq |f \ n * \text{real } (\text{Suc } n) * R'^n| * |x - y|$

**proof** –

**have**  $|f\ n * x^{\wedge} (Suc\ n) - f\ n * y^{\wedge} (Suc\ n)| = (|f\ n| * |x - y|) * |\sum p = 0..<Suc\ n. x^{\wedge} p * y^{\wedge} (n - p)|$   
**unfolding** *right-diff-distrib[symmetric] lemma-realpow-diff-sumr2 abs-mult*  
**by** *auto*  
**also have**  $\dots \leq (|f\ n| * |x - y|) * (|real\ (Suc\ n)| * |R'^{\wedge} n|)$   
**proof** (*rule mult-left-mono*)  
**have**  $|\sum p = 0..<Suc\ n. x^{\wedge} p * y^{\wedge} (n - p)| \leq (\sum p = 0..<Suc\ n. |x^{\wedge} p * y^{\wedge} (n - p)|)$  **by** (*rule setsum-abs*)  
**also have**  $\dots \leq (\sum p = 0..<Suc\ n. R'^{\wedge} n)$   
**proof** (*rule setsum-mono*)  
**fix**  $p$  **assume**  $p \in \{0..<Suc\ n\}$  **hence**  $p \leq n$  **by** *auto*  
**{ fix**  $n$  **fix**  $x :: real$  **assume**  $x \in \{-R' <..< R'\}$   
**hence**  $|x| \leq R'$  **by** *auto*  
**hence**  $|x^{\wedge} n| \leq R'^{\wedge} n$  **unfolding** *power-abs* **by** (*rule power-mono, auto*)  
**}** **from** *mult-mono[OF this[OF  $x \in \{-R' <..< R'\}$ , of  $p$ ] this[OF  $y \in \{-R' <..< R'\}$ , of  $n - p$ ]]*  $\langle 0 < R' \rangle$   
**have**  $|x^{\wedge} p * y^{\wedge} (n - p)| \leq R'^{\wedge} p * R'^{\wedge} (n - p)$  **unfolding** *abs-mult* **by** *auto*  
**thus**  $|x^{\wedge} p * y^{\wedge} (n - p)| \leq R'^{\wedge} n$  **unfolding** *power-add[symmetric]* **using**  $\langle p \leq n \rangle$  **by** *auto*  
**qed**  
**also have**  $\dots = real\ (Suc\ n) * R'^{\wedge} n$  **unfolding** *setsum-constant card-atLeastLessThan real-of-nat-def* **by** *auto*  
**finally show**  $|\sum p = 0..<Suc\ n. x^{\wedge} p * y^{\wedge} (n - p)| \leq |real\ (Suc\ n)| * |R'^{\wedge} n|$  **unfolding** *abs-real-of-nat-cancel abs-of-nonneg[OF zero-le-power[OF less-imp-le[OF  $0 < R'$ ]]]* .  
**show**  $0 \leq |f\ n| * |x - y|$  **unfolding** *abs-mult[symmetric]* **by** *auto*  
**qed**  
**also have**  $\dots = |f\ n * real\ (Suc\ n) * R'^{\wedge} n| * |x - y|$  **unfolding** *abs-mult mult-assoc[symmetric]* **by** *algebra*  
**finally show** *?thesis* .  
**qed** }  
**{ fix**  $n$  **show** *DERIV*  $(\lambda x. ?f\ x\ n)\ x0 :> (?f'\ x0\ n)$   
**by** (*auto intro!: DERIV-intros simp del: power-Suc*) }  
**{ fix**  $x$  **assume**  $x \in \{-R' <..< R'\}$  **hence**  $R' \in \{-R <..< R\}$  **and**  $norm\ x < norm\ R'$  **using** *assms*  $\langle R' < R \rangle$  **by** *auto*  
**have** *summable*  $(\lambda n. f\ n * x^{\wedge} n)$   
**proof** (*rule summable-le2[THEN conjunct1, OF - powser-insidea[OF converges[OF  $R' \in \{-R <..< R\}$ ]]*  $\langle norm\ x < norm\ R' \rangle$ , *rule allI*)  
**fix**  $n$   
**have** *le*:  $|f\ n| * 1 \leq |f\ n| * real\ (Suc\ n)$  **by** (*rule mult-left-mono, auto*)  
**show**  $|f\ n * x^{\wedge} n| \leq norm\ (f\ n * real\ (Suc\ n) * x^{\wedge} n)$  **unfolding** *real-norm-def abs-mult*  
**by** (*rule mult-right-mono, auto simp add: le[unfolding mult-1-right]*)  
**qed**  
**from** *this[THEN summable-mult2[where  $c=x$ ], unfolded mult-assoc, unfolded mult-commute]*  
**show** *summable*  $(?f\ x)$  **by** *auto* }  
**show** *summable*  $(?f'\ x0)$  **using** *converges[OF  $x0 \in \{-R <..< R\}$ ]* .

```

    show  $x0 \in \{-R' <..< R'\}$  using  $\langle x0 \in \{-R' <..< R'\} \rangle$  .
  qed
} note for-subinterval = this
let ?R = (R + |x0|) / 2
have  $|x0| < ?R$  using assms by auto
hence  $- ?R < x0$ 
proof (cases  $x0 < 0$ )
  case True
    hence  $- x0 < ?R$  using  $\langle |x0| < ?R \rangle$  by auto
    thus ?thesis unfolding neg-less-iff-less[symmetric, of - x0] by auto
  next
    case False
    have  $- ?R < 0$  using assms by auto
    also have  $\dots \leq x0$  using False by auto
    finally show ?thesis .
  qed
hence  $0 < ?R$   $?R < R - ?R < x0$  and  $x0 < ?R$  using assms by auto
from for-subinterval[OF this]
show ?thesis .
qed

```

## 76.6 Exponential Function

**definition**

$exp :: 'a \Rightarrow 'a::\{real-normed-field,banach\}$  **where**  
 $exp\ x = (\sum n. x ^ n /_R real\ (fact\ n))$

**lemma** *summable-exp-generic*:

**fixes**  $x :: 'a::\{real-normed-algebra-1,banach\}$   
**defines** *S-def*:  $S \equiv \lambda n. x ^ n /_R real\ (fact\ n)$   
**shows** *summable S*

**proof** –

```

have S-Suc:  $\bigwedge n. S\ (Suc\ n) = (x * S\ n) /_R real\ (Suc\ n)$ 
  unfolding S-def by (simp del: mult-Suc)
obtain  $r :: real$  where  $r0: 0 < r$  and  $r1: r < 1$ 
  using dense [OF zero-less-one] by fast
obtain  $N :: nat$  where  $N: norm\ x < real\ N * r$ 
  using reals-Archimedean3 [OF  $r0$ ] by fast
from  $r1$  show ?thesis
proof (rule ratio-test [rule-format])
  fix  $n :: nat$ 
  assume  $n: N \leq n$ 
  have  $norm\ x \leq real\ N * r$ 
    using  $N$  by (rule order-less-imp-le)
  also have  $real\ N * r \leq real\ (Suc\ n) * r$ 
    using  $r0\ n$  by (simp add: mult-right-mono)
  finally have  $norm\ x * norm\ (S\ n) \leq real\ (Suc\ n) * r * norm\ (S\ n)$ 
    using norm-ge-zero by (rule mult-right-mono)
  hence  $norm\ (x * S\ n) \leq real\ (Suc\ n) * r * norm\ (S\ n)$ 

```

```

    by (rule order-trans [OF norm-mult-ineq])
  hence norm (x * S n) / real (Suc n) ≤ r * norm (S n)
    by (simp add: pos-divide-le-eq mult-ac)
  thus norm (S (Suc n)) ≤ r * norm (S n)
    by (simp add: S-Suc inverse-eq-divide)
qed
qed

```

**lemma** *summable-norm-exp*:

```

  fixes x :: 'a::{real-normed-algebra-1,banach}
  shows summable (λn. norm (x ^ n /R real (fact n)))
proof (rule summable-norm-comparison-test [OF exI, rule-format])
  show summable (λn. norm x ^ n /R real (fact n))
    by (rule summable-exp-generic)
next

```

```

  fix n show norm (x ^ n /R real (fact n)) ≤ norm x ^ n /R real (fact n)
    by (simp add: norm-power-ineq)
qed

```

**lemma** *summable-exp*:  $\text{summable } (\%n. \text{inverse } (\text{real } (\text{fact } n)) * x ^ n)$   
**by** (insert *summable-exp-generic* [where  $x=x$ ], *simp*)

**lemma** *exp-converges*:  $(\lambda n. x ^ n /<sub>R</sub> \text{real } (\text{fact } n)) \text{ sums exp } x$   
**unfolding** *exp-def* **by** (rule *summable-exp-generic* [THEN *summable-sums*])

**lemma** *exp-fdiffs*:

```

  diffs (%n. inverse(real (fact n))) = (%n. inverse(real (fact n)))
by (simp add: diffs-def mult-assoc [symmetric] real-of-nat-def of-nat-mult
    del: mult-Suc of-nat-Suc)

```

**lemma** *diffs-of-real*:  $\text{diffs } (\lambda n. \text{of-real } (f\ n)) = (\lambda n. \text{of-real } (\text{diffs } f\ n))$   
**by** (simp add: *diffs-def*)

**lemma** *lemma-exp-ext*:  $\text{exp} = (\lambda x. \sum n. x ^ n /<sub>R</sub> \text{real } (\text{fact } n))$   
**by** (auto intro!: *ext simp add: exp-def*)

**lemma** *DERIV-exp* [*simp*]: *DERIV* *exp*  $x \Rightarrow \text{exp}(x)$

```

apply (simp add: exp-def)
apply (subst lemma-exp-ext)
apply (subgoal-tac DERIV (λu. ∑ n. of-real (inverse (real (fact n))) * u ^ n) x
  :> (∑ n. diffs (λn. of-real (inverse (real (fact n)))) n * x ^ n))
apply (rule-tac [2]  $K = \text{of-real } (1 + \text{norm } x)$  in termdiffs)
apply (simp-all only: diffs-of-real scaleR-conv-of-real exp-fdiffs)
apply (rule exp-converges [THEN sums-summable, unfolded scaleR-conv-of-real])
apply (simp del: of-real-add)
done

```

**lemma** *isCont-exp* [*simp*]: *isCont* *exp*  $x$

by (rule *DERIV-exp* [THEN *DERIV-isCont*])

### 76.6.1 Properties of the Exponential Function

**lemma** *powser-zero*:

fixes  $f :: \text{nat} \Rightarrow 'a::\{\text{real-normed-algebra-1}\}$

shows  $(\sum n. f\ n * 0 \wedge n) = f\ 0$

**proof** –

have  $(\sum n = 0..<1. f\ n * 0 \wedge n) = (\sum n. f\ n * 0 \wedge n)$

by (rule *sums-unique* [*OF series-zero*], *simp add: power-0-left*)

thus ?thesis **unfolding** *One-nat-def* **by** *simp*

**qed**

**lemma** *exp-zero* [*simp*]:  $\exp\ 0 = 1$

**unfolding** *exp-def* **by** (*simp add: scaleR-conv-of-real powser-zero*)

**lemma** *setsum-cl-ivl-Suc2*:

$(\sum i=m..Suc\ n. f\ i) = (\text{if } Suc\ n < m \text{ then } 0 \text{ else } f\ m + (\sum i=m..n. f\ (Suc\ i)))$

**by** (*simp add: setsum-head-Suc setsum-shift-bounds-cl-Suc-ivl*

*del: setsum-cl-ivl-Suc*)

**lemma** *exp-series-add*:

fixes  $x\ y :: 'a::\{\text{real-field}\}$

defines *S-def*:  $S \equiv \lambda x\ n. x \wedge n /_R \text{real } (fact\ n)$

shows  $S\ (x + y)\ n = (\sum i=0..n. S\ x\ i * S\ y\ (n - i))$

**proof** (*induct n*)

case 0

show ?case

**unfolding** *S-def* **by** *simp*

**next**

case (*Suc n*)

have *S-Suc*:  $\bigwedge x\ n. S\ x\ (Suc\ n) = (x * S\ x\ n) /_R \text{real } (Suc\ n)$

**unfolding** *S-def* **by** (*simp del: mult-Suc*)

hence *times-S*:  $\bigwedge x\ n. x * S\ x\ n = \text{real } (Suc\ n) *_R S\ x\ (Suc\ n)$

**by** *simp*

have  $\text{real } (Suc\ n) *_R S\ (x + y)\ (Suc\ n) = (x + y) * S\ (x + y)\ n$

**by** (*simp only: times-S*)

also have  $\dots = (x + y) * (\sum i=0..n. S\ x\ i * S\ y\ (n-i))$

**by** (*simp only: Suc*)

also have  $\dots = x * (\sum i=0..n. S\ x\ i * S\ y\ (n-i))$

$+ y * (\sum i=0..n. S\ x\ i * S\ y\ (n-i))$

**by** (rule *left-distrib*)

also have  $\dots = (\sum i=0..n. (x * S\ x\ i) * S\ y\ (n-i))$

$+ (\sum i=0..n. S\ x\ i * (y * S\ y\ (n-i)))$

**by** (*simp only: setsum-right-distrib mult-ac*)

also have  $\dots = (\sum i=0..n. \text{real } (Suc\ i) *_R (S\ x\ (Suc\ i) * S\ y\ (n-i)))$

$+ (\sum i=0..n. \text{real } (Suc\ n-i) *_R (S\ x\ i * S\ y\ (Suc\ n-i)))$

**by** (*simp add: times-S Suc-diff-le*)

**also have**  $(\sum_{i=0..n} \text{real } (Suc\ i) *_{\mathbb{R}} (S\ x\ (Suc\ i) * S\ y\ (n-i))) =$   
 $(\sum_{i=0..Suc\ n} \text{real } i *_{\mathbb{R}} (S\ x\ i * S\ y\ (Suc\ n-i)))$   
**by** (*subst setsum-cl-ivl-Suc2, simp*)  
**also have**  $(\sum_{i=0..n} \text{real } (Suc\ n-i) *_{\mathbb{R}} (S\ x\ i * S\ y\ (Suc\ n-i))) =$   
 $(\sum_{i=0..Suc\ n} \text{real } (Suc\ n-i) *_{\mathbb{R}} (S\ x\ i * S\ y\ (Suc\ n-i)))$   
**by** (*subst setsum-cl-ivl-Suc, simp*)  
**also have**  $(\sum_{i=0..Suc\ n} \text{real } i *_{\mathbb{R}} (S\ x\ i * S\ y\ (Suc\ n-i))) +$   
 $(\sum_{i=0..Suc\ n} \text{real } (Suc\ n-i) *_{\mathbb{R}} (S\ x\ i * S\ y\ (Suc\ n-i))) =$   
 $(\sum_{i=0..Suc\ n} \text{real } (Suc\ n) *_{\mathbb{R}} (S\ x\ i * S\ y\ (Suc\ n-i)))$   
**by** (*simp only: setsum-addf [symmetric] scaleR-left-distrib [symmetric]*  
*real-of-nat-add [symmetric], simp*)  
**also have**  $\dots = \text{real } (Suc\ n) *_{\mathbb{R}} (\sum_{i=0..Suc\ n} S\ x\ i * S\ y\ (Suc\ n-i))$   
**by** (*simp only: scaleR-right.setsum*)  
**finally show**  
 $S\ (x + y)\ (Suc\ n) = (\sum_{i=0..Suc\ n} S\ x\ i * S\ y\ (Suc\ n - i))$   
**by** (*simp del: setsum-cl-ivl-Suc*)  
**qed**

**lemma** *exp-add*:  $\exp\ (x + y) = \exp\ x * \exp\ y$   
**unfolding** *exp-def*  
**by** (*simp only: Cauchy-product summable-norm-exp exp-series-add*)

**lemma** *mult-exp-exp*:  $\exp\ x * \exp\ y = \exp\ (x + y)$   
**by** (*rule exp-add [symmetric]*)

**lemma** *exp-of-real*:  $\exp\ (\text{of-real } x) = \text{of-real } (\exp\ x)$   
**unfolding** *exp-def*  
**apply** (*subst of-real.suminf*)  
**apply** (*rule summable-exp-generic*)  
**apply** (*simp add: scaleR-conv-of-real*)  
**done**

**lemma** *exp-not-eq-zero [simp]*:  $\exp\ x \neq 0$   
**proof**  
**have**  $\exp\ x * \exp\ (-x) = 1$  **by** (*simp add: mult-exp-exp*)  
**also assume**  $\exp\ x = 0$   
**finally show** *False* **by** *simp*  
**qed**

**lemma** *exp-minus*:  $\exp\ (-x) = \text{inverse } (\exp\ x)$   
**by** (*rule inverse-unique [symmetric], simp add: mult-exp-exp*)

**lemma** *exp-diff*:  $\exp\ (x - y) = \exp\ x / \exp\ y$   
**unfolding** *diff-minus divide-inverse*  
**by** (*simp add: exp-add exp-minus*)

## 76.6.2 Properties of the Exponential Function on Reals

Comparisons of  $\exp\ x$  with zero.



Proof: because every exponential can be seen as a square.

```
lemma exp-ge-zero [simp]: 0 ≤ exp (x::real)
proof -
  have 0 ≤ exp (x/2) * exp (x/2) by simp
  thus ?thesis by (simp add: exp-add [symmetric])
qed
```

```
lemma exp-gt-zero [simp]: 0 < exp (x::real)
by (simp add: order-less-le)
```

```
lemma not-exp-less-zero [simp]: ¬ exp (x::real) < 0
by (simp add: not-less)
```

```
lemma not-exp-le-zero [simp]: ¬ exp (x::real) ≤ 0
by (simp add: not-le)
```

```
lemma abs-exp-cancel [simp]: |exp x::real| = exp x
by simp
```

```
lemma exp-real-of-nat-mult: exp(real n * x) = exp(x) ^ n
apply (induct n)
apply (auto simp add: real-of-nat-Suc right-distrib exp-add mult-commute)
done
```

Strict monotonicity of exponential.

```
lemma exp-ge-add-one-self-aux: 0 ≤ (x::real) ==> (1 + x) ≤ exp(x)
apply (drule order-le-imp-less-or-eq, auto)
apply (simp add: exp-def)
apply (rule order-trans)
apply (rule-tac [2] n = 2 and f = (%n. inverse (real (fact n)) * x ^ n) in
series-pos-le)
apply (auto intro: summable-exp simp add: numeral-2-eq-2 zero-le-mult-iff)
done
```

```
lemma exp-gt-one: 0 < (x::real) ==> 1 < exp x
proof -
```

```
  assume x: 0 < x
  hence 1 < 1 + x by simp
  also from x have 1 + x ≤ exp x
    by (simp add: exp-ge-add-one-self-aux)
  finally show ?thesis .
qed
```

```
lemma exp-less-mono:
  fixes x y :: real
  assumes x < y shows exp x < exp y
proof -
  from ⟨x < y⟩ have 0 < y - x by simp
  hence 1 < exp (y - x) by (rule exp-gt-one)
```

hence  $1 < \exp y / \exp x$  **by** (*simp only: exp-diff*)  
 thus  $\exp x < \exp y$  **by** *simp*  
**qed**

**lemma** *exp-less-cancel*:  $\exp (x::\text{real}) < \exp y \implies x < y$   
**apply** (*simp add: linorder-not-le [symmetric]*)  
**apply** (*auto simp add: order-le-less exp-less-mono*)  
**done**

**lemma** *exp-less-cancel-iff* [*iff*]:  $\exp (x::\text{real}) < \exp y \iff x < y$   
**by** (*auto intro: exp-less-mono exp-less-cancel*)

**lemma** *exp-le-cancel-iff* [*iff*]:  $\exp (x::\text{real}) \leq \exp y \iff x \leq y$   
**by** (*auto simp add: linorder-not-less [symmetric]*)

**lemma** *exp-inj-iff* [*iff*]:  $\exp (x::\text{real}) = \exp y \iff x = y$   
**by** (*simp add: order-eq-iff*)

Comparisons of  $\exp x$  with one.

**lemma** *one-less-exp-iff* [*simp*]:  $1 < \exp (x::\text{real}) \iff 0 < x$   
**using** *exp-less-cancel-iff* [**where**  $x=0$  **and**  $y=x$ ] **by** *simp*

**lemma** *exp-less-one-iff* [*simp*]:  $\exp (x::\text{real}) < 1 \iff x < 0$   
**using** *exp-less-cancel-iff* [**where**  $x=x$  **and**  $y=0$ ] **by** *simp*

**lemma** *one-le-exp-iff* [*simp*]:  $1 \leq \exp (x::\text{real}) \iff 0 \leq x$   
**using** *exp-le-cancel-iff* [**where**  $x=0$  **and**  $y=x$ ] **by** *simp*

**lemma** *exp-le-one-iff* [*simp*]:  $\exp (x::\text{real}) \leq 1 \iff x \leq 0$   
**using** *exp-le-cancel-iff* [**where**  $x=x$  **and**  $y=0$ ] **by** *simp*

**lemma** *exp-eq-one-iff* [*simp*]:  $\exp (x::\text{real}) = 1 \iff x = 0$   
**using** *exp-inj-iff* [**where**  $x=x$  **and**  $y=0$ ] **by** *simp*

**lemma** *lemma-exp-total*:  $1 \leq y \implies \exists x. 0 \leq x \ \& \ x \leq y - 1 \ \& \ \exp(x::\text{real}) = y$   
**apply** (*rule IVT*)  
**apply** (*auto intro: isCont-exp simp add: le-diff-eq*)  
**apply** (*subgoal-tac 1 + (y - 1) \leq \exp (y - 1)*)  
**apply** *simp*  
**apply** (*rule exp-ge-add-one-self-aux, simp*)  
**done**

**lemma** *exp-total*:  $0 < (y::\text{real}) \implies \exists x. \exp x = y$   
**apply** (*rule-tac x = 1 and y = y in linorder-cases*)  
**apply** (*drule order-less-imp-le [THEN lemma-exp-total]*)  
**apply** (*rule-tac [2] x = 0 in exI*)  
**apply** (*frule-tac [3] one-less-inverse*)  
**apply** (*drule-tac [4] order-less-imp-le [THEN lemma-exp-total], auto*)  
**apply** (*rule-tac x = -x in exI*)

**apply** (*simp add: exp-minus*)  
**done**

## 76.7 Natural Logarithm

**definition**

$\ln :: \text{real} \Rightarrow \text{real}$  **where**  
 $\ln x = (\text{THE } u. \exp u = x)$

**lemma** *ln-exp* [*simp*]:  $\ln (\exp x) = x$   
**by** (*simp add: ln-def*)

**lemma** *exp-ln* [*simp*]:  $0 < x \Longrightarrow \exp (\ln x) = x$   
**by** (*auto dest: exp-total*)

**lemma** *exp-ln-iff* [*simp*]:  $\exp (\ln x) = x \longleftrightarrow 0 < x$   
**apply** (*rule iffI*)  
**apply** (*erule subst, rule exp-gt-zero*)  
**apply** (*erule exp-ln*)  
**done**

**lemma** *ln-unique*:  $\exp y = x \Longrightarrow \ln x = y$   
**by** (*erule subst, rule ln-exp*)

**lemma** *ln-one* [*simp*]:  $\ln 1 = 0$   
**by** (*rule ln-unique, simp*)

**lemma** *ln-mult*:  $\llbracket 0 < x; 0 < y \rrbracket \Longrightarrow \ln (x * y) = \ln x + \ln y$   
**by** (*rule ln-unique, simp add: exp-add*)

**lemma** *ln-inverse*:  $0 < x \Longrightarrow \ln (\text{inverse } x) = - \ln x$   
**by** (*rule ln-unique, simp add: exp-minus*)

**lemma** *ln-div*:  $\llbracket 0 < x; 0 < y \rrbracket \Longrightarrow \ln (x / y) = \ln x - \ln y$   
**by** (*rule ln-unique, simp add: exp-diff*)

**lemma** *ln-realpow*:  $0 < x \Longrightarrow \ln (x ^ n) = \text{real } n * \ln x$   
**by** (*rule ln-unique, simp add: exp-real-of-nat-mult*)

**lemma** *ln-less-cancel-iff* [*simp*]:  $\llbracket 0 < x; 0 < y \rrbracket \Longrightarrow \ln x < \ln y \longleftrightarrow x < y$   
**by** (*subst exp-less-cancel-iff [symmetric], simp*)

**lemma** *ln-le-cancel-iff* [*simp*]:  $\llbracket 0 < x; 0 < y \rrbracket \Longrightarrow \ln x \leq \ln y \longleftrightarrow x \leq y$   
**by** (*simp add: linorder-not-less [symmetric]*)

**lemma** *ln-inj-iff* [*simp*]:  $\llbracket 0 < x; 0 < y \rrbracket \Longrightarrow \ln x = \ln y \longleftrightarrow x = y$   
**by** (*simp add: order-eq-iff*)

**lemma** *ln-add-one-self-le-self* [*simp*]:  $0 \leq x \Longrightarrow \ln (1 + x) \leq x$

**apply** (*rule* *exp-le-cancel-iff* [*THEN* *iffD1*])  
**apply** (*simp* *add: exp-ge-add-one-self-aux*)  
**done**

**lemma** *ln-less-self* [*simp*]:  $0 < x \implies \ln x < x$   
**by** (*rule* *order-less-le-trans* [**where**  $y = \ln(1 + x)$ ]) *simp-all*

**lemma** *ln-ge-zero* [*simp*]:  
**assumes**  $x: 1 \leq x$  **shows**  $0 \leq \ln x$   
**proof** –  
**have**  $0 < x$  **using**  $x$  **by** *arith*  
**hence**  $\exp 0 \leq \exp (\ln x)$   
**by** (*simp* *add: x*)  
**thus** ?thesis **by** (*simp* *only: exp-le-cancel-iff*)  
**qed**

**lemma** *ln-ge-zero-imp-ge-one*:  
**assumes**  $\ln: 0 \leq \ln x$   
**and**  $x: 0 < x$   
**shows**  $1 \leq x$   
**proof** –  
**from**  $\ln$  **have**  $\ln 1 \leq \ln x$  **by** *simp*  
**thus** ?thesis **by** (*simp* *add: x del: ln-one*)  
**qed**

**lemma** *ln-ge-zero-iff* [*simp*]:  $0 < x \implies (0 \leq \ln x) = (1 \leq x)$   
**by** (*blast* *intro: ln-ge-zero ln-ge-zero-imp-ge-one*)

**lemma** *ln-less-zero-iff* [*simp*]:  $0 < x \implies (\ln x < 0) = (x < 1)$   
**by** (*insert* *ln-ge-zero-iff* [*of*  $x$ ], *arith*)

**lemma** *ln-gt-zero*:  
**assumes**  $x: 1 < x$  **shows**  $0 < \ln x$   
**proof** –  
**have**  $0 < x$  **using**  $x$  **by** *arith*  
**hence**  $\exp 0 < \exp (\ln x)$  **by** (*simp* *add: x*)  
**thus** ?thesis **by** (*simp* *only: exp-less-cancel-iff*)  
**qed**

**lemma** *ln-gt-zero-imp-gt-one*:  
**assumes**  $\ln: 0 < \ln x$   
**and**  $x: 0 < x$   
**shows**  $1 < x$   
**proof** –  
**from**  $\ln$  **have**  $\ln 1 < \ln x$  **by** *simp*  
**thus** ?thesis **by** (*simp* *add: x del: ln-one*)  
**qed**

**lemma** *ln-gt-zero-iff* [*simp*]:  $0 < x \implies (0 < \ln x) = (1 < x)$

by (blast intro: ln-gt-zero ln-gt-zero-imp-gt-one)

lemma ln-eq-zero-iff [simp]:  $0 < x \implies (\ln x = 0) = (x = 1)$   
by (insert ln-less-zero-iff [of x] ln-gt-zero-iff [of x], arith)

lemma ln-less-zero:  $[0 < x; x < 1] \implies \ln x < 0$   
by simp

lemma exp-ln-eq:  $\exp u = x \implies \ln x = u$   
by auto

lemma isCont-ln:  $0 < x \implies \text{isCont } \ln x$   
apply (subgoal-tac isCont ln (exp (ln x)), simp)  
apply (rule isCont-inverse-function [where f=exp], simp-all)  
done

lemma DERIV-ln:  $0 < x \implies \text{DERIV } \ln x :> \text{inverse } x$   
apply (rule DERIV-inverse-function [where f=exp and a=0 and b=x+1])  
apply (erule lemma-DERIV-subst [OF DERIV-exp exp-ln])  
apply (simp-all add: abs-if isCont-ln)  
done

lemma DERIV-ln-divide:  $0 < x \implies \text{DERIV } \ln x :> 1 / x$   
by (rule DERIV-ln[THEN DERIV-cong], simp, simp add: divide-inverse)

lemma ln-series: assumes  $0 < x$  and  $x < 2$   
shows  $\ln x = (\sum n. (-1)^n * (1 / \text{real } (n + 1)) * (x - 1)^{\text{Suc } n})$  (is  $\ln x = \text{suminf } (?f (x - 1))$ )

proof -

let  $?f' x n = (-1)^n * (x - 1)^n$

have  $\ln x - \text{suminf } (?f (x - 1)) = \ln 1 - \text{suminf } (?f (1 - 1))$

proof (rule DERIV-isconst3[where x=x])

fix  $x :: \text{real}$  assume  $x \in \{0 < .. < 2\}$  hence  $0 < x$  and  $x < 2$  by auto

have  $\text{norm } (1 - x) < 1$  using  $\langle 0 < x \rangle$  and  $\langle x < 2 \rangle$  by auto

have  $1 / x = 1 / (1 - (1 - x))$  by auto

also have  $\dots = (\sum n. (1 - x)^n)$  using geometric-sums[OF  $\langle \text{norm } (1 - x) < 1 \rangle$ ] by (rule sums-unique)

also have  $\dots = \text{suminf } (?f' x)$  unfolding power-mult-distrib[symmetric] by (rule arg-cong[where f=suminf], rule arg-cong[where f=op ^], auto)

finally have  $\text{DERIV } \ln x :> \text{suminf } (?f' x)$  using DERIV-ln[OF  $\langle 0 < x \rangle$ ]

unfolding divide-inverse by auto

moreover

have repos:  $\bigwedge h x :: \text{real}. h - 1 + x = h + x - 1$  by auto

have  $\text{DERIV } (\lambda x. \text{suminf } (?f x)) (x - 1) :> (\sum n. (-1)^n * (1 / \text{real } (n + 1))) * \text{real } (\text{Suc } n) * (x - 1)^n$

proof (rule DERIV-power-series')

show  $x - 1 \in \{-1 < .. < 1\}$  and  $(0 :: \text{real}) < 1$  using  $\langle 0 < x \rangle$   $\langle x < 2 \rangle$  by auto

```

{ fix x :: real assume x ∈ { - 1 < .. < 1 } hence norm (-x) < 1 by auto
  show summable (λn. -1 ^ n * (1 / real (n + 1)) * real (Suc n) * x ^ n)
    unfolding One-nat-def
    by (auto simp add: power-mult-distrib[symmetric] summable-geometric[OF
      (norm (-x) < 1)])
}
qed
hence DERIV (λx. suminf (?f x)) (x - 1) :> suminf (?f' x) unfolding
One-nat-def by auto
hence DERIV (λx. suminf (?f (x - 1))) x :> suminf (?f' x) unfolding
DERIV-iff repos .
ultimately have DERIV (λx. ln x - suminf (?f (x - 1))) x :> (suminf (?f'
x) - suminf (?f' x))
  by (rule DERIV-diff)
thus DERIV (λx. ln x - suminf (?f (x - 1))) x :> 0 by auto
qed (auto simp add: assms)
thus ?thesis by (auto simp add: suminf-zero)
qed

```

## 76.8 Sine and Cosine

### definition

```

sin-coeff :: nat ⇒ real where
sin-coeff = (λn. if even n then 0 else -1 ^ ((n - Suc 0) div 2) / real (fact n))

```

### definition

```

cos-coeff :: nat ⇒ real where
cos-coeff = (λn. if even n then (-1 ^ (n div 2)) / real (fact n) else 0)

```

### definition

```

sin :: real ⇒ real where
sin x = (∑ n. sin-coeff n * x ^ n)

```

### definition

```

cos :: real ⇒ real where
cos x = (∑ n. cos-coeff n * x ^ n)

```

**lemma** *summable-sin*: *summable (λn. sin-coeff n \* x ^ n)*

**unfolding** *sin-coeff-def*

**apply** (*rule-tac* *g* = (%n. inverse (real (fact n)) \* |x| ^ n) **in** *summable-comparison-test*)

**apply** (*rule-tac* [2] *summable-exp*)

**apply** (*rule-tac* *x* = 0 **in** *exI*)

**apply** (*auto simp add: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff*)

**done**

**lemma** *summable-cos*: *summable (λn. cos-coeff n \* x ^ n)*

**unfolding** *cos-coeff-def*

**apply** (*rule-tac* *g* = (%n. inverse (real (fact n)) \* |x| ^ n) **in** *summable-comparison-test*)

**apply** (*rule-tac* [2] *summable-exp*)

**apply** (*rule-tac*  $x = 0$  **in** *exI*)  
**apply** (*auto simp add: divide-inverse abs-mult power-abs [symmetric] zero-le-mult-iff*)  
**done**

**lemma** *lemma-STAR-sin*:  
 (*if even*  $n$  *then*  $0$   
   *else*  $-1 \wedge ((n - \text{Suc } 0) \text{ div } 2) / (\text{real } (\text{fact } n))) * 0 \wedge n = 0$ )  
**by** (*induct*  $n$ , *auto*)

**lemma** *lemma-STAR-cos*:  
 $0 < n \longleftrightarrow$   
 $-1 \wedge (n \text{ div } 2) / (\text{real } (\text{fact } n)) * 0 \wedge n = 0$   
**by** (*induct*  $n$ , *auto*)

**lemma** *lemma-STAR-cos1*:  
 $0 < n \longleftrightarrow$   
 $(-1) \wedge (n \text{ div } 2) / (\text{real } (\text{fact } n)) * 0 \wedge n = 0$   
**by** (*induct*  $n$ , *auto*)

**lemma** *lemma-STAR-cos2*:  
 $(\sum n=1..<n. \text{ if even } n \text{ then } -1 \wedge (n \text{ div } 2) / (\text{real } (\text{fact } n)) * 0 \wedge n$   
   *else*  $0) = 0$   
**apply** (*induct*  $n$ )  
**apply** (*case-tac*  $[2]$   $n$ , *auto*)  
**done**

**lemma** *sin-converges*:  $(\lambda n. \text{ sin-coeff } n * x \wedge n) \text{ sums } \text{sin}(x)$   
**unfolding** *sin-def* **by** (*rule summable-sin [THEN summable-sums]*)

**lemma** *cos-converges*:  $(\lambda n. \text{ cos-coeff } n * x \wedge n) \text{ sums } \text{cos}(x)$   
**unfolding** *cos-def* **by** (*rule summable-cos [THEN summable-sums]*)

**lemma** *sin-fdiffs*:  $\text{diffs sin-coeff} = \text{cos-coeff}$   
**unfolding** *sin-coeff-def cos-coeff-def*  
**by** (*auto intro!: ext*  
   *simp add: diffs-def divide-inverse real-of-nat-def of-nat-mult*  
   *simp del: mult-Suc of-nat-Suc*)

**lemma** *sin-fdiffs2*:  $\text{diffs sin-coeff } n = \text{cos-coeff } n$   
**by** (*simp only: sin-fdiffs*)

**lemma** *cos-fdiffs*:  $\text{diffs cos-coeff} = (\lambda n. - \text{sin-coeff } n)$   
**unfolding** *sin-coeff-def cos-coeff-def*  
**by** (*auto intro!: ext*  
   *simp add: diffs-def divide-inverse odd-Suc-mult-two-ex real-of-nat-def*  
   *of-nat-mult*  
   *simp del: mult-Suc of-nat-Suc*)

**lemma** *cos-fdiffs2*:  $\text{diffs cos-coeff } n = - \text{sin-coeff } n$

**by** (*simp only: cos-fdiffs*)

Now at last we can get the derivatives of exp, sin and cos

**lemma** *lemma-sin-minus*:  $-\sin x = (\sum n. -(\sin\text{-coeff } n * x ^ n))$   
**by** (*auto intro!: sums-unique sums-minus sin-converges*)

**lemma** *lemma-sin-ext*:  $\sin = (\lambda x. \sum n. \sin\text{-coeff } n * x ^ n)$   
**by** (*auto intro!: ext simp add: sin-def*)

**lemma** *lemma-cos-ext*:  $\cos = (\lambda x. \sum n. \cos\text{-coeff } n * x ^ n)$   
**by** (*auto intro!: ext simp add: cos-def*)

**lemma** *DERIV-sin* [*simp*]:  $\text{DERIV } \sin x :> \cos(x)$   
**apply** (*simp add: cos-def*)  
**apply** (*subst lemma-sin-ext*)  
**apply** (*auto simp add: sin-fdiffs2 [symmetric]*)  
**apply** (*rule-tac K = 1 + |x| in termdiffs*)  
**apply** (*auto intro: sin-converges cos-converges sums-summable intro!: sums-minus [THEN sums-summable] simp add: cos-fdiffs sin-fdiffs*)  
**done**

**lemma** *DERIV-cos* [*simp*]:  $\text{DERIV } \cos x :> -\sin(x)$   
**apply** (*subst lemma-cos-ext*)  
**apply** (*auto simp add: lemma-sin-minus cos-fdiffs2 [symmetric] minus-mult-left*)  
**apply** (*rule-tac K = 1 + |x| in termdiffs*)  
**apply** (*auto intro: sin-converges cos-converges sums-summable intro!: sums-minus [THEN sums-summable] simp add: cos-fdiffs sin-fdiffs diff-minus*)  
**done**

**lemma** *isCont-sin* [*simp*]: *isCont*  $\sin x$   
**by** (*rule DERIV-sin [THEN DERIV-isCont]*)

**lemma** *isCont-cos* [*simp*]: *isCont*  $\cos x$   
**by** (*rule DERIV-cos [THEN DERIV-isCont]*)

**declare**

*DERIV-exp*[*THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros*]  
*DERIV-ln*[*THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros*]  
*DERIV-sin*[*THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros*]  
*DERIV-cos*[*THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros*]

## 76.9 Properties of Sine and Cosine

**lemma** *sin-zero* [*simp*]:  $\sin 0 = 0$   
**unfolding** *sin-def sin-coeff-def* **by** (*simp add: powser-zero*)

**lemma** *cos-zero* [*simp*]:  $\cos 0 = 1$   
**unfolding** *cos-def cos-coeff-def* **by** (*simp add: powser-zero*)



**lemma** *DERIV-sin-sin-mult* [*simp*]:

$DERIV (\%x. \sin(x)*\sin(x)) \ x :> \cos(x) * \sin(x) + \cos(x) * \sin(x)$

**by** (*rule* *DERIV-mult*, *auto*)

**lemma** *DERIV-sin-sin-mult2* [*simp*]:

$DERIV (\%x. \sin(x)*\sin(x)) \ x :> 2 * \cos(x) * \sin(x)$

**apply** (*cut-tac*  $x = x$  **in** *DERIV-sin-sin-mult*)

**apply** (*auto simp add: mult-assoc*)

**done**

**lemma** *DERIV-sin-realpow2* [*simp*]:

$DERIV (\%x. (\sin x)^2) \ x :> \cos(x) * \sin(x) + \cos(x) * \sin(x)$

**by** (*auto simp add: numeral-2-eq-2 mult-assoc [symmetric]*)

**lemma** *DERIV-sin-realpow2a* [*simp*]:

$DERIV (\%x. (\sin x)^2) \ x :> 2 * \cos(x) * \sin(x)$

**by** (*auto simp add: numeral-2-eq-2*)

**lemma** *DERIV-cos-cos-mult* [*simp*]:

$DERIV (\%x. \cos(x)*\cos(x)) \ x :> -\sin(x) * \cos(x) + -\sin(x) * \cos(x)$

**by** (*rule* *DERIV-mult*, *auto*)

**lemma** *DERIV-cos-cos-mult2* [*simp*]:

$DERIV (\%x. \cos(x)*\cos(x)) \ x :> -2 * \cos(x) * \sin(x)$

**apply** (*cut-tac*  $x = x$  **in** *DERIV-cos-cos-mult*)

**apply** (*auto simp add: mult-ac*)

**done**

**lemma** *DERIV-cos-realpow2* [*simp*]:

$DERIV (\%x. (\cos x)^2) \ x :> -\sin(x) * \cos(x) + -\sin(x) * \cos(x)$

**by** (*auto simp add: numeral-2-eq-2 mult-assoc [symmetric]*)

**lemma** *DERIV-cos-realpow2a* [*simp*]:

$DERIV (\%x. (\cos x)^2) \ x :> -2 * \cos(x) * \sin(x)$

**by** (*auto simp add: numeral-2-eq-2*)

**lemma** *lemma-DERIV-subst*: [ $DERIV \ f \ x :> D; D = E$ ]  $\implies DERIV \ f \ x :> E$

**by** *auto*

**lemma** *DERIV-cos-realpow2b*:  $DERIV (\%x. (\cos x)^2) \ x :> -(2 * \cos(x) * \sin(x))$

**by** (*auto intro!: DERIV-intros*)

**lemma** *DERIV-cos-cos-mult3* [*simp*]:

$DERIV (\%x. \cos(x)*\cos(x)) \ x :> -(2 * \cos(x) * \sin(x))$

**by** (*auto intro!: DERIV-intros*)

**lemma** *DERIV-sin-circle-all*:

$\forall x. \text{DERIV } (\%x. (\sin x)^2 + (\cos x)^2) x :>$   
 $(2 * \cos(x) * \sin(x) - 2 * \cos(x) * \sin(x))$

**by** (*auto intro!*: *DERIV-intros*)

**lemma** *DERIV-sin-circle-all-zero* [*simp*]:

$\forall x. \text{DERIV } (\%x. (\sin x)^2 + (\cos x)^2) x :> 0$

**by** (*cut-tac* *DERIV-sin-circle-all*, *auto*)

**lemma** *sin-cos-squared-add* [*simp*]:  $((\sin x)^2) + ((\cos x)^2) = 1$

**apply** (*cut-tac*  $x = x$  **and**  $y = 0$  **in** *DERIV-sin-circle-all-zero* [*THEN* *DERIV-isconst-all*])

**apply** (*auto simp add: numeral-2-eq-2*)

**done**

**lemma** *sin-cos-squared-add2* [*simp*]:  $((\cos x)^2) + ((\sin x)^2) = 1$

**apply** (*subst add-commute*)

**apply** (*rule sin-cos-squared-add*)

**done**

**lemma** *sin-cos-squared-add3* [*simp*]:  $\cos x * \cos x + \sin x * \sin x = 1$

**apply** (*cut-tac*  $x = x$  **in** *sin-cos-squared-add2*)

**apply** (*simp add: power2-eq-square*)

**done**

**lemma** *sin-squared-eq*:  $(\sin x)^2 = 1 - (\cos x)^2$

**apply** (*rule-tac*  $a1 = (\cos x)^2$  **in** *add-right-cancel* [*THEN iffD1*])

**apply** *simp*

**done**

**lemma** *cos-squared-eq*:  $(\cos x)^2 = 1 - (\sin x)^2$

**apply** (*rule-tac*  $a1 = (\sin x)^2$  **in** *add-right-cancel* [*THEN iffD1*])

**apply** *simp*

**done**

**lemma** *abs-sin-le-one* [*simp*]:  $|\sin x| \leq 1$

**by** (*rule power2-le-imp-le*, *simp-all add: sin-squared-eq*)

**lemma** *sin-ge-minus-one* [*simp*]:  $-1 \leq \sin x$

**apply** (*insert abs-sin-le-one* [*of*  $x$ ])

**apply** (*simp add: abs-le-iff del: abs-sin-le-one*)

**done**

**lemma** *sin-le-one* [*simp*]:  $\sin x \leq 1$

**apply** (*insert abs-sin-le-one* [*of*  $x$ ])

**apply** (*simp add: abs-le-iff del: abs-sin-le-one*)

**done**

**lemma** *abs-cos-le-one* [*simp*]:  $|\cos x| \leq 1$

**by** (*rule power2-le-imp-le*, *simp-all add: cos-squared-eq*)

```

lemma cos-ge-minus-one [simp]:  $-1 \leq \cos x$ 
apply (insert abs-cos-le-one [of x])
apply (simp add: abs-le-iff del: abs-cos-le-one)
done

```

```

lemma cos-le-one [simp]:  $\cos x \leq 1$ 
apply (insert abs-cos-le-one [of x])
apply (simp add: abs-le-iff del: abs-cos-le-one)
done

```

```

lemma DERIV-fun-pow:  $DERIV\ g\ x :> m \implies$ 
   $DERIV\ (\%x. (g\ x) ^ n)\ x :> real\ n * (g\ x) ^ (n - 1) * m$ 
unfolding One-nat-def
apply (rule lemma-DERIV-subst)
apply (rule-tac f = (%x. x ^ n) in DERIV-chain2)
apply (rule DERIV-pow, auto)
done

```

```

lemma DERIV-fun-exp:
   $DERIV\ g\ x :> m \implies DERIV\ (\%x. exp(g\ x))\ x :> exp(g\ x) * m$ 
apply (rule lemma-DERIV-subst)
apply (rule-tac f = exp in DERIV-chain2)
apply (rule DERIV-exp, auto)
done

```

```

lemma DERIV-fun-sin:
   $DERIV\ g\ x :> m \implies DERIV\ (\%x. sin(g\ x))\ x :> cos(g\ x) * m$ 
apply (rule lemma-DERIV-subst)
apply (rule-tac f = sin in DERIV-chain2)
apply (rule DERIV-sin, auto)
done

```

```

lemma DERIV-fun-cos:
   $DERIV\ g\ x :> m \implies DERIV\ (\%x. cos(g\ x))\ x :> -sin(g\ x) * m$ 
apply (rule lemma-DERIV-subst)
apply (rule-tac f = cos in DERIV-chain2)
apply (rule DERIV-cos, auto)
done

```

```

lemma lemma-DERIV-sin-cos-add:
   $\forall x.$ 
   $DERIV\ (\%x. (sin\ (x + y) - (sin\ x * cos\ y + cos\ x * sin\ y)) ^ 2 +$ 
     $(cos\ (x + y) - (cos\ x * cos\ y - sin\ x * sin\ y)) ^ 2)\ x :> 0$ 
  by (auto intro!: DERIV-intros simp add: algebra-simps)

```

```

lemma sin-cos-add [simp]:
   $(sin\ (x + y) - (sin\ x * cos\ y + cos\ x * sin\ y)) ^ 2 +$ 

```

```

      (cos (x + y) - (cos x * cos y - sin x * sin y)) ^ 2 = 0
apply (cut-tac y = 0 and x = x and y7 = y
      in lemma-DERIV-sin-cos-add [THEN DERIV-isconst-all])
apply (auto simp add: numeral-2-eq-2)
done

```

```

lemma sin-add: sin (x + y) = sin x * cos y + cos x * sin y
apply (cut-tac x = x and y = y in sin-cos-add)
apply (simp del: sin-cos-add)
done

```

```

lemma cos-add: cos (x + y) = cos x * cos y - sin x * sin y
apply (cut-tac x = x and y = y in sin-cos-add)
apply (simp del: sin-cos-add)
done

```

```

lemma lemma-DERIV-sin-cos-minus:
  ∀ x. DERIV (%x. (sin(-x) + (sin x)) ^ 2 + (cos(-x) - (cos x)) ^ 2) x :> 0
by (auto intro!: DERIV-intros simp add: algebra-simps)

```

```

lemma sin-cos-minus:
  (sin(-x) + (sin x)) ^ 2 + (cos(-x) - (cos x)) ^ 2 = 0
apply (cut-tac y = 0 and x = x
      in lemma-DERIV-sin-cos-minus [THEN DERIV-isconst-all])
apply simp
done

```

```

lemma sin-minus [simp]: sin (-x) = -sin(x)
using sin-cos-minus [where x=x] by simp

```

```

lemma cos-minus [simp]: cos (-x) = cos(x)
using sin-cos-minus [where x=x] by simp

```

```

lemma sin-diff: sin (x - y) = sin x * cos y - cos x * sin y
by (simp add: diff-minus sin-add)

```

```

lemma sin-diff2: sin (x - y) = cos y * sin x - sin y * cos x
by (simp add: sin-diff mult-commute)

```

```

lemma cos-diff: cos (x - y) = cos x * cos y + sin x * sin y
by (simp add: diff-minus cos-add)

```

```

lemma cos-diff2: cos (x - y) = cos y * cos x + sin y * sin x
by (simp add: cos-diff mult-commute)

```

```

lemma sin-double [simp]: sin(2 * x) = 2 * sin x * cos x
using sin-add [where x=x and y=x] by simp

```

**lemma** *cos-double*:  $\cos(2 * x) = ((\cos x)^2) - ((\sin x)^2)$   
**using** *cos-add* [where  $x=x$  and  $y=x$ ]  
**by** (*simp add: power2-eq-square*)

## 76.10 The Constant Pi

**definition**

$\pi :: \text{real}$  **where**  
 $\pi = 2 * (\text{THE } x. 0 \leq (x :: \text{real}) \ \& \ x \leq 2 \ \& \ \cos x = 0)$

Show that there's a least positive  $x$  with  $\cos x = 0$ ; hence define  $\pi$ .

**lemma** *sin-paired*:

$(\%n. -1 \wedge n / (\text{real } (\text{fact } (2 * n + 1))) * x \wedge (2 * n + 1))$   
*sums sin x*

**proof** –

**have**  $(\lambda n. \sum k = n * 2 .. < n * 2 + 2. \text{sin-coeff } k * x \wedge k) \text{ sums sin } x$   
**unfolding** *sin-def*  
**by** (*rule sin-converges [THEN sums-summable, THEN sums-group], simp*)  
**thus** ?thesis **unfolding** *One-nat-def sin-coeff-def* **by** (*simp add: mult-ac*)

**qed**

FIXME: This is a long, ugly proof!

**lemma** *sin-gt-zero*:  $[|0 < x; x < 2|] ==> 0 < \sin x$

**apply** (*subgoal-tac*

$(\lambda n. \sum k = n * 2 .. < n * 2 + 2. -1 \wedge k / \text{real } (\text{fact } (2 * k + 1)) * x \wedge (2 * k + 1))$   
*sums*  $(\sum n. -1 \wedge n / \text{real } (\text{fact } (2 * n + 1)) * x \wedge (2 * n + 1))$ )

**prefer** 2

**apply** (*rule sin-paired [THEN sums-summable, THEN sums-group], simp*)

**apply** (*rotate-tac* 2)

**apply** (*drule sin-paired [THEN sums-unique, THEN ssubst]*)

**unfolding** *One-nat-def*

**apply** (*auto simp del: fact-Suc*)

**apply** (*frule sums-unique*)

**apply** (*auto simp del: fact-Suc*)

**apply** (*rule-tac n1 = 0 in series-pos-less [THEN [2] order-le-less-trans]*)

**apply** (*auto simp del: fact-Suc*)

**apply** (*erule sums-summable*)

**apply** (*case-tac m=0*)

**apply** (*simp (no-asm-simp)*)

**apply** (*subgoal-tac*  $6 * (x * (x * x) / \text{real } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } 0))))))$   
 $< 6 * x$ )

**apply** (*simp only: mult-less-cancel-left, simp*)

**apply** (*simp (no-asm-simp) add: numeral-2-eq-2 [symmetric] mult-assoc [symmetric]*)

**apply** (*subgoal-tac*  $x * x < 2 * 3$ , *simp*)

**apply** (*rule mult-strict-mono*)

**apply** (*auto simp add: real-0-less-add-iff real-of-nat-Suc simp del: fact-Suc*)

**apply** (*subst fact-Suc*)

**apply** (*subst fact-Suc*)

```

apply (subst fact-Suc)
apply (subst fact-Suc)
apply (subst real-of-nat-mult)
apply (subst real-of-nat-mult)
apply (subst real-of-nat-mult)
apply (subst real-of-nat-mult)
apply (simp (no-asm) add: divide-inverse del: fact-Suc)
apply (auto simp add: mult-assoc [symmetric] simp del: fact-Suc)
apply (rule-tac c=real (Suc (Suc (4*m))) in mult-less-imp-less-right)
apply (auto simp add: mult-assoc simp del: fact-Suc)
apply (rule-tac c=real (Suc (Suc (Suc (4*m)))) in mult-less-imp-less-right)
apply (auto simp add: mult-assoc mult-less-cancel-left simp del: fact-Suc)
apply (subgoal-tac x * (x * x ^ (4*m)) = (x ^ (4*m)) * (x * x))
apply (erule ssubst)+
apply (auto simp del: fact-Suc)
apply (subgoal-tac 0 < x ^ (4 * m) )
  prefer 2 apply (simp only: zero-less-power)
apply (simp (no-asm-simp) add: mult-less-cancel-left)
apply (rule mult-strict-mono)
apply (simp-all (no-asm-simp))
done

```

```

lemma sin-gt-zero1: [| 0 < x; x < 2 |] ==> 0 < sin x
by (auto intro: sin-gt-zero)

```

```

lemma cos-double-less-one: [| 0 < x; x < 2 |] ==> cos (2 * x) < 1
apply (cut-tac x = x in sin-gt-zero1)
apply (auto simp add: cos-squared-eq cos-double)
done

```

```

lemma cos-paired:
  (%n. -1 ^ n / (real (fact (2 * n))) * x ^ (2 * n)) sums cos x
proof -
  have (λn. ∑ k = n * 2..<n * 2 + 2. cos-coeff k * x ^ k) sums cos x
    unfolding cos-def
  by (rule cos-converges [THEN sums-summable, THEN sums-group], simp)
  thus ?thesis unfolding cos-coeff-def by (simp add: mult-ac)
qed

```

```

lemma fact-lemma: real (n::nat) * 4 = real (4 * n)
by simp

```

```

lemma real-mult-inverse-cancel:
  [| (0::real) < x; 0 < x1; x1 * y < x * u |]
  ==> inverse x * y < inverse x1 * u
apply (rule-tac c=x in mult-less-imp-less-left)
apply (auto simp add: mult-assoc [symmetric])
apply (simp (no-asm) add: mult-ac)
apply (rule-tac c=x1 in mult-less-imp-less-right)

```

**apply** (*auto simp add: mult-ac*)  
**done**

**lemma** *real-mult-inverse-cancel2*:

$[(0::\text{real}) < x; 0 < x1; x1 * y < x * u] ==> y * \text{inverse } x < u * \text{inverse } x1$   
**apply** (*auto dest: real-mult-inverse-cancel simp add: mult-ac*)  
**done**

**lemma** *realpow-num-eq-if*:

**fixes**  $m :: 'a::\text{power}$   
**shows**  $m \wedge n = (\text{if } n=0 \text{ then } 1 \text{ else } m * m \wedge (n - 1))$   
**by** (*cases n, auto*)

**lemma** *cos-two-less-zero* [*simp*]:  $\cos(2) < 0$

**apply** (*cut-tac x = 2 in cos-paired*)  
**apply** (*drule sums-minus*)  
**apply** (*rule neg-less-iff-less [THEN iffD1]*)  
**apply** (*frule sums-unique, auto*)  
**apply** (*rule-tac y =*  
 $\sum_{n=0..< \text{Suc}(\text{Suc}(\text{Suc } 0)). - (-1 \wedge n / (\text{real}(\text{fact } (2*n))) * 2 \wedge (2*n))$   
*in order-less-trans*)  
**apply** (*simp (no-asm) add: fact-num-eq-if-nat realpow-num-eq-if del: fact-Suc*)  
**apply** (*simp (no-asm) add: mult-assoc del: setsum-op-ivl-Suc*)  
**apply** (*rule sumr-pos-lt-pair*)  
**apply** (*erule sums-summable, safe*)  
**unfolding** *One-nat-def*  
**apply** (*simp (no-asm) add: divide-inverse real-0-less-add-iff mult-assoc [symmetric]*  
  
*del: fact-Suc*)  
**apply** (*rule real-mult-inverse-cancel2*)  
**apply** (*rule real-of-nat-fact-gt-zero*)  
**apply** (*simp (no-asm) add: mult-assoc [symmetric] del: fact-Suc*)  
**apply** (*subst fact-lemma*)  
**apply** (*subst fact-Suc [of Suc (Suc (Suc (Suc (Suc (Suc (Suc (4 \* d)))))))]*)  
**apply** (*simp only: real-of-nat-mult*)  
**apply** (*rule mult-strict-mono, force*)  
**apply** (*rule-tac [3] real-of-nat-ge-zero*)  
**prefer** 2 **apply** *force*  
**apply** (*rule real-of-nat-less-iff [THEN iffD2]*)  
**apply** (*rule fact-less-mono-nat, auto*)  
**done**

**lemmas** *cos-two-neq-zero* [*simp*] = *cos-two-less-zero* [*THEN less-imp-neq*]

**lemmas** *cos-two-le-zero* [*simp*] = *cos-two-less-zero* [*THEN order-less-imp-le*]

**lemma** *cos-is-zero*:  $\text{EX! } x. 0 \leq x \ \& \ x \leq 2 \ \& \ \cos x = 0$

**apply** (*subgoal-tac  $\exists x. 0 \leq x \ \& \ x \leq 2 \ \& \ \cos x = 0$* )  
**apply** (*rule-tac [2] IVT2*)  
**apply** (*auto intro: DERIV-isCont DERIV-cos*)

```

apply (cut-tac  $x = xa$  and  $y = y$  in linorder-less-linear)
apply (rule ccontr)
apply (subgoal-tac ( $\forall x. \cos \text{ differentiable } x$ ) & ( $\forall x. \text{isCont } \cos x$ ) )
apply (auto intro: DERIV-cos DERIV-isCont simp add: differentiable-def)
apply (drule-tac  $f = \cos$  in Rolle)
apply (drule-tac [5]  $f = \cos$  in Rolle)
apply (auto dest!: DERIV-cos [THEN DERIV-unique] simp add: differentiable-def)
apply (metis order-less-le-trans less-le sin-gt-zero)
apply (metis order-less-le-trans less-le sin-gt-zero)
done

lemma pi-half:  $\pi/2 = (\text{THE } x. 0 \leq x \ \& \ x \leq 2 \ \& \ \cos x = 0)$ 
by (simp add: pi-def)

lemma cos-pi-half [simp]:  $\cos (\pi / 2) = 0$ 
by (simp add: pi-half cos-is-zero [THEN theI])

lemma pi-half-gt-zero [simp]:  $0 < \pi / 2$ 
apply (rule order-le-neq-trans)
apply (simp add: pi-half cos-is-zero [THEN theI])
apply (rule notI, drule arg-cong [where  $f=\cos$ ], simp)
done

lemmas pi-half-neq-zero [simp] = pi-half-gt-zero [THEN less-imp-neq, symmetric]
lemmas pi-half-ge-zero [simp] = pi-half-gt-zero [THEN order-less-imp-le]

lemma pi-half-less-two [simp]:  $\pi / 2 < 2$ 
apply (rule order-le-neq-trans)
apply (simp add: pi-half cos-is-zero [THEN theI])
apply (rule notI, drule arg-cong [where  $f=\cos$ ], simp)
done

lemmas pi-half-neq-two [simp] = pi-half-less-two [THEN less-imp-neq]
lemmas pi-half-le-two [simp] = pi-half-less-two [THEN order-less-imp-le]

lemma pi-gt-zero [simp]:  $0 < \pi$ 
by (insert pi-half-gt-zero, simp)

lemma pi-ge-zero [simp]:  $0 \leq \pi$ 
by (rule pi-gt-zero [THEN order-less-imp-le])

lemma pi-neq-zero [simp]:  $\pi \neq 0$ 
by (rule pi-gt-zero [THEN less-imp-neq, THEN not-sym])

lemma pi-not-less-zero [simp]:  $\neg \pi < 0$ 
by (simp add: linorder-not-less)

lemma minus-pi-half-less-zero:  $-(\pi/2) < 0$ 
by simp

```



```

lemma m2pi-less-pi:  $-(2 * \pi) < \pi$ 
proof -
  have  $-(2 * \pi) < 0$  and  $0 < \pi$  by auto
  from order-less-trans[OF this] show ?thesis .
qed

lemma sin-pi-half [simp]:  $\sin(\pi/2) = 1$ 
apply (cut-tac  $x = \pi/2$  in sin-cos-squared-add2)
apply (cut-tac sin-gt-zero [OF pi-half-gt-zero pi-half-less-two])
apply (simp add: power2-eq-1-iff)
done

lemma cos-pi [simp]:  $\cos \pi = -1$ 
by (cut-tac  $x = \pi/2$  and  $y = \pi/2$  in cos-add, simp)

lemma sin-pi [simp]:  $\sin \pi = 0$ 
by (cut-tac  $x = \pi/2$  and  $y = \pi/2$  in sin-add, simp)

lemma sin-cos-eq:  $\sin x = \cos(\pi/2 - x)$ 
by (simp add: diff-minus cos-add)
declare sin-cos-eq [symmetric, simp]

lemma minus-sin-cos-eq:  $-\sin x = \cos(x + \pi/2)$ 
by (simp add: cos-add)
declare minus-sin-cos-eq [symmetric, simp]

lemma cos-sin-eq:  $\cos x = \sin(\pi/2 - x)$ 
by (simp add: diff-minus sin-add)
declare cos-sin-eq [symmetric, simp]

lemma sin-periodic-pi [simp]:  $\sin(x + \pi) = -\sin x$ 
by (simp add: sin-add)

lemma sin-periodic-pi2 [simp]:  $\sin(\pi + x) = -\sin x$ 
by (simp add: sin-add)

lemma cos-periodic-pi [simp]:  $\cos(x + \pi) = -\cos x$ 
by (simp add: cos-add)

lemma sin-periodic [simp]:  $\sin(x + 2*\pi) = \sin x$ 
by (simp add: sin-add cos-double)

lemma cos-periodic [simp]:  $\cos(x + 2*\pi) = \cos x$ 
by (simp add: cos-add cos-double)

lemma cos-npi [simp]:  $\cos(\text{real } n * \pi) = -1 \wedge n$ 
apply (induct  $n$ )
apply (auto simp add: real-of-nat-Suc left-distrib)

```

done

```
lemma cos-npi2 [simp]: cos (pi * real n) = -1 ^ n
proof -
  have cos (pi * real n) = cos (real n * pi) by (simp only: mult-commute)
  also have ... = -1 ^ n by (rule cos-npi)
  finally show ?thesis .
qed
```

```
lemma sin-npi [simp]: sin (real (n::nat) * pi) = 0
apply (induct n)
apply (auto simp add: real-of-nat-Suc left-distrib)
done
```

```
lemma sin-npi2 [simp]: sin (pi * real (n::nat)) = 0
by (simp add: mult-commute [of pi])
```

```
lemma cos-two-pi [simp]: cos (2 * pi) = 1
by (simp add: cos-double)
```

```
lemma sin-two-pi [simp]: sin (2 * pi) = 0
by simp
```

```
lemma sin-gt-zero2: [| 0 < x; x < pi/2 |] ==> 0 < sin x
apply (rule sin-gt-zero, assumption)
apply (rule order-less-trans, assumption)
apply (rule pi-half-less-two)
done
```

```
lemma sin-less-zero:
  assumes lb: - pi/2 < x and x < 0 shows sin x < 0
proof -
  have 0 < sin (- x) using prems by (simp only: sin-gt-zero2)
  thus ?thesis by simp
qed
```

```
lemma pi-less-4: pi < 4
by (cut-tac pi-half-less-two, auto)
```

```
lemma cos-gt-zero: [| 0 < x; x < pi/2 |] ==> 0 < cos x
apply (cut-tac pi-less-4)
apply (cut-tac f = cos and a = 0 and b = x and y = 0 in IVT2-objl, safe,
simp-all)
apply (cut-tac cos-is-zero, safe)
apply (rename-tac y z)
apply (drule-tac x = y in spec)
apply (drule-tac x = pi/2 in spec, simp)
done
```

**lemma** *cos-gt-zero-pi*:  $[| -(pi/2) < x; x < pi/2 |] ==> 0 < \cos x$   
**apply** (*rule-tac*  $x = x$  **and**  $y = 0$  **in** *linorder-cases*)  
**apply** (*rule cos-minus* [*THEN subst*])  
**apply** (*rule cos-gt-zero*)  
**apply** (*auto intro: cos-gt-zero*)  
**done**

**lemma** *cos-ge-zero*:  $[| -(pi/2) \leq x; x \leq pi/2 |] ==> 0 \leq \cos x$   
**apply** (*auto simp add: order-le-less cos-gt-zero-pi*)  
**apply** (*subgoal-tac*  $x = pi/2$ , *auto*)  
**done**

**lemma** *sin-gt-zero-pi*:  $[| 0 < x; x < pi |] ==> 0 < \sin x$   
**apply** (*subst sin-cos-eq*)  
**apply** (*rotate-tac* 1)  
**apply** (*drule real-sum-of-halves* [*THEN ssubst*])  
**apply** (*auto intro!: cos-gt-zero-pi simp del: sin-cos-eq* [*symmetric*])  
**done**

**lemma** *pi-ge-two*:  $2 \leq pi$   
**proof** (*rule ccontr*)  
**assume**  $\neg 2 \leq pi$  **hence**  $pi < 2$  **by** *auto*  
**have**  $\exists y > pi. y < 2 \wedge y < 2 * pi$   
**proof** (*cases*  $2 < 2 * pi$ )  
**case** *True* **with** *dense*[*OF*  $\langle pi < 2 \rangle$ ] **show** *?thesis* **by** *auto*  
**next**  
**case** *False* **have**  $pi < 2 * pi$  **by** *auto*  
**from** *dense*[*OF this*] **and** *False* **show** *?thesis* **by** *auto*  
**qed**  
**then obtain**  $y$  **where**  $pi < y$  **and**  $y < 2$  **and**  $y < 2 * pi$  **by** *blast*  
**hence**  $0 < \sin y$  **using** *sin-gt-zero* **by** *auto*  
**moreover**  
**have**  $\sin y < 0$  **using** *sin-gt-zero-pi*[*of*  $y - pi$ ]  $\langle pi < y \rangle$  **and**  $\langle y < 2 * pi \rangle$   
*sin-periodic-pi*[*of*  $y - pi$ ] **by** *auto*  
**ultimately show** *False* **by** *auto*  
**qed**

**lemma** *sin-ge-zero*:  $[| 0 \leq x; x \leq pi |] ==> 0 \leq \sin x$   
**by** (*auto simp add: order-le-less sin-gt-zero-pi*)

**lemma** *cos-total*:  $[| -1 \leq y; y \leq 1 |] ==> \exists x. 0 \leq x \wedge x \leq pi \wedge (\cos x = y)$   
**apply** (*subgoal-tac*  $\exists x. 0 \leq x \wedge x \leq pi \wedge \cos x = y$ )  
**apply** (*rule-tac* [2] *IVT2*)  
**apply** (*auto intro: order-less-imp-le DERIV-isCont DERIV-cos*)  
**apply** (*cut-tac*  $x = xa$  **and**  $y = y$  **in** *linorder-less-linear*)  
**apply** (*rule ccontr, auto*)  
**apply** (*drule-tac*  $f = \cos$  **in** *Rolle*)

```

apply (drule-tac [5]  $f = \cos$  in Rolle)
apply (auto intro: order-less-imp-le DERIV-isCont DERIV-cos
        dest!: DERIV-cos [THEN DERIV-unique]
        simp add: differentiable-def)
apply (auto dest: sin-gt-zero-pi [OF order-le-less-trans order-less-le-trans])
done

```

**lemma** sin-total:

```

  [|  $-1 \leq y; y \leq 1$  |] ==> EX!  $x. -(pi/2) \leq x \ \& \ x \leq pi/2 \ \& \ (\sin x = y)$ 
apply (rule ccontr)
apply (subgoal-tac  $\forall x. (-(pi/2) \leq x \ \& \ x \leq pi/2 \ \& \ (\sin x = y)) = (0 \leq (x + pi/2) \ \& \ (x + pi/2) \leq pi \ \& \ (\cos (x + pi/2) = -y))$ )
apply (erule contrapos-mp)
apply (simp del: minus-sin-cos-eq [symmetric])
apply (cut-tac  $y = -y$  in cos-total, simp) apply simp
apply (erule ex1E)
apply (rule-tac  $a = x - (pi/2)$  in ex1I)
apply (simp (no-asm) add: add-assoc)
apply (rotate-tac 3)
apply (drule-tac  $x = xa + pi/2$  in spec, safe, simp-all)
done

```

**lemma** reals-Archimedean4:

```

  [|  $0 < y; 0 \leq x$  |] ==>  $\exists n. \text{real } n * y \leq x \ \& \ x < \text{real } (Suc\ n) * y$ 
apply (auto dest!: reals-Archimedean3)
apply (drule-tac  $x = x$  in spec, clarify)
apply (subgoal-tac  $x < \text{real}(LEAST\ m::nat. x < \text{real } m * y) * y$ )
  prefer 2 apply (erule LeastI)
apply (case-tac LEAST  $m::nat. x < \text{real } m * y$ , simp)
apply (subgoal-tac  $\sim x < \text{real } nat * y$ )
  prefer 2 apply (rule not-less-Least, simp, force)
done

```

**lemma** cos-zero-lemma:

```

  [|  $0 \leq x; \cos x = 0$  |] ==>
     $\exists n::nat. \sim \text{even } n \ \& \ x = \text{real } n * (pi/2)$ 
apply (drule pi-gt-zero [THEN reals-Archimedean4], safe)
apply (subgoal-tac  $0 \leq x - \text{real } n * pi \ \& \$ 
         $(x - \text{real } n * pi) \leq pi \ \& \ (\cos (x - \text{real } n * pi) = 0)$  )
apply (auto simp add: algebra-simps real-of-nat-Suc)
  prefer 2 apply (simp add: cos-diff)
apply (simp add: cos-diff)
apply (subgoal-tac EX!  $x. 0 \leq x \ \& \ x \leq pi \ \& \ \cos x = 0$ )
apply (rule-tac [2] cos-total, safe)
apply (drule-tac  $x = x - \text{real } n * pi$  in spec)
apply (drule-tac  $x = pi/2$  in spec)
apply (simp add: cos-diff)
apply (rule-tac  $x = Suc\ (2 * n)$  in exI)

```

**apply** (*simp add: real-of-nat-Suc algebra-simps, auto*)  
**done**

**lemma** *sin-zero-lemma*:

$[[0 \leq x; \sin x = 0]] \implies$   
 $\exists n::\text{nat. even } n \ \& \ x = \text{real } n * (\pi/2)$   
**apply** (*subgoal-tac  $\exists n::\text{nat. } \sim \text{even } n \ \& \ x + \pi/2 = \text{real } n * (\pi/2)$* )  
**apply** (*clarify, rule-tac  $x = n - 1$  in exI*)  
**apply** (*force simp add: odd-Suc-mult-two-ex real-of-nat-Suc left-distrib*)  
**apply** (*rule cos-zero-lemma*)  
**apply** (*simp-all add: add-increasing*)  
**done**

**lemma** *cos-zero-iff*:

$(\cos x = 0) =$   
 $((\exists n::\text{nat. } \sim \text{even } n \ \& \ (x = \text{real } n * (\pi/2))) \mid$   
 $(\exists n::\text{nat. } \sim \text{even } n \ \& \ (x = -(\text{real } n * (\pi/2))))$   
**apply** (*rule iffI*)  
**apply** (*cut-tac linorder-linear [of 0 x], safe*)  
**apply** (*drule cos-zero-lemma, assumption+*)  
**apply** (*cut-tac  $x = -x$  in cos-zero-lemma, simp, simp*)  
**apply** (*force simp add: minus-equation-iff [of x]*)  
**apply** (*auto simp only: odd-Suc-mult-two-ex real-of-nat-Suc left-distrib*)  
**apply** (*auto simp add: cos-add*)  
**done**

**lemma** *sin-zero-iff*:

$(\sin x = 0) =$   
 $((\exists n::\text{nat. even } n \ \& \ (x = \text{real } n * (\pi/2))) \mid$   
 $(\exists n::\text{nat. even } n \ \& \ (x = -(\text{real } n * (\pi/2))))$   
**apply** (*rule iffI*)  
**apply** (*cut-tac linorder-linear [of 0 x], safe*)  
**apply** (*drule sin-zero-lemma, assumption+*)  
**apply** (*cut-tac  $x = -x$  in sin-zero-lemma, simp, simp, safe*)  
**apply** (*force simp add: minus-equation-iff [of x]*)  
**apply** (*auto simp add: even-mult-two-ex*)  
**done**

**lemma** *cos-monotone-0-pi*: **assumes**  $0 \leq y$  **and**  $y < x$  **and**  $x \leq \pi$

**shows**  $\cos x < \cos y$

**proof** –

**have**  $-(x - y) < 0$  **using** *assms* **by** *auto*

**from** *MVT2[OF  $\langle y < x \rangle$  DERIV-cos[THEN impI, THEN all]]*

**obtain**  $z$  **where**  $y < z$  **and**  $z < x$  **and** *cos-diff*:  $\cos x - \cos y = (x - y) * -\sin z$  **by** *auto*

**hence**  $0 < z$  **and**  $z < \pi$  **using** *assms* **by** *auto*

hence  $0 < \sin z$  **using** *sin-gt-zero-pi* **by** *auto*  
 hence  $\cos x - \cos y < 0$  **unfolding** *cos-diff minus-mult-commute[symmetric]*  
**using**  $\langle -(x - y) < 0 \rangle$  **by** *(rule mult-pos-neg2)*  
 thus *?thesis* **by** *auto*  
**qed**

**lemma** *cos-monotone-0-pi'*: **assumes**  $0 \leq y$  **and**  $y \leq x$  **and**  $x \leq \pi$  **shows**  $\cos x \leq \cos y$   
**proof** *(cases  $y < x$ )*  
 case *True* **show** *?thesis* **using** *cos-monotone-0-pi[OF  $\langle 0 \leq y \rangle$  True  $\langle x \leq \pi \rangle$ ]*  
**by** *auto*  
**next**  
 case *False* **hence**  $y = x$  **using**  $\langle y \leq x \rangle$  **by** *auto*  
 thus *?thesis* **by** *auto*  
**qed**

**lemma** *cos-monotone-minus-pi-0*: **assumes**  $-\pi \leq y$  **and**  $y < x$  **and**  $x \leq 0$   
**shows**  $\cos y < \cos x$   
**proof**  $-$   
 have  $0 \leq -x$  **and**  $-x < -y$  **and**  $-y \leq \pi$  **using** *assms* **by** *auto*  
 from *cos-monotone-0-pi[OF this]*  
 show *?thesis* **unfolding** *cos-minus* .  
**qed**

**lemma** *cos-monotone-minus-pi-0'*: **assumes**  $-\pi \leq y$  **and**  $y \leq x$  **and**  $x \leq 0$   
**shows**  $\cos y \leq \cos x$   
**proof** *(cases  $y < x$ )*  
 case *True* **show** *?thesis* **using** *cos-monotone-minus-pi-0[OF  $\langle -\pi \leq y \rangle$  True  $\langle x \leq 0 \rangle$ ]* **by** *auto*  
**next**  
 case *False* **hence**  $y = x$  **using**  $\langle y \leq x \rangle$  **by** *auto*  
 thus *?thesis* **by** *auto*  
**qed**

**lemma** *sin-monotone-2pi'*: **assumes**  $-(\pi / 2) \leq y$  **and**  $y \leq x$  **and**  $x \leq \pi / 2$   
**shows**  $\sin y \leq \sin x$   
**proof**  $-$   
 have  $0 \leq y + \pi / 2$  **and**  $y + \pi / 2 \leq x + \pi / 2$  **and**  $x + \pi / 2 \leq \pi$   
 using *pi-ge-two* **and** *assms* **by** *auto*  
 from *cos-monotone-0-pi'[OF this]* **show** *?thesis* **unfolding** *minus-sin-cos-eq[symmetric]*  
**by** *auto*  
**qed**

## 76.11 Tangent

### definition

*tan* :: *real* => *real* **where**  
*tan*  $x = (\sin x) / (\cos x)$

**lemma** *tan-zero* [*simp*]:  $\tan 0 = 0$   
**by** (*simp add: tan-def*)

**lemma** *tan-pi* [*simp*]:  $\tan \pi = 0$   
**by** (*simp add: tan-def*)

**lemma** *tan-npi* [*simp*]:  $\tan (\text{real } (n::\text{nat}) * \pi) = 0$   
**by** (*simp add: tan-def*)

**lemma** *tan-minus* [*simp*]:  $\tan (-x) = -\tan x$   
**by** (*simp add: tan-def minus-mult-left*)

**lemma** *tan-periodic* [*simp*]:  $\tan (x + 2*\pi) = \tan x$   
**by** (*simp add: tan-def*)

**lemma** *lemma-tan-add1*:  

$$[| \cos x \neq 0; \cos y \neq 0 |]$$

$$\implies 1 - \tan(x)*\tan(y) = \cos (x + y)/(\cos x * \cos y)$$
**apply** (*simp add: tan-def divide-inverse*)  
**apply** (*auto simp del: inverse-mult-distrib*  

$$\text{simp add: inverse-mult-distrib [symmetric] mult-ac}$$
)  
**apply** (*rule-tac c1 = cos x \* cos y in real-mult-right-cancel [THEN subst]*)  
**apply** (*auto simp del: inverse-mult-distrib*  

$$\text{simp add: mult-assoc left-diff-distrib cos-add}$$
)  
**done**

**lemma** *add-tan-eq*:  

$$[| \cos x \neq 0; \cos y \neq 0 |]$$

$$\implies \tan x + \tan y = \sin(x + y)/(\cos x * \cos y)$$
**apply** (*simp add: tan-def*)  
**apply** (*rule-tac c1 = cos x \* cos y in real-mult-right-cancel [THEN subst]*)  
**apply** (*auto simp add: mult-assoc left-distrib*)  
**apply** (*simp add: sin-add*)  
**done**

**lemma** *tan-add*:  

$$[| \cos x \neq 0; \cos y \neq 0; \cos (x + y) \neq 0 |]$$

$$\implies \tan(x + y) = (\tan(x) + \tan(y))/(1 - \tan(x) * \tan(y))$$
**apply** (*simp (no-asm-simp) add: add-tan-eq lemma-tan-add1*)  
**apply** (*simp add: tan-def*)  
**done**

**lemma** *tan-double*:  

$$[| \cos x \neq 0; \cos (2 * x) \neq 0 |]$$

$$\implies \tan (2 * x) = (2 * \tan x)/(1 - (\tan(x) ^ 2))$$
**apply** (*insert tan-add [of x x]*)  
**apply** (*simp add: mult-2 [symmetric]*)  
**apply** (*auto simp add: numeral-2-eq-2*)  
**done**

**lemma** *tan-gt-zero*:  $[| 0 < x; x < \pi/2 |] \implies 0 < \tan x$   
**by** (*simp add: tan-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi*)

**lemma** *tan-less-zero*:  
**assumes** *lb*:  $-\pi/2 < x$  **and**  $x < 0$  **shows**  $\tan x < 0$   
**proof** –  
**have**  $0 < \tan (-x)$  **using** *prems* **by** (*simp only: tan-gt-zero*)  
**thus** *?thesis* **by** *simp*  
**qed**

**lemma** *tan-half*: **fixes**  $x :: \text{real}$  **assumes**  $-\pi/2 < x$  **and**  $x < \pi/2$   
**shows**  $\tan x = \sin (2 * x) / (\cos (2 * x) + 1)$   
**proof** –  
**from** *cos-gt-zero-pi* [*OF*  $\langle -\pi/2 < x \rangle \langle x < \pi/2 \rangle$ ]  
**have**  $\cos x \neq 0$  **by** *auto*

**have** *minus-cos-2x*:  $\bigwedge X. X - \cos (2*x) = X - (\cos x)^2 + (\sin x)^2$   
**unfolding** *cos-double* **by** *algebra*

**have**  $\tan x = (\tan x + \tan x) / 2$  **by** *auto*  
**also have**  $\dots = \sin (x + x) / (\cos x * \cos x) / 2$  **unfolding** *add-tan-eq* [*OF*  $\langle \cos x \neq 0 \rangle \langle \cos x \neq 0 \rangle$ ] **..**  
**also have**  $\dots = \sin (2 * x) / ((\cos x)^2 + (\cos x)^2 + \cos (2*x) - \cos (2*x))$  **unfolding** *divide-divide-eq-left numeral-2-eq-2* **by** *auto*  
**also have**  $\dots = \sin (2 * x) / ((\cos x)^2 + \cos (2*x) + (\sin x)^2)$  **unfolding** *minus-cos-2x* **by** *auto*  
**also have**  $\dots = \sin (2 * x) / (\cos (2*x) + 1)$  **by** *auto*  
**finally show** *?thesis* .  
**qed**

**lemma** *lemma-DERIV-tan*:  
 $\cos x \neq 0 \implies \text{DERIV } (\%x. \sin(x)/\cos(x)) x :> \text{inverse}((\cos x)^2)$   
**by** (*auto intro!: DERIV-intros simp add: field-simps numeral-2-eq-2*)

**lemma** *DERIV-tan [simp]*:  $\cos x \neq 0 \implies \text{DERIV } \tan x :> \text{inverse}((\cos x)^2)$   
**by** (*auto dest: lemma-DERIV-tan simp add: tan-def [symmetric]*)

**lemma** *isCont-tan [simp]*:  $\cos x \neq 0 \implies \text{isCont } \tan x$   
**by** (*rule DERIV-tan [THEN DERIV-isCont]*)

**lemma** *LIM-cos-div-sin [simp]*:  $(\%x. \cos(x)/\sin(x)) \text{ -- } \pi/2 \text{ --> } 0$   
**apply** (*subgoal-tac*  $(\lambda x. \cos x * \text{inverse } (\sin x)) \text{ -- } \pi * \text{inverse } 2 \text{ --> } 0 * 1$ )  
**apply** (*simp add: divide-inverse [symmetric]*)  
**apply** (*rule LIM-mult*)  
**apply** (*rule-tac* [2] *inverse-1 [THEN subst]*)  
**apply** (*rule-tac* [2] *LIM-inverse*)  
**apply** (*simp-all add: divide-inverse [symmetric]*)  
**apply** (*simp-all only: isCont-def [symmetric] cos-pi-half [symmetric] sin-pi-half*)



```
[symmetric])
apply (blast intro!: DERIV-isCont DERIV-sin DERIV-cos)+
done
```

```
lemma lemma-tan-total:  $0 < y \implies \exists x. 0 < x \ \& \ x < \pi/2 \ \& \ y < \tan x$ 
apply (cut-tac LIM-cos-div-sin)
apply (simp only: LIM-eq)
apply (drule-tac  $x = \text{inverse } y$  in spec, safe, force)
apply (drule-tac ?d1.0 = s in pi-half-gt-zero [THEN [2] real-lbound-gt-zero], safe)
apply (rule-tac  $x = (\pi/2) - e$  in exI)
apply (simp (no-asm-simp))
apply (drule-tac  $x = (\pi/2) - e$  in spec)
apply (auto simp add: tan-def)
apply (rule inverse-less-iff-less [THEN iffD1])
apply (auto simp add: divide-inverse)
apply (rule mult-pos-pos)
apply (subgoal-tac [3]  $0 < \sin e \ \& \ 0 < \cos e$ )
apply (auto intro: cos-gt-zero sin-gt-zero2 simp add: mult-commute)
done
```

```
lemma tan-total-pos:  $0 \leq y \implies \exists x. 0 \leq x \ \& \ x < \pi/2 \ \& \ \tan x = y$ 
apply (frule order-le-imp-less-or-eq, safe)
prefer 2 apply force
apply (drule lemma-tan-total, safe)
apply (cut-tac  $f = \tan$  and  $a = 0$  and  $b = x$  and  $y = y$  in IVT-objl)
apply (auto intro!: DERIV-tan [THEN DERIV-isCont])
apply (drule-tac  $y = xa$  in order-le-imp-less-or-eq)
apply (auto dest: cos-gt-zero)
done
```

```
lemma lemma-tan-total1:  $\exists x. -(\pi/2) < x \ \& \ x < (\pi/2) \ \& \ \tan x = y$ 
apply (cut-tac linorder-linear [of 0 y], safe)
apply (drule tan-total-pos)
apply (cut-tac [2]  $y = -y$  in tan-total-pos, safe)
apply (rule-tac [3]  $x = -x$  in exI)
apply (auto intro!: exI)
done
```

```
lemma tan-total: EX!  $x. -(\pi/2) < x \ \& \ x < (\pi/2) \ \& \ \tan x = y$ 
apply (cut-tac  $y = y$  in lemma-tan-total1, auto)
apply (cut-tac  $x = xa$  and  $y = y$  in linorder-less-linear, auto)
apply (subgoal-tac [2]  $\exists z. y < z \ \& \ z < xa \ \& \ \text{DERIV } \tan z \text{ :> } 0$ )
apply (subgoal-tac  $\exists z. xa < z \ \& \ z < y \ \& \ \text{DERIV } \tan z \text{ :> } 0$ )
apply (rule-tac [4] Rolle)
apply (rule-tac [2] Rolle)
apply (auto intro!: DERIV-tan DERIV-isCont exI
    simp add: differentiable-def)
```

Now, simulate TRYALL

```

apply (rule-tac [!] DERIV-tan asm-rl)
apply (auto dest!: DERIV-unique [OF - DERIV-tan]
      simp add: cos-gt-zero-pi [THEN less-imp-neq, THEN not-sym])
done

lemma tan-monotone: assumes  $-(\pi / 2) < y$  and  $y < x$  and  $x < \pi / 2$ 
shows  $\tan y < \tan x$ 
proof -
  have  $\forall x'. y \leq x' \wedge x' \leq x \longrightarrow \text{DERIV } \tan x' :> \text{inverse } (\cos x'^2)$ 
  proof (rule allI, rule impI)
    fix  $x' :: \text{real}$  assume  $y \leq x' \wedge x' \leq x$ 
    hence  $-(\pi/2) < x'$  and  $x' < \pi/2$  using assms by auto
    from cos-gt-zero-pi [OF this]
    have  $\cos x' \neq 0$  by auto
    thus  $\text{DERIV } \tan x' :> \text{inverse } (\cos x'^2)$  by (rule DERIV-tan)
  qed
  from MVT2 [OF  $\langle y < x \rangle$  this]
  obtain  $z$  where  $y < z$  and  $z < x$  and tan-diff:  $\tan x - \tan y = (x - y) * \text{inverse } ((\cos z)^2)$  by auto
  hence  $-(\pi / 2) < z$  and  $z < \pi / 2$  using assms by auto
  hence  $0 < \cos z$  using cos-gt-zero-pi by auto
  hence inv-pos:  $0 < \text{inverse } ((\cos z)^2)$  by auto
  have  $0 < x - y$  using  $\langle y < x \rangle$  by auto
  from mult-pos-pos [OF this inv-pos]
  have  $0 < \tan x - \tan y$  unfolding tan-diff by auto
  thus ?thesis by auto
qed

lemma tan-monotone': assumes  $-(\pi / 2) < y$  and  $y < \pi / 2$  and  $-(\pi / 2) < x$  and  $x < \pi / 2$ 
shows  $(y < x) = (\tan y < \tan x)$ 
proof
  assume  $y < x$  thus  $\tan y < \tan x$  using tan-monotone and  $\langle -(\pi / 2) < y \rangle$  and  $\langle x < \pi / 2 \rangle$  by auto
next
  assume  $\tan y < \tan x$ 
  show  $y < x$ 
  proof (rule ccontr)
    assume  $\neg y < x$  hence  $x \leq y$  by auto
    hence  $\tan x \leq \tan y$ 
    proof (cases  $x = y$ )
      case True thus ?thesis by auto
    next
      case False hence  $x < y$  using  $\langle x \leq y \rangle$  by auto
      from tan-monotone [OF  $\langle -(\pi/2) < x \rangle$  this  $\langle y < \pi / 2 \rangle$ ] show ?thesis by auto
    qed
  thus False using  $\langle \tan y < \tan x \rangle$  by auto
qed

```

qed

**lemma** *tan-inverse*:  $1 / (\tan y) = \tan (\pi / 2 - y)$  **unfolding** *tan-def sin-cos-eq*[of *y*] *cos-sin-eq*[of *y*] **by** *auto*

**lemma** *tan-periodic-pi*[*simp*]:  $\tan (x + \pi) = \tan x$   
**by** (*simp add: tan-def*)

**lemma** *tan-periodic-nat*[*simp*]: **fixes**  $n :: \text{nat}$  **shows**  $\tan (x + \text{real } n * \pi) = \tan x$

**proof** (*induct n arbitrary: x*)

**case** (*Suc n*)

**have** *split-pi-off*:  $x + \text{real } (\text{Suc } n) * \pi = (x + \text{real } n * \pi) + \pi$  **unfolding** *Suc-eq-plus1 real-of-nat-add real-of-one left-distrib* **by** *auto*

**show** ?*case* **unfolding** *split-pi-off* **using** *Suc* **by** *auto*

qed *auto*

**lemma** *tan-periodic-int*[*simp*]: **fixes**  $i :: \text{int}$  **shows**  $\tan (x + \text{real } i * \pi) = \tan x$

**proof** (*cases*  $0 \leq i$ )

**case** *True* **hence** *i-nat*:  $\text{real } i = \text{real } (\text{nat } i)$  **by** *auto*

**show** ?*thesis* **unfolding** *i-nat* **by** *auto*

**next**

**case** *False* **hence** *i-nat*:  $\text{real } i = - \text{real } (\text{nat } (-i))$  **by** *auto*

**have**  $\tan x = \tan (x + \text{real } i * \pi - \text{real } i * \pi)$  **by** *auto*

**also have**  $\dots = \tan (x + \text{real } i * \pi)$  **unfolding** *i-nat mult-minus-left diff-minus-eq-add*  
**by** (*rule tan-periodic-nat*)

**finally show** ?*thesis* **by** *auto*

qed

**lemma** *tan-periodic-n*[*simp*]:  $\tan (x + \text{number-of } n * \pi) = \tan x$

**using** *tan-periodic-int*[of - *number-of n* ] **unfolding** *real-number-of* .

## 76.12 Inverse Trigonometric Functions

**definition**

*arcsin* :: *real* => *real* **where**

*arcsin y* = (*THE*  $x. -( \pi / 2 ) \leq x \ \& \ x \leq \pi / 2 \ \& \ \sin x = y$ )

**definition**

*arccos* :: *real* => *real* **where**

*arccos y* = (*THE*  $x. 0 \leq x \ \& \ x \leq \pi \ \& \ \cos x = y$ )

**definition**

*arctan* :: *real* => *real* **where**

*arctan y* = (*THE*  $x. -( \pi / 2 ) < x \ \& \ x < \pi / 2 \ \& \ \tan x = y$ )

**lemma** *arcsin*:

$[[-1 \leq y; y \leq 1]]$

$\implies -( \pi / 2 ) \leq \text{arcsin } y \ \&$

$\arcsin y \leq \pi/2 \ \& \ \sin(\arcsin y) = y$   
**unfolding** *arcsin-def* **by** (*rule theI'* [*OF sin-total*])

**lemma** *arcsin-pi*:

$[-1 \leq y; y \leq 1] \implies -(\pi/2) \leq \arcsin y \ \& \ \arcsin y \leq \pi \ \& \ \sin(\arcsin y) = y$   
**apply** (*drule* (1) *arcsin*)  
**apply** (*force intro: order-trans*)  
**done**

**lemma** *sin-arcsin* [*simp*]:  $[-1 \leq y; y \leq 1] \implies \sin(\arcsin y) = y$   
**by** (*blast dest: arcsin*)

**lemma** *arcsin-bounded*:

$[-1 \leq y; y \leq 1] \implies -(\pi/2) \leq \arcsin y \ \& \ \arcsin y \leq \pi/2$   
**by** (*blast dest: arcsin*)

**lemma** *arcsin-lbound*:  $[-1 \leq y; y \leq 1] \implies -(\pi/2) \leq \arcsin y$   
**by** (*blast dest: arcsin*)

**lemma** *arcsin-ubound*:  $[-1 \leq y; y \leq 1] \implies \arcsin y \leq \pi/2$   
**by** (*blast dest: arcsin*)

**lemma** *arcsin-lt-bounded*:

$[-1 < y; y < 1] \implies -(\pi/2) < \arcsin y \ \& \ \arcsin y < \pi/2$   
**apply** (*frule order-less-imp-le*)  
**apply** (*frule-tac*  $y = y$  **in** *order-less-imp-le*)  
**apply** (*frule arcsin-bounded*)  
**apply** (*safe, simp*)  
**apply** (*drule-tac*  $y = \arcsin y$  **in** *order-le-imp-less-or-eq*)  
**apply** (*drule-tac* [2]  $y = \pi/2$  **in** *order-le-imp-less-or-eq, safe*)  
**apply** (*drule-tac* [!]  $f = \sin$  **in** *arg-cong, auto*)  
**done**

**lemma** *arcsin-sin*:  $[-(\pi/2) \leq x; x \leq \pi/2] \implies \arcsin(\sin x) = x$   
**apply** (*unfold arcsin-def*)  
**apply** (*rule the1-equality*)  
**apply** (*rule sin-total, auto*)  
**done**

**lemma** *arccos*:

$[-1 \leq y; y \leq 1] \implies 0 \leq \arccos y \ \& \ \arccos y \leq \pi \ \& \ \cos(\arccos y) = y$   
**unfolding** *arccos-def* **by** (*rule theI'* [*OF cos-total*])

**lemma** *cos-arccos* [*simp*]:  $[-1 \leq y; y \leq 1] \implies \cos(\arccos y) = y$   
**by** (*blast dest: arccos*)

**lemma** *arccos-bounded*:  $[-1 \leq y; y \leq 1] \implies 0 \leq \arccos y \ \& \ \arccos y \leq \pi$

**by** (*blast dest: arccos*)

**lemma** *arccos-lbound*:  $\llbracket -1 \leq y; y \leq 1 \rrbracket \implies 0 \leq \arccos y$   
**by** (*blast dest: arccos*)

**lemma** *arccos-ubound*:  $\llbracket -1 \leq y; y \leq 1 \rrbracket \implies \arccos y \leq \pi$   
**by** (*blast dest: arccos*)

**lemma** *arccos-lt-bounded*:  
 $\llbracket -1 < y; y < 1 \rrbracket \implies 0 < \arccos y \ \& \ \arccos y < \pi$   
**apply** (*frule order-less-imp-le*)  
**apply** (*frule-tac y = y in order-less-imp-le*)  
**apply** (*frule arccos-bounded, auto*)  
**apply** (*drule-tac y = arccos y in order-le-imp-less-or-eq*)  
**apply** (*drule-tac [2] y = pi in order-le-imp-less-or-eq, auto*)  
**apply** (*drule-tac [!] f = cos in arg-cong, auto*)  
**done**

**lemma** *arccos-cos*:  $\llbracket 0 \leq x; x \leq \pi \rrbracket \implies \arccos(\cos x) = x$   
**apply** (*simp add: arccos-def*)  
**apply** (*auto intro!: the1-equality cos-total*)  
**done**

**lemma** *arccos-cos2*:  $\llbracket x \leq 0; -\pi \leq x \rrbracket \implies \arccos(\cos x) = -x$   
**apply** (*simp add: arccos-def*)  
**apply** (*auto intro!: the1-equality cos-total*)  
**done**

**lemma** *cos-arcsin*:  $\llbracket -1 \leq x; x \leq 1 \rrbracket \implies \cos(\arcsin x) = \sqrt{1 - x^2}$   
**apply** (*subgoal-tac  $x^2 \leq 1$* )  
**apply** (*rule power2-eq-imp-eq*)  
**apply** (*simp add: cos-squared-eq*)  
**apply** (*rule cos-ge-zero*)  
**apply** (*erule (1) arcsin-lbound*)  
**apply** (*erule (1) arcsin-ubound*)  
**apply** *simp*  
**apply** (*subgoal-tac  $|x|^2 \leq 1^2$ , simp*)  
**apply** (*rule power-mono, simp, simp*)  
**done**

**lemma** *sin-arccos*:  $\llbracket -1 \leq x; x \leq 1 \rrbracket \implies \sin(\arccos x) = \sqrt{1 - x^2}$   
**apply** (*subgoal-tac  $x^2 \leq 1$* )  
**apply** (*rule power2-eq-imp-eq*)  
**apply** (*simp add: sin-squared-eq*)  
**apply** (*rule sin-ge-zero*)  
**apply** (*erule (1) arccos-lbound*)  
**apply** (*erule (1) arccos-ubound*)  
**apply** *simp*

```

apply (subgoal-tac  $|x|^2 \leq 1^2$ , simp)
apply (rule power-mono, simp, simp)
done

```

```

lemma arctan [simp]:
   $-(\pi/2) < \arctan y \ \& \ \arctan y < \pi/2 \ \& \ \tan(\arctan y) = y$ 
unfolding arctan-def by (rule theI' [OF tan-total])

```

```

lemma tan-arctan:  $\tan(\arctan y) = y$ 
by auto

```

```

lemma arctan-bounded:  $-(\pi/2) < \arctan y \ \& \ \arctan y < \pi/2$ 
by (auto simp only: arctan)

```

```

lemma arctan-lbound:  $-(\pi/2) < \arctan y$ 
by auto

```

```

lemma arctan-ubound:  $\arctan y < \pi/2$ 
by (auto simp only: arctan)

```

```

lemma arctan-tan:
   $[|-(\pi/2) < x; x < \pi/2|] ==> \arctan(\tan x) = x$ 
apply (unfold arctan-def)
apply (rule the1-equality)
apply (rule tan-total, auto)
done

```

```

lemma arctan-zero-zero [simp]:  $\arctan 0 = 0$ 
by (insert arctan-tan [of 0], simp)

```

```

lemma cos-arctan-not-zero [simp]:  $\cos(\arctan x) \neq 0$ 
apply (auto simp add: cos-zero-iff)
apply (case-tac n)
apply (case-tac [3] n)
apply (cut-tac [2]  $y = x$  in arctan-ubound)
apply (cut-tac [4]  $y = x$  in arctan-lbound)
apply (auto simp add: real-of-nat-Suc left-distrib mult-less-0-iff)
done

```

```

lemma tan-sec:  $\cos x \neq 0 ==> 1 + \tan(x)^2 = \text{inverse}(\cos x)^2$ 
apply (rule power-inverse [THEN subst])
apply (rule-tac  $c1 = (\cos x)^2$  in real-mult-right-cancel [THEN iffD1])
apply (auto dest: field-power-not-zero
  simp add: power-mult-distrib left-distrib power-divide tan-def
  mult-assoc power-inverse [symmetric])
done

```

```

lemma isCont-inverse-function2:
  fixes  $f g :: \text{real} \Rightarrow \text{real}$  shows

```

```


$$\begin{aligned} & \llbracket a < x; x < b; \\ & \quad \forall z. a \leq z \wedge z \leq b \longrightarrow g(fz) = z; \\ & \quad \forall z. a \leq z \wedge z \leq b \longrightarrow \text{isCont } fz \rrbracket \\ & \implies \text{isCont } g(fx) \end{aligned}$$

apply (rule isCont-inverse-function
  [where f=f and d=min (x - a) (b - x)])
apply (simp-all add: abs-le-iff)
done

```

```

lemma isCont-arcsin:  $\llbracket -1 < x; x < 1 \rrbracket \implies \text{isCont } \arcsin x$ 
apply (subgoal-tac isCont arcsin (sin (arcsin x)), simp)
apply (rule isCont-inverse-function2 [where f=sin])
apply (erule (1) arcsin-lt-bounded [THEN conjunct1])
apply (erule (1) arcsin-lt-bounded [THEN conjunct2])
apply (fast intro: arcsin-sin, simp)
done

```

```

lemma isCont-arccos:  $\llbracket -1 < x; x < 1 \rrbracket \implies \text{isCont } \arccos x$ 
apply (subgoal-tac isCont arccos (cos (arccos x)), simp)
apply (rule isCont-inverse-function2 [where f=cos])
apply (erule (1) arccos-lt-bounded [THEN conjunct1])
apply (erule (1) arccos-lt-bounded [THEN conjunct2])
apply (fast intro: arccos-cos, simp)
done

```

```

lemma isCont-arctan: isCont arctan x
apply (rule arctan-lbound [of x, THEN dense, THEN exE], clarify)
apply (rule arctan-ubound [of x, THEN dense, THEN exE], clarify)
apply (subgoal-tac isCont arctan (tan (arctan x)), simp)
apply (erule (1) isCont-inverse-function2 [where f=tan])
apply (metis arctan-tan order-le-less-trans order-less-le-trans)
apply (metis cos-gt-zero-pi isCont-tan order-less-le-trans less-le)
done

```

```

lemma DERIV-arcsin:
 $\llbracket -1 < x; x < 1 \rrbracket \implies \text{DERIV } \arcsin x :> \text{inverse } (\text{sqrt } (1 - x^2))$ 
apply (rule DERIV-inverse-function [where f=sin and a=-1 and b=1])
apply (rule lemma-DERIV-subst [OF DERIV-sin])
apply (simp add: cos-arcsin)
apply (subgoal-tac  $|x|^2 < 1^2$ , simp)
apply (rule power-strict-mono, simp, simp, simp)
apply assumption
apply assumption
apply simp
apply (erule (1) isCont-arcsin)
done

```

```

lemma DERIV-arccos:
 $\llbracket -1 < x; x < 1 \rrbracket \implies \text{DERIV } \arccos x :> \text{inverse } (- \text{sqrt } (1 - x^2))$ 

```

```

apply (rule DERIV-inverse-function [where  $f=\cos$  and  $a=-1$  and  $b=1$ ])
apply (rule lemma-DERIV-subst [OF DERIV-cos])
apply (simp add: sin-arccos)
apply (subgoal-tac  $|x|^2 < 1^2$ , simp)
apply (rule power-strict-mono, simp, simp, simp)
apply assumption
apply assumption
apply simp
apply (erule (1) isCont-arccos)
done

```

```

lemma DERIV-arctan: DERIV arctan  $x :>$  inverse ( $1 + x^2$ )
apply (rule DERIV-inverse-function [where  $f=\tan$  and  $a=x - 1$  and  $b=x + 1$ ])
apply (rule lemma-DERIV-subst [OF DERIV-tan])
apply (rule cos-arctan-not-zero)
apply (simp add: power-inverse tan-sec [symmetric])
apply (subgoal-tac  $0 < 1 + x^2$ , simp)
apply (simp add: add-pos-nonneg)
apply (simp, simp, simp, rule isCont-arctan)
done

```

**declare**

```

  DERIV-arcsin[THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros]
  DERIV-arccos[THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros]
  DERIV-arctan[THEN DERIV-chain2, THEN DERIV-cong, DERIV-intros]

```

### 76.13 More Theorems about Sin and Cos

**lemma** *cos-45*:  $\cos (\pi / 4) = \text{sqrt } 2 / 2$

**proof** –

```

  let ?c =  $\cos (\pi / 4)$  and ?s =  $\sin (\pi / 4)$ 
  have nonneg:  $0 \leq ?c$ 
    by (rule cos-ge-zero, rule order-trans [where  $y=0$ ], simp-all)
  have  $0 = \cos (\pi / 4 + \pi / 4)$ 
    by simp
  also have  $\cos (\pi / 4 + \pi / 4) = ?c^2 - ?s^2$ 
    by (simp only: cos-add power2-eq-square)
  also have  $\dots = 2 * ?c^2 - 1$ 
    by (simp add: sin-squared-eq)
  finally have  $?c^2 = (\text{sqrt } 2 / 2)^2$ 
    by (simp add: power-divide)
  thus ?thesis
    using nonneg by (rule power2-eq-imp-eq) simp
qed

```

**lemma** *cos-30*:  $\cos (\pi / 6) = \text{sqrt } 3 / 2$

**proof** –

```

  let ?c =  $\cos (\pi / 6)$  and ?s =  $\sin (\pi / 6)$ 

```



```

have pos-c: 0 < ?c
  by (rule cos-gt-zero, simp, simp)
have 0 = cos (pi / 6 + pi / 6 + pi / 6)
  by simp
also have ... = (?c * ?c - ?s * ?s) * ?c - (?s * ?c + ?c * ?s) * ?s
  by (simp only: cos-add sin-add)
also have ... = ?c * (?c2 - 3 * ?s2)
  by (simp add: algebra-simps power2-eq-square)
finally have ?c2 = (sqrt 3 / 2)2
  using pos-c by (simp add: sin-squared-eq power-divide)
thus ?thesis
  using pos-c [THEN order-less-imp-le]
  by (rule power2-eq-imp-eq) simp
qed

```

```

lemma sin-45: sin (pi / 4) = sqrt 2 / 2
proof -
  have sin (pi / 4) = cos (pi / 2 - pi / 4) by (rule sin-cos-eq)
  also have pi / 2 - pi / 4 = pi / 4 by simp
  also have cos (pi / 4) = sqrt 2 / 2 by (rule cos-45)
  finally show ?thesis .
qed

```

```

lemma sin-60: sin (pi / 3) = sqrt 3 / 2
proof -
  have sin (pi / 3) = cos (pi / 2 - pi / 3) by (rule sin-cos-eq)
  also have pi / 2 - pi / 3 = pi / 6 by simp
  also have cos (pi / 6) = sqrt 3 / 2 by (rule cos-30)
  finally show ?thesis .
qed

```

```

lemma cos-60: cos (pi / 3) = 1 / 2
apply (rule power2-eq-imp-eq)
apply (simp add: cos-squared-eq sin-60 power-divide)
apply (rule cos-ge-zero, rule order-trans [where y=0], simp-all)
done

```

```

lemma sin-30: sin (pi / 6) = 1 / 2
proof -
  have sin (pi / 6) = cos (pi / 2 - pi / 6) by (rule sin-cos-eq)
  also have pi / 2 - pi / 6 = pi / 3 by simp
  also have cos (pi / 3) = 1 / 2 by (rule cos-60)
  finally show ?thesis .
qed

```

```

lemma tan-30: tan (pi / 6) = 1 / sqrt 3
unfolding tan-def by (simp add: sin-30 cos-30)

```

```

lemma tan-45: tan (pi / 4) = 1

```

**unfolding** *tan-def* **by** (*simp add: sin-45 cos-45*)

**lemma** *tan-60*:  $\tan (\pi / 3) = \text{sqrt } 3$

**unfolding** *tan-def* **by** (*simp add: sin-60 cos-60*)

**lemma** *DERIV-sin-add* [*simp*]:  $\text{DERIV } (\%x. \sin (x + k)) \text{ } xa :> \cos (xa + k)$   
**by** (*auto intro!: DERIV-intros*)

**lemma** *sin-cos-npi* [*simp*]:  $\sin (\text{real } (\text{Suc } (2 * n)) * \pi / 2) = (-1) ^ n$

**proof** –

**have**  $\sin ((\text{real } n + 1/2) * \pi) = \cos (\text{real } n * \pi)$

**by** (*auto simp add: algebra-simps sin-add*)

**thus** *?thesis*

**by** (*simp add: real-of-nat-Suc left-distrib add-divide-distrib  
mult-commute [of pi]*)

**qed**

**lemma** *cos-2npi* [*simp*]:  $\cos (2 * \text{real } (n::\text{nat}) * \pi) = 1$

**by** (*simp add: cos-double mult-assoc power-add [symmetric] numeral-2-eq-2*)

**lemma** *cos-3over2-pi* [*simp*]:  $\cos (3 / 2 * \pi) = 0$

**apply** (*subgoal-tac cos (pi + pi/2) = 0, simp*)

**apply** (*subst cos-add, simp*)

**done**

**lemma** *sin-2npi* [*simp*]:  $\sin (2 * \text{real } (n::\text{nat}) * \pi) = 0$

**by** (*auto simp add: mult-assoc*)

**lemma** *sin-3over2-pi* [*simp*]:  $\sin (3 / 2 * \pi) = - 1$

**apply** (*subgoal-tac sin (pi + pi/2) = - 1, simp*)

**apply** (*subst sin-add, simp*)

**done**

**lemma** *cos-pi-eq-zero* [*simp*]:  $\cos (\pi * \text{real } (\text{Suc } (2 * m)) / 2) = 0$

**by** (*simp only: cos-add sin-add real-of-nat-Suc left-distrib right-distrib add-divide-distrib,  
auto*)

**lemma** *DERIV-cos-add* [*simp*]:  $\text{DERIV } (\%x. \cos (x + k)) \text{ } xa :> - \sin (xa + k)$

**by** (*auto intro!: DERIV-intros*)

**lemma** *sin-zero-abs-cos-one*:  $\sin x = 0 ==> |\cos x| = 1$

**by** (*auto simp add: sin-zero-iff even-mult-two-ex*)

**lemma** *cos-one-sin-zero*:  $\cos x = 1 ==> \sin x = 0$

**by** (*cut-tac x = x in sin-cos-squared-add3, auto*)

## 76.14 Machins formula

**lemma** *tan-total-pi4*: **assumes**  $|x| < 1$

shows  $\exists z. -(pi / 4) < z \wedge z < pi / 4 \wedge \tan z = x$   
**proof** –  
 obtain  $z$  where  $-(pi / 2) < z$  and  $z < pi / 2$  and  $\tan z = x$  using *tan-total*  
 by *blast*  
 have  $\tan (pi / 4) = 1$  and  $\tan (-(pi / 4)) = -1$  using *tan-45 tan-minus* by *auto*  
 have  $z \neq pi / 4$   
**proof** (*rule ccontr*)  
 assume  $\neg (z \neq pi / 4)$  hence  $z = pi / 4$  by *auto*  
 have  $\tan z = 1$  unfolding  $\langle z = pi / 4 \rangle \langle \tan (pi / 4) = 1 \rangle$ ..  
 thus *False* unfolding  $\langle \tan z = x \rangle$  using  $\langle |x| < 1 \rangle$  by *auto*  
 qed  
 have  $z \neq -(pi / 4)$   
**proof** (*rule ccontr*)  
 assume  $\neg (z \neq -(pi / 4))$  hence  $z = -(pi / 4)$  by *auto*  
 have  $\tan z = -1$  unfolding  $\langle z = -(pi / 4) \rangle \langle \tan (-(pi / 4)) = -1 \rangle$ ..  
 thus *False* unfolding  $\langle \tan z = x \rangle$  using  $\langle |x| < 1 \rangle$  by *auto*  
 qed  
 have  $z < pi / 4$   
**proof** (*rule ccontr*)  
 assume  $\neg (z < pi / 4)$  hence  $pi / 4 < z$  using  $\langle z \neq pi / 4 \rangle$  by *auto*  
 have  $-(pi / 2) < pi / 4$  using *m2pi-less-pi* by *auto*  
 from *tan-monotone*[*OF* this  $\langle pi / 4 < z \rangle \langle z < pi / 2 \rangle$ ]  
 have  $1 < x$  unfolding  $\langle \tan z = x \rangle \langle \tan (pi / 4) = 1 \rangle$ ..  
 thus *False* using  $\langle |x| < 1 \rangle$  by *auto*  
 qed  
 moreover  
 have  $-(pi / 4) < z$   
**proof** (*rule ccontr*)  
 assume  $\neg (-(pi / 4) < z)$  hence  $z < -(pi / 4)$  using  $\langle z \neq -(pi / 4) \rangle$  by *auto*  
 have  $-(pi / 4) < pi / 2$  using *m2pi-less-pi* by *auto*  
 from *tan-monotone*[*OF*  $\langle -(pi / 2) < z \rangle \langle z < -(pi / 4) \rangle$  this]  
 have  $x < -1$  unfolding  $\langle \tan z = x \rangle \langle \tan (-(pi / 4)) = -1 \rangle$ ..  
 thus *False* using  $\langle |x| < 1 \rangle$  by *auto*  
 qed  
 ultimately show *?thesis* using  $\langle \tan z = x \rangle$  by *auto*  
 qed  
**lemma** *arctan-add*: assumes  $|x| \leq 1$  and  $|y| < 1$   
 shows  $\arctan x + \arctan y = \arctan ((x + y) / (1 - x * y))$   
**proof** –  
 obtain  $y'$  where  $-(pi/4) < y'$  and  $y' < pi/4$  and  $\tan y' = y$  using *tan-total-pi4*[*OF*  $\langle |y| < 1 \rangle$ ] by *blast*  
 have  $pi / 4 < pi / 2$  by *auto*  
 have  $\exists x'. -(pi/4) \leq x' \wedge x' \leq pi/4 \wedge \tan x' = x$

**proof** (cases  $|x| < 1$ )  
**case** *True* **from** *tan-total-pi4*[*OF this*] **obtain**  $x'$  **where**  $-(\pi/4) < x'$  **and**  $x' < \pi/4$  **and**  $\tan x' = x$  **by** *blast*  
**hence**  $-(\pi/4) \leq x'$  **and**  $x' \leq \pi/4$  **and**  $\tan x' = x$  **by** *auto*  
**thus** *?thesis* **by** *auto*  
**next**  
**case** *False*  
**show** *?thesis*  
**proof** (cases  $x = 1$ )  
**case** *True* **hence**  $\tan(\pi/4) = x$  **using** *tan-45* **by** *auto*  
**moreover**  
**have**  $-\pi \leq \pi$  **unfolding** *minus-le-self-iff* **by** *auto*  
**hence**  $-(\pi/4) \leq \pi/4$  **and**  $\pi/4 \leq \pi/4$  **by** *auto*  
**ultimately show** *?thesis* **by** *blast*  
**next**  
**case** *False* **hence**  $x = -1$  **using**  $\langle \neg |x| < 1 \rangle$  **and**  $\langle |x| \leq 1 \rangle$  **by** *auto*  
**hence**  $\tan(-(\pi/4)) = x$  **using** *tan-45 tan-minus* **by** *auto*  
**moreover**  
**have**  $-\pi \leq \pi$  **unfolding** *minus-le-self-iff* **by** *auto*  
**hence**  $-(\pi/4) \leq \pi/4$  **and**  $-(\pi/4) \leq -(\pi/4)$  **by** *auto*  
**ultimately show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**then obtain**  $x'$  **where**  $-(\pi/4) \leq x'$  **and**  $x' \leq \pi/4$  **and**  $\tan x' = x$  **by** *blast*  
**hence**  $-(\pi/2) < x'$  **and**  $x' < \pi/2$  **using** *order-le-less-trans*[*OF*  $\langle x' \leq \pi/4 \rangle$ ]  
 $\langle \pi/4 < \pi/2 \rangle$  **by** *auto*  
  
**have**  $\cos x' \neq 0$  **using** *cos-gt-zero-pi*[*THEN less-imp-neq*] **and**  $\langle -(\pi/2) < x' \rangle$   
**and**  $\langle x' < \pi/2 \rangle$  **by** *auto*  
**moreover have**  $\cos y' \neq 0$  **using** *cos-gt-zero-pi*[*THEN less-imp-neq*] **and**  $\langle -(\pi/4) < y' \rangle$   
**and**  $\langle y' < \pi/4 \rangle$  **by** *auto*  
**ultimately have**  $\cos x' * \cos y' \neq 0$  **by** *auto*  
  
**have** *divide-nonzero-divide*:  $\bigwedge A B C :: \text{real. } C \neq 0 \implies (A / C) / (B / C) = A / B$  **by** *auto*  
**have** *divide-mult-commute*:  $\bigwedge A B C D :: \text{real. } A * B / (C * D) = (A / C) * (B / D)$  **by** *auto*  
  
**have**  $\tan(x' + y') = \sin(x' + y') / (\cos x' * \cos y' - \sin x' * \sin y')$  **unfolding** *tan-def cos-add* ..  
**also have**  $\dots = (\tan x' + \tan y') / ((\cos x' * \cos y' - \sin x' * \sin y') / (\cos x' * \cos y'))$  **unfolding** *add-tan-eq*[*OF*  $\langle \cos x' \neq 0 \rangle \langle \cos y' \neq 0 \rangle$ ] *divide-nonzero-divide*[*OF*  $\langle \cos x' * \cos y' \neq 0 \rangle$ ] ..  
**also have**  $\dots = (\tan x' + \tan y') / (1 - \tan x' * \tan y')$  **unfolding** *tan-def diff-divide-distrib divide-self*[*OF*  $\langle \cos x' * \cos y' \neq 0 \rangle$ ] **unfolding** *divide-mult-commute* ..  
**finally have** *tan-eq*:  $\tan(x' + y') = (x + y) / (1 - x * y)$  **unfolding**  $\langle \tan y' = y \rangle \langle \tan x' = x \rangle$  .

```

have arctan (tan (x' + y')) = x' + y' using ⟨-(pi/4) < y'⟩ ⟨-(pi/4) ≤ x'⟩ ⟨y'
< pi/4⟩ and ⟨x' ≤ pi/4⟩ by (auto intro!: arctan-tan)
moreover have arctan (tan (x')) = x' using ⟨-(pi/2) < x'⟩ and ⟨x' < pi/2⟩
by (auto intro!: arctan-tan)
moreover have arctan (tan (y')) = y' using ⟨-(pi/4) < y'⟩ and ⟨y' < pi/4⟩
by (auto intro!: arctan-tan)
ultimately have arctan x + arctan y = arctan (tan (x' + y')) unfolding ⟨tan
y' = y⟩ [symmetric] ⟨tan x' = x⟩ [symmetric] by auto
thus arctan x + arctan y = arctan ((x + y) / (1 - x * y)) unfolding tan-eq .
qed

```

```

lemma arctan1-eq-pi4: arctan 1 = pi / 4 unfolding tan-45[symmetric] by (rule
arctan-tan, auto simp add: m2pi-less-pi)

```

```

theorem machin: pi / 4 = 4 * arctan (1/5) - arctan (1 / 239)

```

```

proof -

```

```

  have |1 / 5| < (1 :: real) by auto
  from arctan-add[OF less-imp-le[OF this] this]
  have 2 * arctan (1 / 5) = arctan (5 / 12) by auto
  moreover
  have |5 / 12| < (1 :: real) by auto
  from arctan-add[OF less-imp-le[OF this] this]
  have 2 * arctan (5 / 12) = arctan (120 / 119) by auto
  moreover
  have |1| ≤ (1::real) and |1 / 239| < (1::real) by auto
  from arctan-add[OF this]
  have arctan 1 + arctan (1 / 239) = arctan (120 / 119) by auto
  ultimately have arctan 1 + arctan (1 / 239) = 4 * arctan (1 / 5) by auto
  thus ?thesis unfolding arctan1-eq-pi4 by algebra

```

```

qed

```

## 76.15 Introducing the arcus tangens power series

```

lemma monoseq-arctan-series: fixes x :: real

```

```

  assumes |x| ≤ 1 shows monoseq (λ n. 1 / real (n*2+1) * x^(n*2+1)) (is
monoseq ?a)

```

```

proof (cases x = 0) case True thus ?thesis unfolding monoseq-def One-nat-def
by auto

```

```

next

```

```

  case False

```

```

  have norm x ≤ 1 and x ≤ 1 and -1 ≤ x using assms by auto

```

```

  show monoseq ?a

```

```

  proof -

```

```

    { fix n fix x :: real assume 0 ≤ x and x ≤ 1

```

```

      have 1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≤ 1 / real (Suc (n *
2)) * x ^ Suc (n * 2)

```

```

      proof (rule mult-mono)

```

```

        show 1 / real (Suc (Suc n * 2)) ≤ 1 / real (Suc (n * 2)) by (rule frac-le)
simp-all

```

```

    show  $0 \leq 1 / \text{real } (\text{Suc } (n * 2))$  by auto
    show  $x \wedge \text{Suc } (\text{Suc } n * 2) \leq x \wedge \text{Suc } (n * 2)$  by (rule power-decreasing)
  (simp-all add:  $\langle 0 \leq x \rangle \langle x \leq 1 \rangle$ )
    show  $0 \leq x \wedge \text{Suc } (\text{Suc } n * 2)$  by (rule zero-le-power) (simp add:  $\langle 0 \leq x \rangle$ )
  qed
} note mono = this

```

```

show ?thesis
proof (cases  $0 \leq x$ )
  case True from mono[OF this  $\langle x \leq 1 \rangle$ , THEN allI]
    show ?thesis unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI2)
  next
    case False hence  $0 \leq -x$  and  $-x \leq 1$  using  $\langle -1 \leq x \rangle$  by auto
    from mono[OF this]
      have  $\bigwedge n. 1 / \text{real } (\text{Suc } (\text{Suc } n * 2)) * x \wedge \text{Suc } (\text{Suc } n * 2) \geq 1 / \text{real } (\text{Suc } (n * 2)) * x \wedge \text{Suc } (n * 2)$  using  $\langle 0 \leq -x \rangle$  by auto
      thus ?thesis unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI1[OF allI])
    qed
  qed
qed

```

```

lemma zeroseq-arctan-series: fixes  $x :: \text{real}$ 
  assumes  $|x| \leq 1$  shows  $(\lambda n. 1 / \text{real } (n*2+1) * x^{(n*2+1)}) \text{ ----> } 0$  (is
  ?a ----> 0)
proof (cases  $x = 0$ ) case True thus ?thesis unfolding One-nat-def by (auto
simp add: LIMSEQ-const)
next
  case False
    have  $\text{norm } x \leq 1$  and  $x \leq 1$  and  $-1 \leq x$  using assms by auto
    show ?a ----> 0
    proof (cases  $|x| < 1$ )
      case True hence  $\text{norm } x < 1$  by auto
      from LIMSEQ-mult[OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero[OF
 $\langle \text{norm } x < 1 \rangle$ , THEN LIMSEQ-Suc]]
        have  $(\lambda n. 1 / \text{real } (n + 1) * x^{(n + 1)}) \text{ ----> } 0$ 
        unfolding inverse-eq-divide Suc-eq-plus1 by simp
        then show ?thesis using pos2 by (rule LIMSEQ-linear)
      next
        case False hence  $x = -1 \vee x = 1$  using  $\langle |x| \leq 1 \rangle$  by auto
        hence  $n\text{-eq}: \bigwedge n. x^{(n * 2 + 1)} = x$  unfolding One-nat-def by auto
        from LIMSEQ-mult[OF LIMSEQ-inverse-real-of-nat[THEN LIMSEQ-linear,
OF pos2, unfolded inverse-eq-divide] LIMSEQ-const[of x]]
          show ?thesis unfolding n-eq Suc-eq-plus1 by auto
        qed
    qed
  qed

```

```

lemma summable-arctan-series: fixes  $x :: \text{real}$  and  $n :: \text{nat}$ 
  assumes  $|x| \leq 1$  shows summable  $(\lambda k. (-1)^k * (1 / \text{real } (k*2+1)) * x^{(k*2+1)})$ 

```

$(k*2+1)))$  (is summable ( $?c\ x$ ))  
 by (rule summable-Leibniz(1), rule zeroseq-arctan-series[OF assms], rule monoseq-arctan-series[OF  
 assms])

**lemma less-one-imp-sqr-less-one:** fixes  $x :: \text{real}$  assumes  $|x| < 1$  shows  $x^2 < 1$

**proof** –

from mult-mono1[OF less-imp-le[OF  $|x| < 1$ ] abs-ge-zero[of  $x$ ]]  
 have  $|x^2| < 1$  using  $|x| < 1$  unfolding numeral-2-eq-2 power-Suc2 by  
 auto  
 thus ?thesis using zero-le-power2 by auto  
**qed**

**lemma DERIV-arctan-series:** assumes  $|x| < 1$

shows  $\text{DERIV } (\lambda x'. \sum k. (-1)^k * (1 / \text{real } (k*2+1)) * x'^k * (k*2+1)))\ x :>$   
 $(\sum k. (-1)^k * x^{(k*2)})$  (is  $\text{DERIV } ?\text{arctan} - :> ?\text{Int}$ )

**proof** –

let  $?f\ n = \text{if even } n \text{ then } (-1)^{(n \text{ div } 2)} * 1 / \text{real } (\text{Suc } n) \text{ else } 0$

{ fix  $n :: \text{nat}$  assume even  $n$  hence  $2 * (n \text{ div } 2) = n$  by presburger } note  
 $n\text{-even}=\text{this}$

have if-eq:  $\bigwedge n\ x'. ?f\ n * \text{real } (\text{Suc } n) * x'^n = (\text{if even } n \text{ then } (-1)^{(n \text{ div } 2)} * x'^{(2 * (n \text{ div } 2))} \text{ else } 0)$  using  $n\text{-even}$  by auto

{ fix  $x :: \text{real}$  assume  $|x| < 1$  hence  $x^2 < 1$  by (rule less-one-imp-sqr-less-one)  
 have summable  $(\lambda n. -1^n * x^{(2*n)})$   
 by (rule summable-Leibniz(1), auto intro!: LIMSEQ-realpow-zero monoseq-realpow  
 $(x^2 < 1)$  order-less-imp-le[OF  $(x^2 < 1)$ ])  
 hence summable  $(\lambda n. -1^n * x^{(2*n)})$  unfolding power-mult .  
 } note summable-Integral = this

{ fix  $f :: \text{nat} \Rightarrow \text{real}$

have  $\bigwedge x. f \text{ sums } x = (\lambda n. \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0) \text{ sums } x$

**proof**

fix  $x :: \text{real}$  assume  $f \text{ sums } x$   
 from sums-if[OF sums-zero this]  
 show  $(\lambda n. \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0) \text{ sums } x$  by auto  
**next**

fix  $x :: \text{real}$  assume  $(\lambda n. \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0) \text{ sums } x$

from LIMSEQ-linear[OF this[unfolded sums-def] pos2, unfolded sum-split-even-odd[unfolded  
 mult-commute]]

show  $f \text{ sums } x$  unfolding sums-def by auto

**qed**

hence  $\text{op sums } f = \text{op sums } (\lambda n. \text{if even } n \text{ then } f\ (n \text{ div } 2) \text{ else } 0) ..$

} note sums-even = this

have Int-eq:  $(\sum n. ?f\ n * \text{real } (\text{Suc } n) * x^n) = ?\text{Int}$  unfolding if-eq mult-commute[of  
 $-2$ ] suminf-def sums-even[of  $\lambda n. -1^n * x^{(2 * n)}$ , symmetric]  
 by auto

```

{ fix x :: real
  have if-eq':  $\bigwedge n. (if\ even\ n\ then\ -1^{(n\ div\ 2)} * 1 / real\ (Suc\ n)\ else\ 0) * x^{Suc\ n} =$ 
    (if even n then  $-1^{(n\ div\ 2)} * (1 / real\ (Suc\ (2 * (n\ div\ 2)))) * x^{Suc\ (2 * (n\ div\ 2))}$  else 0)
  using n-even by auto
  have idx-eq:  $\bigwedge n. n * 2 + 1 = Suc\ (2 * n)$  by auto
  have ( $\sum n. ?f\ n * x^{(Suc\ n)}$ ) = ?arctan x unfolding if-eq' idx-eq suminf-def
    sums-even[of  $\lambda n. -1^{(n\ div\ 2)} * (1 / real\ (Suc\ (2 * n))) * x^{Suc\ (2 * n)}$ ], symmetric]
    by auto
} note arctan-eq = this

have DERIV ( $\lambda x. \sum n. ?f\ n * x^{(Suc\ n)}$ ) x := ( $\sum n. ?f\ n * real\ (Suc\ n) * x^{n+1}$ )
proof (rule DERIV-power-series')
  show  $x \in \{-1 <..< 1\}$  using  $\langle |x| < 1 \rangle$  by auto
  { fix x' :: real assume x'-bounds:  $x' \in \{-1 <..< 1\}$ 
    hence  $|x'| < 1$  by auto

    let ?S =  $\sum n. (-1)^n * x'^{(2 * n)}$ 
    show summable ( $\lambda n. ?f\ n * real\ (Suc\ n) * x'^{n+1}$ ) unfolding if-eq
      by (rule sums-summable[where l=0 + ?S], rule sums-if, rule sums-zero,
        rule summable-sums, rule summable-Integral[OF  $\langle |x'| < 1 \rangle$ ])
    }
  qed auto
  thus ?thesis unfolding Int-eq arctan-eq .
qed

lemma arctan-series: assumes  $|x| \leq 1$ 
  shows  $arctan\ x = (\sum k. (-1)^k * (1 / real\ (k*2+1)) * x^{(k*2+1)})$  (is - =
    suminf ( $\lambda n. ?c\ x\ n$ ))
proof -
  let ?c' x n =  $(-1)^n * x^{(n*2)}$ 

  { fix r x :: real assume  $0 < r$  and  $r < 1$  and  $|x| < r$ 
    have  $|x| < 1$  using  $\langle r < 1 \rangle$  and  $\langle |x| < r \rangle$  by auto
    from DERIV-arctan-series[OF this]
    have DERIV ( $\lambda x. suminf\ (?c\ x)$ ) x := (suminf (?c' x)) .
  } note DERIV-arctan-suminf = this

  { fix x :: real assume  $|x| \leq 1$  note summable-Leibniz[OF zeroseq-arctan-series[OF
    this] monoseq-arctan-series[OF this]] }
  note arctan-series-borders = this

  { fix x :: real assume  $|x| < 1$  have  $arctan\ x = (\sum k. ?c\ x\ k)$ 
    proof -
      obtain r where  $|x| < r$  and  $r < 1$  using dense[OF  $\langle |x| < 1 \rangle$ ] by blast
      hence  $0 < r$  and  $-r < x$  and  $x < r$  by auto
    }
  }

```



**have** *suminf-eq-arctan-bounded*:  $\bigwedge x\ a\ b. \llbracket -r < a ; b < r ; a < b ; a \leq x ; x \leq b \rrbracket \implies \text{suminf } (?c\ x) - \arctan\ x = \text{suminf } (?c\ a) - \arctan\ a$   
**proof** –  
**fix**  $x\ a\ b$  **assume**  $-r < a$  **and**  $b < r$  **and**  $a < b$  **and**  $a \leq x$  **and**  $x \leq b$   
**hence**  $|x| < r$  **by** *auto*  
**show**  $\text{suminf } (?c\ x) - \arctan\ x = \text{suminf } (?c\ a) - \arctan\ a$   
**proof** (*rule DERIV-isconst2*[*of a b*])  
**show**  $a < b$  **and**  $a \leq x$  **and**  $x \leq b$  **using**  $\langle a < b \rangle \langle a \leq x \rangle \langle x \leq b \rangle$  **by** *auto*  
**have**  $\forall x. -r < x \wedge x < r \longrightarrow \text{DERIV } (\lambda x. \text{suminf } (?c\ x) - \arctan\ x) x$   
 $:=> 0$   
**proof** (*rule allI, rule impI*)  
**fix**  $x$  **assume**  $-r < x \wedge x < r$  **hence**  $|x| < r$  **by** *auto*  
**hence**  $|x| < 1$  **using**  $\langle r < 1 \rangle$  **by** *auto*  
**have**  $|-(x^2)| < 1$  **using** *less-one-imp-sqr-less-one*[*OF*  $\langle |x| < 1 \rangle$ ] **by** *auto*  
**hence**  $(\lambda n. (- (x^2)) ^ n \text{ sums } (1 / (1 - (- (x^2))))$  **unfolding** *real-norm-def*[*symmetric*] **by** (*rule geometric-sums*)  
**hence**  $(?c'\ x) \text{ sums } (1 / (1 - (- (x^2))))$  **unfolding** *power-mult-distrib*[*symmetric*] *power-mult nat-mult-commute*[*of - 2*] **by** *auto*  
**hence** *suminf-c'-eq-geom*:  $\text{inverse } (1 + x^2) = \text{suminf } (?c'\ x)$  **using** *sums-unique* **unfolding** *inverse-eq-divide* **by** *auto*  
**have**  $\text{DERIV } (\lambda x. \text{suminf } (?c\ x))\ x :=> (\text{inverse } (1 + x^2))$  **unfolding** *suminf-c'-eq-geom*  
**by** (*rule DERIV-arctan-suminf*[*OF*  $\langle 0 < r \rangle \langle r < 1 \rangle \langle |x| < r \rangle$ ])  
**from** *DERIV-add-minus*[*OF this DERIV-arctan*]  
**show**  $\text{DERIV } (\lambda x. \text{suminf } (?c\ x) - \arctan\ x)\ x :=> 0$  **unfolding** *diff-minus*  
**by** *auto*  
**qed**  
**hence** *DERIV-in-rball*:  $\forall y. a \leq y \wedge y \leq b \longrightarrow \text{DERIV } (\lambda x. \text{suminf } (?c\ x) - \arctan\ x)\ y :=> 0$  **using**  $\langle -r < a \rangle \langle b < r \rangle$  **by** *auto*  
**thus**  $\forall y. a < y \wedge y < b \longrightarrow \text{DERIV } (\lambda x. \text{suminf } (?c\ x) - \arctan\ x)\ y :=> 0$  **using**  $\langle |x| < r \rangle$  **by** *auto*  
**show**  $\forall y. a \leq y \wedge y \leq b \longrightarrow \text{isCont } (\lambda x. \text{suminf } (?c\ x) - \arctan\ x)\ y$   
**using** *DERIV-in-rball* *DERIV-isCont* **by** *auto*  
**qed**  
**qed**  
**have** *suminf-arctan-zero*:  $\text{suminf } (?c\ 0) - \arctan\ 0 = 0$   
**unfolding** *Suc-eq-plus1*[*symmetric*] *power-Suc2* *mult-zero-right* *arctan-zero-zero* *suminf-zero* **by** *auto*  
**have**  $\text{suminf } (?c\ x) - \arctan\ x = 0$   
**proof** (*cases*  $x = 0$ )  
**case** *True* **thus** *?thesis* **using** *suminf-arctan-zero* **by** *auto*  
**next**  
**case** *False* **hence**  $0 < |x|$  **and**  $-|x| < |x|$  **by** *auto*  
**have**  $\text{suminf } (?c\ (-|x|)) - \arctan\ (-|x|) = \text{suminf } (?c\ 0) - \arctan\ 0$   
**by** (*rule suminf-eq-arctan-bounded*[*where*  $x=0$  **and**  $a=-|x|$  **and**  $b=|x|$ ],

```

symmetric])
  (simp-all only: ⟨|x| < r⟩ ⟨-|x| < |x|⟩ neg-less-iff-less)
  moreover
  have suminf (?c x) - arctan x = suminf (?c (-|x|)) - arctan (-|x|)
    by (rule suminf-eq-arctan-bounded[where x=x and a=-|x| and b=|x|])
    (simp-all only: ⟨|x| < r⟩ ⟨-|x| < |x|⟩ neg-less-iff-less)
  ultimately
  show ?thesis using suminf-arctan-zero by auto
qed
thus ?thesis by auto
qed } note when-less-one = this

show arctan x = suminf (λ n. ?c x n)
proof (cases |x| < 1)
  case True thus ?thesis by (rule when-less-one)
next case False hence |x| = 1 using ⟨|x| ≤ 1⟩ by auto
  let ?a x n = |1 / real (n*2+1) * x^(n*2+1)|
  let ?diff x n = |arctan x - (∑ i = 0..<n. ?c x i)|
  { fix n :: nat
    have 0 < (1 :: real) by auto
    moreover
    { fix x :: real assume 0 < x and x < 1 hence |x| ≤ 1 and |x| < 1 by auto
      from ⟨0 < x⟩ have 0 < 1 / real (0 * 2 + (1::nat)) * x ^ (0 * 2 + 1) by
auto
      note bounds = mp[OF arctan-series-borders(2)[OF ⟨|x| ≤ 1⟩] this, unfolded
when-less-one[OF ⟨|x| < 1⟩, symmetric], THEN spec]
      have 0 < 1 / real (n*2+1) * x^(n*2+1) by (rule mult-pos-pos, auto simp
only: zero-less-power[OF ⟨0 < x⟩], auto)
      hence a-pos: ?a x n = 1 / real (n*2+1) * x^(n*2+1) by (rule abs-of-pos)
      have ?diff x n ≤ ?a x n
      proof (cases even n)
        case True hence sgn-pos: (-1)^n = (1::real) by auto
        from ⟨even n⟩ obtain m where 2 * m = n unfolding even-mult-two-ex
by auto
        from bounds[of m, unfolded this atLeastAtMost-iff]
        have |arctan x - (∑ i = 0..<n. (?c x i))| ≤ (∑ i = 0..<n + 1. (?c x i))
- (∑ i = 0..<n. (?c x i)) by auto
        also have ... = ?c x n unfolding One-nat-def by auto
        also have ... = ?a x n unfolding sgn-pos a-pos by auto
        finally show ?thesis .
      next
        case False hence sgn-neg: (-1)^n = (-1::real) by auto
        from ⟨odd n⟩ obtain m where m-def: 2 * m + 1 = n unfolding
odd-Suc-mult-two-ex by auto
        hence m-plus: 2 * (m + 1) = n + 1 by auto
        from bounds[of m + 1, unfolded this atLeastAtMost-iff, THEN conjunct1]
bounds[of m, unfolded m-def atLeastAtMost-iff, THEN conjunct2]
        have |arctan x - (∑ i = 0..<n. (?c x i))| ≤ (∑ i = 0..<n. (?c x i)) -
(∑ i = 0..<n+1. (?c x i)) by auto

```

```

    also have ... = - ?c x n unfolding One-nat-def by auto
    also have ... = ?a x n unfolding sgn-neg a-pos by auto
    finally show ?thesis .
qed
hence 0 ≤ ?a x n - ?diff x n by auto
}
hence ∀ x ∈ { 0 <..< 1 }, 0 ≤ ?a x n - ?diff x n by auto
moreover have ∧x. isCont (λ x. ?a x n - ?diff x n) x
  unfolding diff-def divide-inverse
by (auto intro!: isCont-add isCont-rabs isCont-ident isCont-minus isCont-arctan
isCont-inverse isCont-mult isCont-power isCont-const isCont-setsum)
ultimately have 0 ≤ ?a 1 n - ?diff 1 n by (rule LIM-less-bound)
hence ?diff 1 n ≤ ?a 1 n by auto
}
have ?a 1 -----> 0
  unfolding LIMSEQ-rabs-zero power-one divide-inverse One-nat-def
  by (auto intro!: LIMSEQ-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat)
have ?diff 1 -----> 0
proof (rule LIMSEQ-I)
  fix r :: real assume 0 < r
  obtain N :: nat where N-I: ∧ n. N ≤ n ⇒ ?a 1 n < r using LIMSEQ-D[OF
(?a 1 -----> 0) (0 < r)] by auto
  { fix n assume N ≤ n from ( ?diff 1 n ≤ ?a 1 n) N-I[OF this]
    have norm (?diff 1 n - 0) < r by auto }
  thus ∃ N. ∀ n ≥ N. norm (?diff 1 n - 0) < r by blast
qed
from this[unfolded LIMSEQ-rabs-zero diff-def add-commute[of arctan 1], THEN
LIMSEQ-add-const, of - arctan 1, THEN LIMSEQ-minus]
have (?c 1) sums (arctan 1) unfolding sums-def by auto
hence arctan 1 = (∑ i. ?c 1 i) by (rule sums-unique)

show ?thesis
proof (cases x = 1, simp add: (arctan 1 = (∑ i. ?c 1 i)))
  assume x ≠ 1 hence x = -1 using (|x| = 1) by auto

  have - (pi / 2) < 0 using pi-gt-zero by auto
  have - (2 * pi) < 0 using pi-gt-zero by auto

  have c-minus-minus: ∧ i. ?c (- 1) i = - ?c 1 i unfolding One-nat-def by
auto

  have arctan (- 1) = arctan (tan (-(pi / 4))) unfolding tan-45 tan-minus
..
  also have ... = - (pi / 4) by (rule arctan-tan, auto simp add: order-less-trans[OF
( - (pi / 2) < 0) pi-gt-zero])
  also have ... = - (arctan (tan (pi / 4))) unfolding neg-equal-iff-equal by
(rule arctan-tan[symmetric], auto simp add: order-less-trans[OF (2 * pi) < 0)
pi-gt-zero])
  also have ... = - (arctan 1) unfolding tan-45 ..

```

also have  $\dots = -(\sum i. ?c\ 1\ i)$  using  $\langle \arctan\ 1 = (\sum i. ?c\ 1\ i) \rangle$  by auto  
 also have  $\dots = (\sum i. ?c\ (-1)\ i)$  using  $\text{suminf-minus}[OF\ \text{sums-summable}[OF\ \langle ?c\ 1 \rangle\ \text{sums}\ (\arctan\ 1)]]$  unfolding  $c\text{-minus-minus}$  by auto  
 finally show  $?thesis$  using  $\langle x = -1 \rangle$  by auto  
 qed  
 qed  
 qed

lemma *arctan-half*: fixes  $x :: \text{real}$   
 shows  $\arctan\ x = 2 * \arctan\ (x / (1 + \sqrt{1 + x^2}))$   
 proof –  
 obtain  $y$  where  $\text{low}: -(pi / 2) < y$  and  $\text{high}: y < pi / 2$  and  $y\text{-eq}: \tan y = x$  using  $\text{tan-total}$  by blast  
 hence  $\text{low2}: -(pi / 2) < y / 2$  and  $\text{high2}: y / 2 < pi / 2$  by auto  
  
 have  $\text{divide-nonzero-divide}: \bigwedge A\ B\ C :: \text{real}. C \neq 0 \implies A / B = (A / C) / (B / C)$  by auto  
  
 have  $0 < \cos y$  using  $\text{cos-gt-zero-pi}[OF\ \text{low}\ \text{high}]$ .  
 hence  $\cos y \neq 0$  and  $\text{cos-sqrt}: \sqrt{(\cos y)^2} = \cos y$  by auto  
  
 have  $1 + (\tan y)^2 = 1 + \sin y^2 / \cos y^2$  unfolding  $\text{tan-def}\ \text{power-divide}$ ..  
 also have  $\dots = \cos y^2 / \cos y^2 + \sin y^2 / \cos y^2$  using  $\langle \cos y \neq 0 \rangle$  by auto  
 also have  $\dots = 1 / \cos y^2$  unfolding  $\text{add-divide-distrib}[\text{symmetric}]\ \text{sin-cos-squared-add2}$ ..  
 finally have  $1 + (\tan y)^2 = 1 / \cos y^2$ .

have  $\sin y / (\cos y + 1) = \tan y / ((\cos y + 1) / \cos y)$  unfolding  $\text{tan-def}\ \text{divide-nonzero-divide}[OF\ \langle \cos y \neq 0 \rangle, \text{symmetric}]$ ..  
 also have  $\dots = \tan y / (1 + 1 / \cos y)$  using  $\langle \cos y \neq 0 \rangle$  unfolding  $\text{add-divide-distrib}$  by auto  
 also have  $\dots = \tan y / (1 + 1 / \sqrt{\cos y^2})$  unfolding  $\text{cos-sqrt}$ ..  
 also have  $\dots = \tan y / (1 + \sqrt{1 / \cos y^2})$  unfolding  $\text{real-sqrt-divide}$  by auto  
 finally have  $\text{eq}: \sin y / (\cos y + 1) = \tan y / (1 + \sqrt{1 + (\tan y)^2})$  unfolding  $\langle 1 + (\tan y)^2 = 1 / \cos y^2 \rangle$ .

have  $\arctan\ x = y$  using  $\text{arctan-tan}\ \text{low}\ \text{high}\ y\text{-eq}$  by auto  
 also have  $\dots = 2 * (\arctan\ (\tan\ (y/2)))$  using  $\text{arctan-tan}[OF\ \text{low2}\ \text{high2}]$  by auto  
 also have  $\dots = 2 * (\arctan\ (\sin y / (\cos y + 1)))$  unfolding  $\text{tan-half}[OF\ \text{low2}\ \text{high2}]$  by auto  
 finally show  $?thesis$  unfolding  $\text{eq}\ \langle \tan y = x \rangle$ .  
 qed

lemma *arctan-monotone*: assumes  $x < y$   
 shows  $\arctan\ x < \arctan\ y$   
 proof –

**obtain**  $z$  **where**  $-(\pi / 2) < z$  **and**  $z < \pi / 2$  **and**  $\tan z = x$  **using** *tan-total*  
**by** *blast*  
**obtain**  $w$  **where**  $-(\pi / 2) < w$  **and**  $w < \pi / 2$  **and**  $\tan w = y$  **using** *tan-total*  
**by** *blast*  
**have**  $z < w$  **unfolding** *tan-monotone'*[*OF*  $\langle -(\pi / 2) < z \rangle \langle z < \pi / 2 \rangle \langle -(\pi / 2) < w \rangle \langle w < \pi / 2 \rangle$ ]  $\langle \tan z = x \rangle \langle \tan w = y \rangle$  **using**  $\langle x < y \rangle$  .  
**thus** *?thesis*  
**unfolding**  $\langle \tan z = x \rangle$ [*symmetric*] *arctan-tan*[*OF*  $\langle -(\pi / 2) < z \rangle \langle z < \pi / 2 \rangle$ ]  
**unfolding**  $\langle \tan w = y \rangle$ [*symmetric*] *arctan-tan*[*OF*  $\langle -(\pi / 2) < w \rangle \langle w < \pi / 2 \rangle$ ]  
 $\rangle$ ] .  
**qed**

**lemma** *arctan-monotone'*: **assumes**  $x \leq y$  **shows**  $\arctan x \leq \arctan y$   
**proof** (*cases*  $x = y$ )  
**case** *False* **hence**  $x < y$  **using**  $\langle x \leq y \rangle$  **by** *auto* **from** *arctan-monotone*[*OF this*]  
**show** *?thesis* **by** *auto*  
**qed** *auto*

**lemma** *arctan-minus*:  $\arctan (-x) = -\arctan x$   
**proof** –  
**obtain**  $y$  **where**  $-(\pi / 2) < y$  **and**  $y < \pi / 2$  **and**  $\tan y = x$  **using** *tan-total*  
**by** *blast*  
**thus** *?thesis* **unfolding**  $\langle \tan y = x \rangle$ [*symmetric*] *tan-minus*[*symmetric*] **using**  
*arctan-tan*[*of y*] *arctan-tan*[*of -y*] **by** *auto*  
**qed**

**lemma** *arctan-inverse*: **assumes**  $x \neq 0$  **shows**  $\arctan (1 / x) = \text{sgn } x * \pi / 2 - \arctan x$   
**proof** –  
**obtain**  $y$  **where**  $-(\pi / 2) < y$  **and**  $y < \pi / 2$  **and**  $\tan y = x$  **using** *tan-total*  
**by** *blast*  
**hence**  $y = \arctan x$  **unfolding**  $\langle \tan y = x \rangle$ [*symmetric*] **using** *arctan-tan* **by**  
*auto*

**{** **fix**  $y \ x :: \text{real}$  **assume**  $0 < y$  **and**  $y < \pi / 2$  **and**  $y = \arctan x$  **and**  $\tan y =$   
 $x$  **hence**  $-(\pi / 2) < y$  **by** *auto*  
**have**  $\tan y > 0$  **using** *tan-monotone'*[*OF*  $\langle -(\pi / 2) < y \rangle \langle y < \pi / 2 \rangle$ , *of*  
 $0$ ] *tan-zero*  $\langle 0 < y \rangle$  **by** *auto*  
**hence**  $x > 0$  **using**  $\langle \tan y = x \rangle$  **by** *auto*  
  
**have**  $-(\pi / 2) < \pi / 2 - y$  **using**  $\langle y > 0 \rangle \langle y < \pi / 2 \rangle$  **by** *auto*  
**moreover** **have**  $\pi / 2 - y < \pi / 2$  **using**  $\langle y > 0 \rangle \langle y < \pi / 2 \rangle$  **by** *auto*  
**ultimately** **have**  $\arctan (1 / x) = \pi / 2 - y$  **unfolding**  $\langle \tan y = x \rangle$ [*symmetric*]  
*tan-inverse* **using** *arctan-tan* **by** *auto*  
**hence**  $\arctan (1 / x) = \text{sgn } x * \pi / 2 - \arctan x$  **unfolding**  $\langle y = \arctan x \rangle$   
*real-sgn-pos*[*OF*  $\langle x > 0 \rangle$ ] **by** *auto*  
**}** **note**  $\text{pos-}y = \text{this}$

**show** *?thesis*

**proof** (cases  $y > 0$ )  
**case** *True* **from** *pos-y* [OF this  $\langle y < \pi / 2 \rangle \langle y = \arctan x \rangle \langle \tan y = x \rangle$ ] **show** *?thesis* .  
**next**  
**case** *False* **hence**  $y \leq 0$  **by** *auto*  
**moreover** **have**  $y \neq 0$   
**proof** (rule *ccontr*)  
**assume**  $\neg y \neq 0$  **hence**  $y = 0$  **by** *auto*  
**have**  $x = 0$  **unfolding**  $\langle \tan y = x \rangle$  [symmetric]  $\langle y = 0 \rangle$  *tan-zero* ..  
**thus** *False* **using**  $\langle x \neq 0 \rangle$  **by** *auto*  
**qed**  
**ultimately** **have**  $y < 0$  **by** *auto*  
**hence**  $0 < -y$  **and**  $-y < \pi / 2$  **using**  $\langle -(\pi / 2) < y \rangle$  **by** *auto*  
**moreover** **have**  $-y = \arctan (-x)$  **unfolding** *arctan-minus*  $\langle y = \arctan x \rangle$  ..  
**moreover** **have**  $\tan (-y) = -x$  **unfolding** *tan-minus*  $\langle \tan y = x \rangle$  ..  
**ultimately** **have**  $\arctan (1 / -x) = \text{sgn } (-x) * \pi / 2 - \arctan (-x)$  **using** *pos-y* **by** *blast*  
**hence**  $\arctan (-(1 / x)) = -(\text{sgn } x * \pi / 2 - \arctan x)$  **unfolding** *arctan-minus* [of  $x$ ] *divide-minus-right* *sgn-minus* **by** *auto*  
**thus** *?thesis* **unfolding** *arctan-minus* *neg-equal-iff-equal* .  
**qed**  
**qed**

**theorem** *pi-series*:  $\pi / 4 = (\sum k. (-1)^k * 1 / \text{real } (k*2+1))$  (is - = ?SUM)  
**proof** -  
**have**  $\pi / 4 = \arctan 1$  **using** *arctan1-eq-pi4* **by** *auto*  
**also** **have** ... = ?SUM **using** *arctan-series* [of 1] **by** *auto*  
**finally** **show** *?thesis* **by** *auto*  
**qed**

## 76.16 Existence of Polar Coordinates

**lemma** *cos-x-y-le-one*:  $|x / \text{sqrt } (x^2 + y^2)| \leq 1$   
**apply** (rule *power2-le-imp-le* [OF - zero-le-one])  
**apply** (*simp add: power-divide divide-le-eq not-sum-power2-lt-zero*)  
**done**

**lemma** *cos-arccos-abs*:  $|y| \leq 1 \implies \cos (\arccos y) = y$   
**by** (*simp add: abs-le-iff*)

**lemma** *sin-arccos-abs*:  $|y| \leq 1 \implies \sin (\arccos y) = \text{sqrt } (1 - y^2)$   
**by** (*simp add: sin-arccos abs-le-iff*)

**lemmas** *cos-arccos-lemma1* = *cos-arccos-abs* [OF *cos-x-y-le-one*]

**lemmas** *sin-arccos-lemma1* = *sin-arccos-abs* [OF *cos-x-y-le-one*]

**lemma** *polar-ex1*:

$0 < y \implies \exists r a. x = r * \cos a \ \& \ y = r * \sin a$

```

apply (rule-tac  $x = \text{sqrt } (x^2 + y^2)$  in exI)
apply (rule-tac  $x = \arccos (x / \text{sqrt } (x^2 + y^2))$  in exI)
apply (simp add: cos-arccos-lemma1)
apply (simp add: sin-arccos-lemma1)
apply (simp add: power-divide)
apply (simp add: real-sqrt-mult [symmetric])
apply (simp add: right-diff-distrib)
done

lemma polar-ex2:
   $y < 0 \implies \exists r a. x = r * \cos a \ \& \ y = r * \sin a$ 
apply (insert polar-ex1 [where x=x and y=-y], simp, clarify)
apply (metis cos-minus minus-minus minus-mult-right sin-minus)
done

lemma polar-Ex:  $\exists r a. x = r * \cos a \ \& \ y = r * \sin a$ 
apply (rule-tac x=0 and y=y in linorder-cases)
apply (erule polar-ex1)
apply (rule-tac x=x in exI, rule-tac x=0 in exI, simp)
apply (erule polar-ex2)
done

end

```

## 77 Complex: Complex Numbers: Rectangular and Polar Representations

```

theory Complex
imports Transcendental
begin

datatype complex = Complex real real

primrec
   $Re :: \text{complex} \Rightarrow \text{real}$ 
where
   $Re: Re \ (Complex \ x \ y) = x$ 

primrec
   $Im :: \text{complex} \Rightarrow \text{real}$ 
where
   $Im: Im \ (Complex \ x \ y) = y$ 

lemma complex-surj [simp]:  $Complex \ (Re \ z) \ (Im \ z) = z$ 
  by (induct z) simp

lemma complex-equality [intro?]:  $\llbracket Re \ x = Re \ y; Im \ x = Im \ y \rrbracket \implies x = y$ 

```

**by** (*induct x, induct y*) *simp*

**lemma** *expand-complex-eq*:  $x = y \longleftrightarrow \text{Re } x = \text{Re } y \wedge \text{Im } x = \text{Im } y$   
**by** (*induct x, induct y*) *simp*

**lemmas** *complex-Re-Im-cancel-iff* = *expand-complex-eq*

## 77.1 Addition and Subtraction

**instantiation** *complex* :: *ab-group-add*  
**begin**

**definition**  
*complex-zero-def*:  $0 = \text{Complex } 0 \ 0$

**definition**  
*complex-add-def*:  $x + y = \text{Complex } (\text{Re } x + \text{Re } y) (\text{Im } x + \text{Im } y)$

**definition**  
*complex-minus-def*:  $- x = \text{Complex } (- \text{Re } x) (- \text{Im } x)$

**definition**  
*complex-diff-def*:  $x - (y :: \text{complex}) = x + - y$

**lemma** *Complex-eq-0* [*simp*]:  $\text{Complex } a \ b = 0 \longleftrightarrow a = 0 \wedge b = 0$   
**by** (*simp add: complex-zero-def*)

**lemma** *complex-Re-zero* [*simp*]:  $\text{Re } 0 = 0$   
**by** (*simp add: complex-zero-def*)

**lemma** *complex-Im-zero* [*simp*]:  $\text{Im } 0 = 0$   
**by** (*simp add: complex-zero-def*)

**lemma** *complex-add* [*simp*]:  
 $\text{Complex } a \ b + \text{Complex } c \ d = \text{Complex } (a + c) (b + d)$   
**by** (*simp add: complex-add-def*)

**lemma** *complex-Re-add* [*simp*]:  $\text{Re } (x + y) = \text{Re } x + \text{Re } y$   
**by** (*simp add: complex-add-def*)

**lemma** *complex-Im-add* [*simp*]:  $\text{Im } (x + y) = \text{Im } x + \text{Im } y$   
**by** (*simp add: complex-add-def*)

**lemma** *complex-minus* [*simp*]:  
 $-(\text{Complex } a \ b) = \text{Complex } (- a) (- b)$   
**by** (*simp add: complex-minus-def*)

**lemma** *complex-Re-minus* [*simp*]:  $\text{Re } (- x) = - \text{Re } x$   
**by** (*simp add: complex-minus-def*)



**lemma** *complex-Im-minus* [simp]:  $\text{Im } (-x) = -\text{Im } x$   
**by** (simp add: complex-minus-def)

**lemma** *complex-diff* [simp]:  
 $\text{Complex } a \ b - \text{Complex } c \ d = \text{Complex } (a - c) \ (b - d)$   
**by** (simp add: complex-diff-def)

**lemma** *complex-Re-diff* [simp]:  $\text{Re } (x - y) = \text{Re } x - \text{Re } y$   
**by** (simp add: complex-diff-def)

**lemma** *complex-Im-diff* [simp]:  $\text{Im } (x - y) = \text{Im } x - \text{Im } y$   
**by** (simp add: complex-diff-def)

**instance**  
**by** intro-classes (simp-all add: complex-add-def complex-diff-def)

**end**

## 77.2 Multiplication and Division

**instantiation** *complex* :: *field-inverse-zero*  
**begin**

**definition**  
*complex-one-def*:  $1 = \text{Complex } 1 \ 0$

**definition**  
*complex-mult-def*:  $x * y =$   
 $\text{Complex } (\text{Re } x * \text{Re } y - \text{Im } x * \text{Im } y) \ (\text{Re } x * \text{Im } y + \text{Im } x * \text{Re } y)$

**definition**  
*complex-inverse-def*:  $\text{inverse } x =$   
 $\text{Complex } (\text{Re } x / ((\text{Re } x)^2 + (\text{Im } x)^2)) \ (-\text{Im } x / ((\text{Re } x)^2 + (\text{Im } x)^2))$

**definition**  
*complex-divide-def*:  $x / (y::\text{complex}) = x * \text{inverse } y$

**lemma** *Complex-eq-1* [simp]:  $(\text{Complex } a \ b = 1) = (a = 1 \wedge b = 0)$   
**by** (simp add: complex-one-def)

**lemma** *complex-Re-one* [simp]:  $\text{Re } 1 = 1$   
**by** (simp add: complex-one-def)

**lemma** *complex-Im-one* [simp]:  $\text{Im } 1 = 0$   
**by** (simp add: complex-one-def)

**lemma** *complex-mult* [simp]:  
 $\text{Complex } a \ b * \text{Complex } c \ d = \text{Complex } (a * c - b * d) \ (a * d + b * c)$

```

  by (simp add: complex-mult-def)

lemma complex-Re-mult [simp]:  $\text{Re } (x * y) = \text{Re } x * \text{Re } y - \text{Im } x * \text{Im } y$ 
  by (simp add: complex-mult-def)

lemma complex-Im-mult [simp]:  $\text{Im } (x * y) = \text{Re } x * \text{Im } y + \text{Im } x * \text{Re } y$ 
  by (simp add: complex-mult-def)

lemma complex-inverse [simp]:
   $\text{inverse } (\text{Complex } a \ b) = \text{Complex } (a / (a^2 + b^2)) \ (- \ b / (a^2 + b^2))$ 
  by (simp add: complex-inverse-def)

lemma complex-Re-inverse:
   $\text{Re } (\text{inverse } x) = \text{Re } x / ((\text{Re } x)^2 + (\text{Im } x)^2)$ 
  by (simp add: complex-inverse-def)

lemma complex-Im-inverse:
   $\text{Im } (\text{inverse } x) = - \ \text{Im } x / ((\text{Re } x)^2 + (\text{Im } x)^2)$ 
  by (simp add: complex-inverse-def)

instance
  by intro-classes (simp-all add: complex-mult-def
    right-distrib left-distrib right-diff-distrib left-diff-distrib
    complex-inverse-def complex-divide-def
    power2-eq-square add-divide-distrib [symmetric]
    expand-complex-eq)

end

77.3 Numerals and Arithmetic

instantiation complex :: number-ring
begin

definition number-of-complex where
   $\text{complex-number-of-def: number-of } w = (\text{of-int } w :: \text{complex})$ 

instance
  by intro-classes (simp only: complex-number-of-def)

end

lemma complex-Re-of-nat [simp]:  $\text{Re } (\text{of-nat } n) = \text{of-nat } n$ 
  by (induct n) simp-all

lemma complex-Im-of-nat [simp]:  $\text{Im } (\text{of-nat } n) = 0$ 
  by (induct n) simp-all

lemma complex-Re-of-int [simp]:  $\text{Re } (\text{of-int } z) = \text{of-int } z$ 

```

**by** (*cases z rule: int-diff-cases*) *simp*

**lemma** *complex-Im-of-int* [*simp*]:  $Im\ (of-int\ z) = 0$   
**by** (*cases z rule: int-diff-cases*) *simp*

**lemma** *complex-Re-number-of* [*simp*]:  $Re\ (number-of\ v) = number-of\ v$   
**unfolding** *number-of-eq* **by** (*rule complex-Re-of-int*)

**lemma** *complex-Im-number-of* [*simp*]:  $Im\ (number-of\ v) = 0$   
**unfolding** *number-of-eq* **by** (*rule complex-Im-of-int*)

**lemma** *Complex-eq-number-of* [*simp*]:  
 $(Complex\ a\ b = number-of\ w) = (a = number-of\ w \wedge b = 0)$   
**by** (*simp add: expand-complex-eq*)

## 77.4 Scalar Multiplication

**instantiation** *complex* :: *real-field*  
**begin**

**definition**  
*complex-scaleR-def*:  $scaleR\ r\ x = Complex\ (r * Re\ x)\ (r * Im\ x)$

**lemma** *complex-scaleR* [*simp*]:  
 $scaleR\ r\ (Complex\ a\ b) = Complex\ (r * a)\ (r * b)$   
**unfolding** *complex-scaleR-def* **by** *simp*

**lemma** *complex-Re-scaleR* [*simp*]:  $Re\ (scaleR\ r\ x) = r * Re\ x$   
**unfolding** *complex-scaleR-def* **by** *simp*

**lemma** *complex-Im-scaleR* [*simp*]:  $Im\ (scaleR\ r\ x) = r * Im\ x$   
**unfolding** *complex-scaleR-def* **by** *simp*

**instance**

**proof**

**fix** *a b* :: *real* **and** *x y* :: *complex*  
**show**  $scaleR\ a\ (x + y) = scaleR\ a\ x + scaleR\ a\ y$   
**by** (*simp add: expand-complex-eq right-distrib*)  
**show**  $scaleR\ (a + b)\ x = scaleR\ a\ x + scaleR\ b\ x$   
**by** (*simp add: expand-complex-eq left-distrib*)  
**show**  $scaleR\ a\ (scaleR\ b\ x) = scaleR\ (a * b)\ x$   
**by** (*simp add: expand-complex-eq mult-assoc*)  
**show**  $scaleR\ 1\ x = x$   
**by** (*simp add: expand-complex-eq*)  
**show**  $scaleR\ a\ x * y = scaleR\ a\ (x * y)$   
**by** (*simp add: expand-complex-eq algebra-simps*)  
**show**  $x * scaleR\ a\ y = scaleR\ a\ (x * y)$   
**by** (*simp add: expand-complex-eq algebra-simps*)

**qed**

end

## 77.5 Properties of Embedding from Reals

### abbreviation

$\text{complex-of-real} :: \text{real} \Rightarrow \text{complex}$  **where**  
 $\text{complex-of-real} \equiv \text{of-real}$

**lemma** *complex-of-real-def*:  $\text{complex-of-real } r = \text{Complex } r \ 0$   
**by** (*simp add: of-real-def complex-scaleR-def*)

**lemma** *Re-complex-of-real* [*simp*]:  $\text{Re } (\text{complex-of-real } z) = z$   
**by** (*simp add: complex-of-real-def*)

**lemma** *Im-complex-of-real* [*simp*]:  $\text{Im } (\text{complex-of-real } z) = 0$   
**by** (*simp add: complex-of-real-def*)

**lemma** *Complex-add-complex-of-real* [*simp*]:  
 $\text{Complex } x \ y + \text{complex-of-real } r = \text{Complex } (x+r) \ y$   
**by** (*simp add: complex-of-real-def*)

**lemma** *complex-of-real-add-Complex* [*simp*]:  
 $\text{complex-of-real } r + \text{Complex } x \ y = \text{Complex } (r+x) \ y$   
**by** (*simp add: complex-of-real-def*)

**lemma** *Complex-mult-complex-of-real*:  
 $\text{Complex } x \ y * \text{complex-of-real } r = \text{Complex } (x*r) \ (y*r)$   
**by** (*simp add: complex-of-real-def*)

**lemma** *complex-of-real-mult-Complex*:  
 $\text{complex-of-real } r * \text{Complex } x \ y = \text{Complex } (r*x) \ (r*y)$   
**by** (*simp add: complex-of-real-def*)

## 77.6 Vector Norm

**instantiation** *complex* :: *real-normed-field*  
**begin**

**definition** *complex-norm-def*:  
 $\text{norm } z = \text{sqrt } ((\text{Re } z)^2 + (\text{Im } z)^2)$

### abbreviation

$\text{cmod} :: \text{complex} \Rightarrow \text{real}$  **where**  
 $\text{cmod} \equiv \text{norm}$

**definition** *complex-sgn-def*:  
 $\text{sgn } x = x \ /_R \ \text{cmod } x$

**definition** *dist-complex-def*:

$dist\ x\ y = cmod\ (x - y)$

**definition** *open-complex-def* [*code del*]:

$open\ (S :: complex\ set) \longleftrightarrow (\forall x \in S. \exists e > 0. \forall y. dist\ y\ x < e \longrightarrow y \in S)$

**lemmas** *cmod-def* = *complex-norm-def*

**lemma** *complex-norm* [*simp*]:  $cmod\ (Complex\ x\ y) = sqrt\ (x^2 + y^2)$

**by** (*simp add: complex-norm-def*)

**instance proof**

**fix**  $r :: real$  **and**  $x\ y :: complex$  **and**  $S :: complex\ set$

**show**  $0 \leq norm\ x$

**by** (*induct x simp*)

**show**  $(norm\ x = 0) = (x = 0)$

**by** (*induct x simp*)

**show**  $norm\ (x + y) \leq norm\ x + norm\ y$

**by** (*induct x, induct y*)

(*simp add: real-sqrt-sum-squares-triangle-ineq*)

**show**  $norm\ (scaleR\ r\ x) = |r| * norm\ x$

**by** (*induct x*)

(*simp add: power-mult-distrib right-distrib [symmetric] real-sqrt-mult*)

**show**  $norm\ (x * y) = norm\ x * norm\ y$

**by** (*induct x, induct y*)

(*simp add: real-sqrt-mult [symmetric] power2-eq-square algebra-simps*)

**show**  $sgn\ x = x /_R\ cmod\ x$

**by** (*rule complex-sgn-def*)

**show**  $dist\ x\ y = cmod\ (x - y)$

**by** (*rule dist-complex-def*)

**show**  $open\ S \longleftrightarrow (\forall x \in S. \exists e > 0. \forall y. dist\ y\ x < e \longrightarrow y \in S)$

**by** (*rule open-complex-def*)

**qed**

**end**

**lemma** *cmod-unit-one* [*simp*]:  $cmod\ (Complex\ (cos\ a)\ (sin\ a)) = 1$

**by** *simp*

**lemma** *cmod-complex-polar* [*simp*]:

$cmod\ (complex-of-real\ r * Complex\ (cos\ a)\ (sin\ a)) = abs\ r$

**by** (*simp add: norm-mult*)

**lemma** *complex-Re-le-cmod*:  $Re\ x \leq cmod\ x$

**unfolding** *complex-norm-def*

**by** (*rule real-sqrt-sum-squares-ge1*)

**lemma** *complex-mod-minus-le-complex-mod* [*simp*]:  $- cmod\ x \leq cmod\ x$

**by** (*rule order-trans [OF - norm-ge-zero], simp*)

**lemma** *complex-mod-triangle-ineq2* [*simp*]:  $cmod(b + a) - cmod\ b \leq cmod\ a$   
**by** (*rule ord-le-eq-trans* [*OF norm-triangle-ineq2*], *simp*)

**lemmas** *real-sum-squared-expand* = *power2-sum* [**where** 'a=real]

**lemma** *abs-Re-le-cmod*:  $|Re\ x| \leq cmod\ x$   
**by** (*cases x*) *simp*

**lemma** *abs-Im-le-cmod*:  $|Im\ x| \leq cmod\ x$   
**by** (*cases x*) *simp*

## 77.7 Completeness of the Complexes

**interpretation** *Re*: *bounded-linear Re*  
**apply** (*unfold-locales*, *simp*, *simp*)  
**apply** (*rule-tac x=1 in exI*)  
**apply** (*simp add: complex-norm-def*)  
**done**

**interpretation** *Im*: *bounded-linear Im*  
**apply** (*unfold-locales*, *simp*, *simp*)  
**apply** (*rule-tac x=1 in exI*)  
**apply** (*simp add: complex-norm-def*)  
**done**

**lemma** *tendsto-Complex* [*tendsto-intros*]:  
**assumes** ( $f \dashrightarrow a$ ) *net* **and** ( $g \dashrightarrow b$ ) *net*  
**shows** ( $(\lambda x. Complex\ (f\ x)\ (g\ x)) \dashrightarrow Complex\ a\ b$ ) *net*  
**proof** (*rule tendstoI*)  
**fix**  $r :: real$  **assume**  $0 < r$   
**hence**  $0 < r / \sqrt{2}$  **by** (*simp add: divide-pos-pos*)  
**have** *eventually* ( $\lambda x. dist\ (f\ x)\ a < r / \sqrt{2}$ ) *net*  
**using** ( $\langle f \dashrightarrow a \rangle$ ) *net* **and** ( $0 < r / \sqrt{2}$ ) **by** (*rule tendstoD*)  
**moreover**  
**have** *eventually* ( $\lambda x. dist\ (g\ x)\ b < r / \sqrt{2}$ ) *net*  
**using** ( $\langle g \dashrightarrow b \rangle$ ) *net* **and** ( $0 < r / \sqrt{2}$ ) **by** (*rule tendstoD*)  
**ultimately**  
**show** *eventually* ( $\lambda x. dist\ (Complex\ (f\ x)\ (g\ x))\ (Complex\ a\ b) < r$ ) *net*  
**by** (*rule eventually-elim2*)  
*(simp add: dist-norm real-sqrt-sum-squares-less)*  
**qed**

**lemma** *LIMSEQ-Complex*:  
 $\llbracket X \dashrightarrow a; Y \dashrightarrow b \rrbracket \implies (\lambda n. Complex\ (X\ n)\ (Y\ n)) \dashrightarrow Complex\ a\ b$   
**by** (*rule tendsto-Complex*)

**instance** *complex* :: *banach*  
**proof**

```

fix  $X :: \text{nat} \Rightarrow \text{complex}$ 
assume  $X: \text{Cauchy } X$ 
from  $\text{Re.Cauchy } [OF\ X]$  have  $1: (\lambda n. \text{Re } (X\ n)) \text{ ----> } \lim (\lambda n. \text{Re } (X\ n))$ 
  by ( $\text{simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff}$ )
from  $\text{Im.Cauchy } [OF\ X]$  have  $2: (\lambda n. \text{Im } (X\ n)) \text{ ----> } \lim (\lambda n. \text{Im } (X\ n))$ 
  by ( $\text{simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff}$ )
have  $X \text{ ----> } \text{Complex } (\lim (\lambda n. \text{Re } (X\ n))) (\lim (\lambda n. \text{Im } (X\ n)))$ 
  using  $\text{LIMSEQ-Complex } [OF\ 1\ 2]$  by  $\text{simp}$ 
thus  $\text{convergent } X$ 
  by ( $\text{rule convergentI}$ )
qed

```

## 77.8 The Complex Number $i$

### definition

$ii :: \text{complex } (i)$  **where**  
 $i\text{-def}: ii \equiv \text{Complex } 0\ 1$

**lemma**  $\text{complex-Re-}i$  [ $\text{simp}$ ]:  $\text{Re } ii = 0$   
**by** ( $\text{simp add: } i\text{-def}$ )

**lemma**  $\text{complex-Im-}i$  [ $\text{simp}$ ]:  $\text{Im } ii = 1$   
**by** ( $\text{simp add: } i\text{-def}$ )

**lemma**  $\text{Complex-eq-}i$  [ $\text{simp}$ ]:  $(\text{Complex } x\ y = ii) = (x = 0 \wedge y = 1)$   
**by** ( $\text{simp add: } i\text{-def}$ )

**lemma**  $\text{complex-}i\text{-not-zero}$  [ $\text{simp}$ ]:  $ii \neq 0$   
**by** ( $\text{simp add: expand-complex-eq}$ )

**lemma**  $\text{complex-}i\text{-not-one}$  [ $\text{simp}$ ]:  $ii \neq 1$   
**by** ( $\text{simp add: expand-complex-eq}$ )

**lemma**  $\text{complex-}i\text{-not-number-of}$  [ $\text{simp}$ ]:  $ii \neq \text{number-of } w$   
**by** ( $\text{simp add: expand-complex-eq}$ )

**lemma**  $i\text{-mult-Complex}$  [ $\text{simp}$ ]:  $ii * \text{Complex } a\ b = \text{Complex } (-\ b)\ a$   
**by** ( $\text{simp add: expand-complex-eq}$ )

**lemma**  $\text{Complex-mult-}i$  [ $\text{simp}$ ]:  $\text{Complex } a\ b * ii = \text{Complex } (-\ b)\ a$   
**by** ( $\text{simp add: expand-complex-eq}$ )

**lemma**  $i\text{-complex-of-real}$  [ $\text{simp}$ ]:  $ii * \text{complex-of-real } r = \text{Complex } 0\ r$   
**by** ( $\text{simp add: } i\text{-def complex-of-real-def}$ )

**lemma**  $\text{complex-of-real-}i$  [ $\text{simp}$ ]:  $\text{complex-of-real } r * ii = \text{Complex } 0\ r$   
**by** ( $\text{simp add: } i\text{-def complex-of-real-def}$ )

**lemma**  $i\text{-squared}$  [ $\text{simp}$ ]:  $ii * ii = -1$

**by** (*simp add: i-def*)

**lemma** *power2-i* [*simp*]:  $i^2 = -1$   
**by** (*simp add: power2-eq-square*)

**lemma** *inverse-i* [*simp*]:  $\text{inverse } ii = -\ ii$   
**by** (*rule inverse-unique, simp*)

## 77.9 Complex Conjugation

**definition**

$\text{cnj} :: \text{complex} \Rightarrow \text{complex}$  **where**  
 $\text{cnj } z = \text{Complex } (\text{Re } z) (-\ \text{Im } z)$

**lemma** *complex-cnj* [*simp*]:  $\text{cnj } (\text{Complex } a\ b) = \text{Complex } a\ (-\ b)$   
**by** (*simp add: cnj-def*)

**lemma** *complex-Re-cnj* [*simp*]:  $\text{Re } (\text{cnj } x) = \text{Re } x$   
**by** (*simp add: cnj-def*)

**lemma** *complex-Im-cnj* [*simp*]:  $\text{Im } (\text{cnj } x) = -\ \text{Im } x$   
**by** (*simp add: cnj-def*)

**lemma** *complex-cnj-cancel-iff* [*simp*]:  $(\text{cnj } x = \text{cnj } y) = (x = y)$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-cnj* [*simp*]:  $\text{cnj } (\text{cnj } z) = z$   
**by** (*simp add: cnj-def*)

**lemma** *complex-cnj-zero* [*simp*]:  $\text{cnj } 0 = 0$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-zero-iff* [*iff*]:  $(\text{cnj } z = 0) = (z = 0)$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-add*:  $\text{cnj } (x + y) = \text{cnj } x + \text{cnj } y$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-diff*:  $\text{cnj } (x - y) = \text{cnj } x - \text{cnj } y$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-minus*:  $\text{cnj } (-\ x) = -\ \text{cnj } x$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-one* [*simp*]:  $\text{cnj } 1 = 1$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-mult*:  $\text{cnj } (x * y) = \text{cnj } x * \text{cnj } y$   
**by** (*simp add: expand-complex-eq*)



**lemma** *complex-cnj-inverse*:  $\text{cnj } (\text{inverse } x) = \text{inverse } (\text{cnj } x)$   
**by** (*simp add: complex-inverse-def*)

**lemma** *complex-cnj-divide*:  $\text{cnj } (x / y) = \text{cnj } x / \text{cnj } y$   
**by** (*simp add: complex-divide-def complex-cnj-mult complex-cnj-inverse*)

**lemma** *complex-cnj-power*:  $\text{cnj } (x ^ n) = \text{cnj } x ^ n$   
**by** (*induct n, simp-all add: complex-cnj-mult*)

**lemma** *complex-cnj-of-nat* [*simp*]:  $\text{cnj } (\text{of-nat } n) = \text{of-nat } n$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-of-int* [*simp*]:  $\text{cnj } (\text{of-int } z) = \text{of-int } z$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-number-of* [*simp*]:  $\text{cnj } (\text{number-of } w) = \text{number-of } w$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-scaleR*:  $\text{cnj } (\text{scaleR } r x) = \text{scaleR } r (\text{cnj } x)$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-mod-cnj* [*simp*]:  $\text{cmod } (\text{cnj } z) = \text{cmod } z$   
**by** (*simp add: complex-norm-def*)

**lemma** *complex-cnj-complex-of-real* [*simp*]:  $\text{cnj } (\text{of-real } x) = \text{of-real } x$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-cnj-i* [*simp*]:  $\text{cnj } ii = - ii$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-add-cnj*:  $z + \text{cnj } z = \text{complex-of-real } (2 * \text{Re } z)$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-diff-cnj*:  $z - \text{cnj } z = \text{complex-of-real } (2 * \text{Im } z) * ii$   
**by** (*simp add: expand-complex-eq*)

**lemma** *complex-mult-cnj*:  $z * \text{cnj } z = \text{complex-of-real } ((\text{Re } z)^2 + (\text{Im } z)^2)$   
**by** (*simp add: expand-complex-eq power2-eq-square*)

**lemma** *complex-mod-mult-cnj*:  $\text{cmod } (z * \text{cnj } z) = (\text{cmod } z)^2$   
**by** (*simp add: norm-mult power2-eq-square*)

**interpretation** *cnj*: *bounded-linear cnj*  
**apply** (*unfold-locales*)  
**apply** (*rule complex-cnj-add*)  
**apply** (*rule complex-cnj-scaleR*)  
**apply** (*rule-tac x=1 in exI, simp*)  
**done**

**77.10 The Functions *sgn* and *arg***

————— Argand —————

**definition**

*arg* :: complex => real **where**  
*arg* z = (SOME a. Re(*sgn* z) = cos a & Im(*sgn* z) = sin a & -pi < a & a ≤ pi)

**lemma** *sgn-eq*: *sgn* z = z / complex-of-real (cmod z)

**by** (simp add: complex-sgn-def divide-inverse scaleR-conv-of-real mult-commute)

**lemma** *i-mult-eq*: ii \* ii = complex-of-real (-1)

**by** (simp add: i-def complex-of-real-def)

**lemma** *i-mult-eq2* [simp]: ii \* ii = -(1::complex)

**by** (simp add: i-def complex-one-def)

**lemma** *complex-eq-cancel-iff2* [simp]:

(Complex x y = complex-of-real xa) = (x = xa & y = 0)

**by** (simp add: complex-of-real-def)

**lemma** *Re-sgn* [simp]: Re(*sgn* z) = Re(z)/cmod z

**by** (simp add: complex-sgn-def divide-inverse)

**lemma** *Im-sgn* [simp]: Im(*sgn* z) = Im(z)/cmod z

**by** (simp add: complex-sgn-def divide-inverse)

**lemma** *complex-inverse-complex-split*:

inverse(complex-of-real x + ii \* complex-of-real y) =

complex-of-real(x/(x ^ 2 + y ^ 2)) -

ii \* complex-of-real(y/(x ^ 2 + y ^ 2))

**by** (simp add: complex-of-real-def i-def diff-minus divide-inverse)

**lemma** *cos-arg-i-mult-zero-pos*:

0 < y ==> cos (arg (Complex 0 y)) = 0

**apply** (simp add: arg-def abs-if)

**apply** (rule-tac a = pi/2 in someI2, auto)

**apply** (rule order-less-trans [of - 0], auto)

**done**

**lemma** *cos-arg-i-mult-zero-neg*:

y < 0 ==> cos (arg (Complex 0 y)) = 0

**apply** (simp add: arg-def abs-if)

**apply** (rule-tac a = - pi/2 in someI2, auto)

**apply** (*rule order-trans [of - 0], auto*)  
**done**

**lemma** *cos-arg-i-mult-zero [simp]:*

$y \neq 0 \implies \cos(\arg(\text{Complex } 0\ y)) = 0$

**by** (*auto simp add: linorder-neq-iff cos-arg-i-mult-zero-pos cos-arg-i-mult-zero-neg*)

## 77.11 Finally! Polar Form for Complex Numbers

**definition**

*cis* :: *real* => *complex* **where**  
*cis* *a* = *Complex* (*cos a*) (*sin a*)

**definition**

*rcis* :: [*real*, *real*] => *complex* **where**  
*rcis* *r a* = *complex-of-real* *r* \* *cis a*

**definition**

*expi* :: *complex* => *complex* **where**  
*expi* *z* = *complex-of-real*(*exp* (*Re z*)) \* *cis* (*Im z*)

**lemma** *complex-split-polar:*

$\exists r\ a.\ z = \text{complex-of-real } r * (\text{Complex } (\cos a) (\sin a))$

**apply** (*induct z*)

**apply** (*auto simp add: polar-Ex complex-of-real-mult-Complex*)

**done**

**lemma** *rcis-Ex:  $\exists r\ a.\ z = \text{rcis } r\ a$*

**apply** (*induct z*)

**apply** (*simp add: rcis-def cis-def polar-Ex complex-of-real-mult-Complex*)

**done**

**lemma** *Re-rcis [simp]:  $\text{Re}(\text{rcis } r\ a) = r * \cos a$*

**by** (*simp add: rcis-def cis-def*)

**lemma** *Im-rcis [simp]:  $\text{Im}(\text{rcis } r\ a) = r * \sin a$*

**by** (*simp add: rcis-def cis-def*)

**lemma** *sin-cos-squared-add2-mult:  $(r * \cos a)^2 + (r * \sin a)^2 = r^2$*

**proof** —

**have**  $(r * \cos a)^2 + (r * \sin a)^2 = r^2 * ((\cos a)^2 + (\sin a)^2)$

**by** (*simp only: power-mult-distrib right-distrib*)

**thus** *?thesis* **by** *simp*

**qed**

**lemma** *complex-mod-rcis* [simp]:  $\text{cmod}(\text{rcis } r \ a) = \text{abs } r$   
**by** (simp add: rcis-def cis-def sin-cos-squared-add2-mult)

**lemma** *complex-mod-sqrt-Re-mult-cnj*:  $\text{cmod } z = \text{sqrt } (\text{Re } (z * \text{cnj } z))$   
**by** (simp add: cmod-def power2-eq-square)

**lemma** *complex-In-mult-cnj-zero* [simp]:  $\text{Im } (z * \text{cnj } z) = 0$   
**by** simp

**lemma** *cis-rcis-eq*:  $\text{cis } a = \text{rcis } 1 \ a$   
**by** (simp add: rcis-def)

**lemma** *rcis-mult*:  $\text{rcis } r1 \ a * \text{rcis } r2 \ b = \text{rcis } (r1 * r2) \ (a + b)$   
**by** (simp add: rcis-def cis-def cos-add sin-add right-distrib right-diff-distrib  
 complex-of-real-def)

**lemma** *cis-mult*:  $\text{cis } a * \text{cis } b = \text{cis } (a + b)$   
**by** (simp add: cis-rcis-eq rcis-mult)

**lemma** *cis-zero* [simp]:  $\text{cis } 0 = 1$   
**by** (simp add: cis-def complex-one-def)

**lemma** *rcis-zero-mod* [simp]:  $\text{rcis } 0 \ a = 0$   
**by** (simp add: rcis-def)

**lemma** *rcis-zero-arg* [simp]:  $\text{rcis } r \ 0 = \text{complex-of-real } r$   
**by** (simp add: rcis-def)

**lemma** *complex-of-real-minus-one*:  
 $\text{complex-of-real } (-(1::\text{real})) = -(1::\text{complex})$   
**by** (simp add: complex-of-real-def complex-one-def)

**lemma** *complex-i-mult-minus* [simp]:  $i * (i * x) = -x$   
**by** (simp add: mult-assoc [symmetric])

**lemma** *cis-real-of-nat-Suc-mult*:  
 $\text{cis } (\text{real } (\text{Suc } n) * a) = \text{cis } a * \text{cis } (\text{real } n * a)$   
**by** (simp add: cis-def real-of-nat-Suc left-distrib cos-add sin-add right-distrib)

**lemma** *DeMoivre*:  $(\text{cis } a) ^ n = \text{cis } (\text{real } n * a)$   
**apply** (induct-tac n)  
**apply** (auto simp add: cis-real-of-nat-Suc-mult)  
**done**

**lemma** *DeMoivre2*:  $(rcis\ r\ a) ^ n = rcis\ (r ^ n)\ (real\ n * a)$   
**by** (*simp add: rcis-def power-mult-distrib DeMoivre*)

**lemma** *cis-inverse* [*simp*]:  $inverse(cis\ a) = cis\ (-a)$   
**by** (*simp add: cis-def complex-inverse-complex-split diff-minus*)

**lemma** *rcis-inverse*:  $inverse(rcis\ r\ a) = rcis\ (1/r)\ (-a)$   
**by** (*simp add: divide-inverse rcis-def*)

**lemma** *cis-divide*:  $cis\ a / cis\ b = cis\ (a - b)$   
**by** (*simp add: complex-divide-def cis-mult diff-def*)

**lemma** *rcis-divide*:  $rcis\ r1\ a / rcis\ r2\ b = rcis\ (r1/r2)\ (a - b)$   
**apply** (*simp add: complex-divide-def*)  
**apply** (*case-tac r2=0, simp*)  
**apply** (*simp add: rcis-inverse rcis-mult diff-def*)  
**done**

**lemma** *Re-cis* [*simp*]:  $Re(cis\ a) = cos\ a$   
**by** (*simp add: cis-def*)

**lemma** *Im-cis* [*simp*]:  $Im(cis\ a) = sin\ a$   
**by** (*simp add: cis-def*)

**lemma** *cos-n-Re-cis-pow-n*:  $cos\ (real\ n * a) = Re(cis\ a ^ n)$   
**by** (*auto simp add: DeMoivre*)

**lemma** *sin-n-Im-cis-pow-n*:  $sin\ (real\ n * a) = Im(cis\ a ^ n)$   
**by** (*auto simp add: DeMoivre*)

**lemma** *expi-add*:  $expi(a + b) = expi(a) * expi(b)$   
**by** (*simp add: expi-def exp-add cis-mult [symmetric] mult-ac*)

**lemma** *expi-zero* [*simp*]:  $expi\ (0::complex) = 1$   
**by** (*simp add: expi-def*)

**lemma** *complex-expi-Ex*:  $\exists a\ r. z = complex-of-real\ r * expi\ a$   
**apply** (*insert rcis-Ex [of z]*)  
**apply** (*auto simp add: expi-def rcis-def mult-assoc [symmetric]*)  
**apply** (*rule-tac x = ii \* complex-of-real a in exI, auto*)  
**done**

**lemma** *expi-two-pi-i* [*simp*]:  $expi((2::complex) * complex-of-real\ pi * ii) = 1$   
**by** (*simp add: expi-def cis-def*)

**end**

## 78 Log: Logarithms: Standard Version

```
theory Log
imports Transcendental
begin
```

### definition

```
powr :: [real,real] => real    (infixr powr 80) where
  — exponentiation with real exponent
  x powr a = exp(a * ln x)
```

### definition

```
log :: [real,real] => real where
  — logarithm of x to base a
  log a x = ln x / ln a
```

```
lemma powr-one-eq-one [simp]: 1 powr a = 1
by (simp add: powr-def)
```

```
lemma powr-zero-eq-one [simp]: x powr 0 = 1
by (simp add: powr-def)
```

```
lemma powr-one-gt-zero-iff [simp]: (x powr 1 = x) = (0 < x)
by (simp add: powr-def)
declare powr-one-gt-zero-iff [THEN iffD2, simp]
```

```
lemma powr-mult:
  [| 0 < x; 0 < y |] ==> (x * y) powr a = (x powr a) * (y powr a)
by (simp add: powr-def exp-add [symmetric] ln-mult right-distrib)
```

```
lemma powr-gt-zero [simp]: 0 < x powr a
by (simp add: powr-def)
```

```
lemma powr-ge-pzero [simp]: 0 <= x powr y
by (rule order-less-imp-le, rule powr-gt-zero)
```

```
lemma powr-not-zero [simp]: x powr a ≠ 0
by (simp add: powr-def)
```

### lemma powr-divide:

```
  [| 0 < x; 0 < y |] ==> (x / y) powr a = (x powr a)/(y powr a)
apply (simp add: divide-inverse positive-imp-inverse-positive powr-mult)
apply (simp add: powr-def exp-minus [symmetric] exp-add [symmetric] ln-inverse)
done
```

```
lemma powr-divide2: x powr a / x powr b = x powr (a - b)
  apply (simp add: powr-def)
```

```

apply (subst exp-diff [THEN sym])
apply (simp add: left-diff-distrib)
done

```

```

lemma powr-add:  $x \text{ powr } (a + b) = (x \text{ powr } a) * (x \text{ powr } b)$ 
by (simp add: powr-def exp-add [symmetric] left-distrib)

```

```

lemma powr-powr:  $(x \text{ powr } a) \text{ powr } b = x \text{ powr } (a * b)$ 
by (simp add: powr-def)

```

```

lemma powr-powr-swap:  $(x \text{ powr } a) \text{ powr } b = (x \text{ powr } b) \text{ powr } a$ 
by (simp add: powr-powr mult-commute)

```

```

lemma powr-minus:  $x \text{ powr } (-a) = \text{inverse } (x \text{ powr } a)$ 
by (simp add: powr-def exp-minus [symmetric])

```

```

lemma powr-minus-divide:  $x \text{ powr } (-a) = 1 / (x \text{ powr } a)$ 
by (simp add: divide-inverse powr-minus)

```

```

lemma powr-less-mono:  $[[ a < b; 1 < x ]] ==> x \text{ powr } a < x \text{ powr } b$ 
by (simp add: powr-def)

```

```

lemma powr-less-cancel:  $[[ x \text{ powr } a < x \text{ powr } b; 1 < x ]] ==> a < b$ 
by (simp add: powr-def)

```

```

lemma powr-less-cancel-iff [simp]:  $1 < x ==> (x \text{ powr } a < x \text{ powr } b) = (a < b)$ 
by (blast intro: powr-less-cancel powr-less-mono)

```

```

lemma powr-le-cancel-iff [simp]:  $1 < x ==> (x \text{ powr } a \leq x \text{ powr } b) = (a \leq b)$ 
by (simp add: linorder-not-less [symmetric])

```

```

lemma log-ln:  $\ln x = \log (\exp(1)) x$ 
by (simp add: log-def)

```

```

lemma DERIV-log:  $x > 0 ==> \text{DERIV } (\%y. \log b y) x :> 1 / (\ln b * x)$ 
apply (subst log-def)
apply (subgoal-tac (%y.  $\ln y / \ln b = (\%y. (1 / \ln b) * \ln y)$ )
apply (erule ssubst)
apply (subgoal-tac  $1 / (\ln b * x) = (1 / \ln b) * (1 / x)$ )
apply (erule ssubst)
apply (rule DERIV-cmult)
apply (erule DERIV-ln-divide)
apply auto
done

```

```

lemma powr-log-cancel [simp]:
   $[[ 0 < a; a \neq 1; 0 < x ]] ==> a \text{ powr } (\log a x) = x$ 
by (simp add: powr-def log-def)

```

**lemma** *log-powr-cancel* [simp]:  $[[0 < a; a \neq 1]] \implies \log a (a \text{ powr } y) = y$   
**by** (simp add: log-def powr-def)

**lemma** *log-mult*:  
 $[[0 < a; a \neq 1; 0 < x; 0 < y]] \implies \log a (x * y) = \log a x + \log a y$   
**by** (simp add: log-def ln-mult divide-inverse left-distrib)

**lemma** *log-eq-div-ln-mult-log*:  
 $[[0 < a; a \neq 1; 0 < b; b \neq 1; 0 < x]] \implies \log a x = (\ln b / \ln a) * \log b x$   
**by** (simp add: log-def divide-inverse)

Base 10 logarithms

**lemma** *log-base-10-eq1*:  $0 < x \implies \log 10 x = (\ln (\exp 1) / \ln 10) * \ln x$   
**by** (simp add: log-def)

**lemma** *log-base-10-eq2*:  $0 < x \implies \log 10 x = (\log 10 (\exp 1)) * \ln x$   
**by** (simp add: log-def)

**lemma** *log-one* [simp]:  $\log a 1 = 0$   
**by** (simp add: log-def)

**lemma** *log-eq-one* [simp]:  $[[0 < a; a \neq 1]] \implies \log a a = 1$   
**by** (simp add: log-def)

**lemma** *log-inverse*:  
 $[[0 < a; a \neq 1; 0 < x]] \implies \log a (\text{inverse } x) = - \log a x$   
**apply** (rule-tac a1 =  $\log a x$  **in** add-left-cancel [THEN iffD1])  
**apply** (simp add: log-mult [symmetric])  
**done**

**lemma** *log-divide*:  
 $[[0 < a; a \neq 1; 0 < x; 0 < y]] \implies \log a (x/y) = \log a x - \log a y$   
**by** (simp add: log-mult divide-inverse log-inverse)

**lemma** *log-less-cancel-iff* [simp]:  
 $[[1 < a; 0 < x; 0 < y]] \implies (\log a x < \log a y) = (x < y)$   
**apply** safe  
**apply** (rule-tac [2] powr-less-cancel)  
**apply** (drule-tac a =  $\log a x$  **in** powr-less-mono, auto)  
**done**

**lemma** *log-inj*: **assumes**  $1 < b$  **shows** *inj-on* ( $\log b$ )  $\{0 < ..\}$   
**proof** (rule inj-onI, simp)  
**fix**  $x y$  **assume**  $pos: 0 < x < y$  **and**  $*: \log b x = \log b y$   
**show**  $x = y$   
**proof** (cases rule: linorder-cases)  
**assume**  $x < y$  **hence**  $\log b x < \log b y$



```

    using log-less-cancel-iff[OF <1 < b>] pos by simp
  thus ?thesis using * by simp
next
  assume y < x hence log b y < log b x
  using log-less-cancel-iff[OF <1 < b>] pos by simp
  thus ?thesis using * by simp
qed simp
qed

```

```

lemma log-le-cancel-iff [simp]:
  [| 1 < a; 0 < x; 0 < y |] ==> (log a x ≤ log a y) = (x ≤ y)
by (simp add: linorder-not-less [symmetric])

```

```

lemma powr-realpow: 0 < x ==> x powr (real n) = x ^ n
  apply (induct n, simp)
  apply (subgoal-tac real (Suc n) = real n + 1)
  apply (erule ssubst)
  apply (subst powr-add, simp, simp)
done

```

```

lemma powr-realpow2: 0 ≤ x ==> 0 < n ==> x ^ n = (if (x = 0) then 0
  else x powr (real n))
  apply (case-tac x = 0, simp, simp)
  apply (rule powr-realpow [THEN sym], simp)
done

```

```

lemma ln-powr: 0 < x ==> 0 < y ==> ln(x powr y) = y * ln x
by (unfold powr-def, simp)

```

```

lemma log-powr: 0 < x ==> 0 ≤ y ==> log b (x powr y) = y * log b x
  apply (case-tac y = 0)
  apply force
  apply (auto simp add: log-def ln-powr field-simps)
done

```

```

lemma log-nat-power: 0 < x ==> log b (x ^ n) = real n * log b x
  apply (subst powr-realpow [symmetric])
  apply (auto simp add: log-powr)
done

```

```

lemma ln-bound: 1 ≤ x ==> ln x ≤ x
  apply (subgoal-tac ln(1 + (x - 1)) ≤ x - 1)
  apply simp
  apply (rule ln-add-one-self-le-self, simp)
done

```

```

lemma powr-mono: a ≤ b ==> 1 ≤ x ==> x powr a ≤ x powr b
  apply (case-tac x = 1, simp)

```

```

  apply (case-tac a = b, simp)
  apply (rule order-less-imp-le)
  apply (rule powr-less-mono, auto)
done

```

```

lemma ge-one-powr-ge-zero: 1 <= x ==> 0 <= a ==> 1 <= x powr a
  apply (subst powr-zero-eq-one [THEN sym])
  apply (rule powr-mono, assumption+)
done

```

```

lemma powr-less-mono2: 0 < a ==> 0 < x ==> x < y ==> x powr a <
  y powr a
  apply (unfold powr-def)
  apply (rule exp-less-mono)
  apply (rule mult-strict-left-mono)
  apply (subst ln-less-cancel-iff, assumption)
  apply (rule order-less-trans)
  prefer 2
  apply assumption+
done

```

```

lemma powr-less-mono2-neg: a < 0 ==> 0 < x ==> x < y ==> y powr a <
  x powr a
  apply (unfold powr-def)
  apply (rule exp-less-mono)
  apply (rule mult-strict-left-mono-neg)
  apply (subst ln-less-cancel-iff)
  apply assumption
  apply (rule order-less-trans)
  prefer 2
  apply assumption+
done

```

```

lemma powr-mono2: 0 <= a ==> 0 < x ==> x <= y ==> x powr a <= y
  powr a
  apply (case-tac a = 0, simp)
  apply (case-tac x = y, simp)
  apply (rule order-less-imp-le)
  apply (rule powr-less-mono2, auto)
done

```

```

lemma ln-powr-bound: 1 <= x ==> 0 < a ==> ln x <= (x powr a) / a
  apply (rule mult-imp-le-div-pos)
  apply (assumption)
  apply (subst mult-commute)
  apply (subst ln-powr [THEN sym])
  apply auto
  apply (rule ln-bound)
  apply (erule ge-one-powr-ge-zero)

```

**apply** (*erule order-less-imp-le*)  
**done**

**lemma** *ln-powr-bound2*:  $1 < x \implies 0 < a \implies (\ln x) \text{ powr } a \leq (a \text{ powr } a) * x$

**proof** –

**assume**  $1 < x$  **and**  $0 < a$

**then have**  $\ln x \leq (x \text{ powr } (1 / a)) / (1 / a)$

**apply** (*intro ln-powr-bound*)

**apply** (*erule order-less-imp-le*)

**apply** (*rule divide-pos-pos*)

**apply** *simp-all*

**done**

**also have**  $\dots = a * (x \text{ powr } (1 / a))$

**by** *simp*

**finally have**  $(\ln x) \text{ powr } a \leq (a * (x \text{ powr } (1 / a))) \text{ powr } a$

**apply** (*intro powr-mono2*)

**apply** (*rule order-less-imp-le, rule prems*)

**apply** (*rule ln-gt-zero*)

**apply** (*rule prems*)

**apply** *assumption*

**done**

**also have**  $\dots = (a \text{ powr } a) * ((x \text{ powr } (1 / a)) \text{ powr } a)$

**apply** (*rule powr-mult*)

**apply** (*rule prems*)

**apply** (*rule powr-gt-zero*)

**done**

**also have**  $(x \text{ powr } (1 / a)) \text{ powr } a = x \text{ powr } ((1 / a) * a)$

**by** (*rule powr-powr*)

**also have**  $\dots = x$

**apply** *simp*

**apply** (*subgoal-tac a ~ = 0*)

**apply** (*insert prems, auto*)

**done**

**finally show** *?thesis* .

**qed**

**lemma** *LIMSEQ-neg-powr*:  $0 < s \implies (\%x. (\text{real } x) \text{ powr } -s) \dashrightarrow 0$

**apply** (*unfold LIMSEQ-iff*)

**apply** *clarsimp*

**apply** (*rule-tac x = natfloor(r powr (1 / -s)) + 1 in exI*)

**apply** *clarify*

**proof** –

**fix** *r* **fix** *n*

**assume**  $0 < s$  **and**  $0 < r$  **and**  $\text{natfloor } (r \text{ powr } (1 / -s)) + 1 \leq n$

**have**  $r \text{ powr } (1 / -s) < \text{real}(\text{natfloor}(r \text{ powr } (1 / -s))) + 1$

**by** (*rule real-natfloor-add-one-gt*)

**also have**  $\dots = \text{real}(\text{natfloor}(r \text{ powr } (1 / -s)) + 1)$

**by** *simp*

```

    also have ... <= real n
    apply (subst real-of-nat-le-iff)
    apply (rule prems)
    done
  finally have  $r \text{ powr } (1 / - s) < \text{real } n$ .
  then have  $\text{real } n \text{ powr } (- s) < (r \text{ powr } (1 / - s)) \text{ powr } - s$ 
    apply (intro powr-less-mono2-neg)
    apply (auto simp add: prems)
    done
  also have ... = r
    by (simp add: powr-powr prems less-imp-neq [THEN not-sym])
  finally show  $\text{real } n \text{ powr } - s < r$  .
qed

end

```

## 79 Ln: Properties of ln

```

theory Ln
imports Transcendental
begin

```

```

lemma exp-first-two-terms:  $\exp x = 1 + x + \text{suminf } (\%n. \text{inverse}(\text{real } (\text{fact } (n+2))) * (x ^ (n+2)))$ 
proof -
  have  $\exp x = \text{suminf } (\%n. \text{inverse}(\text{real } (\text{fact } n)) * (x ^ n))$ 
    by (simp add: exp-def)
  also from summable-exp have ... =  $(\text{SUM } n : \{0..<2\}. \text{inverse}(\text{real } (\text{fact } n)) * (x ^ n)) + \text{suminf } (\%n. \text{inverse}(\text{real } (\text{fact } (n+2))) * (x ^ (n+2)))$  (is - = ?a + -)
    by (rule suminf-split-initial-segment)
  also have ?a =  $1 + x$ 
    by (simp add: numerals)
  finally show ?thesis .
qed

```

```

lemma exp-tail-after-first-two-terms-summable:
  summable (%n.  $\text{inverse}(\text{real } (\text{fact } (n+2))) * (x ^ (n+2))$ )
proof -
  note summable-exp
  thus ?thesis
    by (frule summable-ignore-initial-segment)
qed

```

```

lemma aux1: assumes  $a: 0 \leq x$  and  $b: x \leq 1$ 
  shows  $\text{inverse } (\text{real } (\text{fact } ((n::nat) + 2))) * x ^ (n + 2) \leq (x^2/2) * ((1/2)^n)$ 
proof (induct n)

```

```

show  $\text{inverse}(\text{real}(\text{fact}((0::\text{nat}) + 2))) * x ^ (0 + 2) <=$ 
 $x ^ 2 / 2 * (1 / 2) ^ 0$ 
by (simp add: real-of-nat-Suc power2-eq-square)
next
fix  $n :: \text{nat}$ 
assume  $c: \text{inverse}(\text{real}(\text{fact}(n + 2))) * x ^ (n + 2)$ 
 $<= x ^ 2 / 2 * (1 / 2) ^ n$ 
show  $\text{inverse}(\text{real}(\text{fact}(\text{Suc } n + 2))) * x ^ (\text{Suc } n + 2)$ 
 $<= x ^ 2 / 2 * (1 / 2) ^ \text{Suc } n$ 
proof –
have  $\text{inverse}(\text{real}(\text{fact}(\text{Suc } n + 2))) <=$ 
 $(1 / 2) * \text{inverse}(\text{real}(\text{fact}(n + 2)))$ 
proof –
have  $\text{Suc } n + 2 = \text{Suc } (n + 2)$  by simp
then have  $\text{fact}(\text{Suc } n + 2) = \text{Suc } (n + 2) * \text{fact}(n + 2)$ 
by simp
then have  $\text{real}(\text{fact}(\text{Suc } n + 2)) = \text{real}(\text{Suc } (n + 2) * \text{fact}(n + 2))$ 
apply (rule subst)
apply (rule refl)
done
also have  $\dots = \text{real}(\text{Suc } (n + 2)) * \text{real}(\text{fact}(n + 2))$ 
by (rule real-of-nat-mult)
finally have  $\text{real}(\text{fact}(\text{Suc } n + 2)) =$ 
 $\text{real}(\text{Suc } (n + 2)) * \text{real}(\text{fact}(n + 2))$  .
then have  $\text{inverse}(\text{real}(\text{fact}(\text{Suc } n + 2))) =$ 
 $\text{inverse}(\text{real}(\text{Suc } (n + 2))) * \text{inverse}(\text{real}(\text{fact}(n + 2)))$ 
apply (rule ssubst)
apply (rule inverse-mult-distrib)
done
also have  $\dots <= (1/2) * \text{inverse}(\text{real}(\text{fact}(n + 2)))$ 
apply (rule mult-right-mono)
apply (subst inverse-eq-divide)
apply simp
apply (rule inv-real-of-nat-fact-ge-zero)
done
finally show ?thesis .
qed
moreover have  $x ^ (\text{Suc } n + 2) <= x ^ (n + 2)$ 
apply (simp add: mult-compare-simps)
apply (simp add: prems)
apply (subgoal-tac 0 <= x * (x * x^n))
apply force
apply (rule mult-nonneg-nonneg, rule a) +
apply (rule zero-le-power, rule a)
done
ultimately have  $\text{inverse}(\text{real}(\text{fact}(\text{Suc } n + 2))) * x ^ (\text{Suc } n + 2) <=$ 
 $(1 / 2 * \text{inverse}(\text{real}(\text{fact}(n + 2)))) * x ^ (n + 2)$ 
apply (rule mult-mono)
apply (rule mult-nonneg-nonneg)

```

```

    apply simp
    apply (subst inverse-nonnegative-iff-nonnegative)
    apply (rule real-of-nat-ge-zero)
    apply (rule zero-le-power)
    apply (rule a)
  done
also have ... = 1 / 2 * (inverse (real (fact (n + 2))) * x ^ (n + 2))
  by simp
also have ... <= 1 / 2 * (x ^ 2 / 2 * (1 / 2) ^ n)
  apply (rule mult-left-mono)
  apply (rule prems)
  apply simp
done
also have ... = x ^ 2 / 2 * (1 / 2 * (1 / 2) ^ n)
  by auto
also have (1::real) / 2 * (1 / 2) ^ n = (1 / 2) ^ (Suc n)
  by (rule power-Suc [THEN sym])
finally show ?thesis .
qed
qed

lemma aux2: (%n. (x::real) ^ 2 / 2 * (1 / 2) ^ n) sums x^2
proof -
  have (%n. (1 / 2::real) ^ n) sums (1 / (1 - (1/2)))
    apply (rule geometric-sums)
    by (simp add: abs-less-iff)
  also have (1::real) / (1 - 1/2) = 2
    by simp
  finally have (%n. (1 / 2::real) ^ n) sums 2 .
  then have (%n. x ^ 2 / 2 * (1 / 2) ^ n) sums (x^2 / 2 * 2)
    by (rule sums-mult)
  also have x^2 / 2 * 2 = x^2
    by simp
  finally show ?thesis .
qed

lemma exp-bound: 0 <= (x::real) ==> x <= 1 ==> exp x <= 1 + x + x^2
proof -
  assume a: 0 <= x
  assume b: x <= 1
  have c: exp x = 1 + x + suminf (%n. inverse(real (fact (n+2))) *
    (x ^ (n+2)))
    by (rule exp-first-two-terms)
  moreover have suminf (%n. inverse(real (fact (n+2))) * (x ^ (n+2))) <= x^2
  proof -
    have suminf (%n. inverse(real (fact (n+2))) * (x ^ (n+2))) <=
      suminf (%n. (x^2/2) * ((1/2) ^ n))
      apply (rule summable-le)
      apply (auto simp only: aux1 prems)

```

```

    apply (rule exp-tail-after-first-two-terms-summable)
    by (rule sums-summable, rule aux2)
  also have ... =  $x^2$ 
    by (rule sums-unique [THEN sym], rule aux2)
  finally show ?thesis .
qed
ultimately show ?thesis
  by auto
qed

```

**lemma** *aux4*:  $0 \leq (x::real) \implies x \leq 1 \implies \exp(x - x^2) \leq 1 + x$   
**proof** –

```

  assume a:  $0 \leq x$  and b:  $x \leq 1$ 
  have  $\exp(x - x^2) = \exp x / \exp(x^2)$ 
    by (rule exp-diff)
  also have ...  $\leq (1 + x + x^2) / \exp(x^2)$ 
    apply (rule divide-right-mono)
    apply (rule exp-bound)
    apply (rule a, rule b)
    apply simp
  done
  also have ...  $\leq (1 + x + x^2) / (1 + x^2)$ 
    apply (rule divide-left-mono)
    apply (auto simp add: exp-ge-add-one-self-aux)
    apply (rule add-nonneg-nonneg)
    apply (insert prems, auto)
    apply (rule mult-pos-pos)
    apply auto
    apply (rule add-pos-nonneg)
    apply auto
  done
  also from a have ...  $\leq 1 + x$ 
    by (simp add: field-simps zero-compare-simps)
  finally show ?thesis .
qed

```

**lemma** *ln-one-plus-pos-lower-bound*:  $0 \leq x \implies x \leq 1 \implies$   
 $x - x^2 \leq \ln(1 + x)$

```

proof –
  assume a:  $0 \leq x$  and b:  $x \leq 1$ 
  then have  $\exp(x - x^2) \leq 1 + x$ 
    by (rule aux4)
  also have ... =  $\exp(\ln(1 + x))$ 
    proof –
      from a have  $0 < 1 + x$  by auto
      thus ?thesis
        by (auto simp only: exp-ln-iff [THEN sym])
    qed
  finally have  $\exp(x - x^2) \leq \exp(\ln(1 + x))$  .

```

thus ?thesis by (auto simp only: exp-le-cancel-iff)  
qed

lemma ln-one-minus-pos-upper-bound:  $0 \leq x \implies x < 1 \implies \ln(1 - x) \leq -x$

proof -

assume  $a: 0 \leq (x::\text{real})$  and  $b: x < 1$   
have  $(1 - x) * (1 + x + x^2) = (1 - x^3)$   
by (simp add: algebra-simps power2-eq-square power3-eq-cube)  
also have  $\dots \leq 1$   
by (auto simp add: a)  
finally have  $(1 - x) * (1 + x + x^2) \leq 1$ .  
moreover have  $0 < 1 + x + x^2$   
apply (rule add-pos-nonneg)  
apply (insert a, auto)  
done

ultimately have  $1 - x \leq 1 / (1 + x + x^2)$   
by (elim mult-imp-le-div-pos)

also have  $\dots \leq 1 / \exp x$   
apply (rule divide-left-mono)  
apply (rule exp-bound, rule a)  
apply (insert prems, auto)  
apply (rule mult-pos-pos)  
apply (rule add-pos-nonneg)  
apply auto  
done

also have  $\dots = \exp(-x)$   
by (auto simp add: exp-minus divide-inverse)

finally have  $1 - x \leq \exp(-x)$ .  
also have  $1 - x = \exp(\ln(1 - x))$

proof -

have  $0 < 1 - x$   
by (insert b, auto)  
thus ?thesis  
by (auto simp only: exp-ln-iff [THEN sym])

qed

finally have  $\exp(\ln(1 - x)) \leq \exp(-x)$ .  
thus ?thesis by (auto simp only: exp-le-cancel-iff)

qed

lemma aux5:  $x < 1 \implies \ln(1 - x) = -\ln(1 + x / (1 - x))$

proof -

assume  $a: x < 1$   
have  $\ln(1 - x) = -\ln(1 / (1 - x))$   
proof -  
have  $\ln(1 - x) = -(-\ln(1 - x))$   
by auto  
also have  $-\ln(1 - x) = \ln 1 - \ln(1 - x)$   
by simp



```

    also have ... =  $\ln(1 / (1 - x))$ 
    apply (rule ln-div [THEN sym])
    by (insert a, auto)
    finally show ?thesis .
qed
    also have  $1 / (1 - x) = 1 + x / (1 - x)$  using a by(simp add:field-simps)
    finally show ?thesis .
qed

```

**lemma** *ln-one-minus-pos-lower-bound*:  $0 \leq x \implies x \leq (1 / 2) \implies -x - 2 * x^2 \leq \ln(1 - x)$

```

proof -
  assume a:  $0 \leq x$  and b:  $x \leq (1 / 2)$ 
  from b have c:  $x < 1$ 
  by auto
  then have  $\ln(1 - x) = -\ln(1 + x / (1 - x))$ 
  by (rule aux5)
  also have  $-(x / (1 - x)) \leq \dots$ 
  proof -
    have  $\ln(1 + x / (1 - x)) \leq x / (1 - x)$ 
    apply (rule ln-add-one-self-le-self)
    apply (rule divide-nonneg-pos)
    by (insert a c, auto)
    thus ?thesis
    by auto
  qed
  also have  $-(x / (1 - x)) = -x / (1 - x)$ 
  by auto
  finally have d:  $-x / (1 - x) \leq \ln(1 - x)$  .
  have  $0 < 1 - x$  using prems by simp
  hence e:  $-x - 2 * x^2 \leq -x / (1 - x)$ 
    using mult-right-le-one-le[of  $x * x$   $2 * x$ ] prems
    by(simp add:field-simps power2-eq-square)
  from e d show  $-x - 2 * x^2 \leq \ln(1 - x)$ 
    by (rule order-trans)
qed

```

**lemma** *exp-ge-add-one-self* [simp]:  $1 + (x::real) \leq \exp x$

```

  apply (case-tac  $0 \leq x$ )
  apply (erule exp-ge-add-one-self-aux)
  apply (case-tac  $x \leq -1$ )
  apply (subgoal-tac  $1 + x \leq 0$ )
  apply (erule order-trans)
  apply simp
  apply simp
  apply (subgoal-tac  $1 + x = \exp(\ln(1 + x))$ )
  apply (erule ssubst)
  apply (subst exp-le-cancel-iff)
  apply (subgoal-tac  $\ln(1 - (-x)) \leq -(-x)$ )

```

```

apply simp
apply (rule ln-one-minus-pos-upper-bound)
apply auto
done

```

```

lemma ln-add-one-self-le-self2:  $-1 < x \implies \ln(1 + x) \leq x$ 
apply (subgoal-tac  $x = \ln(\exp x)$ )
apply (erule ssubst)back
apply (subst ln-le-cancel-iff)
apply auto
done

```

```

lemma abs-ln-one-plus-x-minus-x-bound-nonneg:
 $0 \leq x \implies x \leq 1 \implies \text{abs}(\ln(1 + x) - x) \leq x^2$ 
proof -
  assume  $x: 0 \leq x$ 
  assume  $x \leq 1$ 
  from  $x$  have  $\ln(1 + x) \leq x$ 
    by (rule ln-add-one-self-le-self)
  then have  $\ln(1 + x) - x \leq 0$ 
    by simp
  then have  $\text{abs}(\ln(1 + x) - x) = -(\ln(1 + x) - x)$ 
    by (rule abs-of-nonpos)
  also have  $\dots = x - \ln(1 + x)$ 
    by simp
  also have  $\dots \leq x^2$ 
  proof -
    from prems have  $x - x^2 \leq \ln(1 + x)$ 
      by (intro ln-one-plus-pos-lower-bound)
    thus ?thesis
      by simp
  qed
finally show ?thesis .
qed

```

```

lemma abs-ln-one-plus-x-minus-x-bound-nonpos:
 $-(1 / 2) \leq x \implies x \leq 0 \implies \text{abs}(\ln(1 + x) - x) \leq 2 * x^2$ 
proof -
  assume  $-(1 / 2) \leq x$ 
  assume  $x \leq 0$ 
  have  $\text{abs}(\ln(1 + x) - x) = x - \ln(1 - (-x))$ 
    apply (subst abs-of-nonpos)
    apply simp
    apply (rule ln-add-one-self-le-self2)
    apply (insert prems, auto)
  done
  also have  $\dots \leq 2 * x^2$ 
    apply (subgoal-tac  $-(-x) - 2 * (-x)^2 \leq \ln(1 - (-x))$ )
    apply (simp add: algebra-simps)

```

```

    apply (rule ln-one-minus-pos-lower-bound)
    apply (insert prems, auto)
  done
  finally show ?thesis .
qed

```

```

lemma abs-ln-one-plus-x-minus-x-bound:
  abs x <= 1 / 2 ==> abs(ln (1 + x) - x) <= 2 * x^2
  apply (case-tac 0 <= x)
  apply (rule order-trans)
  apply (rule abs-ln-one-plus-x-minus-x-bound-nonneg)
  apply auto
  apply (rule abs-ln-one-plus-x-minus-x-bound-nonpos)
  apply auto
done

```

```

lemma ln-x-over-x-mono: exp 1 <= x ==> x <= y ==> (ln y / y) <= (ln x /
x)
proof -
  assume exp 1 <= x and x <= y
  have a: 0 < x and b: 0 < y
    apply (insert prems)
    apply (subgoal-tac 0 < exp (1::real))
    apply arith
    apply auto
    apply (subgoal-tac 0 < exp (1::real))
    apply arith
    apply auto
  done
  have x * ln y - x * ln x = x * (ln y - ln x)
    by (simp add: algebra-simps)
  also have ... = x * ln(y / x)
    apply (subst ln-div)
    apply (rule b, rule a, rule refl)
  done
  also have y / x = (x + (y - x)) / x
    by simp
  also have ... = 1 + (y - x) / x using a prems by (simp add: field-simps)
  also have x * ln(1 + (y - x) / x) <= x * ((y - x) / x)
    apply (rule mult-left-mono)
    apply (rule ln-add-one-self-le-self)
    apply (rule divide-nonneg-pos)
    apply (insert prems a, simp-all)
  done
  also have ... = y - x using a by simp
  also have ... = (y - x) * ln (exp 1) by simp
  also have ... <= (y - x) * ln x
    apply (rule mult-left-mono)
    apply (subst ln-le-cancel-iff)

```

```

    apply force
    apply (rule a)
    apply (rule prems)
    apply (insert prems, simp)
    done
  also have ... = y * ln x - x * ln x
    by (rule left-diff-distrib)
  finally have x * ln y <= y * ln x
    by arith
  then have ln y <= (y * ln x) / x using a by (simp add: field_simps)
  also have ... = y * (ln x / x) by simp
  finally show ?thesis using b by (simp add: field_simps)
qed

end

```

## 80 MacLaurin: MacLaurin Series

```

theory MacLaurin
imports Transcendental
begin

```

### 80.1 Maclaurin’s Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it’s been broken down into lemmas.

**lemma** *Maclaurin-lemma*:

```

  0 < h ==>
    ∃ B. f h = (∑ m=0..<n. (j m / real (fact m)) * (h ^ m)) +
              (B * ((h ^ n) / real (fact n)))
  apply (rule-tac x = (f h - (∑ m=0..<n. (j m / real (fact m)) * h ^ m)) *
    real (fact n) / (h ^ n)
    in exI)
  apply (simp)
  done

```

**lemma** *eq-diff-eq'*:  $(x = y - z) = (y = x + (z::real))$   
**by** *arith*

**lemma** *fact-diff-Suc* [rule-format]:

```

  n < Suc m ==> fact (Suc m - n) = (Suc m - n) * fact (m - n)
  by (subst fact-reduce-nat, auto)

```

**lemma** *Maclaurin-lemma2*:

```

  assumes diff: ∀ m t. m < n ∧ 0 ≤ t ∧ t ≤ h ⟶ DERIV (diff m) t :> diff (Suc m) t

```

```

assumes  $n: n = \text{Suc } k$ 
assumes  $\text{difg}: \text{difg} =$ 
   $(\lambda m \ t. \text{diff } m \ t -$ 
     $((\sum p = 0..<n - m. \text{diff } (m + p) \ 0 / \text{real } (\text{fact } p) * t ^ p) +$ 
     $B * (t ^ (n - m) / \text{real } (\text{fact } (n - m))))))$ 
shows
   $\forall m \ t. m < n \ \& \ 0 \leq t \ \& \ t \leq h \longrightarrow \text{DERIV } (\text{difg } m) \ t :> \text{difg } (\text{Suc } m) \ t$ 
unfolding  $\text{difg}$ 
apply  $\text{clarify}$ 
apply  $(\text{rule DERIV-diff})$ 
apply  $(\text{simp add: diff})$ 
apply  $(\text{simp only: } n)$ 
apply  $(\text{rule DERIV-add})$ 
apply  $(\text{rule-tac } [2] \text{ DERIV-cmult})$ 
apply  $(\text{rule-tac } [2] \text{ lemma-DERIV-subst})$ 
apply  $(\text{rule-tac } [2] \text{ DERIV-quotient})$ 
  apply  $(\text{rule-tac } [3] \text{ DERIV-const})$ 
  apply  $(\text{rule-tac } [2] \text{ DERIV-pow})$ 
prefer 3

apply  $(\text{simp add: fact-diff-Suc})$ 
prefer 2 apply  $\text{simp}$ 
apply  $(\text{frule less-iff-Suc-add } [THEN \text{iffD1}], \text{clarify})$ 
apply  $(\text{simp del: setsum-op-ivl-Suc})$ 
apply  $(\text{insert sumr-offset4 } [of \ \text{Suc } 0])$ 
apply  $(\text{simp del: setsum-op-ivl-Suc fact-Suc power-Suc})$ 
apply  $(\text{rule lemma-DERIV-subst})$ 
apply  $(\text{rule DERIV-add})$ 
  apply  $(\text{rule-tac } [2] \text{ DERIV-const})$ 
apply  $(\text{rule DERIV-sumr, clarify})$ 
prefer 2 apply  $\text{simp}$ 
apply  $(\text{simp (no-asm) add: divide-inverse mult-assoc del: fact-Suc power-Suc})$ 
apply  $(\text{rule DERIV-cmult})$ 
apply  $(\text{rule lemma-DERIV-subst})$ 
  apply  $(\text{best intro!: DERIV-intros})$ 
apply  $(\text{subst fact-Suc})$ 
apply  $(\text{subst real-of-nat-mult})$ 
apply  $(\text{simp add: mult-ac})$ 
done

lemma  $\text{MacLaurin}$ :
assumes  $h: 0 < h$ 
assumes  $n: 0 < n$ 
assumes  $\text{diff-0}: \text{diff } 0 = f$ 
assumes  $\text{diff-Suc}$ :
   $\forall m \ t. m < n \ \& \ 0 \leq t \ \& \ t \leq h \longrightarrow \text{DERIV } (\text{diff } m) \ t :> \text{diff } (\text{Suc } m) \ t$ 
shows
   $\exists t. 0 < t \ \& \ t < h \ \&$ 
     $f \ h =$ 

```

$$\text{setsum } (\%m. (\text{diff } m \ 0 \ / \ \text{real } (\text{fact } m)) * h \wedge m) \{0..<n\} + \\ (\text{diff } n \ t \ / \ \text{real } (\text{fact } n)) * h \wedge n$$

**proof** –

**from**  $n$  **obtain**  $m$  **where**  $m: n = \text{Suc } m$   
**by**  $(\text{cases } n, \text{ simp add: } n)$

**obtain**  $B$  **where**  $f\text{-}h: f \ h =$   
 $(\sum m = 0..<n. \text{diff } m \ (0::\text{real}) \ / \ \text{real } (\text{fact } m) * h \wedge m) +$   
 $B * (h \wedge n \ / \ \text{real } (\text{fact } n))$   
**using**  $\text{Maclaurin-lemma } [OF \ h] \ ..$

**obtain**  $g$  **where**  $g\text{-def}: g = (\%t. f \ t -$   
 $(\text{setsum } (\%m. (\text{diff } m \ 0 \ / \ \text{real } (\text{fact } m)) * t \wedge m) \{0..<n\}$   
 $+ (B * (t \wedge n \ / \ \text{real } (\text{fact } n)))) \text{ by } \text{blast}$

**have**  $g2: g \ 0 = 0 \ \& \ g \ h = 0$   
**apply**  $(\text{simp add: } m \ f\text{-}h \ g\text{-def del: setsum-op-ivl-Suc)$   
**apply**  $(\text{cut-tac } n = m \ \text{and } k = \text{Suc } 0 \ \text{in } \text{sumr-offset2})$   
**apply**  $(\text{simp add: eq-diff-eq' diff-0 del: setsum-op-ivl-Suc})$   
**done**

**obtain**  $\text{difg}$  **where**  $\text{difg-def}: \text{difg} = (\%m \ t. \text{diff } m \ t -$   
 $(\text{setsum } (\%p. (\text{diff } (m + p) \ 0 \ / \ \text{real } (\text{fact } p)) * (t \wedge p)) \{0..<n-m\}$   
 $+ (B * ((t \wedge (n - m)) \ / \ \text{real } (\text{fact } (n - m))))) \text{ by } \text{blast}$

**have**  $\text{difg-0}: \text{difg } 0 = g$   
**unfolding**  $\text{difg-def } g\text{-def}$  **by**  $(\text{simp add: difg-0})$

**have**  $\text{difg-Suc}: \forall (m::\text{nat}) \ t::\text{real}.$   
 $m < n \wedge (0::\text{real}) \leq t \wedge t \leq h \longrightarrow \text{DERIV } (\text{difg } m) \ t :> \text{difg } (\text{Suc } m) \ t$   
**using**  $\text{difg-Suc } m \ \text{difg-def}$  **by**  $(\text{rule } \text{Maclaurin-lemma2})$

**have**  $\text{difg-eq-0}: \forall m. m < n \longrightarrow \text{difg } m \ 0 = 0$   
**apply**  $\text{clarify}$   
**apply**  $(\text{simp add: } m \ \text{difg-def})$   
**apply**  $(\text{frule less-iff-Suc-add } [THEN \ \text{iffD1}], \text{ clarify})$   
**apply**  $(\text{simp del: setsum-op-ivl-Suc})$   
**apply**  $(\text{insert sumr-offset4 } [of \ \text{Suc } 0])$   
**apply**  $(\text{simp del: setsum-op-ivl-Suc fact-Suc})$   
**done**

**have**  $\text{isCont-difg}: \bigwedge m \ x. \llbracket m < n; 0 \leq x; x \leq h \rrbracket \Longrightarrow \text{isCont } (\text{difg } m) \ x$   
**by**  $(\text{rule } \text{DERIV-isCont } [OF \ \text{difg-Suc } [rule-format]]) \text{ simp}$

**have**  $\text{differentiable-difg}: \bigwedge m \ x. \llbracket m < n; 0 \leq x; x \leq h \rrbracket \Longrightarrow \text{difg } m \ \text{differentiable } x$   
**by**  $(\text{rule } \text{differentiableI } [OF \ \text{difg-Suc } [rule-format]]) \text{ simp}$

**have**  $\text{difg-Suc-eq-0}: \bigwedge m \ t. \llbracket m < n; 0 \leq t; t \leq h; \text{DERIV } (\text{difg } m) \ t :> 0 \rrbracket$

$\implies \text{difg } (\text{Suc } m) \ t = 0$   
**by** (rule DERIV-unique [OF difg-Suc [rule-format]]) simp

**have**  $m < n$  **using**  $m$  **by** simp

**have**  $\exists t. 0 < t \wedge t < h \wedge \text{DERIV } (\text{difg } m) \ t :> 0$

**using**  $\langle m < n \rangle$

**proof** (induct  $m$ )

**case** 0

**show** ?case

**proof** (rule Rolle)

**show**  $0 < h$  **by** fact

**show**  $\text{difg } 0 \ 0 = \text{difg } 0 \ h$  **by** (simp add: difg-0 g2)

**show**  $\forall x. 0 \leq x \wedge x \leq h \longrightarrow \text{isCont } (\text{difg } (0::\text{nat})) \ x$

**by** (simp add: isCont-difg n)

**show**  $\forall x. 0 < x \wedge x < h \longrightarrow \text{difg } (0::\text{nat}) \text{ differentiable } x$

**by** (simp add: differentiable-difg n)

**qed**

**next**

**case** (Suc  $m'$ )

**hence**  $\exists t. 0 < t \wedge t < h \wedge \text{DERIV } (\text{difg } m') \ t :> 0$  **by** simp

**then obtain**  $t$  **where**  $t: 0 < t < h \text{ DERIV } (\text{difg } m') \ t :> 0$  **by** fast

**have**  $\exists t'. 0 < t' \wedge t' < t \wedge \text{DERIV } (\text{difg } (\text{Suc } m')) \ t' :> 0$

**proof** (rule Rolle)

**show**  $0 < t$  **by** fact

**show**  $\text{difg } (\text{Suc } m') \ 0 = \text{difg } (\text{Suc } m') \ t$

**using**  $t \langle \text{Suc } m' < n \rangle$  **by** (simp add: difg-Suc-eq-0 difg-eq-0)

**show**  $\forall x. 0 \leq x \wedge x \leq t \longrightarrow \text{isCont } (\text{difg } (\text{Suc } m')) \ x$

**using**  $\langle t < h \rangle \langle \text{Suc } m' < n \rangle$  **by** (simp add: isCont-difg)

**show**  $\forall x. 0 < x \wedge x < t \longrightarrow \text{difg } (\text{Suc } m') \text{ differentiable } x$

**using**  $\langle t < h \rangle \langle \text{Suc } m' < n \rangle$  **by** (simp add: differentiable-difg)

**qed**

**thus** ?case

**using**  $\langle t < h \rangle$  **by** auto

**qed**

**then obtain**  $t$  **where**  $0 < t < h \text{ DERIV } (\text{difg } m) \ t :> 0$  **by** fast

**hence**  $\text{difg } (\text{Suc } m) \ t = 0$

**using**  $\langle m < n \rangle$  **by** (simp add: difg-Suc-eq-0)

**show** ?thesis

**proof** (intro exI conjI)

**show**  $0 < t$  **by** fact

**show**  $t < h$  **by** fact

**show**  $f \ h =$

$(\sum m = 0..<n. \text{diff } m \ 0 / \text{real } (\text{fact } m) * h ^ m) +$

$\text{diff } n \ t / \text{real } (\text{fact } n) * h ^ n$

**using**  $\langle \text{difg } (\text{Suc } m) \ t = 0 \rangle$

by (simp add: m f-h difg-def del: fact-Suc)  
qed

qed

**lemma** *Maclaurin-objl*:

$0 < h \ \& \ n > 0 \ \& \ \text{diff } 0 = f \ \&$   
 $(\forall m \ t. \ m < n \ \& \ 0 \leq t \ \& \ t \leq h \ \longrightarrow \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff } (\text{Suc } m) \ t)$   
 $\longrightarrow (\exists t. \ 0 < t \ \& \ t < h \ \&$   
 $f \ h = (\sum m=0..<n. \ \text{diff } m \ 0 / \text{real } (\text{fact } m) * h \wedge m) +$   
 $\text{diff } n \ t / \text{real } (\text{fact } n) * h \wedge n)$

by (blast intro: Maclaurin)

**lemma** *Maclaurin2*:

$[\![ \ 0 < h; \ \text{diff } 0 = f; \]$   
 $\forall m \ t.$   
 $m < n \ \& \ 0 \leq t \ \& \ t \leq h \ \longrightarrow \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff } (\text{Suc } m) \ t \ ]]$   
 $\implies \exists t. \ 0 < t \ \&$   
 $t \leq h \ \&$   
 $f \ h =$   
 $(\sum m=0..<n. \ \text{diff } m \ 0 / \text{real } (\text{fact } m) * h \wedge m) +$   
 $\text{diff } n \ t / \text{real } (\text{fact } n) * h \wedge n$

apply (case-tac n, auto)

apply (drule Maclaurin, auto)

done

**lemma** *Maclaurin2-objl*:

$0 < h \ \& \ \text{diff } 0 = f \ \&$   
 $(\forall m \ t.$   
 $m < n \ \& \ 0 \leq t \ \& \ t \leq h \ \longrightarrow \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff } (\text{Suc } m) \ t)$   
 $\longrightarrow (\exists t. \ 0 < t \ \&$   
 $t \leq h \ \&$   
 $f \ h =$   
 $(\sum m=0..<n. \ \text{diff } m \ 0 / \text{real } (\text{fact } m) * h \wedge m) +$   
 $\text{diff } n \ t / \text{real } (\text{fact } n) * h \wedge n)$

by (blast intro: Maclaurin2)

**lemma** *Maclaurin-minus*:

$[\![ \ h < 0; \ n > 0; \ \text{diff } 0 = f; \]$   
 $\forall m \ t. \ m < n \ \& \ h \leq t \ \& \ t \leq 0 \ \longrightarrow \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff } (\text{Suc } m) \ t \ ]]$   
 $\implies \exists t. \ h < t \ \&$   
 $t < 0 \ \&$   
 $f \ h =$   
 $(\sum m=0..<n. \ \text{diff } m \ 0 / \text{real } (\text{fact } m) * h \wedge m) +$   
 $\text{diff } n \ t / \text{real } (\text{fact } n) * h \wedge n$

apply (cut-tac f = %x. f (-x))

and diff = %n x. (-1 ^ n) \* diff n (-x)

and h = -h and n = n in Maclaurin-objl)



```

apply (simp)
apply safe
apply (subst minus-mult-right)
apply (rule DERIV-cmult)
apply (rule lemma-DERIV-subst)
apply (rule DERIV-chain2 [where g=uminus])
apply (rule-tac [2] DERIV-minus, rule-tac [2] DERIV-ident)
prefer 2 apply force
apply force
apply (rule-tac x = -t in exI, auto)
apply (subgoal-tac ( $\sum m = 0..<n. -1 \wedge m * \text{diff } m \ 0 * (-h) \wedge m / \text{real}(\text{fact } m)$ )
=
 $(\sum m = 0..<n. \text{diff } m \ 0 * h \wedge m / \text{real}(\text{fact } m))$ )
apply (rule-tac [2] setsum-cong[OF refl])
apply (auto simp add: divide-inverse power-mult-distrib [symmetric])
done

```

**lemma** *Maclaurin-minus-objl*:

$$\begin{aligned}
& (h < 0 \ \& \ n > 0 \ \& \ \text{diff } 0 = f \ \& \\
& (\forall m \ t. \\
& \quad m < n \ \& \ h \leq t \ \& \ t \leq 0 \ \longrightarrow \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff } (\text{Suc } m) \ t)) \\
& \longrightarrow (\exists t. h < t \ \& \\
& \quad t < 0 \ \& \\
& \quad f \ h = \\
& \quad (\sum m=0..<n. \text{diff } m \ 0 / \text{real } (\text{fact } m) * h \wedge m) + \\
& \quad \text{diff } n \ t / \text{real } (\text{fact } n) * h \wedge n)
\end{aligned}$$

**by** (*blast intro: Maclaurin-minus*)

## 80.2 More Convenient “Bidirectional” Version.

**lemma** *Maclaurin-bi-le-lemma* [*rule-format*]:

$$\begin{aligned}
& n > 0 \longrightarrow \\
& \text{diff } 0 \ 0 = \\
& (\sum m = 0..<n. \text{diff } m \ 0 * 0 \wedge m / \text{real } (\text{fact } m)) + \\
& \text{diff } n \ 0 * 0 \wedge n / \text{real } (\text{fact } n)
\end{aligned}$$

**by** (*induct n, auto*)

**lemma** *Maclaurin-bi-le*:

$$\begin{aligned}
& [] \ \text{diff } 0 = f; \\
& \quad \forall m \ t. m < n \ \& \ \text{abs } t \leq \text{abs } x \ \longrightarrow \text{DERIV } (\text{diff } m) \ t \text{ :> } \text{diff } (\text{Suc } m) \ t \ [] \\
& \implies \exists t. \text{abs } t \leq \text{abs } x \ \& \\
& \quad f \ x = \\
& \quad (\sum m=0..<n. \text{diff } m \ 0 / \text{real } (\text{fact } m) * x \wedge m) + \\
& \quad \text{diff } n \ t / \text{real } (\text{fact } n) * x \wedge n
\end{aligned}$$

**apply** (*case-tac* *n = 0, force*)

**apply** (*case-tac* *x = 0*)

**apply** (*rule-tac* *x = 0 in exI*)

**apply** (*force simp add: Maclaurin-bi-le-lemma*)

**apply** (*cut-tac* *x = x and y = 0 in linorder-less-linear, auto*)

Case 1, where  $x < 0$

```

apply (cut-tac  $f = \text{diff } 0$  and  $\text{diff} = \text{diff}$  and  $h = x$  and  $n = n$  in Maclaurin-minus-objl,
safe)
  apply (simp add: abs-if)
  apply (rule-tac  $x = t$  in exI)
  apply (simp add: abs-if)

```

Case 2, where  $0 < x$

```

apply (cut-tac  $f = \text{diff } 0$  and  $\text{diff} = \text{diff}$  and  $h = x$  and  $n = n$  in Maclaurin-objl,
safe)
  apply (simp add: abs-if)
  apply (rule-tac  $x = t$  in exI)
  apply (simp add: abs-if)
done

```

**lemma** *Maclaurin-all-lt*:

```

  [|  $\text{diff } 0 = f$ ;
     $\forall m x. \text{DERIV } (\text{diff } m) x :> \text{diff } (\text{Suc } m) x$ ;
     $x \sim 0$ ;  $n > 0$ 
  |] ==>  $\exists t. 0 < \text{abs } t \ \& \ \text{abs } t < \text{abs } x \ \&$ 
         $f x = (\sum m=0..<n. (\text{diff } m \ 0 / \text{real } (\text{fact } m)) * x ^ m) +$ 
         $(\text{diff } n \ t / \text{real } (\text{fact } n)) * x ^ n$ 
apply (rule-tac  $x = x$  and  $y = 0$  in linorder-cases)
prefer 2 apply blast
apply (drule-tac [2]  $\text{diff} = \text{diff}$  in Maclaurin)
apply (drule-tac  $\text{diff} = \text{diff}$  in Maclaurin-minus, simp-all, safe)
apply (rule-tac [!]  $x = t$  in exI, auto)
done

```

**lemma** *Maclaurin-all-lt-objl*:

```

   $\text{diff } 0 = f \ \&$ 
   $(\forall m x. \text{DERIV } (\text{diff } m) x :> \text{diff } (\text{Suc } m) x) \ \&$ 
   $x \sim 0 \ \& \ n > 0$ 
  -->  $(\exists t. 0 < \text{abs } t \ \& \ \text{abs } t < \text{abs } x \ \&$ 
         $f x = (\sum m=0..<n. (\text{diff } m \ 0 / \text{real } (\text{fact } m)) * x ^ m) +$ 
         $(\text{diff } n \ t / \text{real } (\text{fact } n)) * x ^ n)$ 
by (blast intro: Maclaurin-all-lt)

```

**lemma** *Maclaurin-zero* [*rule-format*]:

```

   $x = (0::\text{real})$ 
  ==>  $n \neq 0$  -->
     $(\sum m=0..<n. (\text{diff } m \ (0::\text{real}) / \text{real } (\text{fact } m)) * x ^ m) =$ 
     $\text{diff } 0 \ 0$ 
by (induct  $n$ , auto)

```

**lemma** *Maclaurin-all-le*: [|  $\text{diff } 0 = f$ ;

```

   $\forall m x. \text{DERIV } (\text{diff } m) x :> \text{diff } (\text{Suc } m) x$ 
  |] ==>  $\exists t. \text{abs } t \leq \text{abs } x \ \&$ 
         $f x = (\sum m=0..<n. (\text{diff } m \ 0 / \text{real } (\text{fact } m)) * x ^ m) +$ 

```

```

      (diff n t / real (fact n)) * x ^ n
apply (cases n=0)
apply (force)
apply (case-tac x = 0)
apply (frule-tac diff = diff and n = n in Maclaurin-zero, assumption)
apply (drule not0-implies-Suc)
apply (rule-tac x = 0 in exI, force)
apply (frule-tac diff = diff and n = n in Maclaurin-all-lt, auto)
apply (rule-tac x = t in exI, auto)
done

```

**lemma** *Maclaurin-all-le-objl*:  $\text{diff } 0 = f \ \&$   
 $(\forall m \ x. \text{DERIV } (\text{diff } m) \ x :> \text{diff } (\text{Suc } m) \ x)$   
 $--> (\exists t. \text{abs } t \leq \text{abs } x \ \&$   
 $f \ x = (\sum m=0..<n. (\text{diff } m \ 0 / \text{real } (\text{fact } m)) * x ^ m) +$   
 $(\text{diff } n \ t / \text{real } (\text{fact } n)) * x ^ n)$   
**by** (*blast intro: Maclaurin-all-le*)

### 80.3 Version for Exponential Function

**lemma** *Maclaurin-exp-lt*:  $[\![ \ x \sim 0; \ n > 0 \ ]\!]$   
 $==> (\exists t. \ 0 < \text{abs } t \ \&$   
 $\text{abs } t < \text{abs } x \ \&$   
 $\text{exp } x = (\sum m=0..<n. (x ^ m) / \text{real } (\text{fact } m)) +$   
 $(\text{exp } t / \text{real } (\text{fact } n)) * x ^ n)$   
**by** (*cut-tac diff = %n. exp and f = exp and x = x and n = n in Maclaurin-all-lt-objl, auto*)

**lemma** *Maclaurin-exp-le*:  
 $\exists t. \text{abs } t \leq \text{abs } x \ \&$   
 $\text{exp } x = (\sum m=0..<n. (x ^ m) / \text{real } (\text{fact } m)) +$   
 $(\text{exp } t / \text{real } (\text{fact } n)) * x ^ n$   
**by** (*cut-tac diff = %n. exp and f = exp and x = x and n = n in Maclaurin-all-le-objl, auto*)

### 80.4 Version for Sine Function

**lemma** *mod-exhaust-less-4*:  
 $m \bmod 4 = 0 \mid m \bmod 4 = 1 \mid m \bmod 4 = 2 \mid m \bmod 4 = (3::\text{nat})$   
**by** *auto*

**lemma** *Suc-Suc-mult-two-diff-two* [*rule-format, simp*]:  
 $n \neq 0 --> \text{Suc } (\text{Suc } (2 * n - 2)) = 2 * n$   
**by** (*induct n, auto*)

**lemma** *lemma-Suc-Suc-4n-diff-2* [*rule-format, simp*]:  
 $n \neq 0 --> \text{Suc } (\text{Suc } (4 * n - 2)) = 4 * n$   
**by** (*induct n, auto*)

**lemma** *Suc-mult-two-diff-one* [rule-format, simp]:  
 $n \neq 0 \longrightarrow \text{Suc } (2 * n - 1) = 2 * n$   
**by** (induct n, auto)

It is unclear why so many variant results are needed.

**lemma** *sin-expansion-lemma*:  
 $\sin (x + \text{real } (\text{Suc } m) * \pi / 2) =$   
 $\cos (x + \text{real } (m) * \pi / 2)$   
**by** (simp only: cos-add sin-add real-of-nat-Suc add-divide-distrib left-distrib, auto)

**lemma** *Maclaurin-sin-expansion2*:  
 $\exists t. \text{abs } t \leq \text{abs } x \ \&$   
 $\sin x =$   
 $(\sum m=0..<n. (\text{if even } m \text{ then } 0$   
 $\text{else } (-1 \wedge ((m - \text{Suc } 0) \text{ div } 2)) / \text{real } (\text{fact } m)) *$   
 $x \wedge m)$   
 $+ ((\sin(t + 1/2 * \text{real } (n) * \pi) / \text{real } (\text{fact } n)) * x \wedge n)$   
**apply** (cut-tac f = sin **and** n = n **and** x = x  
**and** diff = %n x. sin (x + 1/2\*real n \* pi) **in** Maclaurin-all-lt-objl)  
**apply** safe  
**apply** (simp (no-asm))  
**apply** (simp (no-asm) add: sin-expansion-lemma)  
**apply** (case-tac n, clarify, simp, simp add: lemma-STAR-sin)  
**apply** (rule ccontr, simp)  
**apply** (drule-tac x = x **in** spec, simp)  
**apply** (erule ssubst)  
**apply** (rule-tac x = t **in** exI, simp)  
**apply** (rule setsum-cong[OF refl])  
**apply** (auto simp add: sin-zero-iff odd-Suc-mult-two-ex)  
**done**

**lemma** *Maclaurin-sin-expansion*:  
 $\exists t. \sin x =$   
 $(\sum m=0..<n. (\text{if even } m \text{ then } 0$   
 $\text{else } (-1 \wedge ((m - \text{Suc } 0) \text{ div } 2)) / \text{real } (\text{fact } m)) *$   
 $x \wedge m)$   
 $+ ((\sin(t + 1/2 * \text{real } (n) * \pi) / \text{real } (\text{fact } n)) * x \wedge n)$   
**apply** (insert Maclaurin-sin-expansion2 [of x n])  
**apply** (blast intro: elim:)  
**done**

**lemma** *Maclaurin-sin-expansion3*:  
 $[| n > 0; 0 < x |] \implies$   
 $\exists t. 0 < t \ \& \ t < x \ \&$   
 $\sin x =$   
 $(\sum m=0..<n. (\text{if even } m \text{ then } 0$   
 $\text{else } (-1 \wedge ((m - \text{Suc } 0) \text{ div } 2)) / \text{real } (\text{fact } m)) *$   
 $x \wedge m)$   
 $+ ((\sin(t + 1/2 * \text{real } (n) * \pi) / \text{real } (\text{fact } n)) * x \wedge n)$

```

apply (cut-tac f = sin and n = n and h = x and diff = %n x. sin (x + 1/2*real
(n) *pi) in Maclaurin-objl)
apply safe
apply simp
apply (simp (no-asm) add: sin-expansion-lemma)
apply (erule ssubst)
apply (rule-tac x = t in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: sin-zero-iff odd-Suc-mult-two-ex)
done

```

**lemma** Maclaurin-sin-expansion4:

```

0 < x ==>
  ∃ t. 0 < t & t ≤ x &
    sin x =
      (∑ m=0.. $n$ . (if even m then 0
        else  $-1^{((m - \text{Suc } 0) \text{ div } 2)}$ ) / real (fact m)) *
        x ^ m)
    + ((sin(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)
apply (cut-tac f = sin and n = n and h = x and diff = %n x. sin (x + 1/2*real
(n) *pi) in Maclaurin2-objl)
apply safe
apply simp
apply (simp (no-asm) add: sin-expansion-lemma)
apply (erule ssubst)
apply (rule-tac x = t in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: sin-zero-iff odd-Suc-mult-two-ex)
done

```

## 80.5 Maclaurin Expansion for Cosine Function

**lemma** sumr-cos-zero-one [simp]:

```

(∑ m=0.. $(\text{Suc } n)$ .
  (if even m then  $-1^{(m \text{ div } 2)}$  / (real (fact m)) else 0) * 0 ^ m) = 1
by (induct n, auto)

```

**lemma** cos-expansion-lemma:

```

cos (x + real(Suc m) * pi / 2) = -sin (x + real m * pi / 2)
by (simp only: cos-add sin-add real-of-nat-Suc left-distrib add-divide-distrib, auto)

```

**lemma** Maclaurin-cos-expansion:

```

∃ t. abs t ≤ abs x &
  cos x =
    (∑ m=0.. $n$ . (if even m
      then  $-1^{(m \text{ div } 2)}$  / (real (fact m))
      else 0) *
      x ^ m)
    + ((cos(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)

```

```

apply (cut-tac f = cos and n = n and x = x and diff = %n x. cos (x + 1/2*real
(n) *pi) in Maclaurin-all-lt-objl)
apply safe
apply (simp (no-asm))
apply (simp (no-asm) add: cos-expansion-lemma)
apply (case-tac n, simp)
apply (simp del: setsum-op-ivl-Suc)
apply (rule ccontr, simp)
apply (drule-tac x = x in spec, simp)
apply (erule ssubst)
apply (rule-tac x = t in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: cos-zero-iff even-mult-two-ex)
done

```

**lemma** Maclaurin-cos-expansion2:

```

[[ 0 < x; n > 0 ]] ==>
  ∃ t. 0 < t & t < x &
    cos x =
      (∑ m=0.. $n$ . (if even m
        then  $-1^{(m \text{ div } 2)}$  / (real (fact m))
        else 0) *
        x ^ m)
      + ((cos(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)
apply (cut-tac f = cos and n = n and h = x and diff = %n x. cos (x + 1/2*real
(n) *pi) in Maclaurin-objl)
apply safe
apply simp
apply (simp (no-asm) add: cos-expansion-lemma)
apply (erule ssubst)
apply (rule-tac x = t in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: cos-zero-iff even-mult-two-ex)
done

```

**lemma** Maclaurin-minus-cos-expansion:

```

[[ x < 0; n > 0 ]] ==>
  ∃ t. x < t & t < 0 &
    cos x =
      (∑ m=0.. $n$ . (if even m
        then  $-1^{(m \text{ div } 2)}$  / (real (fact m))
        else 0) *
        x ^ m)
      + ((cos(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)
apply (cut-tac f = cos and n = n and h = x and diff = %n x. cos (x + 1/2*real
(n) *pi) in Maclaurin-minus-objl)
apply safe
apply simp
apply (simp (no-asm) add: cos-expansion-lemma)

```

```

apply (erule ssubst)
apply (rule-tac  $x = t$  in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: cos-zero-iff even-mult-two-ex)
done

```

**lemma** sin-bound-lemma:

```

  [| $x = y$ ;  $\text{abs } u \leq (v::\text{real})$ |] ==>  $|(x + u) - y| \leq v$ 
by auto

```

**lemma** Maclaurin-sin-bound:

```

  abs(sin  $x$  - ( $\sum m=0..<n$ . (if even  $m$  then 0 else  $(-1)^{(m - \text{Suc } 0) \text{ div } 2}$ )) /
  real (fact  $m$ )) *
   $x^m$ )  $\leq$  inverse(real (fact  $n$ )) *  $|x|^n$ 
proof -
  have !!  $x$  ( $y::\text{real}$ ).  $x \leq 1 \implies 0 \leq y \implies x * y \leq 1 * y$ 
    by (rule-tac mult-right-mono, simp-all)
  note est = this[simplified]
  let ?diff =  $\lambda(n::\text{nat}) x$ . if  $n \bmod 4 = 0$  then sin( $x$ ) else if  $n \bmod 4 = 1$  then
  cos( $x$ ) else if  $n \bmod 4 = 2$  then  $-\text{sin}(x)$  else  $-\text{cos}(x)$ 
  have diff-0: ?diff 0 = sin by simp
  have DERIV-diff:  $\forall m x$ . DERIV (?diff  $m$ )  $x$  :> ?diff (Suc  $m$ )  $x$ 
    apply (clarify)
    apply (subst (1 2 3) mod-Suc-eq-Suc-mod)
    apply (cut-tac  $m=m$  in mod-exhaust-less-4)
    apply (safe, auto intro!: DERIV-intros)
  done
from Maclaurin-all-le [OF diff-0 DERIV-diff]
obtain  $t$  where  $t1: |t| \leq |x|$  and
   $t2: \text{sin } x = (\sum m = 0..<n. ?diff m 0 / \text{real (fact } m) * x^m) +$ 
  ?diff  $n$   $t$  / real (fact  $n$ ) *  $x^n$  by fast
have diff-m-0:
   $\bigwedge m. ?diff m 0 = (\text{if even } m \text{ then } 0$ 
     $\text{else } (-1)^{(m - \text{Suc } 0) \text{ div } 2})$ 
    apply (subst even-even-mod-4-iff)
    apply (cut-tac  $m=m$  in mod-exhaust-less-4)
    apply (elim disjE, simp-all)
    apply (safe dest!: mod-eqD, simp-all)
  done
show ?thesis
  apply (subst  $t2$ )
  apply (rule sin-bound-lemma)
  apply (rule setsum-cong[OF refl])
  apply (subst diff-m-0, simp)
  apply (auto intro: mult-right-mono [where  $b=1$ , simplified] mult-right-mono

```

```

      simp add: est mult-nonneg-nonneg mult-ac divide-inverse
              power-abs [symmetric] abs-mult)
    done
  qed
end

```

## 81 Taylor: Taylor series

```

theory Taylor
imports MacLaurin
begin

```

We use MacLaurin and the translation of the expansion point  $c$  to  $0$  to prove Taylor’s theorem.

**lemma** *taylor-up*:

```

  assumes INIT:  $n > 0$  diff  $0 = f$ 
  and DERIV:  $(\forall m t. m < n \ \& \ a \leq t \ \& \ t \leq b \longrightarrow \text{DERIV } (\text{diff } m) t :> (\text{diff } (\text{Suc } m) t))$ 
  and INTERV:  $a \leq c \ c < b$ 
  shows  $\exists t. c < t \ \& \ t < b \ \& \$ 

$$f b = \text{setsum } (\%m. (\text{diff } m \ c / \text{real } (\text{fact } m)) * (b - c) ^ m) \ \{0..<n\} +$$


$$(\text{diff } n \ t / \text{real } (\text{fact } n)) * (b - c) ^ n$$

  proof -
    from INTERV have  $0 < b - c$  by arith
    moreover
    from INIT have  $n > 0 \ ((\lambda m x. \text{diff } m (x + c)) \ 0) = (\lambda x. f (x + c))$  by auto
    moreover
    have ALL  $m t. m < n \ \& \ 0 \leq t \ \& \ t \leq b - c \longrightarrow \text{DERIV } (\%x. \text{diff } m (x + c)) \ t :> \text{diff } (\text{Suc } m) (t + c)$ 
    proof (intro strip)
      fix  $m t$ 
      assume  $m < n \ \& \ 0 \leq t \ \& \ t \leq b - c$ 
      with DERIV and INTERV have  $\text{DERIV } (\text{diff } m) (t + c) :> \text{diff } (\text{Suc } m) (t + c)$  by auto
      moreover
      from DERIV-ident and DERIV-const have  $\text{DERIV } (\%x. x + c) \ t :> 1 + 0$  by (rule DERIV-add)
      ultimately have  $\text{DERIV } (\%x. \text{diff } m (x + c)) \ t :> \text{diff } (\text{Suc } m) (t + c) * (1 + 0)$ 
      by (rule DERIV-chain2)
      thus  $\text{DERIV } (\%x. \text{diff } m (x + c)) \ t :> \text{diff } (\text{Suc } m) (t + c)$  by simp
    qed
    ultimately
    have EX:  $\exists t > 0. t < b - c \ \& \$ 

$$f (b - c + c) = (\text{SUM } m = 0..<n. \text{diff } m (0 + c) / \text{real } (\text{fact } m) * (b - c) ^$$


$$m) +$$


$$\text{diff } n (t + c) / \text{real } (\text{fact } n) * (b - c) ^ n$$


```



by (rule Maclaurin)  
 show ?thesis  
 proof –  
 from EX obtain x where  
 $X: 0 < x \ \& \ x < b - c \ \& \ f(b - c + c) = (\sum m = 0..<n. \text{diff } m \ (0 + c) / \text{real } (\text{fact } m) * (b - c) ^ m) +$   
 $\text{diff } n \ (x + c) / \text{real } (\text{fact } n) * (b - c) ^ n ..$   
 let ?H = x + c  
 from X have c < ?H & ?H < b  $\wedge f b = (\sum m = 0..<n. \text{diff } m \ c / \text{real } (\text{fact } m) * (b - c) ^ m) +$   
 $\text{diff } n \ ?H / \text{real } (\text{fact } n) * (b - c) ^ n$   
 by fastsimp  
 thus ?thesis by fastsimp  
 qed  
 qed

lemma *taylor-down*:

assumes INIT:  $n > 0 \ \text{diff } 0 = f$   
 and DERIV:  $(\forall m \ t. \ m < n \ \& \ a \leq t \ \& \ t \leq b \longrightarrow \text{DERIV } (\text{diff } m) \ t :> (\text{diff } (\text{Suc } m) \ t))$   
 and INTERV:  $a < c \leq b$   
 shows  $\exists t. \ a < t \ \& \ t < c \ \& \ f a = \text{setsum } (\% m. (\text{diff } m \ c / \text{real } (\text{fact } m)) * (a - c) ^ m) \ \{0..<n\} +$   
 $(\text{diff } n \ t / \text{real } (\text{fact } n)) * (a - c) ^ n$   
 proof –  
 from INTERV have  $a - c < 0$  by arith  
 moreover  
 from INIT have  $n > 0 \ ((\lambda m \ x. \ \text{diff } m \ (x + c)) \ 0) = (\lambda x. \ f \ (x + c))$  by auto  
 moreover  
 have ALL  $m \ t. \ m < n \ \& \ a - c \leq t \ \& \ t \leq 0 \longrightarrow \text{DERIV } (\%x. \ \text{diff } m \ (x + c)) \ t :> \text{diff } (\text{Suc } m) \ (t + c)$   
 proof (rule allI impI)+  
 fix m t  
 assume  $m < n \ \& \ a - c \leq t \ \& \ t \leq 0$   
 with DERIV and INTERV have  $\text{DERIV } (\text{diff } m) \ (t + c) :> \text{diff } (\text{Suc } m) \ (t + c)$  by auto  
 moreover  
 from DERIV-ident and DERIV-const have  $\text{DERIV } (\%x. \ x + c) \ t :> 1 + 0$   
 by (rule DERIV-add)  
 ultimately have  $\text{DERIV } (\%x. \ \text{diff } m \ (x + c)) \ t :> \text{diff } (\text{Suc } m) \ (t + c) * (1 + 0)$  by (rule DERIV-chain2)  
 thus  $\text{DERIV } (\%x. \ \text{diff } m \ (x + c)) \ t :> \text{diff } (\text{Suc } m) \ (t + c)$  by simp  
 qed  
 ultimately  
 have EX:  $EX \ t > a - c. \ t < 0 \ \& \ f(a - c + c) = (\text{SUM } m = 0..<n. \ \text{diff } m \ (0 + c) / \text{real } (\text{fact } m) * (a - c) ^ m) +$   
 $\text{diff } n \ (t + c) / \text{real } (\text{fact } n) * (a - c) ^ n$

by (rule Maclaurin-minus)  
 show ?thesis  
 proof –  
 from EX obtain x where X:  $a - c < x \ \& \ x < 0 \ \&$   
 $f(a - c + c) = (SUM \ m = 0..<n. \ diff \ m \ (0 + c) / \ real \ (fact \ m) * (a - c)$   
 $^m) +$   
 $diff \ n \ (x + c) / \ real \ (fact \ n) * (a - c) ^ n ..$   
 let ?H =  $x + c$   
 from X have  $a < ?H \ \& \ ?H < c \wedge f \ a = (\sum \ m = 0..<n. \ diff \ m \ c / \ real \ (fact$   
 $m) * (a - c) ^ m) +$   
 $diff \ n \ ?H / \ real \ (fact \ n) * (a - c) ^ n$   
 by fastsimp  
 thus ?thesis by fastsimp  
 qed  
 qed

lemma taylor:

assumes INIT:  $n > 0 \ \diff \ 0 = f$   
 and DERIV:  $(\forall \ m \ t. \ m < n \ \& \ a \leq t \ \& \ t \leq b \longrightarrow DERIV \ (\diff \ m) \ t :> (\diff$   
 $(Suc \ m) \ t))$   
 and INTERV:  $a \leq c \ \ c \leq b \ a \leq x \ x \leq b \ x \neq c$   
 shows  $\exists \ t. \ (if \ x < c \ then \ (x < t \ \& \ t < c) \ else \ (c < t \ \& \ t < x)) \ \&$   
 $f \ x = setsum \ (\% \ m. \ (\diff \ m \ c / \ real \ (fact \ m)) * (x - c) ^ m) \ \{0..<n\} +$   
 $(\diff \ n \ t / \ real \ (fact \ n)) * (x - c) ^ n$   
 proof (cases  $x < c$ )  
 case True  
 note INIT  
 moreover from DERIV and INTERV  
 have  $\forall \ m \ t. \ m < n \wedge x \leq t \wedge t \leq b \longrightarrow DERIV \ (\diff \ m) \ t :> \diff \ (Suc \ m) \ t$   
 by fastsimp  
 moreover note True  
 moreover from INTERV have  $c \leq b$  by simp  
 ultimately have EX:  $\exists \ t > x. \ t < c \wedge f \ x =$   
 $(\sum \ m = 0..<n. \ diff \ m \ c / \ real \ (fact \ m) * (x - c) ^ m) +$   
 $diff \ n \ t / \ real \ (fact \ n) * (x - c) ^ n$   
 by (rule taylor-down)  
 with True show ?thesis by simp  
 next  
 case False  
 note INIT  
 moreover from DERIV and INTERV  
 have  $\forall \ m \ t. \ m < n \wedge a \leq t \wedge t \leq x \longrightarrow DERIV \ (\diff \ m) \ t :> \diff \ (Suc \ m) \ t$   
 by fastsimp  
 moreover from INTERV have  $a \leq c$  by arith  
 moreover from False and INTERV have  $c < x$  by arith  
 ultimately have EX:  $\exists \ t > c. \ t < x \wedge f \ x =$   
 $(\sum \ m = 0..<n. \ diff \ m \ c / \ real \ (fact \ m) * (x - c) ^ m) +$   
 $diff \ n \ t / \ real \ (fact \ n) * (x - c) ^ n$   
 by (rule taylor-up)

```
  with False show ?thesis by simp
qed

end
```

## 82 Complex-Main: Comprehensive Complex Theory

```
theory Complex-Main
imports
  Main
  Real
  SupInf
  Complex
  Log
  Ln
  Taylor
  Deriv
begin

end
```

## References

- [1] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.