

Isabelle/HOL — Higher-Order Logic

June 21, 2010

Contents

1	Code-Generator: Loading the code generator modules	18
2	HOL: The basis of Higher-Order Logic	18
2.1	Primitive logic	19
2.1.1	Core syntax	19
2.1.2	Additional concrete syntax	20
2.1.3	Axioms and basic definitions	21
2.2	Fundamental rules	22
2.2.1	Equality	22
2.2.2	Congruence rules for application	23
2.2.3	Equality of booleans – iff	23
2.2.4	True	23
2.2.5	Universal quantifier	24
2.2.6	False	24
2.2.7	Negation	24
2.2.8	Implication	25
2.2.9	Existential quantifier	25
2.2.10	Conjunction	26
2.2.11	Disjunction	26
2.2.12	Classical logic	26
2.2.13	Unique existence	27
2.2.14	THE: definite description operator	27
2.2.15	Classical intro rules for disjunction and existential quantifiers	28
2.2.16	Intuitionistic Reasoning	29
2.2.17	Atomizing meta-level connectives	29
2.2.18	Atomizing elimination rules	30
2.3	Package setup	30
2.3.1	Sledgehammer setup	30
2.3.2	Classical Reasoner setup	30
2.3.3	Simplifier	32

2.3.4	Generic cases and induction	38
2.3.5	Coherent logic	40
2.3.6	Reorienting equalities	40
2.4	Other simple lemmas and lemma duplicates	40
2.5	Basic ML bindings	41
2.6	Code generator setup	41
2.6.1	SML code generator setup	41
2.6.2	Generic code generator preprocessor setup	41
2.6.3	Equality	41
2.6.4	Generic code generator foundation	42
2.6.5	Generic code generator target languages	43
2.6.6	Evaluation and normalization by evaluation	44
2.7	Counterexample Search Units	45
2.7.1	Quickcheck	45
2.7.2	Nitpick setup	45
2.8	Preprocessing for the predicate compiler	45
2.9	Legacy tactics and ML bindings	45
3	Orderings: Abstract orderings	45
3.1	Syntactic orders	45
3.2	Quasi orders	46
3.3	Partial orders	47
3.4	Linear (total) orders	48
3.5	Reasoning tools setup	51
3.6	Bounded quantifiers	53
3.7	Transitivity reasoning	54
3.8	Monotonicity, least value operator and min/max	58
3.9	Top and bottom elements	60
3.10	Dense orders	60
3.11	Wellorders	60
3.12	Order on bool	61
3.13	Order on functions	62
3.14	Name duplicates	63
4	Groups: Groups, also combined with orderings	64
4.1	Fact collections	64
4.2	Abstract structures	64
4.3	Generic operations	65
4.4	Semigroups and Monoids	66
4.5	Groups	68
4.6	(Partially) Ordered Groups	70
4.7	Support for reasoning about signs	72
4.8	Tools setup	80

5	Lattices: Abstract lattices	81
5.1	Abstract semilattice	81
5.2	Idempotent semigroup	81
5.3	Concrete lattices	81
5.3.1	Intro and elim rules	82
5.3.2	Equational laws	84
5.3.3	Strict order	86
5.4	Distributive lattices	86
5.5	Bounded lattices and boolean algebras	87
5.6	Uniqueness of inf and sup	89
5.7	\min/\max on linear orders as special case of $op \sqcap/op \sqcup$	90
5.8	Bool as lattice	90
5.9	Fun as lattice	91
6	Set: Set theory for higher-order logic	92
6.1	Sets as predicates	92
6.2	Subsets and bounded quantifiers	94
6.3	Basic operations	98
6.3.1	Subsets	98
6.3.2	Equality	99
6.3.3	The universal set – UNIV	100
6.3.4	The empty set	100
6.3.5	The Powerset operator – Pow	101
6.3.6	Set complement	101
6.3.7	Binary union – Un	102
6.3.8	Binary intersection – Int	102
6.3.9	Set difference	103
6.3.10	Augmenting a set – <i>insert</i>	103
6.3.11	Singletons, using insert	104
6.3.12	Image of a set under a function	105
6.3.13	Some rules with <i>if</i>	106
6.4	Further operations and lemmas	107
6.4.1	The “proper subset” relation	107
6.4.2	Derived rules involving subsets.	108
6.4.3	Equalities involving union, intersection, inclusion, etc.	108
6.4.4	Monotonicity of various operations	118
6.4.5	Inverse image of a function	119
6.4.6	Getting the Contents of a Singleton Set	121
6.4.7	Least value operator	121
6.5	Misc	121
7	Typedef: HOL type definitions	122

8 Complete-Lattice: Complete lattices, with special focus on sets	123
8.1 Syntactic infimum and supremum operations	123
8.2 Abstract complete lattices	123
8.3 <i>bool</i> and $- \Rightarrow -$ as complete lattice	126
8.4 Union	127
8.5 Unions of families	128
8.6 Inter	132
8.7 Intersections of families	133
8.8 Distributive laws	135
8.9 Complement	136
8.10 Miniscoping and maxiscoping	136
9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions	139
9.1 Least and greatest fixed points	139
9.2 Proof of Knaster-Tarski Theorem using <i>lfp</i>	139
9.3 General induction rules for least fixed points	140
9.4 Proof of Knaster-Tarski Theorem using <i>gfp</i>	141
9.5 Coinduction rules for greatest fixed points	141
9.6 Even Stronger Coinduction Rule, by Martin Coen	142
9.7 Inductive predicates and sets	143
9.8 Inductive datatypes and primitive recursion	143
10 Fun: Notions about functions	143
10.1 The Identity Function <i>id</i>	144
10.2 The Composition Operator $f \circ g$	144
10.3 The Forward Composition Operator <i>fcomp</i>	145
10.4 Injectivity and Surjectivity	145
10.5 Function Updating	150
10.6 <i>override-on</i>	151
10.7 <i>swap</i>	151
10.8 Inversion of injective functions	152
10.9 Proof tool setup	153
10.10 Code generator setup	153
11 Product-Type: Cartesian products	154
11.1 <i>bool</i> is a datatype	154
11.2 The <i>unit</i> type	154
11.3 The product type	156
11.3.1 Type definition	156
11.3.2 Tuple syntax	156
11.3.3 Code generator setup	157
11.3.4 Fundamental operations and properties	158

11.3.5	Derived operations	163
11.4	Inductively defined sets	170
11.5	Legacy theorem bindings and duplicates	170
12	Sum-Type: The Disjoint Sum of Two Types	171
12.1	Construction of the sum type and its basic abstract operations	171
12.2	Projections	172
12.3	The Disjoint Sum of Sets	173
13	Rings: Rings	174
14	Fields: Fields	195
15	Nat: Natural numbers	208
15.1	Type <i>ind</i>	208
15.2	Type <i>nat</i>	208
15.3	Arithmetic operators	210
15.3.1	Addition	211
15.3.2	Difference	212
15.3.3	Multiplication	213
15.4	Orders on <i>nat</i>	214
15.4.1	Operation definition	214
15.4.2	Introduction properties	215
15.4.3	Elimination properties	215
15.4.4	Inductive (?) properties	216
15.4.5	<i>min</i> and <i>max</i>	219
15.4.6	Monotonicity of Addition	220
15.4.7	Additional theorems about <i>op</i> \leq	221
15.4.8	More results about difference	224
15.4.9	Monotonicity of Multiplication	225
15.5	Natural operation of natural numbers on functions	226
15.6	Embedding of the Naturals into any <i>semiring-1: of-nat</i>	227
15.7	The Set of Natural Numbers	230
15.8	Further Arithmetic Facts Concerning the Natural Numbers	230
15.9	The divides relation on <i>nat</i>	233
15.10	size of a datatype value	234
15.11	code module namespace	234
16	Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums	235
16.1	The datatype universe	235
16.2	Freeness: Distinctness of Constructors	237
16.3	Set Constructions	240

17 Record: Extensible records with structural subtyping	244
17.1 Introduction	244
17.2 Operators and lemmas for types isomorphic to tuples	245
17.3 Logical infrastructure for records	246
17.4 Concrete record syntax	251
17.5 Record package	252
18 Power: Exponentiation	252
18.1 Powers for Arbitrary Monoids	252
18.2 Exponentiation for the Natural Numbers	258
18.3 Code generator tweak	259
19 Option: Datatype option	259
19.0.1 Operations	260
19.0.2 Code generator setup	261
20 Finite-Set: Finite sets	262
20.1 Predicate for finite sets	262
20.2 Class <i>finite</i>	267
20.3 A basic fold functional for finite sets	268
20.3.1 From <i>fold-graph</i> to <i>fold</i>	269
20.3.2 Expressing set operations via <i>fold</i>	270
20.4 The derived combinator <i>fold-image</i>	272
20.5 A fold functional for non-empty sets	274
20.5.1 Determinacy for <i>fold1Set</i>	276
20.5.2 Lemmas about <i>fold1</i>	276
20.6 Locales as mini-packages for fold operations	276
20.6.1 The natural case	276
20.6.2 The natural case with idempotency	278
20.6.3 The image case with fixed function	278
20.6.4 The image case with flexible function	280
20.6.5 The image case with fixed function and idempotency	281
20.6.6 The image case with flexible function and idempotency	281
20.6.7 The neutral-less case	281
20.6.8 The neutral-less case with idempotency	283
20.7 Finite cardinality	283
20.7.1 Cardinality of image	287
20.7.2 Cardinality of sums	287
20.7.3 Cardinality of the Powerset	288
20.7.4 Relating injectivity and surjectivity	288

21 Relation: Relations	288
21.1 Definitions	288
21.2 The identity relation	290
21.3 Diagonal: identity over a set	290
21.4 Composition of two relations	291
21.5 Reflexivity	292
21.6 Antisymmetry	293
21.7 Symmetry	293
21.8 Transitivity	293
21.9 Irreflexivity	294
21.10 Totality	294
21.11 Converse	294
21.12 Domain	295
21.13 Range	297
21.14 Field	298
21.15 Image of a set under a relation	298
21.16 Single valued relations	299
21.17 Graphs given by <i>Collect</i>	300
21.18 Inverse image	300
21.19 Finiteness	300
21.20 Miscellaneous	301
22 Predicate: Predicates as relations and enumerations	301
22.1 Predicates as (complete) lattices	301
22.1.1 Equality	302
22.1.2 Order relation	302
22.1.3 Top and bottom elements	302
22.1.4 Binary union	303
22.1.5 Binary intersection	303
22.1.6 Unions of families	304
22.1.7 Intersections of families	305
22.2 Predicates as relations	305
22.2.1 Composition	305
22.2.2 Converse	306
22.2.3 Domain	306
22.2.4 Range	307
22.2.5 Inverse image	307
22.2.6 Powerset	307
22.2.7 Properties of relations	307
22.3 Predicates as enumerations	308
22.3.1 The type of predicate enumerations (a monad)	308
22.3.2 Emptiness check and definite choice	310
22.3.3 Derived operations	312
22.3.4 Implementation	313

23 Transitive-Closure: Reflexive and Transitive closure of a relation	316
23.1 Reflexive closure	318
23.2 Reflexive-transitive closure	318
23.3 Transitive closure	321
23.4 The power operation on relations	326
23.5 Setup of transitivity reasoner	328
24 Wellfounded: Well-founded Recursion	329
24.1 Basic Definitions	329
24.2 Basic Results	330
24.3 Well-Foundedness Results for Unions	332
24.4 Acyclic relations	332
24.5 <i>nat</i> is well-founded	333
24.6 Accessible Part	334
24.7 Tools for building wellfounded relations	336
24.8 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.	338
24.9 size of a datatype value	338
25 FunDef: Function Definitions and Termination Proofs	338
25.1 Definitions with default value.	339
25.2 Measure Functions	340
25.3 Congruence Rules	340
25.4 Simp rules for termination proofs	341
25.5 Decomposition	341
25.6 Reduction Pairs	341
25.7 Concrete orders for SCNP termination proofs	342
25.8 Tool setup	343
26 Extraction: Program extraction for HOL	343
26.1 Setup	343
26.2 Type of extracted program	344
26.3 Realizability	345
26.4 Computational content of basic inference rules	346
27 Plain: Plain HOL	351
28 Big-Operators: Big operators and finite (non-empty) sets	351
28.1 Generic monoid operation over a set	351
28.2 Generalized summation over a set	352
28.2.1 Properties in more restricted classes of structures . . .	356
28.2.2 Cardinality as special case of <i>setsum</i>	359
28.2.3 Cardinality of products	360

28.3	Generalized product over a set	360
28.3.1	Properties in more restricted classes of structures . . .	363
28.4	Versions of <i>inf</i> and <i>sup</i> on non-empty sets	365
28.5	Versions of <i>min</i> and <i>max</i> on non-empty sets	368

29 Equiv-Relations: Equivalence Relations in Higher-Order Set

Theory	372
29.1 Equivalence relations	372
29.2 Equivalence classes	373
29.3 Quotients	374
29.4 Defining unary operations upon equivalence classes	375
29.5 Defining binary operations upon equivalence classes	375
29.6 Quotients and finiteness	377

30 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

ural Numbers	377
30.1 The equivalence relation underlying the integers	377
30.2 Construction of the Integers	378
30.3 Arithmetic Operations	379
30.4 The \leq Ordering	380
30.5 Embedding of the Integers into any <i>ring-1: of-int</i>	381
30.6 Magnitude of an Integer, as a Natural Number: <i>nat</i>	383
30.7 Lemmas about the Function <i>of-nat</i> and Orderings	385
30.8 Cases and induction	386
30.9 Binary representation	386
30.9.1 The constructors <i>Bit0</i> , <i>Bit1</i> , <i>Pls</i> and <i>Min</i>	386
30.9.2 Successor and predecessor functions	388
30.9.3 Binary arithmetic	388
30.9.4 Binary comparisons	390
30.10 Converting Numerals to Rings: <i>number-of</i>	392
30.10.1 Equality of Binary Numbers	394
30.10.2 Comparisons, for Ordered Rings	395
30.10.3 The Less-Than Relation	395
30.10.4 Simplification of arithmetic operations on integer constants.	396
30.10.5 Simplification of arithmetic when nested to the right.	397
30.11 The Set of Integers	397
30.12 <i>setsum</i> and <i>setprod</i>	399
30.13 Inequality Reasoning for the Arithmetic Simproc	399
30.14 Special Arithmetic Rules for Abstract 0 and 1	400
30.15 Setting up simplification procedures	401
30.16 Lemmas About Small Numerals	401
30.17 More Inequality Reasoning	401
30.18 The functions <i>nat</i> and <i>int</i>	402

30.19	Induction principles for <code>int</code>	403
30.20	Intermediate value theorems	404
30.21	Products and 1, by T. M. Rasmussen	404
30.22	Further theorems on numerals	405
30.22.1	Special Simplification for Constants	405
30.22.2	Optional Simplification Rules Involving Constants	407
30.23	The divides relation	408
30.24	Configuration of the code generator	409
30.25	Legacy theorems	413
31	Nat-Numeral: Binary numerals for the natural numbers	414
31.1	Numerals for natural numbers	414
31.2	Special case: squares and cubes	415
31.3	Predicate for negative binary numbers	419
31.4	Function <i>nat</i> : Coercion from Type <i>int</i> to <i>nat</i>	420
31.5	Function <i>int</i> : Coercion from Type <i>nat</i> to <i>int</i>	420
31.5.1	Successor	420
31.5.2	Addition	421
31.5.3	Subtraction	421
31.5.4	Multiplication	422
31.6	Comparisons	422
31.6.1	Equals (=)	422
31.6.2	Less-than (<)	422
31.6.3	Less-than-or-equal (<=)	422
31.7	Powers with Numeric Exponents	422
31.7.1	Nat	423
31.7.2	Arith	423
31.8	Comparisons involving (0::nat)	423
31.9	Comparisons involving <i>Suc</i>	424
31.10	Max and Min Combined with <i>Suc</i>	424
31.11	Literal arithmetic involving powers	425
31.12	Literal arithmetic and <i>of-nat</i>	427
31.12.1	For simplifying <i>Suc m - K</i> and <i>K - Suc m</i>	427
31.12.2	For <i>nat-case</i> and <i>nat-rec</i>	428
31.12.3	Various Other Lemmas	428
32	Nat-Transfer: Generic transfer machinery; specific transfer from nats to ints and back.	429
32.1	Generic transfer machinery	429
32.2	Set up transfer from nat to int	429
32.3	Set up transfer from int to nat	433

33 Divides: The division operators <code>div</code> and <code>mod</code>	436
33.1 Syntactic division operations	436
33.2 Abstract division in commutative semirings.	436
33.3 Division on <i>nat</i>	441
33.3.1 Quotient	443
33.3.2 Remainder	443
33.3.3 Quotient and Remainder	444
33.3.4 Further Facts about Quotient and Remainder	445
33.3.5 An “induction” law for modulus arithmetic.	446
33.4 Division on <i>int</i>	448
33.4.1 Uniqueness and Monotonicity of Quotients and Re- mainders	450
33.4.2 Correctness of <i>posDivAlg</i> , the Algorithm for Non-Negative Dividends	450
33.4.3 Correctness of <i>negDivAlg</i> , the Algorithm for Negative Dividends	451
33.4.4 Existence Shown by Proving the Division Algorithm to be Correct	451
33.4.5 General Properties of <code>div</code> and <code>mod</code>	452
33.4.6 Laws for <code>div</code> and <code>mod</code> with Unary Minus	453
33.4.7 Division of a Number by Itself	454
33.4.8 Computation of Division and Remainder	454
33.4.9 Monotonicity in the First Argument (Dividend)	456
33.4.10 Monotonicity in the Second Argument (Divisor)	456
33.4.11 More Algebraic Laws for <code>div</code> and <code>mod</code>	457
33.4.12 Proving $a \text{ div } (b * c) = a \text{ div } b \text{ div } c$	458
33.4.13 Splitting Rules for <code>div</code> and <code>mod</code>	459
33.4.14 Speeding up the Division Algorithm with Shifting	459
33.4.15 Computing <code>mod</code> by Shifting (proofs resemble those for <code>div</code>)	460
33.4.16 Quotients of Signs	460
33.4.17 The Divides Relation	461
33.4.18 Nitpick	464
33.4.19 Code generation	464
34 Code-Numeral: Type of target language numerals	465
34.1 Datatype of target language numerals	465
34.2 Indices as datatype of ints	467
34.3 Basic arithmetic	467
34.3.1 Lazy Evaluation of an indexed function	470
34.4 Code generator setup	470
35 Numeral-Simprocs: Combination and Cancellation Simprocs for Numeral Expressions	471

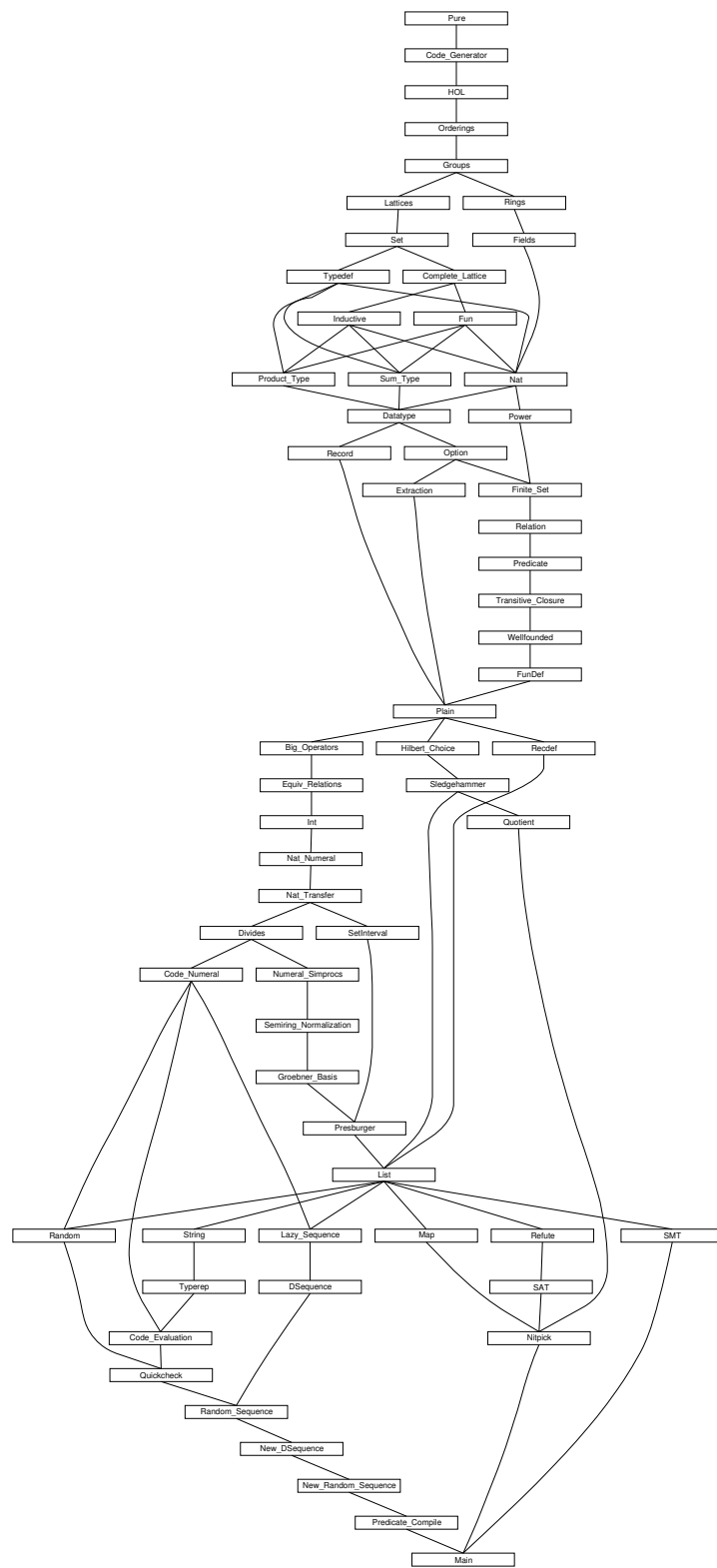
36 Semiring-Normalization: Semiring normalization	473
37 Groebner-Basis: Groebner bases	477
37.1 Groebner Bases	477
38 SetInterval: Set intervals	478
38.1 Various equivalences	480
38.2 Logical Equivalences for Set Inclusion and Equality	480
38.3 Two-sided intervals	481
38.3.1 Emptiness, singletons, subset	482
38.3.2 Intersection	483
38.4 Intervals of natural numbers	484
38.4.1 The Constant <i>lessThan</i>	484
38.4.2 The Constant <i>greaterThan</i>	484
38.4.3 The Constant <i>atLeast</i>	484
38.4.4 The Constant <i>atMost</i>	484
38.4.5 The Constant <i>atLeastLessThan</i>	485
38.4.6 Intervals of nats with <i>Suc</i>	485
38.4.7 Image	486
38.4.8 Finiteness	486
38.4.9 Proving Inclusions and Equalities between Unions	487
38.4.10 Cardinality	488
38.5 Intervals of integers	489
38.5.1 Finiteness	489
38.5.2 Cardinality	489
38.6 Lemmas useful with the summation operator <i>setsum</i>	490
38.6.1 Disjoint Unions	490
38.6.2 Disjoint Intersections	491
38.6.3 Some Differences	492
38.6.4 Some Subset Conditions	492
38.7 Summation indexed over intervals	492
38.8 Shifting bounds	495
38.9 The formula for geometric sums	495
38.10 The formula for arithmetic sums	495
38.11 Products indexed over intervals	496
38.12 Transfer setup	497
39 Presburger: Decision Procedure for Presburger Arithmetic	497
39.1 The $-\infty$ and $+\infty$ Properties	498
39.2 The A and B sets	498
39.3 Cooper's Theorem $-\infty$ and $+\infty$ Version	500
39.3.1 First some trivial facts about periodic sets or predicates	500
39.3.2 The $-\infty$ Version	500
39.3.3 The $+\infty$ Version	500

40 Hilbert-Choice: Hilbert’s Epsilon-Operator and the Axiom of Choice	502
40.1 Hilbert’s epsilon	503
40.2 Hilbert’s Epsilon-operator	503
40.3 Axiom of Choice, Proved Using the Description Operator	504
40.4 Function Inverse	504
40.5 Other Consequences of Hilbert’s Epsilon	506
40.6 Least value operator	507
40.7 Greatest value operator	508
40.8 The Meson proof procedure	509
40.8.1 Negation Normal Form	509
40.8.2 Pulling out the existential quantifiers	509
40.8.3 Generating clauses for the Meson Proof Procedure	510
40.9 Lemmas for Meson, the Model Elimination Procedure	510
40.9.1 Lemmas for Forward Proof	511
40.10 Meson package	511
40.11 Specification package – Hilbertized version	511
41 Sledgehammer: Sledgehammer: Isabelle–ATP Linkup	511
41.1 Setup of external ATPs	513
41.2 The MESON prover	513
41.3 The Metis prover	513
42 Recdef: TFL: recursive function definitions	513
42.1 Well-Founded Recursion	514
43 List: The datatype of finite lists	516
43.1 Basic list processing functions	516
43.1.1 List comprehension	522
43.1.2 $[]$ and $op \#$	523
43.1.3 <i>length</i>	524
43.1.4 $@$ – append	525
43.1.5 <i>map</i>	527
43.1.6 <i>rev</i>	529
43.1.7 <i>set</i>	530
43.1.8 <i>filter</i>	532
43.1.9 List partitioning	533
43.1.10 <i>concat</i>	534
43.1.11 <i>nth</i>	534
43.1.12 <i>list-update</i>	536
43.1.13 <i>last</i> and <i>butlast</i>	537
43.1.14 <i>take</i> and <i>drop</i>	539
43.1.15 <i>takeWhile</i> and <i>dropWhile</i>	542
43.1.16 <i>zip</i>	545

43.1.17	<i>list-all2</i>	548
43.1.18	<i>foldl</i> and <i>foldr</i>	550
43.1.19	List summation: <i>listsum</i> and \sum	553
43.1.20	<i>upt</i>	555
43.1.21	<i>upto</i> : interval-list on <i>int</i>	557
43.1.22	<i>distinct</i> and <i>remdups</i>	557
43.1.23	<i>insert</i>	560
43.1.24	<i>remove1</i>	560
43.1.25	<i>removeAll</i>	561
43.1.26	<i>replicate</i>	562
43.1.27	<i>rotate1</i> and <i>rotate</i>	564
43.1.28	<i>sublist</i> — a generalization of <i>nth</i> to sets	565
43.1.29	<i>splice</i>	566
43.1.30	Transpose	567
43.1.31	(In)finiteness	568
43.2	Sorting	568
43.2.1	<i>transpose</i> on sorted lists	572
43.2.2	<i>sorted-list-of-set</i>	573
43.2.3	<i>lists</i> : the list-forming operator over sets	574
43.2.4	Inductive definition for membership	575
43.2.5	Lists as Cartesian products	575
43.3	Relations on Lists	576
43.3.1	Length Lexicographic Ordering	576
43.3.2	Lexicographic Ordering	577
43.4	Lexicographic combination of measure functions	578
43.4.1	Lifting a Relation on List Elements to the Lists	578
43.5	Size function	579
43.6	Transfer	580
43.7	Code generator	580
43.7.1	Setup	580
43.7.2	Generation of efficient code	581
44	Random: A HOL random engine	587
44.1	Auxiliary functions	587
44.2	Random seeds	587
44.3	Base selectors	588
44.4	<i>ML</i> interface	589
45	String: Character and string types	589
45.1	Characters	590
45.2	Strings	591
45.3	Strings as dedicated datatype	592
45.4	Code generator	592

46 Typerep: Reflecting Pure types into HOL	593
47 Code-Evaluation: Term evaluation using the generic code generator	594
47.1 Term representation	594
47.1.1 Terms and class <i>term-of</i>	594
47.1.2 <i>term-of</i> instances	595
47.1.3 Code generator setup	595
47.1.4 Syntax	596
47.2 Numeric types	596
47.3 Obfuscate	597
47.4 Tracing of generated and evaluated code	597
47.5 Evaluation setup	597
48 Quickcheck: A simple counterexample generator	597
48.1 The <i>random</i> class	598
48.2 Fundamental and numeric types	598
48.3 Complex generators	599
48.4 Code setup	600
48.5 The Random-Predicate Monad	600
49 Lazy-Sequence: Lazy sequences	602
49.1 Code setup	604
49.2 With Hit Bound Value	605
50 DSequence: Depth-Limited Sequences with failure element	606
51 New-DSequence: Depth-Limited Sequences with failure element	609
51.1 Positive Depth-Limited Sequence	609
51.2 Negative Depth-Limited Sequence	610
51.3 Negation	610
52 Predicate-Compile: A compiler for predicates defined by introduction rules	613
53 Map: Maps	614
53.1 <i>empty</i>	615
53.2 <i>map-upd</i>	616
53.3 <i>map-of</i>	616
53.4 <i>Option.map</i> related	618
53.5 <i>map-comp</i> related	618
53.6 <i>++</i>	618
53.7 <i>restrict-map</i>	619
53.8 <i>map-upds</i>	620

53.9	<i>dom</i>	621
53.10	<i>ran</i>	623
53.11	<i>map-le</i>	623
53.12	Various	624
54	Quotient: Definition of Quotient Types	625
54.1	Respects predicate	626
54.2	Function map and function relation	626
54.3	Quotient Predicate	627
54.4	lemmas for regularisation of ball and bex	629
54.5	Bounded abstraction	631
54.6	<i>Bex1-rel</i> quantifier	632
54.7	Various respects and preserve lemmas	633
54.8	ML setup	635
54.9	Methods / Interface	636
55	Refute: Refute	636
56	SAT: Reconstructing external resolution proofs for propositional logic	638
57	Nitpick: Nitpick: Yet Another Counterexample Generator for Isabelle/HOL	639
58	SMT: Bindings to Satisfiability Modulo Theories (SMT) solvers	643
58.1	Triggers for quantifier instantiation	643
58.2	Higher-order encoding	644
58.3	First-order logic	644
58.4	Integer division and modulo for Z3	644
58.5	Setup	645
58.6	Configuration	645
58.7	General configuration options	645
58.8	Certificates	645
58.9	Tracing	646
58.10	Z3-specific options	646
58.11	Schematic rules for Z3 proof reconstruction	646
59	Main: Main HOL	648



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

  ⟨ML⟩

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

```

⟨ML⟩

2.1 Primitive logic

2.1.1 Core syntax

```

classes type
default-sort type
⟨ML⟩

```

arities

```

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

```

global

typedec1 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)

```

notation (output)

abbreviation

notation (output)

notation (*xsymbols*)notation (*HTML* output)

abbreviation (*iff*)

notation (*xsymbols*)

nonterminals

syntax

$$\text{-case2} \quad :: [case\text{-syn}, cases\text{-syn}] \Rightarrow cases\text{-syn} \quad (-/ \mid -)$$

translations

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

$\langle ML \rangle$

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

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

— 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

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

$impI: \quad (P \implies Q) \implies P \longrightarrow Q$
 $mp: \quad [| P \longrightarrow Q; \ P |] \implies Q$

defs

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

axioms

iff: $(P \dashv\dashv Q) \dashv\dashv (Q \dashv\dashv P) \dashv\dashv (P=Q)$
True-or-False: $(P=True) \mid (P=False)$

defs

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

finalconsts

op =
op $\dashv\dashv$
The

axiomatization

undefined :: 'a

class *default* =
fixes *default* :: 'a

2.2 Fundamental rules**2.2.1 Equality**

lemma *sym*: $s = t \implies t = s$
 $\langle proof \rangle$

lemma *ssubst*: $t = s \implies P\ s \implies P\ t$
 $\langle proof \rangle$

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

lemma *meta-eq-to-obj-eq*:
assumes *meq*: $A == B$
shows $A = B$
 $\langle proof \rangle$

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

lemma *box-equals*: $[| a=b; a=c; b=d |] \implies c=d$
 $\langle proof \rangle$

For calculational reasoning:

lemma *forw-subst*: $a = b \implies P\ b \implies P\ a$
 $\langle proof \rangle$

lemma *back-subst*: $P\ a \implies a = b \implies P\ b$
 $\langle proof \rangle$

2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon’s HOL.

lemma *fun-cong*: $(f::'a=>'b) = g ==> f(x)=g(x)$
 $\langle proof \rangle$

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

lemma *arg-cong*: $x=y ==> f(x)=f(y)$
 $\langle proof \rangle$

lemma *arg-cong2*: $\llbracket a = b; c = d \rrbracket \implies f a c = f b d$
 $\langle proof \rangle$

lemma *cong*: $\llbracket f = g; (x::'a) = y \rrbracket ==> f x = g y$
 $\langle proof \rangle$

$\langle ML \rangle$

2.2.3 Equality of booleans – iff

lemma *iffI*: **assumes** $P ==> Q$ **and** $Q ==> P$ **shows** $P=Q$
 $\langle proof \rangle$

lemma *iffD2*: $\llbracket P=Q; Q \rrbracket ==> P$
 $\langle proof \rangle$

lemma *rev-iffD2*: $\llbracket Q; P=Q \rrbracket ==> P$
 $\langle proof \rangle$

lemma *iffD1*: $Q = P \implies Q \implies P$
 $\langle proof \rangle$

lemma *rev-iffD1*: $Q \implies Q = P \implies P$
 $\langle proof \rangle$

lemma *iffE*:
assumes *major*: $P=Q$
and *minor*: $\llbracket P \dashv\dashv Q; Q \dashv\dashv P \rrbracket ==> R$
shows R
 $\langle proof \rangle$

2.2.4 True

lemma *TrueI*: *True*
 $\langle proof \rangle$

lemma *eqTrueI*: $P ==> P = True$
 $\langle proof \rangle$

lemma *eqTrueE*: $P = \text{True} \implies P$
 $\langle \text{proof} \rangle$

2.2.5 Universal quantifier

lemma *allI*: **assumes** $!x::'a. P(x)$ **shows** $\text{ALL } x. P(x)$
 $\langle \text{proof} \rangle$

lemma *spec*: $\text{ALL } x::'a. P(x) \implies P(x)$
 $\langle \text{proof} \rangle$

lemma *allE*:
assumes *major*: $\text{ALL } x. P(x)$
and *minor*: $P(x) \implies R$
shows R
 $\langle \text{proof} \rangle$

lemma *all-dupE*:
assumes *major*: $\text{ALL } x. P(x)$
and *minor*: $[| P(x); \text{ALL } x. P(x) |] \implies R$
shows R
 $\langle \text{proof} \rangle$

2.2.6 False

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

lemma *FalseE*: $\text{False} \implies P$
 $\langle \text{proof} \rangle$

lemma *False-neg-True*: $\text{False} = \text{True} \implies P$
 $\langle \text{proof} \rangle$

2.2.7 Negation

lemma *notI*:
assumes $P \implies \text{False}$
shows $\sim P$
 $\langle \text{proof} \rangle$

lemma *False-not-True*: $\text{False} \sim = \text{True}$
 $\langle \text{proof} \rangle$

lemma *True-not-False*: $\text{True} \sim = \text{False}$
 $\langle \text{proof} \rangle$

lemma *notE*: $[| \sim P; P |] \implies R$
 $\langle \text{proof} \rangle$

lemma *notI2*: $(P \implies \neg Pa) \implies (P \implies Pa) \implies \neg P$
 $\langle proof \rangle$

2.2.8 Implication

lemma *impE*:
 assumes $P \longrightarrow Q$ P $Q \implies R$
 shows R
 $\langle proof \rangle$

lemma *rev-mp*: $[| P; P \longrightarrow Q |] \implies Q$
 $\langle proof \rangle$

lemma *contrapos-nn*:
 assumes *major*: $\sim Q$
 and *minor*: $P \implies Q$
 shows $\sim P$
 $\langle proof \rangle$

lemma *contrapos-pn*:
 assumes *major*: Q
 and *minor*: $P \implies \sim Q$
 shows $\sim P$
 $\langle proof \rangle$

lemma *not-sym*: $t \sim s \implies s \sim t$
 $\langle proof \rangle$

lemma *eq-neq-eq-imp-neq*: $[| x = a ; a \sim b ; b = y |] \implies x \sim y$
 $\langle proof \rangle$

lemma *rev-contrapos*:
 assumes *pq*: $P \implies Q$
 and *nq*: $\sim Q$
 shows $\sim P$
 $\langle proof \rangle$

2.2.9 Existential quantifier

lemma *exI*: $P\ x \implies EX\ x::'a. P\ x$
 $\langle proof \rangle$

lemma *exE*:
 assumes *major*: $EX\ x::'a. P(x)$
 and *minor*: $!!x. P(x) \implies Q$
 shows Q
 $\langle proof \rangle$

2.2.10 Conjunction

lemma *conjI*: $[[P; Q]] ==> P \& Q$
 $\langle proof \rangle$

lemma *conjunct1*: $[[P \& Q]] ==> P$
 $\langle proof \rangle$

lemma *conjunct2*: $[[P \& Q]] ==> Q$
 $\langle proof \rangle$

lemma *conjE*:
 assumes *major*: $P \& Q$
 and *minor*: $[[P; Q]] ==> R$
 shows R
 $\langle proof \rangle$

lemma *context-conjI*:
 assumes $P \implies Q$ shows $P \& Q$
 $\langle proof \rangle$

2.2.11 Disjunction

lemma *disjI1*: $P ==> P | Q$
 $\langle proof \rangle$

lemma *disjI2*: $Q ==> P | Q$
 $\langle proof \rangle$

lemma *disjE*:
 assumes *major*: $P | Q$
 and *minorP*: $P ==> R$
 and *minorQ*: $Q ==> R$
 shows R
 $\langle proof \rangle$

2.2.12 Classical logic

lemma *classical*:
 assumes *prem*: $\sim P ==> P$
 shows P
 $\langle proof \rangle$

lemmas *ccontr* = *FalseE* [THEN *classical*, *standard*]

lemma *rev-notE*:
 assumes *premp*: P
 and *premnnot*: $\sim R ==> \sim P$
 shows R

$\langle proof \rangle$

lemma *notnotD*: $\sim\sim P \implies P$
 $\langle proof \rangle$

lemma *contrapos-pp*:
 assumes $p1: Q$
 and $p2: \sim P \implies \sim Q$
 shows P
 $\langle proof \rangle$

2.2.13 Unique existence

lemma *exII*:
 assumes $P\ a\ !!x. P(x) \implies x=a$
 shows $EX!\ x. P(x)$
 $\langle proof \rangle$

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

lemma *ex-exII*:
 assumes *ex-prem*: $EX\ x. P(x)$
 and *eq*: $!!x\ y. [P(x); P(y)] \implies x=y$
 shows $EX!\ x. P(x)$
 $\langle proof \rangle$

lemma *exIE*:
 assumes *major*: $EX!\ x. P(x)$
 and *minor*: $!!x. [P(x); ALL\ y. P(y) \longrightarrow y=x] \implies R$
 shows R
 $\langle proof \rangle$

lemma *exI-implies-ex*: $EX!\ x. P\ x \implies EX\ x. P\ x$
 $\langle proof \rangle$

2.2.14 THE: definite description operator

lemma *the-equality*:
 assumes *prema*: $P\ a$
 and *premx*: $!!x. P\ x \implies x=a$
 shows $(THE\ x. P\ x) = a$
 $\langle proof \rangle$

lemma *theI*:
 assumes $P\ a$ and $!!x. P\ x \implies x=a$
 shows $P\ (THE\ x. P\ x)$
 $\langle proof \rangle$

lemma *theI'*: $EX!\ x. P\ x \implies P\ (THE\ x. P\ x)$
 $\langle proof \rangle$

lemma *theI2*:

assumes $P\ a\ !!x. P\ x ==> x=a\ !!x. P\ x ==> Q\ x$

shows $Q\ (THE\ x. P\ x)$

$\langle proof \rangle$

lemma *the1I2*: **assumes** $EX!\ x. P\ x \wedge x. P\ x \implies Q\ x$ **shows** $Q\ (THE\ x. P\ x)$

$\langle proof \rangle$

lemma *the1-equality* [*elim?*]: $[[\ EX!\ x. P\ x; P\ a\]] ==> (THE\ x. P\ x) = a$

$\langle proof \rangle$

lemma *the-sym-eq-trivial*: $(THE\ y. x=y) = x$

$\langle proof \rangle$

2.2.15 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:

assumes $\sim Q ==> P$ **shows** $P | Q$

$\langle proof \rangle$

lemma *excluded-middle*: $\sim P | P$

$\langle proof \rangle$

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*: $P ==> Q$

and *prem2*: $\sim P ==> Q$

shows Q

$\langle proof \rangle$

lemma *impCE*:

assumes *major*: $P \dashrightarrow Q$

and *minor*: $\sim P ==> R\ Q ==> R$

shows R

$\langle proof \rangle$

lemma *impCE'*:

assumes *major*: $P \dashrightarrow Q$

and *minor*: $Q ==> R\ \sim P ==> R$

shows R

$\langle proof \rangle$

lemma *iffCE*:
 assumes *major*: $P=Q$
 and *minor*: $[| P; Q |] ==> R \quad [| \sim P; \sim Q |] ==> R$
 shows R
 $\langle proof \rangle$

lemma *exCI*:
 assumes $ALL\ x. \sim P(x) ==> P(a)$
 shows $EX\ x. P(x)$
 $\langle proof \rangle$

2.2.16 Intuitionistic Reasoning

lemma *impE'*:
 assumes $1: P \dashv\vdash Q$
 and $2: Q ==> R$
 and $3: P \dashv\vdash Q ==> P$
 shows R
 $\langle proof \rangle$

lemma *allE'*:
 assumes $1: ALL\ x. P\ x$
 and $2: P\ x ==> ALL\ x. P\ x ==> Q$
 shows Q
 $\langle proof \rangle$

lemma *notE'*:
 assumes $1: \sim P$
 and $2: \sim P ==> P$
 shows R
 $\langle proof \rangle$

lemma *TrueE*: $True ==> P ==> P \langle proof \rangle$
lemma *notFalseE*: $\sim False ==> P ==> P \langle proof \rangle$

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 $[Pure.elim?] = iffD1\ iffD2\ impE$

$\langle ML \rangle$

2.2.17 Atomizing meta-level connectives

axiomatization where

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

lemma *atomize-all* [*atomize*]: $(!!x. P\ x) == \text{Trueprop}\ (ALL\ x. P\ x)$
 $\langle proof \rangle$

lemma *atomize-imp* [*atomize*]: $(A ==> B) == \text{Trueprop}\ (A --> B)$
 $\langle proof \rangle$

lemma *atomize-not*: $(A ==> False) == \text{Trueprop}\ (\sim A)$
 $\langle proof \rangle$

lemma *atomize-eq* [*atomize*]: $(x == y) == \text{Trueprop}\ (x = y)$
 $\langle proof \rangle$

lemma *atomize-conj* [*atomize*]: $(A \&\&\& B) == \text{Trueprop}\ (A \& B)$
 $\langle proof \rangle$

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

2.2.18 Atomizing elimination rules

$\langle ML \rangle$

lemma *atomize-exL*[*atomize-elim*]: $(!!x. P\ x ==> Q) == ((EX\ x. P\ x) ==> Q)$
 $\langle proof \rangle$

lemma *atomize-conjL*[*atomize-elim*]: $(A ==> B ==> C) == (A \& B ==> C)$
 $\langle proof \rangle$

lemma *atomize-disjL*[*atomize-elim*]: $((A ==> C) ==> (B ==> C) ==> C) == ((A \mid B ==> C) ==> C)$
 $\langle proof \rangle$

lemma *atomize-elimL*[*atomize-elim*]: $(!!B. (A ==> B) ==> B) == \text{Trueprop}\ A$
 $\langle proof \rangle$

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.

$\langle ML \rangle$

2.3.2 Classical Reasoner setup

lemma *imp-elim*: $P --> Q ==> (\sim R ==> P) ==> (Q ==> R) ==> R$
 $\langle proof \rangle$

lemma *swap*: $\sim P \implies (\sim R \implies P) \implies R$
 $\langle proof \rangle$

lemma *thin-refl*:
 $\bigwedge X. \llbracket x=x; PROP\ W \rrbracket \implies PROP\ W \langle proof \rangle$

$\langle ML \rangle$

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*]

$\langle ML \rangle$

lemma *contrapos-np*: $\sim Q \implies (\sim P \implies Q) \implies P$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

$\langle ML \rangle$ **2.3.3 Simplifier****lemma** *eta-contract-eq*: $(\%s. f\ s) = f\ \langle proof \rangle$ **lemma** *simp-thms*:**shows** *not-not*: $(\sim \sim P) = P$ **and** *Not-eq-iff*: $((\sim P) = (\sim Q)) = (P = Q)$ **and** $(P \sim = Q) = (P = (\sim Q))$ $(P \mid \sim P) = True \quad (\sim P \mid P) = True$ $(x = x) = True$ **and** *not-True-eq-False* [code]: $(\neg True) = False$ **and** *not-False-eq-True* [code]: $(\neg False) = True$ **and** $(\sim P) \sim = P \quad P \sim = (\sim P)$ $(True = P) = P$ **and** *eq-True*: $(P = True) = P$ **and** $(False = P) = (\sim P)$ **and** *eq-False*: $(P = False) = (\neg P)$ **and** $(True \dashrightarrow P) = P \quad (False \dashrightarrow P) = True$ $(P \dashrightarrow True) = True \quad (P \dashrightarrow P) = True$ $(P \dashrightarrow False) = (\sim P) \quad (P \dashrightarrow \sim P) = (\sim P)$ $(P \ \& \ True) = P \quad (True \ \& \ P) = P$ $(P \ \& \ False) = False \quad (False \ \& \ P) = False$ $(P \ \& \ P) = P \quad (P \ \& \ (P \ \& \ Q)) = (P \ \& \ Q)$ $(P \ \& \ \sim P) = False \quad (\sim P \ \& \ P) = False$ $(P \mid True) = True \quad (True \mid P) = True$ $(P \mid False) = P \quad (False \mid P) = P$ $(P \mid P) = P \quad (P \mid (P \mid Q)) = (P \mid Q)$ **and** $(ALL\ x. P) = P \quad (EX\ x. P) = P \quad EX\ x. x=t \quad EX\ x. t=x$ **and** $!!P. (EX\ x. x=t \ \& \ P(x)) = P(t)$ $!!P. (EX\ x. t=x \ \& \ P(x)) = P(t)$ $!!P. (ALL\ x. x=t \dashrightarrow P(x)) = P(t)$ $!!P. (ALL\ x. t=x \dashrightarrow P(x)) = P(t)$ $\langle proof \rangle$ **lemma** *disj-absorb*: $(A \mid A) = A$ $\langle proof \rangle$ **lemma** *disj-left-absorb*: $(A \mid (A \mid B)) = (A \mid B)$ $\langle proof \rangle$ **lemma** *conj-absorb*: $(A \ \& \ A) = A$ $\langle proof \rangle$

lemma *conj-left-absorb*: $(A \ \& \ (A \ \& \ B)) = (A \ \& \ B)$
 $\langle proof \rangle$

lemma *eq-ac*:
shows *eq-commute*: $(a=b) = (b=a)$
and *eq-left-commute*: $(P=(Q=R)) = (Q=(P=R))$
and *eq-assoc*: $((P=Q)=R) = (P=(Q=R))$ $\langle proof \rangle$
lemma *neq-commute*: $(a \sim b) = (b \sim a)$ $\langle proof \rangle$

lemma *conj-comms*:
shows *conj-commute*: $(P \ \& \ Q) = (Q \ \& \ P)$
and *conj-left-commute*: $(P \ \& \ (Q \ \& \ R)) = (Q \ \& \ (P \ \& \ R))$ $\langle proof \rangle$
lemma *conj-assoc*: $((P \ \& \ Q) \ \& \ R) = (P \ \& \ (Q \ \& \ R))$ $\langle proof \rangle$

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

lemma *disj-comms*:
shows *disj-commute*: $(P \ | \ Q) = (Q \ | \ P)$
and *disj-left-commute*: $(P \ | \ (Q \ | \ R)) = (Q \ | \ (P \ | \ R))$ $\langle proof \rangle$
lemma *disj-assoc*: $((P \ | \ Q) \ | \ R) = (P \ | \ (Q \ | \ R))$ $\langle proof \rangle$

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

lemma *conj-disj-distribL*: $(P \ \& \ (Q \ | \ R)) = (P \ \& \ Q \ | \ P \ \& \ R)$ $\langle proof \rangle$
lemma *conj-disj-distribR*: $((P \ | \ Q) \ \& \ R) = (P \ \& \ R \ | \ Q \ \& \ R)$ $\langle proof \rangle$

lemma *disj-conj-distribL*: $(P \ | \ (Q \ \& \ R)) = ((P \ | \ Q) \ \& \ (P \ | \ R))$ $\langle proof \rangle$
lemma *disj-conj-distribR*: $((P \ \& \ Q) \ | \ R) = ((P \ \& \ R) \ \& \ (Q \ \& \ R))$ $\langle proof \rangle$

lemma *imp-conjR*: $(P \ \longrightarrow \ (Q \ \& \ R)) = ((P \ \longrightarrow \ Q) \ \& \ (P \ \longrightarrow \ R))$ $\langle proof \rangle$
lemma *imp-conjL*: $((P \ \& \ Q) \ \longrightarrow \ R) = (P \ \longrightarrow \ (Q \ \longrightarrow \ R))$ $\langle proof \rangle$
lemma *imp-disjL*: $((P \ | \ Q) \ \longrightarrow \ R) = ((P \ \longrightarrow \ R) \ \& \ (Q \ \longrightarrow \ R))$ $\langle proof \rangle$

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

lemma *imp-disj-not1*: $(P \ \longrightarrow \ Q \ | \ R) = (\sim Q \ \longrightarrow \ P \ \longrightarrow \ R)$ $\langle proof \rangle$
lemma *imp-disj-not2*: $(P \ \longrightarrow \ Q \ | \ R) = (\sim R \ \longrightarrow \ P \ \longrightarrow \ Q)$ $\langle proof \rangle$

lemma *imp-disj1*: $((P \ \longrightarrow \ Q) \ | \ R) = (P \ \longrightarrow \ Q \ | \ R)$ $\langle proof \rangle$
lemma *imp-disj2*: $(Q \ | \ (P \ \longrightarrow \ R)) = (P \ \longrightarrow \ Q \ | \ R)$ $\langle proof \rangle$

lemma *imp-cong*: $(P = P') \implies (P' \implies (Q = Q')) \implies ((P \ \longrightarrow \ Q) = (P' \ \longrightarrow \ Q'))$
 $\langle proof \rangle$

lemma *de-Morgan-disj*: $(\sim(P \ | \ Q)) = (\sim P \ \& \ \sim Q)$ $\langle proof \rangle$
lemma *de-Morgan-conj*: $(\sim(P \ \& \ Q)) = (\sim P \ | \ \sim Q)$ $\langle proof \rangle$
lemma *not-imp*: $(\sim(P \ \longrightarrow \ Q)) = (P \ \& \ \sim Q)$ $\langle proof \rangle$
lemma *not-iff*: $(P \sim Q) = (P = (\sim Q))$ $\langle proof \rangle$
lemma *disj-not1*: $(\sim P \ | \ Q) = (P \ \longrightarrow \ Q)$ $\langle proof \rangle$

lemma *disj-not2*: $(P \mid \sim Q) = (Q \dashrightarrow P)$ — changes orientation :-(
 $\langle proof \rangle$)

lemma *imp-conv-disj*: $(P \dashrightarrow Q) = ((\sim P) \mid Q)$ $\langle proof \rangle$

lemma *iff-conv-conj-imp*: $(P = Q) = ((P \dashrightarrow Q) \& (Q \dashrightarrow P))$ $\langle proof \rangle$

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.

$\langle proof \rangle$

lemma *not-all*: $(\sim (! x. P(x))) = (? x. \sim P(x))$ $\langle proof \rangle$

lemma *imp-all*: $((! x. P x) \dashrightarrow Q) = (? x. P x \dashrightarrow Q)$ $\langle proof \rangle$

lemma *not-ex*: $(\sim (? x. P(x))) = (! x. \sim P(x))$ $\langle proof \rangle$

lemma *imp-ex*: $((? x. P x) \dashrightarrow Q) = (! x. P x \dashrightarrow Q)$ $\langle proof \rangle$

lemma *all-not-ex*: $(ALL x. P x) = (\sim (EX x. \sim P x))$ $\langle proof \rangle$

declare *All-def* [no-atp]

lemma *ex-disj-distrib*: $(? x. P(x) \mid Q(x)) = ((? x. P(x)) \mid (? x. Q(x)))$ $\langle proof \rangle$

lemma *all-conj-distrib*: $(!x. P(x) \& Q(x)) = ((! x. P(x)) \& (! x. Q(x)))$ $\langle proof \rangle$

The $\&$ congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

lemma *conj-cong*:

$(P = P') ==> (P' ==> (Q = Q')) ==> ((P \& Q) = (P' \& Q'))$

$\langle proof \rangle$

lemma *rev-conj-cong*:

$(Q = Q') ==> (Q' ==> (P = P')) ==> ((P \& Q) = (P' \& Q'))$

$\langle proof \rangle$

The \mid congruence rule: not included by default!

lemma *disj-cong*:

$(P = P') ==> (\sim P' ==> (Q = Q')) ==> ((P \mid Q) = (P' \mid Q'))$

$\langle proof \rangle$

if-then-else rules

lemma *if-True* [code]: $(if\ True\ then\ x\ else\ y) = x$

$\langle proof \rangle$

lemma *if-False* [code]: $(if\ False\ then\ x\ else\ y) = y$

$\langle proof \rangle$

lemma *if-P*: $P ==> (if\ P\ then\ x\ else\ y) = x$

$\langle proof \rangle$

lemma *if-not-P*: $\sim P \implies (\text{if } P \text{ then } x \text{ else } y) = y$
 $\langle \text{proof} \rangle$

lemma *split-if*: $P (\text{if } Q \text{ then } x \text{ else } y) = ((Q \multimap P(x)) \ \& \ (\sim Q \multimap P(y)))$
 $\langle \text{proof} \rangle$

lemma *split-if-asm*: $P (\text{if } Q \text{ then } x \text{ else } y) = (\sim((Q \ \& \ \sim P \ x) \mid (\sim Q \ \& \ \sim P \ y)))$
 $\langle \text{proof} \rangle$

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

lemma *if-cancel*: $(\text{if } c \text{ then } x \text{ else } x) = x$
 $\langle \text{proof} \rangle$

lemma *if-eq-cancel*: $(\text{if } x = y \text{ then } y \text{ else } x) = x$
 $\langle \text{proof} \rangle$

lemma *if-bool-eq-conj*: $(\text{if } P \text{ then } Q \text{ else } R) = ((P \multimap Q) \ \& \ (\sim P \multimap R))$
 — This form is useful for expanding *ifs* on the RIGHT of the \implies symbol.
 $\langle \text{proof} \rangle$

lemma *if-bool-eq-disj*: $(\text{if } P \text{ then } Q \text{ else } R) = ((P \ \& \ Q) \mid (\sim P \ \& \ R))$
 — And this form is useful for expanding *ifs* on the LEFT.
 $\langle \text{proof} \rangle$

lemma *Eq-TrueI*: $P \implies P == \text{True}$ $\langle \text{proof} \rangle$

lemma *Eq-FalseI*: $\sim P \implies P == \text{False}$ $\langle \text{proof} \rangle$

let rules for *simproc*

lemma *Let-folded*: $f \ x \equiv g \ x \implies \text{Let } x \ f \equiv \text{Let } x \ g$
 $\langle \text{proof} \rangle$

lemma *Let-unfold*: $f \ x \equiv g \implies \text{Let } x \ f \equiv g$
 $\langle \text{proof} \rangle$

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* \implies 1) **where**
 $[code \ del]: \text{simp-implies} \equiv op \implies$

lemma *simp-impliesI*:
assumes *PQ*: $(PROP \ P \implies PROP \ Q)$
shows $PROP \ P =_{\text{simp}\implies} PROP \ Q$
 $\langle \text{proof} \rangle$

lemma *simp-impliesE*:
assumes *PQ*: $PROP \ P =_{\text{simp}\implies} PROP \ Q$
and *P*: $PROP \ P$

and $QR: PROP\ Q \implies PROP\ R$
shows $PROP\ R$
 $\langle proof \rangle$

lemma *simp-implies-cong*:
assumes $PP': PROP\ P == PROP\ P'$
and $P'QQ': PROP\ P' ==> (PROP\ Q == PROP\ Q')$
shows $(PROP\ P ==_{simp} PROP\ Q) == (PROP\ P' ==_{simp} PROP\ Q')$
 $\langle proof \rangle$

lemma *uncurry*:
assumes $P \longrightarrow Q \longrightarrow R$
shows $P \wedge Q \longrightarrow R$
 $\langle proof \rangle$

lemma *iff-allI*:
assumes $\bigwedge x. P\ x = Q\ x$
shows $(\forall x. P\ x) = (\forall x. Q\ x)$
 $\langle proof \rangle$

lemma *iff-exI*:
assumes $\bigwedge x. P\ x = Q\ x$
shows $(\exists x. P\ x) = (\exists x. Q\ x)$
 $\langle proof \rangle$

lemma *all-comm*:
 $(\forall x\ y. P\ x\ y) = (\forall y\ x. P\ x\ y)$
 $\langle proof \rangle$

lemma *ex-comm*:
 $(\exists x\ y. P\ x\ y) = (\exists y\ x. P\ x\ y)$
 $\langle proof \rangle$

$\langle ML \rangle$

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

$\langle ML \rangle$

lemma *True-implies-equals*: $(True \implies PROP\ P) \equiv PROP\ P$
 $\langle proof \rangle$

lemma *ex-simps*:
 $!!P\ Q. (EX\ x. P\ x \ \&\ Q) = ((EX\ x. P\ x) \ \&\ Q)$
 $!!P\ Q. (EX\ x. P \ \&\ Q\ x) = (P \ \&\ (EX\ x. Q\ x))$
 $!!P\ Q. (EX\ x. P\ x \mid Q) = ((EX\ x. P\ x) \mid Q)$
 $!!P\ Q. (EX\ x. P \mid Q\ x) = (P \mid (EX\ x. Q\ x))$
 $!!P\ Q. (EX\ x. P\ x \dashv\dashv Q) = ((ALL\ x. P\ x) \dashv\dashv Q)$
 $!!P\ Q. (EX\ x. P \dashv\dashv Q\ x) = (P \dashv\dashv (EX\ x. Q\ x))$
 — Miniscoping: pushing in existential quantifiers.

$\langle \text{proof} \rangle$

lemma *all-simps*:

$!!P\ Q. (ALL\ x. P\ x \ \&\ Q) = ((ALL\ x. P\ x) \ \&\ Q)$
 $!!P\ Q. (ALL\ x. P \ \&\ Q\ x) = (P \ \&\ (ALL\ x. Q\ x))$
 $!!P\ Q. (ALL\ x. P\ x \ | \ Q) = ((ALL\ x. P\ x) \ | \ Q)$
 $!!P\ Q. (ALL\ x. P \ | \ Q\ x) = (P \ | \ (ALL\ x. Q\ x))$
 $!!P\ Q. (ALL\ x. P\ x \ --> Q) = ((EX\ x. P\ x) \ --> Q)$
 $!!P\ Q. (ALL\ x. P \ --> Q\ x) = (P \ --> (ALL\ x. Q\ x))$

— Miniscoping: pushing in universal quantifiers.

$\langle \text{proof} \rangle$

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*

$\langle ML \rangle$

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

$\langle proof \rangle$

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

lemma *if-weak-cong* [*cong*]:
assumes $b = c$
shows $(if\ b\ then\ x\ else\ y) = (if\ c\ then\ x\ else\ y)$
 $\langle proof \rangle$

Prevents simplification of t : much faster

lemma *let-weak-cong*:
assumes $a = b$
shows $(let\ x = a\ in\ t\ x) = (let\ x = b\ in\ t\ x)$
 $\langle proof \rangle$

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

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
 $\langle proof \rangle$

lemma *if-distrib*:
 $f\ (if\ c\ then\ x\ else\ y) = (if\ c\ then\ f\ x\ else\ f\ y)$
 $\langle proof \rangle$

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 $(x = z) = (y = z)$
 $\langle proof \rangle$

2.3.4 Generic cases and induction

Rule projections:

$\langle ML \rangle$

definition *induct-forall* **where**
 $induct-forall\ P == \forall x. P\ x$

definition *induct-implies* **where**
 $induct-implies\ A\ B == A \longrightarrow B$

definition *induct-equal* **where**
 $induct-equal\ x\ y == x = y$

definition *induct-conj* **where**

induct-conj $A \ B == A \wedge B$

definition *induct-true* **where**

induct-true == *True*

definition *induct-false* **where**

induct-false == *False*

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

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

lemma *induct-equal-eq*: $(x == y) == \text{Trueprop} \ (\text{induct-equal} \ x \ y)$
 ⟨proof⟩

lemma *induct-conj-eq*: $(A \ \&\&\& \ B) == \text{Trueprop} \ (\text{induct-conj} \ A \ B)$
 ⟨proof⟩

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*: $\text{induct-forall} \ (\lambda x. \text{induct-conj} \ (A \ x) \ (B \ x)) =$
 $\text{induct-conj} \ (\text{induct-forall} \ A) \ (\text{induct-forall} \ B)$
 ⟨proof⟩

lemma *induct-implies-conj*: $\text{induct-implies} \ C \ (\text{induct-conj} \ A \ B) =$
 $\text{induct-conj} \ (\text{induct-implies} \ C \ A) \ (\text{induct-implies} \ C \ B)$
 ⟨proof⟩

lemma *induct-conj-curry*: $(\text{induct-conj} \ A \ B ==> \text{PROP} \ C) == (A ==> B ==>$
 $\text{PROP} \ C)$
 ⟨proof⟩

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

lemma *induct-trueI*: *induct-true*
 ⟨proof⟩

Method setup.

⟨ML⟩

Pre-simplification of induction and cases rules

lemma *[induct-simp]*: $(\llbracket x. \text{induct-equal } x \ t \implies \text{PROP } P \ x \rrbracket == \text{PROP } P \ t$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $(\llbracket x. \text{induct-equal } t \ x \implies \text{PROP } P \ x \rrbracket == \text{PROP } P \ t$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $(\text{induct-false} \implies P) == \text{Trueprop induct-true}$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $(\text{induct-true} \implies \text{PROP } P) == \text{PROP } P$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $(\text{PROP } P \implies \text{induct-true}) == \text{Trueprop induct-true}$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $(\llbracket x. \text{induct-true} \rrbracket == \text{Trueprop induct-true}$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $\text{induct-implies induct-true } P == P$
 $\langle \text{proof} \rangle$

lemma *[induct-simp]*: $(x = x) = \text{True}$
 $\langle \text{proof} \rangle$

hide-const *induct-forall induct-implies induct-equal induct-conj induct-true induct-false*
 $\langle \text{ML} \rangle$

2.3.5 Coherent logic

$\langle \text{ML} \rangle$

2.3.6 Reorienting equalities

$\langle \text{ML} \rangle$

2.4 Other simple lemmas and lemma duplicates

lemma *ex1-eq [iff]*: $\text{EX! } x. x = t \text{ EX! } x. t = x$
 $\langle \text{proof} \rangle$

lemma *choice-eq*: $(\text{ALL } x. \text{EX! } y. P \ x \ y) = (\text{EX! } f. \text{ALL } x. P \ x \ (f \ x))$
 $\langle \text{proof} \rangle$

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

lemma *nnf-simps*:

$(\neg(P \wedge Q)) = (\neg P \vee \neg Q) \ (\neg(P \vee Q)) = (\neg P \wedge \neg Q) \ (P \longrightarrow Q) = (\neg P \vee$
 $Q)$
 $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q)) \ (\neg(P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$

$(\neg \neg(P)) = P$
 $\langle proof \rangle$

2.5 Basic ML bindings

$\langle ML \rangle$

2.6 Code generator setup

2.6.1 SML code generator setup

$\langle ML \rangle$

```
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 -))
```

$\langle ML \rangle$

2.6.2 Generic code generator preprocessor setup

$\langle ML \rangle$

2.6.3 Equality

```
class eq =
  fixes eq :: 'a ⇒ 'a ⇒ bool
  assumes eq-equals: eq x y ⟷ x = y
begin
```

lemma *eq* [*code-unfold*, *code-inline del*]: $eq = (op =)$
 $\langle proof \rangle$

lemma *eq-refl*: $eq\ x\ x \longleftrightarrow True$
 $\langle proof \rangle$

lemma *equals-eq*: $(op =) \equiv eq$
 $\langle proof \rangle$

declare *equals-eq* [*symmetric*, *code-post*]

end

declare *equals-eq* [*code*]

$\langle ML \rangle$

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 $(P \implies R) \equiv Trueprop\ (P \longrightarrow R) \langle proof \rangle$

lemma [*code*]:
shows $False \wedge P \longleftrightarrow False$
and $True \wedge P \longleftrightarrow P$
and $P \wedge False \longleftrightarrow False$
and $P \wedge True \longleftrightarrow P \langle proof \rangle$

lemma [*code*]:
shows $False \vee P \longleftrightarrow P$
and $True \vee P \longleftrightarrow True$
and $P \vee False \longleftrightarrow P$
and $P \vee True \longleftrightarrow True \langle proof \rangle$

lemma [*code*]:
shows $(False \longrightarrow P) \longleftrightarrow True$
and $(True \longrightarrow P) \longleftrightarrow P$
and $(P \longrightarrow False) \longleftrightarrow \neg P$

```

and ( $P \longrightarrow \text{True}$ )  $\longleftrightarrow \text{True}$   $\langle \text{proof} \rangle$ 

instantiation itself :: (type) eq
begin

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

instance  $\langle \text{proof} \rangle$ 

end

lemma eq-itself-code [code]:
  eq-class.eq TYPE('a) TYPE('a)  $\longleftrightarrow \text{True}$ 
   $\langle \text{proof} \rangle$ 

Equality

declare simp-thms(6) [code nbe]

 $\langle \text{ML} \rangle$ 

lemma equals-alias-cert: OFCLASS('a, eq-class)  $\equiv ((\text{op} = :: 'a \Rightarrow 'a \Rightarrow \text{bool}) \equiv$ 
eq) (is ?ofclass  $\equiv$  ?eq)
 $\langle \text{proof} \rangle$ 

 $\langle \text{ML} \rangle$ 

hide-const (open) eq

Cases

lemma Let-case-cert:
  assumes CASE  $\equiv (\lambda x. \text{Let } x \text{ } f)$ 
  shows CASE x  $\equiv f \text{ } x$ 
   $\langle \text{proof} \rangle$ 

lemma If-case-cert:
  assumes CASE  $\equiv (\lambda b. \text{If } b \text{ } f \text{ } g)$ 
  shows (CASE True  $\equiv f$ ) &&& (CASE False  $\equiv g$ )
   $\langle \text{proof} \rangle$ 

 $\langle \text{ML} \rangle$ 

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*

$\langle ML \rangle$

2.7 Counterexample Search Units

2.7.1 Quickcheck

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

2.7.2 Nitpick setup

$\langle ML \rangle$

2.8 Preprocessing for the predicate compiler

$\langle ML \rangle$

2.9 Legacy tactics and ML bindings

$\langle ML \rangle$

`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* :: $'a \Rightarrow 'a \Rightarrow bool$

`and` *less* :: $'a \Rightarrow 'a \Rightarrow bool$

`begin`

`notation`

less-eq (*op* \leq) `and`

less-eq ((*-/* \leq *-*) $[51, 51]$ 50) `and`

less (*op* $<$) `and`

less ((*-/* $<$ *-*) $[51, 51]$ 50)

`notation` (*xsymbols*)

less-eq (*op* \leq) `and`

less-eq ((*-/* \leq *-*) $[51, 51]$ 50)

notation (*HTML output*)
less-eq (*op* \leq) **and**
less-eq (*(-/* \leq *-)* [51, 51] 50)

abbreviation (*input*)
greater-eq (**infix** \geq 50) **where**
 $x \geq y \equiv y \leq x$

notation (*input*)
greater-eq (**infix** \geq 50)

abbreviation (*input*)
greater (**infix** $>$ 50) **where**
 $x > y \equiv y < x$

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.
 $\langle proof \rangle$

lemma *less-irrefl* [*iff*]: $\neg x < x$
 $\langle proof \rangle$

lemma *less-imp-le*: $x < y \implies x \leq y$
 $\langle proof \rangle$

Asymmetry.

lemma *less-not-sym*: $x < y \implies \neg (y < x)$
 $\langle proof \rangle$

lemma *less-asy*: $x < y \implies (\neg P \implies y < x) \implies P$
 $\langle proof \rangle$

Transitivity.

lemma *less-trans*: $x < y \implies y < z \implies x < z$
 $\langle proof \rangle$

lemma *le-less-trans*: $x \leq y \implies y < z \implies x < z$
 $\langle proof \rangle$

lemma *less-le-trans*: $x < y \implies y \leq z \implies x < z$
 $\langle \text{proof} \rangle$

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

lemma *less-imp-not-less*: $x < y \implies (\neg y < x) \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

lemma *less-imp-triv*: $x < y \implies (y < x \longrightarrow P) \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

Transitivity rules for calculational reasoning

lemma *less-asym'*: $a < b \implies b < a \implies P$
 $\langle \text{proof} \rangle$

Dual order

lemma *dual-preorder*:
 $\text{class.preorder } (op \geq) (op >)$
 $\langle \text{proof} \rangle$

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*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
 $\langle \text{proof} \rangle$

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x = y$
 — NOT suitable for iff, since it can cause PROOF FAILED.
 $\langle \text{proof} \rangle$

lemma *le-imp-less-or-eq*: $x \leq y \implies x < y \vee x = y$
 $\langle \text{proof} \rangle$

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

lemma *less-imp-not-eq*: $x < y \implies (x = y) \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *less-imp-not-eq2*: $x < y \implies (y = x) \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

Transitivity rules for calculational reasoning

lemma *neq-le-trans*: $a \neq b \implies a \leq b \implies a < b$

$\langle proof \rangle$

lemma *le-neq-trans*: $a \leq b \implies a \neq b \implies a < b$
 $\langle proof \rangle$

Asymmetry.

lemma *eq-iff*: $x = y \iff x \leq y \wedge y \leq x$
 $\langle proof \rangle$

lemma *antisym-conv*: $y \leq x \implies x \leq y \iff x = y$
 $\langle proof \rangle$

lemma *less-imp-neq*: $x < y \implies x \neq y$
 $\langle proof \rangle$

Least value operator

definition (*in ord*)
 $Least :: ('a \Rightarrow bool) \Rightarrow 'a$ (**binder** *LEAST* 10) **where**
 $Least\ P = (THE\ x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \leq y))$

lemma *Least-equality*:
assumes $P\ x$
and $\bigwedge y.\ P\ y \implies x \leq y$
shows $Least\ P = x$
 $\langle proof \rangle$

lemma *LeastI2-order*:
assumes $P\ x$
and $\bigwedge y.\ P\ y \implies x \leq y$
and $\bigwedge x.\ P\ x \implies \forall y.\ P\ y \longrightarrow x \leq y \implies Q\ x$
shows $Q\ (Least\ P)$
 $\langle proof \rangle$

Dual order

lemma *dual-order*:
 $class.order\ (op\ \geq)\ (op\ >)$
 $\langle proof \rangle$

end

3.4 Linear (total) orders

class *linorder* = *order* +
assumes *linear*: $x \leq y \vee y \leq x$
begin

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

lemma *le-less-linear*: $x \leq y \vee y < x$
 $\langle \text{proof} \rangle$

lemma *le-cases* [*case-names le ge*]:
 $(x \leq y \implies P) \implies (y \leq x \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *linorder-cases* [*case-names less equal greater*]:
 $(x < y \implies P) \implies (x = y \implies P) \implies (y < x \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *not-less*: $\neg x < y \longleftrightarrow y \leq x$
 $\langle \text{proof} \rangle$

lemma *not-less-iff-gr-or-eq*:
 $\neg(x < y) \longleftrightarrow (x > y \mid x = y)$
 $\langle \text{proof} \rangle$

lemma *not-le*: $\neg x \leq y \longleftrightarrow y < x$
 $\langle \text{proof} \rangle$

lemma *neq-iff*: $x \neq y \longleftrightarrow x < y \vee y < x$
 $\langle \text{proof} \rangle$

lemma *neqE*: $x \neq y \implies (x < y \implies R) \implies (y < x \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *antisym-conv1*: $\neg x < y \implies x \leq y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *antisym-conv2*: $x \leq y \implies \neg x < y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *antisym-conv3*: $\neg y < x \implies \neg x < y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *leI*: $\neg x < y \implies y \leq x$
 $\langle \text{proof} \rangle$

lemma *leD*: $y \leq x \implies \neg x < y$
 $\langle \text{proof} \rangle$

lemma *not-leE*: $\neg y \leq x \implies x < y$
 $\langle \text{proof} \rangle$

Dual order

lemma *dual-linorder*:
 $\text{class.linorder } (op \geq) (op >)$

$\langle proof \rangle$

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 $min-le-iff-disj$:
 $min\ x\ y \leq z \longleftrightarrow x \leq z \vee y \leq z$
 $\langle proof \rangle$

lemma $le-max-iff-disj$:
 $z \leq max\ x\ y \longleftrightarrow z \leq x \vee z \leq y$
 $\langle proof \rangle$

lemma $min-less-iff-disj$:
 $min\ x\ y < z \longleftrightarrow x < z \vee y < z$
 $\langle proof \rangle$

lemma $less-max-iff-disj$:
 $z < max\ x\ y \longleftrightarrow z < x \vee z < y$
 $\langle proof \rangle$

lemma $min-less-iff-conj$ [simp]:
 $z < min\ x\ y \longleftrightarrow z < x \wedge z < y$
 $\langle proof \rangle$

lemma $max-less-iff-conj$ [simp]:
 $max\ x\ y < z \longleftrightarrow x < z \wedge y < z$
 $\langle proof \rangle$

lemma $split-min$ [no-atp]:
 $P\ (min\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ i) \wedge (\neg\ i \leq j \longrightarrow P\ j)$
 $\langle proof \rangle$

lemma $split-max$ [no-atp]:
 $P\ (max\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ j) \wedge (\neg\ i \leq j \longrightarrow P\ i)$
 $\langle proof \rangle$

end

Explicit dictionaries for code generation

lemma $min-ord-min$ [code, code-unfold, code-inline del]:
 $min = ord.min\ (op\ \leq)$
 $\langle proof \rangle$

declare $ord.min-def$ [code]

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

max = *ord.max* (*op* \leq)

<proof>

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

3.5 Reasoning tools setup

<ML>

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* \Rightarrow '*a* \Rightarrow *bool op* \leq *op* $<$]

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

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

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

declare *eq-refl* [*order add eqD1*: *order op* = :: '*a* \Rightarrow '*a* \Rightarrow *bool op* \leq *op* $<$]

declare *sym* [*THEN eq-refl*, *order add eqD2*: *order op* = :: '*a* \Rightarrow '*a* \Rightarrow *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 *neg-le-trans* [*order add neg-le-trans*: *order op* = :: '*a* \Rightarrow '*a* \Rightarrow *bool op* \leq *op* $<$]

```

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

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

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

end

context linorder
begin

declare [[order del: order op = :: 'a => 'a => bool op <= op <]]

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

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

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

declare not-less [THEN iffD2, order add not-lessI: linorder op = :: 'a => 'a =>
bool op <= op <]

declare not-le [THEN iffD2, order add not-leI: linorder op = :: 'a => 'a => 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

```

$op \leq op <$]

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

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

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

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

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

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

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

end

$\langle ML \rangle$

3.6 Bounded quantifiers

syntax

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

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

syntax (*xsymbols*)

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

-All-greater :: [idt, 'a, bool] $\Rightarrow bool$ (($\exists \forall \text{ ->-. / -}$) [0, 0, 10] 10)
 -Ex-greater :: [idt, 'a, bool] $\Rightarrow bool$ (($\exists \exists \text{ ->-. / -}$) [0, 0, 10] 10)

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

translations

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

$\langle ML \rangle$

3.7 Transitivity reasoning

context ord

begin

lemma ord-le-eq-trans: $a \leq b \Longrightarrow b = c \Longrightarrow a \leq c$
 $\langle proof \rangle$

lemma ord-eq-le-trans: $a = b \Longrightarrow b \leq c \Longrightarrow a \leq c$
 $\langle proof \rangle$

lemma ord-less-eq-trans: $a < b \Longrightarrow b = c \Longrightarrow a < c$
 $\langle proof \rangle$

lemma ord-eq-less-trans: $a = b \Longrightarrow b < c \Longrightarrow a < c$
 $\langle proof \rangle$

end

lemma *order-less-subst2*: $(a::'a::order) < b \implies f\ b < (c::'c::order) \implies$
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies f\ a < c$
 $\langle proof \rangle$

lemma *order-less-subst1*: $(a::'a::order) < f\ b \implies (b::'b::order) < c \implies$
 $(!!x\ y. x < y \implies f\ x < f\ y) \implies a < f\ c$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

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

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*:

$a = b \implies b > c \implies a > c$
 $a > b \implies b = c \implies a > c$
 $a = b \implies b \geq c \implies a \geq c$
 $a \geq b \implies b = c \implies a \geq c$
 $(x::'a::\text{order}) \geq y \implies y \geq x \implies x = y$
 $(x::'a::\text{order}) \geq y \implies y \geq z \implies x \geq z$
 $(x::'a::\text{order}) > y \implies y \geq z \implies x > z$
 $(x::'a::\text{order}) \geq y \implies y > z \implies x > z$
 $(a::'a::\text{order}) > b \implies b > a \implies P$
 $(x::'a::\text{order}) > y \implies y > z \implies x > z$
 $(a::'a::\text{order}) \geq b \implies a \sim b \implies a > b$
 $(a::'a::\text{order}) \sim b \implies a \geq b \implies a > b$
 $a = f b \implies b > c \implies (!x y. x > y \implies f x > f y) \implies a > f c$
 $a > b \implies f b = c \implies (!x y. x > y \implies f x > f y) \implies f a > c$
 $a = f b \implies b \geq c \implies (!x y. x \geq y \implies f x \geq f y) \implies a \geq f c$
 $a \geq b \implies f b = c \implies (!x y. x \geq y \implies f x \geq f y) \implies f a \geq c$
 $\langle \text{proof} \rangle$

lemma *xt2*:

$(a::'a::\text{order}) \geq f b \implies b \geq c \implies (!x y. x \geq y \implies f x \geq f y) \implies$
 $a \geq f c$
 $\langle \text{proof} \rangle$

lemma *xt3*: $(a::'a::order) \geq b \implies (f\ b::'b::order) \geq c \implies$
 $(!!x\ y. x \geq y \implies f\ x \geq f\ y) \implies f\ a \geq c$
 $\langle proof \rangle$

lemma *xt4*: $(a::'a::order) > f\ b \implies (b::'b::order) \geq c \implies$
 $(!!x\ y. x \geq y \implies f\ x \geq f\ y) \implies a > f\ c$
 $\langle proof \rangle$

lemma *xt5*: $(a::'a::order) > b \implies (f\ b::'b::order) \geq c \implies$
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies f\ a > c$
 $\langle proof \rangle$

lemma *xt6*: $(a::'a::order) \geq f\ b \implies b > c \implies$
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies a > f\ c$
 $\langle proof \rangle$

lemma *xt7*: $(a::'a::order) \geq b \implies (f\ b::'b::order) > c \implies$
 $(!!x\ y. x \geq y \implies f\ x \geq f\ y) \implies f\ a > c$
 $\langle proof \rangle$

lemma *xt8*: $(a::'a::order) > f\ b \implies (b::'b::order) > c \implies$
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies a > f\ c$
 $\langle proof \rangle$

lemma *xt9*: $(a::'a::order) > b \implies (f\ b::'b::order) > c \implies$
 $(!!x\ y. x > y \implies f\ x > f\ y) \implies f\ a > c$
 $\langle proof \rangle$

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

lemma *monoI* [*intro?*]:
fixes $f :: 'a \Rightarrow 'b::order$
shows $(\bigwedge x\ y. x \leq y \implies f\ x \leq f\ y) \implies mono\ f$
 $\langle proof \rangle$

lemma *monoD* [*dest?*]:
fixes $f :: 'a \Rightarrow 'b::order$
shows $mono\ f \implies x \leq y \implies f\ x \leq f\ y$
 $\langle proof \rangle$

definition *strict-mono* :: ('a \Rightarrow 'b::order) \Rightarrow bool **where**
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
 \langle proof \rangle

lemma *strict-monoD* [dest?]:
strict-mono f $\Longrightarrow x < y \Longrightarrow f\ x < f\ y$
 \langle proof \rangle

lemma *strict-mono-mono* [dest?]:
 assumes *strict-mono* f
 shows *mono* f
 \langle proof \rangle

end

context *linorder*
begin

lemma *strict-mono-eq*:
 assumes *strict-mono* f
 shows $f\ x = f\ y \longleftrightarrow x = y$
 \langle proof \rangle

lemma *strict-mono-less-eq*:
 assumes *strict-mono* f
 shows $f\ x \leq f\ y \longleftrightarrow x \leq y$
 \langle proof \rangle

lemma *strict-mono-less*:
 assumes *strict-mono* f
 shows $f\ x < f\ y \longleftrightarrow x < y$
 \langle proof \rangle

lemma *min-of-mono*:
 fixes f :: 'a \Rightarrow 'b::linorder
 shows *mono* f $\Longrightarrow \min\ (f\ m)\ (f\ n) = f\ (\min\ m\ n)$
 \langle proof \rangle

lemma *max-of-mono*:
 fixes f :: 'a \Rightarrow 'b::linorder
 shows *mono* f $\Longrightarrow \max\ (f\ m)\ (f\ n) = f\ (\max\ m\ n)$
 \langle proof \rangle

end

lemma *min-leastL*: $(!!x. \text{least} \leq x) \implies \min \text{least } x = \text{least}$
 $\langle \text{proof} \rangle$

lemma *max-leastL*: $(!!x. \text{least} \leq x) \implies \max \text{least } x = x$
 $\langle \text{proof} \rangle$

lemma *min-leastR*: $(\bigwedge x :: 'a :: \text{order}. \text{least} \leq x) \implies \min x \text{ least} = \text{least}$
 $\langle \text{proof} \rangle$

lemma *max-leastR*: $(\bigwedge x :: 'a :: \text{order}. \text{least} \leq x) \implies \max x \text{ least} = x$
 $\langle \text{proof} \rangle$

3.9 Top and bottom elements

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

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

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 $y \leq z$
 $\langle \text{proof} \rangle$

lemma *dense-le-bounded*:
 fixes $x\ y\ z :: 'a$
 assumes $x < y$
 assumes *: $\bigwedge w. [x < w ; w < y] \implies w \leq z$
 shows $y \leq z$
 $\langle \text{proof} \rangle$

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 $\text{Least}I: P\ (\text{LEAST}\ x.\ P\ x)$ **and** $\text{Least-le}: (\text{LEAST}\ x.\ P\ x) \leq k$
 $\langle \text{proof} \rangle$

lemma *LeastI-ex*: $\exists x.\ P\ x \implies P\ (\text{Least}\ P)$
 $\langle \text{proof} \rangle$

lemma *LeastI2*:

$P\ a \implies (\bigwedge x.\ P\ x \implies Q\ x) \implies Q\ (\text{Least}\ P)$
 $\langle \text{proof} \rangle$

lemma *LeastI2-ex*:

$\exists a.\ P\ a \implies (\bigwedge x.\ P\ x \implies Q\ x) \implies Q\ (\text{Least}\ P)$
 $\langle \text{proof} \rangle$

lemma *not-less-Least*: $k < (\text{LEAST}\ x.\ P\ x) \implies \neg P\ k$
 $\langle \text{proof} \rangle$

end

3.12 Order on bool

instantiation $\text{bool} :: \{\text{order}, \text{top}, \text{bot}\}$

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: $\text{bot} = \text{False}$

instance $\langle \text{proof} \rangle$

end

lemma *le-boolI*: $(P \implies Q) \implies P \leq Q$
 $\langle \text{proof} \rangle$

lemma *le-boolI'*: $P \longrightarrow Q \implies P \leq Q$
 $\langle \text{proof} \rangle$

lemma *le-boolE*: $P \leq Q \implies P \implies (Q \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *le-boolD*: $P \leq Q \implies P \longrightarrow Q$
 $\langle \text{proof} \rangle$

lemma *bot-boolE*: $\text{bot} \implies P$
 $\langle \text{proof} \rangle$

lemma *top-boolI*: top
 $\langle \text{proof} \rangle$

lemma [*code*]:
 $\text{False} \leq b \longleftrightarrow \text{True}$
 $\text{True} \leq b \longleftrightarrow b$
 $\text{False} < b \longleftrightarrow b$
 $\text{True} < b \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

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 $\langle \text{proof} \rangle$

end

instance *fun* :: (*type*, *preorder*) *preorder* $\langle \text{proof} \rangle$

instance *fun* :: (*type*, *order*) *order* $\langle \text{proof} \rangle$

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

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

instance $\langle \text{proof} \rangle$

end

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

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

instance $\langle proof \rangle$

end

lemma *le-funI*: $(\bigwedge x. f\ x \leq g\ x) \implies f \leq g$
 $\langle proof \rangle$

lemma *le-funE*: $f \leq g \implies (f\ x \leq g\ x \implies P) \implies P$
 $\langle proof \rangle$

lemma *le-funD*: $f \leq g \implies f\ x \leq g\ x$
 $\langle proof \rangle$

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>

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>

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

<ML>

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  $\Rightarrow$  'a  $\Rightarrow$  '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*]:

$b * (a * c) = a * (b * c)$
 $\langle proof \rangle$

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* $\langle proof \rangle$

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* $0 f = f 0$

$\langle proof \rangle$

lemma *Let-1* [*simp*]: *Let* $1 f = f 1$

$\langle proof \rangle$

$\langle ML \rangle$

class *plus* =

fixes $plus :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** + 65)

class *minus* =

fixes $minus :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** - 65)

class *uminus* =

```

fixes uminus :: 'a ⇒ 'a  (– - [81] 80)

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

⟨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⟩

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

sublocale ab-semigroup-add < add!: abel-semigroup plus ⟨proof⟩

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⟩

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⟩

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* \langle *proof* \rangle

lemma *zero-reorient*: $0 = x \longleftrightarrow x = 0$
 \langle *proof* \rangle

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

sublocale *comm-monoid-add* < *add!*: *comm-monoid plus 0* \langle *proof* \rangle

subclass (**in** *comm-monoid-add*) *monoid-add* \langle *proof* \rangle

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* \langle *proof* \rangle

lemma *one-reorient*: $1 = x \longleftrightarrow x = 1$
 \langle *proof* \rangle

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

sublocale *comm-monoid-mult* < *mult!*: *comm-monoid times 1* \langle *proof* \rangle

subclass (**in** *comm-monoid-mult*) *monoid-mult* \langle *proof* \rangle

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*]:
 $a + b = a + c \longleftrightarrow b = c$
 \langle *proof* \rangle

lemma *add-right-cancel* [*simp*]:
 $b + a = c + a \longleftrightarrow b = c$
 \langle *proof* \rangle

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>

```

```

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>

```

```

lemmas equals-zero-I = minus-unique

```

```

lemma minus-zero [simp]:  $- 0 = 0$ 
  <proof>

```

```

lemma minus-minus [simp]:  $- (- a) = a$ 
  <proof>

```

```

lemma right-minus [simp]:  $a + - a = 0$ 
  <proof>

```

```

lemma minus-add-cancel:  $- a + (a + b) = b$ 
  <proof>

```

```

lemma add-minus-cancel:  $a + (- a + b) = b$ 
  <proof>

```

```

lemma minus-add:  $-(a + b) = - b + - a$ 
  <proof>

```

```

lemma right-minus-eq:  $a - b = 0 \iff a = b$ 
  <proof>

```

```

lemma diff-self [simp]:  $a - a = 0$ 
  <proof>

```

```

lemma diff-0 [simp]:  $0 - a = - a$ 

```

$\langle proof \rangle$

lemma *diff-0-right* [*simp*]: $a - 0 = a$
 $\langle proof \rangle$

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + b$
 $\langle proof \rangle$

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

lemma *neg-equal-0-iff-equal* [*simp*]:
 $- a = 0 \longleftrightarrow a = 0$
 $\langle proof \rangle$

lemma *neg-0-equal-iff-equal* [*simp*]:
 $0 = - a \longleftrightarrow 0 = a$
 $\langle proof \rangle$

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*:
 $a = - b \longleftrightarrow b = - a$
 $\langle proof \rangle$

lemma *minus-equation-iff*:
 $- a = b \longleftrightarrow - b = a$
 $\langle proof \rangle$

lemma *diff-add-cancel*: $a - b + b = a$
 $\langle proof \rangle$

lemma *add-diff-cancel*: $a + b - b = a$
 $\langle proof \rangle$

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

lemma *eq-neg-iff-add-eq-0*: $a = - b \longleftrightarrow a + b = 0$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

subclass *cancel-comm-monoid-add*

$\langle proof \rangle$

lemma *uminus-add-conv-diff* [*algebra-simps*, *field-simps*]:

$$- a + b = b - a$$

$\langle proof \rangle$

lemma *minus-add-distrib* [*simp*]:

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

$\langle proof \rangle$

lemma *minus-diff-eq* [*simp*]:

$$-(a - b) = b - a$$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *diff-eq-eq* [*algebra-simps*, *field-simps*]: $a - b = c \longleftrightarrow a = c + b$

$\langle proof \rangle$

lemma *eq-diff-eq* [*algebra-simps*, *field-simps*]: $a = c - b \longleftrightarrow a + b = c$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$

$\langle proof \rangle$

lemma *diff-eq-0-iff-eq* [*simp*, *no-atp*]: $a - b = 0 \longleftrightarrow a = b$

$\langle proof \rangle$

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:
   $a \leq b \implies a + c \leq b + c$ 
 $\langle$ proof $\rangle$ 
```

non-strict, in both arguments

```
lemma add-mono:
   $a \leq b \implies c \leq d \implies a + c \leq b + d$ 
 $\langle$ proof $\rangle$ 
```

```
end
```

```
class ordered-cancel-ab-semigroup-add =
  ordered-ab-semigroup-add + cancel-ab-semigroup-add
begin
```

```
lemma add-strict-left-mono:
   $a < b \implies c + a < c + b$ 
 $\langle$ proof $\rangle$ 
```

```
lemma add-strict-right-mono:
   $a < b \implies a + c < b + c$ 
 $\langle$ proof $\rangle$ 
```

Strict monotonicity in both arguments

```
lemma add-strict-mono:
   $a < b \implies c < d \implies a + c < b + d$ 
 $\langle$ proof $\rangle$ 
```

```
lemma add-less-le-mono:
   $a < b \implies c \leq d \implies a + c < b + d$ 
 $\langle$ proof $\rangle$ 
```

```
lemma add-le-less-mono:
   $a \leq b \implies c < d \implies a + c < b + d$ 
 $\langle$ proof $\rangle$ 
```

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>

lemma *add-less-imp-less-right*:
 $a + c < b + c \implies a < b$
<proof>

lemma *add-less-cancel-left* [*simp*]:
 $c + a < c + b \longleftrightarrow a < b$
<proof>

lemma *add-less-cancel-right* [*simp*]:
 $a + c < b + c \longleftrightarrow a < b$
<proof>

lemma *add-le-cancel-left* [*simp*]:
 $c + a \leq c + b \longleftrightarrow a \leq b$
<proof>

lemma *add-le-cancel-right* [*simp*]:
 $a + c \leq b + c \longleftrightarrow a \leq b$
<proof>

lemma *add-le-imp-le-right*:
 $a + c \leq b + c \implies a \leq b$
<proof>

lemma *max-add-distrib-left*:
 $\max x \ y + z = \max (x + z) (y + z)$
<proof>

lemma *min-add-distrib-left*:
 $\min x \ y + z = \min (x + z) (y + z)$
<proof>

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>

lemma add-pos-pos:
  assumes  $0 < a$  and  $0 < b$  shows  $0 < a + b$ 
  <proof>

lemma add-nonneg-pos:
  assumes  $0 \leq a$  and  $0 < b$  shows  $0 < a + b$ 
  <proof>

lemma add-nonneg-nonneg [simp]:
  assumes  $0 \leq a$  and  $0 \leq b$  shows  $0 \leq a + b$ 
  <proof>

lemma add-neg-nonpos:
  assumes  $a < 0$  and  $b \leq 0$  shows  $a + b < 0$ 
  <proof>

lemma add-neg-neg:
  assumes  $a < 0$  and  $b < 0$  shows  $a + b < 0$ 
  <proof>

lemma add-nonpos-neg:
  assumes  $a \leq 0$  and  $b < 0$  shows  $a + b < 0$ 
  <proof>

lemma add-nonpos-nonpos:
  assumes  $a \leq 0$  and  $b \leq 0$  shows  $a + b \leq 0$ 
  <proof>

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>

end

class ordered-ab-group-add =
  ab-group-add + ordered-ab-semigroup-add
begin

```

subclass *ordered-cancel-ab-semigroup-add* $\langle \text{proof} \rangle$

subclass *ordered-ab-semigroup-add-imp-le*
 $\langle \text{proof} \rangle$

subclass *ordered-comm-monoid-add* $\langle \text{proof} \rangle$

lemma *max-diff-distrib-left*:
shows $\max x y - z = \max (x - z) (y - z)$
 $\langle \text{proof} \rangle$

lemma *min-diff-distrib-left*:
shows $\min x y - z = \min (x - z) (y - z)$
 $\langle \text{proof} \rangle$

lemma *le-imp-neg-le*:
assumes $a \leq b$ **shows** $-b \leq -a$
 $\langle \text{proof} \rangle$

lemma *neg-le-iff-le* [simp]: $-b \leq -a \longleftrightarrow a \leq b$
 $\langle \text{proof} \rangle$

lemma *neg-le-0-iff-le* [simp]: $-a \leq 0 \longleftrightarrow 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *neg-0-le-iff-le* [simp]: $0 \leq -a \longleftrightarrow a \leq 0$
 $\langle \text{proof} \rangle$

lemma *neg-less-iff-less* [simp]: $-b < -a \longleftrightarrow a < b$
 $\langle \text{proof} \rangle$

lemma *neg-less-0-iff-less* [simp]: $-a < 0 \longleftrightarrow 0 < a$
 $\langle \text{proof} \rangle$

lemma *neg-0-less-iff-less* [simp]: $0 < -a \longleftrightarrow a < 0$
 $\langle \text{proof} \rangle$

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < -b \longleftrightarrow b < -a$
 $\langle \text{proof} \rangle$

lemma *minus-less-iff*: $-a < b \longleftrightarrow -b < a$
 $\langle \text{proof} \rangle$

lemma *le-minus-iff*: $a \leq -b \longleftrightarrow b \leq -a$
 $\langle \text{proof} \rangle$

lemma *minus-le-iff*: $-a \leq b \longleftrightarrow -b \leq a$

$\langle proof \rangle$

lemma *less-iff-diff-less-0*: $a < b \iff a - b < 0$
 $\langle proof \rangle$

lemma *diff-less-eq[algebra-simps, field-simps]*: $a - b < c \iff a < c + b$
 $\langle proof \rangle$

lemma *less-diff-eq[algebra-simps, field-simps]*: $a < c - b \iff a + b < c$
 $\langle proof \rangle$

lemma *diff-le-eq[algebra-simps, field-simps]*: $a - b \leq c \iff a \leq c + b$
 $\langle proof \rangle$

lemma *le-diff-eq[algebra-simps, field-simps]*: $a \leq c - b \iff a + b \leq c$
 $\langle proof \rangle$

lemma *le-iff-diff-le-0*: $a \leq b \iff a - b \leq 0$
 $\langle proof \rangle$

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* $\langle proof \rangle$

subclass *ordered-ab-semigroup-add-imp-le*
 $\langle proof \rangle$

end

class *linordered-ab-group-add* = *linorder* + *ordered-ab-group-add*
begin

subclass *linordered-cancel-ab-semigroup-add* $\langle proof \rangle$

lemma *neg-less-eq-nonneg [simp]*:
 $-a \leq a \iff 0 \leq a$
 $\langle proof \rangle$

lemma *neg-less-nonneg [simp]*:
 $-a < a \iff 0 < a$
 $\langle proof \rangle$

lemma *less-eq-neg-nonpos* [simp]:

$$a \leq -a \longleftrightarrow a \leq 0$$

$\langle \text{proof} \rangle$

lemma *equal-neg-zero* [simp]:

$$a = -a \longleftrightarrow a = 0$$

$\langle \text{proof} \rangle$

lemma *neg-equal-zero* [simp]:

$$-a = a \longleftrightarrow a = 0$$

$\langle \text{proof} \rangle$

lemma *double-zero* [simp]:

$$a + a = 0 \longleftrightarrow a = 0$$

$\langle \text{proof} \rangle$

lemma *double-zero-sym* [simp]:

$$0 = a + a \longleftrightarrow a = 0$$

$\langle \text{proof} \rangle$

lemma *zero-less-double-add-iff-zero-less-single-add* [simp]:

$$0 < a + a \longleftrightarrow 0 < a$$

$\langle \text{proof} \rangle$

lemma *zero-le-double-add-iff-zero-le-single-add* [simp]:

$$0 \leq a + a \longleftrightarrow 0 \leq a$$

$\langle \text{proof} \rangle$

lemma *double-add-less-zero-iff-single-add-less-zero* [simp]:

$$a + a < 0 \longleftrightarrow a < 0$$

$\langle \text{proof} \rangle$

lemma *double-add-le-zero-iff-single-add-le-zero* [simp]:

$$a + a \leq 0 \longleftrightarrow a \leq 0$$

$\langle \text{proof} \rangle$

lemma *le-minus-self-iff*:

$$a \leq -a \longleftrightarrow a \leq 0$$

$\langle \text{proof} \rangle$

lemma *minus-le-self-iff*:

$$-a \leq a \longleftrightarrow 0 \leq a$$

$\langle \text{proof} \rangle$

lemma *minus-max-eq-min*:

$$-\max x y = \min (-x) (-y)$$

$\langle \text{proof} \rangle$

lemma *minus-min-eq-max*:

– $\min x \ y = \max (-x) \ (-y)$
 $\langle \text{proof} \rangle$

end

context *ordered-comm-monoid-add*
begin

lemma *add-increasing*:
 $0 \leq a \implies b \leq c \implies b \leq a + c$
 $\langle \text{proof} \rangle$

lemma *add-increasing2*:
 $0 \leq c \implies b \leq a \implies b \leq a + c$
 $\langle \text{proof} \rangle$

lemma *add-strict-increasing*:
 $0 < a \implies b \leq c \implies b < a + c$
 $\langle \text{proof} \rangle$

lemma *add-strict-increasing2*:
 $0 \leq a \implies b < c \implies b < a + c$
 $\langle \text{proof} \rangle$

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*]: $\text{sgn } 0 = 0$

```

    <proof>

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:  $-|a| \leq 0$ 
  <proof>

lemma abs-of-nonneg [simp]:
  assumes nonneg:  $0 \leq a$  shows  $|a| = a$ 
  <proof>

lemma abs-idempotent [simp]:  $||a|| = |a|$ 
  <proof>

lemma abs-eq-0 [simp]:  $|a| = 0 \iff a = 0$ 
  <proof>

lemma abs-zero [simp]:  $|0| = 0$ 
  <proof>

lemma abs-0-eq [simp, no-atp]:  $0 = |a| \iff a = 0$ 
  <proof>

lemma abs-le-zero-iff [simp]:  $|a| \leq 0 \iff a = 0$ 
  <proof>

lemma zero-less-abs-iff [simp]:  $0 < |a| \iff a \neq 0$ 
  <proof>

lemma abs-not-less-zero [simp]:  $\neg |a| < 0$ 
  <proof>

lemma abs-ge-minus-self:  $-a \leq |a|$ 
  <proof>

lemma abs-minus-commute:
   $|a - b| = |b - a|$ 
  <proof>

lemma abs-of-pos:  $0 < a \implies |a| = a$ 
  <proof>

```

lemma *abs-of-nonpos* [*simp*]:

assumes $a \leq 0$ shows $|a| = -a$
 $\langle proof \rangle$

lemma *abs-of-neg*: $a < 0 \implies |a| = -a$
 $\langle proof \rangle$

lemma *abs-le-D1*: $|a| \leq b \implies a \leq b$
 $\langle proof \rangle$

lemma *abs-le-D2*: $|a| \leq b \implies -a \leq b$
 $\langle proof \rangle$

lemma *abs-le-iff*: $|a| \leq b \iff a \leq b \wedge -a \leq b$
 $\langle proof \rangle$

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$
 $\langle proof \rangle$

lemma *abs-triangle-ineq2-sym*: $|a| - |b| \leq |b - a|$
 $\langle proof \rangle$

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
 $\langle proof \rangle$

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
 $\langle proof \rangle$

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
 $\langle proof \rangle$

lemma *abs-add-abs* [*simp*]:

$||a| + |b|| = |a| + |b|$ (is ?L = ?R)
 $\langle proof \rangle$

end

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: $((x::'a::ab-group-add) + (y + z) = y + u) = (x + z = u)$
 $\langle proof \rangle$

lemma *add-cancel-end*: $(x + (y + z) = y) = (x = - (z::'a::ab-group-add))$
 $\langle proof \rangle$

lemma *less-eqI*: $(x::'a::ordered-ab-group-add) - y = x' - y' \implies (x < y) = (x' < y')$
 $\langle proof \rangle$

lemma *le-eqI*: $(x::'a::\text{ordered-ab-group-add}) - y = x' - y' \implies (y \leq x) = (y' \leq x')$
 <proof>

lemma *eq-eqI*: $(x::'a::\text{ab-group-add}) - y = x' - y' \implies (x = y) = (x' = y')$
 <proof>

lemma *diff-def*: $(x::'a::\text{ab-group-add}) - y == x + (-y)$
 <proof>

lemma *le-add-right-mono*:
 assumes
 $a \leq b + (c::'a::\text{ordered-ab-group-add})$
 $c \leq d$
 shows $a \leq b + d$
 <proof>

4.8 Tools setup

lemma *add-mono-thms-linordered-semiring* [no-atp]:
 fixes $i\ j\ k :: 'a::\text{ordered-ab-semigroup-add}$
 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 $i = j \wedge k = l \implies i + k = j + l$
 <proof>

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 $i < j \wedge k < l \implies i + k < j + l$
 <proof>

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>

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*]:
 $f\ a\ (f\ a\ b) = f\ a\ b$
 $\langle proof \rangle$

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* $\langle proof \rangle$

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  $\geq$ ) (op  $>$ ) sup
   $\langle$ proof $\rangle$ 

```

end

```

class lattice = semilattice-inf + semilattice-sup

```

5.3.1 Intro and elim rules

```

context semilattice-inf
begin

```

```

lemma le-infI1:
   $a \sqsubseteq x \Longrightarrow a \sqcap b \sqsubseteq x$ 
   $\langle$ proof $\rangle$ 

```

```

lemma le-infI2:
   $b \sqsubseteq x \Longrightarrow a \sqcap b \sqsubseteq x$ 
   $\langle$ proof $\rangle$ 

```

```

lemma le-infI:  $x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcap b$ 
   $\langle$ proof $\rangle$ 

```

```

lemma le-infE:  $x \sqsubseteq a \sqcap b \Longrightarrow (x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow P) \Longrightarrow P$ 
   $\langle$ proof $\rangle$ 

```

```

lemma le-inf-iff [simp]:
   $x \sqsubseteq y \sqcap z \longleftrightarrow x \sqsubseteq y \wedge x \sqsubseteq z$ 
   $\langle$ proof $\rangle$ 

```

```

lemma le-iff-inf:
   $x \sqsubseteq y \longleftrightarrow x \sqcap y = x$ 
   $\langle$ proof $\rangle$ 

```

lemma *inf-mono*: $a \sqsubseteq c \implies b \leq d \implies a \sqcap b \sqsubseteq c \sqcap d$
 $\langle proof \rangle$

lemma *mono-inf*:
fixes $f :: 'a \Rightarrow 'b :: semilattice-inf$
shows $mono\ f \implies f\ (A \sqcap B) \sqsubseteq f\ A \sqcap f\ B$
 $\langle proof \rangle$

end

context *semilattice-sup*
begin

lemma *le-supI1*:
 $x \sqsubseteq a \implies x \sqsubseteq a \sqcup b$
 $\langle proof \rangle$

lemma *le-supI2*:
 $x \sqsubseteq b \implies x \sqsubseteq a \sqcup b$
 $\langle proof \rangle$

lemma *le-supI*:
 $a \sqsubseteq x \implies b \sqsubseteq x \implies a \sqcup b \sqsubseteq x$
 $\langle proof \rangle$

lemma *le-supE*:
 $a \sqcup b \sqsubseteq x \implies (a \sqsubseteq x \implies b \sqsubseteq x \implies P) \implies P$
 $\langle proof \rangle$

lemma *le-sup-iff* [*simp*]:
 $x \sqcup y \sqsubseteq z \longleftrightarrow x \sqsubseteq z \wedge y \sqsubseteq z$
 $\langle proof \rangle$

lemma *le-iff-sup*:
 $x \sqsubseteq y \longleftrightarrow x \sqcup y = y$
 $\langle proof \rangle$

lemma *sup-mono*: $a \sqsubseteq c \implies b \leq d \implies a \sqcup b \sqsubseteq c \sqcup d$
 $\langle proof \rangle$

lemma *mono-sup*:
fixes $f :: 'a \Rightarrow 'b :: semilattice-sup$
shows $mono\ f \implies f\ A \sqcup f\ B \sqsubseteq f\ (A \sqcup B)$
 $\langle proof \rangle$

end

5.3.2 Equational laws

sublocale *semilattice-inf* < *inf!*: *semilattice inf*
 ⟨*proof*⟩

context *semilattice-inf*
begin

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
 ⟨*proof*⟩

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
 ⟨*proof*⟩

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
 ⟨*proof*⟩

lemma *inf-idem*: $x \sqcap x = x$
 ⟨*proof*⟩

lemma *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$
 ⟨*proof*⟩

lemma *inf-absorb1*: $x \sqsubseteq y \implies x \sqcap y = x$
 ⟨*proof*⟩

lemma *inf-absorb2*: $y \sqsubseteq x \implies x \sqcap y = y$
 ⟨*proof*⟩

lemmas *inf-aci* = *inf-commute inf-assoc inf-left-commute inf-left-idem*

end

sublocale *semilattice-sup* < *sup!*: *semilattice sup*
 ⟨*proof*⟩

context *semilattice-sup*
begin

lemma *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
 ⟨*proof*⟩

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
 ⟨*proof*⟩

lemma *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
 ⟨*proof*⟩

lemma *sup-idem*: $x \sqcup x = x$
 ⟨*proof*⟩

lemma *sup-left-idem*: $x \sqcup (x \sqcap y) = x \sqcup y$
 $\langle \text{proof} \rangle$

lemma *sup-absorb1*: $y \sqsubseteq x \implies x \sqcup y = x$
 $\langle \text{proof} \rangle$

lemma *sup-absorb2*: $x \sqsubseteq y \implies x \sqcup y = y$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *inf-sup-absorb*: $x \sqcap (x \sqcup y) = x$
 $\langle \text{proof} \rangle$

lemma *sup-inf-absorb*: $x \sqcup (x \sqcap y) = x$
 $\langle \text{proof} \rangle$

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*: $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$
 $\langle \text{proof} \rangle$

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$
 $\langle \text{proof} \rangle$

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 $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
 $\langle \text{proof} \rangle$

lemma *distrib-imp2*:
assumes *D*: $\forall x y z. x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
 $\langle \text{proof} \rangle$

end

5.3.3 Strict order

context *semilattice-inf*
begin

lemma *less-infI1*:
 $a \sqsubseteq x \implies a \sqcap b \sqsubseteq x$
 $\langle \text{proof} \rangle$

lemma *less-infI2*:
 $b \sqsubseteq x \implies a \sqcap b \sqsubseteq x$
 $\langle \text{proof} \rangle$

end

context *semilattice-sup*
begin

lemma *less-supI1*:
 $x \sqsubseteq a \implies x \sqsubseteq a \sqcup b$
 $\langle \text{proof} \rangle$

lemma *less-supI2*:
 $x \sqsubseteq b \implies x \sqsubseteq a \sqcup b$
 $\langle \text{proof} \rangle$

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*:
 $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
 $\langle \text{proof} \rangle$

lemma *inf-sup-distrib1*:
 $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
 $\langle \text{proof} \rangle$

lemma *inf-sup-distrib2*:
 $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
 $\langle \text{proof} \rangle$

```

lemma dual-distrib-lattice:
  class.distrib-lattice (op  $\geq$ ) (op  $>$ ) sup inf
   $\langle proof \rangle$ 

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]:
   $\perp \sqcap x = \perp$ 
   $\langle proof \rangle$ 

lemma inf-bot-right [simp]:
   $x \sqcap \perp = \perp$ 
   $\langle proof \rangle$ 

lemma sup-bot-left [simp]:
   $\perp \sqcup x = x$ 
   $\langle proof \rangle$ 

lemma sup-bot-right [simp]:
   $x \sqcup \perp = x$ 
   $\langle proof \rangle$ 

lemma sup-eq-bot-iff [simp]:
   $x \sqcup y = \perp \iff x = \perp \wedge y = \perp$ 
   $\langle proof \rangle$ 

```

```

end

```

```

class bounded-lattice-top = lattice + top
begin

```

```

lemma sup-top-left [simp]:
   $\top \sqcup x = \top$ 
   $\langle proof \rangle$ 

```

lemma *sup-top-right* [*simp*]:

$$x \sqcup \top = \top$$

<proof>

lemma *inf-top-left* [*simp*]:

$$\top \sqcap x = x$$

<proof>

lemma *inf-top-right* [*simp*]:

$$x \sqcap \top = x$$

<proof>

lemma *inf-eq-top-iff* [*simp*]:

$$x \sqcap y = \top \iff x = \top \wedge y = \top$$

<proof>

end

class *bounded-lattice* = *bounded-lattice-bot* + *bounded-lattice-top*
begin

lemma *dual-bounded-lattice*:

$$\text{class.bounded-lattice } (op \geq) (op >) (op \sqcup) (op \sqcap) \top \perp$$

<proof>

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*:

$$\text{class.boolean-algebra } (\lambda x y. x \sqcup - y) \text{ uminus } (op \geq) (op >) (op \sqcup) (op \sqcap) \top \perp$$

<proof>

lemma *compl-inf-bot*:

$$- x \sqcap x = \perp$$

<proof>

lemma *compl-sup-top*:

$$- x \sqcup x = \top$$

<proof>

lemma *compl-unique*:

assumes $x \sqcap y = \perp$

and $x \sqcup y = \top$

shows $- x = y$

$\langle proof \rangle$

lemma *double-compl* [*simp*]:

– $(- x) = x$

$\langle proof \rangle$

lemma *compl-eq-compl-iff* [*simp*]:

– $x = - y \longleftrightarrow x = y$

$\langle proof \rangle$

lemma *compl-bot-eq* [*simp*]:

– $\perp = \top$

$\langle proof \rangle$

lemma *compl-top-eq* [*simp*]:

– $\top = \perp$

$\langle proof \rangle$

lemma *compl-inf* [*simp*]:

– $(x \sqcap y) = - x \sqcup - y$

$\langle proof \rangle$

lemma *compl-sup* [*simp*]:

– $(x \sqcup y) = - x \sqcap - y$

$\langle proof \rangle$

lemma *compl-mono*:

$x \sqsubseteq y \implies - y \sqsubseteq - x$

$\langle proof \rangle$

lemma *compl-le-compl-iff*:

– $x \leq - y \longleftrightarrow y \leq x$

$\langle proof \rangle$

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 $x \sqcap y = x \triangle y$

$\langle proof \rangle$

lemma (in *semilattice-sup*) *sup-unique*:

fixes f (**infixl** ∇ 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 $x \sqcup y = x \nabla y$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *inf-min*: $\text{inf} = (\text{min} :: 'a :: \{\text{semilattice-inf}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$
 $\langle \text{proof} \rangle$

lemma *sup-max*: $\text{sup} = (\text{max} :: 'a :: \{\text{semilattice-sup}, \text{linorder}\} \Rightarrow 'a \Rightarrow 'a)$
 $\langle \text{proof} \rangle$

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 $\langle \text{proof} \rangle$

end

lemma *sup-boolI1*:

$P \Longrightarrow P \sqcup Q$

$\langle \text{proof} \rangle$

lemma *sup-boolI2*:

$Q \Longrightarrow P \sqcup Q$

$\langle proof \rangle$

lemma *sup-boolE*:

$P \sqcup Q \implies (P \implies R) \implies (Q \implies R) \implies R$
 $\langle proof \rangle$

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 $\langle proof \rangle$

end

instance *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*
 $\langle proof \rangle$

instance *fun* :: (*type*, *bounded-lattice*) *bounded-lattice* $\langle proof \rangle$

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

definition

fun-Compl-def: $- A = (\lambda x. - A\ x)$

instance $\langle proof \rangle$

end

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

definition

fun-diff-def: $A - B = (\lambda x. A\ x - B\ x)$

instance $\langle proof \rangle$

end

instance *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*
 $\langle proof \rangle$

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* \Rightarrow *bool***consts**

Collect :: (*'a* \Rightarrow *bool*) \Rightarrow *'a set* — comprehension
op : :: *'a* \Rightarrow *'a set* \Rightarrow *bool* — membership

local**notation**

op : (*op* :) **and**
op : ((-/ : -) [50, 51] 50)

defs

mem-def [code]: $x : S == S\ x$
Collect-def [code]: $\text{Collect } P == P$

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 : $(op \in)$ **and**
op : $((-/ \in -) [50, 51] 50)$ **and**
not-mem $(op \notin)$ **and**
not-mem $((-/ \notin -) [50, 51] 50)$

Set comprehensions

syntax

-Coll :: $pttrn \Rightarrow bool \Rightarrow 'a\ set$ $((1\{-./-\})$

translations

$\{x. P\} == CONST\ Collect\ (\%x. P)$

syntax

-Collect :: $idt \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set$ $((1\{-./-\})$

syntax (*xsymbols*)

-Collect :: $idt \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set$ $((1\{- \in / -./ -\})$

translations

$\{x:A. P\} \Rightarrow \{x. x:A \ \& \ P\}$

lemma *mem-Collect-eq* [*iff*]: $(a : \{x. P(x)\}) = P(a)$

$\langle proof \rangle$

lemma *Collect-mem-eq* [*simp*]: $\{x. x:A\} = A$

$\langle proof \rangle$

lemma *CollectI*: $P(a) \Rightarrow a : \{x. P(x)\}$

$\langle proof \rangle$

lemma *CollectD*: $a : \{x. P(x)\} \Rightarrow P(a)$

$\langle proof \rangle$

lemma *Collect-cong*: $(!!x. P\ x = Q\ x) \Rightarrow \{x. P(x)\} = \{x. Q(x)\}$

$\langle proof \rangle$

Simproc for pulling $x=t$ in $\{x. \dots \ \& \ x=t \ \& \ \dots\}$ to the front (and similarly for $t=x$):

$\langle ML \rangle$

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* => 'a set ($\{(-)\}$)

translations

$\{x, xs\} == \text{CONST insert } x \{xs\}$

$\{x\} == \text{CONST insert } x \{\}$

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* => (*'a* => *bool*) => *bool* — bounded universal quantifiers

Bex :: *'a set* => (*'a* => *bool*) => *bool* — bounded existential quantifiers

local**defs**

Ball-def: *Ball A P* == *ALL x. x:A --> P(x)*

Bex-def: *Bex A P* == *EX x. x:A & P(x)*

syntax

-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3ALL* :-./ -) [0, 0, 10] 10)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3EX* :-./ -) [0, 0, 10] 10)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3EX!* :-./ -) [0, 0, 10] 10)
-Bleast :: *id* => *'a set* => *bool* => *'a* ((*3LEAST* :-./ -) [0, 0, 10] 10)

syntax (HOL)

-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3!* :-./ -) [0, 0, 10] 10)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3?* :-./ -) [0, 0, 10] 10)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3?!* :-./ -) [0, 0, 10] 10)

syntax (xsymbols)

-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3V* -∈./ -) [0, 0, 10] 10)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3E* -∈./ -) [0, 0, 10] 10)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3E!* -∈./ -) [0, 0, 10] 10)
-Bleast :: *id* => *'a set* => *bool* => *'a* ((*3LEAST* -∈./ -) [0, 0, 10] 10)

syntax (HTML output)

-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3V* -∈./ -) [0, 0, 10] 10)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3E* -∈./ -) [0, 0, 10] 10)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3E!* -∈./ -) [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 == *EX! x. x:A & P*

LEAST x:A. P == *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)
-setleAll :: [*idt, 'a, bool*] => *bool* ((*3ALL* -<=./ -) [0, 0, 10] 10)
-setleEx :: [*idt, 'a, bool*] => *bool* ((*3EX* -<=./ -) [0, 0, 10] 10)
-setleEx1 :: [*idt, 'a, bool*] => *bool* ((*3EX!* -<=./ -) [0, 0, 10] 10)

syntax (*xsymbols*)

$\text{-setlessAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \forall \text{-}\subset\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setlessEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \exists \text{-}\subset\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \forall \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \exists \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleEx1} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \exists !\text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$

syntax (*HOL output*)

$\text{-setlessAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists ! \text{-}\subset\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setlessEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists ? \text{-}\subset\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists ! \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists ? \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleEx1} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists ? ! \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$

syntax (*HTML output*)

$\text{-setlessAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \forall \text{-}\subset\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setlessEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \exists \text{-}\subset\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleAll} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \forall \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleEx} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \exists \text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$
 $\text{-setleEx1} :: [\text{idt}, 'a, \text{bool}] \Rightarrow \text{bool} \quad ((\exists \exists !\text{-}\subseteq\text{-}/\text{-}) [0, 0, 10] 10)$

translations

$\forall A \subset B. P \Rightarrow ALL A. A \subset B \dashrightarrow P$
 $\exists A \subset B. P \Rightarrow EX A. A \subset B \ \& \ P$
 $\forall A \subseteq B. P \Rightarrow ALL A. A \subseteq B \dashrightarrow P$
 $\exists A \subseteq B. P \Rightarrow EX A. A \subseteq B \ \& \ P$
 $\exists ! A \subseteq B. P \Rightarrow EX ! A. A \subseteq B \ \& \ P$

$\langle ML \rangle$

Translate between $\{e \mid x1 \dots xn. P\}$ and $\{u. EX x1 \dots xn. u = e \ \& \ P\}$; $\{y. EX x1 \dots xn. y = e \ \& \ P\}$ is only translated if $[0..n] \text{ subset } bvs(e)$.

syntax

$\text{-Setcompr} :: 'a \Rightarrow \text{idts} \Rightarrow \text{bool} \Rightarrow 'a \text{ set} \quad ((1\{- \mid / \text{-} / \text{-}\})$

$\langle ML \rangle$

lemma *ballI* [*intro!*]: $(!!x. x:A \Rightarrow P \ x) \Rightarrow ALL x:A. P \ x$
 $\langle \text{proof} \rangle$

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: $ALL x:A. P \ x \Rightarrow x:A \Rightarrow P \ x$
 $\langle \text{proof} \rangle$

Gives better instantiation for bound:

$\langle ML \rangle$

lemma *ballE* [*elim*]: $ALL\ x:A.\ P\ x \implies (P\ x \implies Q) \implies (x \sim: A \implies Q) \implies Q$
 $\langle proof \rangle$

lemma *bexI* [*intro*]: $P\ x \implies x:A \implies EX\ x:A.\ P\ x$
 — Normally the best argument order: $P\ x$ constrains the choice of $x \in A$.
 $\langle proof \rangle$

lemma *rev-bexI* [*intro?*]: $x:A \implies P\ x \implies EX\ x:A.\ P\ x$
 — The best argument order when there is only one $x \in A$.
 $\langle proof \rangle$

lemma *bexCI*: $(ALL\ x:A.\ \sim P\ x \implies P\ a) \implies a:A \implies EX\ x:A.\ P\ x$
 $\langle proof \rangle$

lemma *bexE* [*elim!*]: $EX\ x:A.\ P\ x \implies (!x.\ x:A \implies P\ x \implies Q) \implies Q$
 $\langle proof \rangle$

lemma *ball-triv* [*simp*]: $(ALL\ x:A.\ P) = ((EX\ x.\ x:A) \dashrightarrow P)$
 — Trivial rewrite rule.
 $\langle proof \rangle$

lemma *bex-triv* [*simp*]: $(EX\ x:A.\ P) = ((EX\ x.\ x:A) \& P)$
 — Dual form for existentials.
 $\langle proof \rangle$

lemma *bex-triv-one-point1* [*simp*]: $(EX\ x:A.\ x = a) = (a:A)$
 $\langle proof \rangle$

lemma *bex-triv-one-point2* [*simp*]: $(EX\ x:A.\ a = x) = (a:A)$
 $\langle proof \rangle$

lemma *bex-one-point1* [*simp*]: $(EX\ x:A.\ x = a \& P\ x) = (a:A \& P\ a)$
 $\langle proof \rangle$

lemma *bex-one-point2* [*simp*]: $(EX\ x:A.\ a = x \& P\ x) = (a:A \& P\ a)$
 $\langle proof \rangle$

lemma *ball-one-point1* [*simp*]: $(ALL\ x:A.\ x = a \dashrightarrow P\ x) = (a:A \dashrightarrow P\ a)$
 $\langle proof \rangle$

lemma *ball-one-point2* [*simp*]: $(ALL\ x:A.\ a = x \dashrightarrow P\ x) = (a:A \dashrightarrow P\ a)$
 $\langle proof \rangle$

Congruence rules

lemma *ball-cong*:
 $A = B \implies (!x.\ x:B \implies P\ x = Q\ x) \implies$
 $(ALL\ x:A.\ P\ x) = (ALL\ x:B.\ Q\ x)$

$\langle proof \rangle$

lemma *strong-ball-cong* [*cong*]:

$A = B \implies (!x. x:B \implies P\ x = Q\ x) \implies$
 $(ALL\ x:A. P\ x) = (ALL\ x:B. Q\ x)$
 $\langle proof \rangle$

lemma *bex-cong*:

$A = B \implies (!x. x:B \implies P\ x = Q\ x) \implies$
 $(EX\ x:A. P\ x) = (EX\ x:B. Q\ x)$
 $\langle proof \rangle$

lemma *strong-bex-cong* [*cong*]:

$A = B \implies (!x. x:B \implies P\ x = Q\ x) \implies$
 $(EX\ x:A. P\ x) = (EX\ x:B. Q\ x)$
 $\langle proof \rangle$

6.3 Basic operations

6.3.1 Subsets

lemma *subsetI* [*intro!*]: $(\bigwedge x. x \in A \implies x \in B) \implies A \subseteq B$
 $\langle proof \rangle$

Map the type *'a set => anything* to just *'a*; for overloading constants whose first argument has type *'a set*.

lemma *subsetD* [*elim, intro?*]: $A \subseteq B \implies c \in A \implies c \in B$
 $\langle proof \rangle$

lemma *rev-subsetD* [*no-atp,intro?*]: $c \in A \implies A \subseteq B \implies c \in B$
 — The same, with reversed premises for use with *erule* – cf *rev-mp*.
 $\langle proof \rangle$

Converts $A \subseteq B$ to $x \in A \implies x \in B$.

lemma *subsetCE* [*no-atp,elim*]: $A \subseteq B \implies (c \notin A \implies P) \implies (c \in B \implies P) \implies P$
 — Classical elimination rule.
 $\langle proof \rangle$

lemma *subset-eq* [*no-atp*]: $A \leq B = (\forall x \in A. x \in B)$ $\langle proof \rangle$

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B \implies c \notin B \implies c \notin A$
 $\langle proof \rangle$

lemma *subset-refl* [*simp*]: $A \subseteq A$
 $\langle proof \rangle$

lemma *subset-trans*: $A \subseteq B \implies B \subseteq C \implies A \subseteq C$

$\langle proof \rangle$

lemma *set-rev-mp*: $x:A ==> A \subseteq B ==> x:B$
 $\langle proof \rangle$

lemma *set-mp*: $A \subseteq B ==> x:A ==> x:B$
 $\langle proof \rangle$

lemma *eq-mem-trans*: $a=b ==> b \in A ==> a \in A$
 $\langle proof \rangle$

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** $A = B$
 $\langle proof \rangle$

lemma *expand-set-eq*: $(A = B) = (ALL\ x. (x:A) = (x:B))$
 $\langle proof \rangle$

lemma *subset-antisym* [*intro!*]: $A \subseteq B ==> B \subseteq A ==> A = B$
 — Anti-symmetry of the subset relation.
 $\langle proof \rangle$

Equality rules from ZF set theory – are they appropriate here?

lemma *equalityD1*: $A = B ==> A \subseteq B$
 $\langle proof \rangle$

lemma *equalityD2*: $A = B ==> B \subseteq A$
 $\langle proof \rangle$

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*: $A = B ==> (A \subseteq B ==> B \subseteq A ==> P) ==> P$
 $\langle proof \rangle$

lemma *equalityCE* [*elim*]:
 $A = B ==> (c \in A ==> c \in B ==> P) ==> (c \notin A ==> c \notin B ==> P)$
 $==> P$
 $\langle proof \rangle$

lemma *eqset-imp-iff*: $A = B ==> (x : A) = (x : B)$
 $\langle proof \rangle$

lemma *equelem-imp-iff*: $x = y ==> (x : A) = (y : A)$
 $\langle proof \rangle$

6.3.3 The universal set – UNIV

abbreviation *UNIV* :: 'a set where

UNIV \equiv top

lemma *UNIV-def*:

UNIV = {*x*. True}

<proof>

lemma *UNIV-I* [*simp*]: *x* : *UNIV*

<proof>

declare *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

lemma *UNIV-witness* [*intro?*]: *EX x. x* : *UNIV*

<proof>

lemma *subset-UNIV* [*simp*]: *A* \subseteq *UNIV*

<proof>

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*

<proof>

lemma *bex-UNIV* [*simp*]: *Bex UNIV P* = *Ex P*

<proof>

lemma *UNIV-eq-I*: $(\bigwedge x. x \in A) \implies UNIV = A$

<proof>

6.3.4 The empty set

lemma *empty-def*:

{ } = {*x*. False}

<proof>

lemma *empty-iff* [*simp*]: (*c* : { }) = *False*

<proof>

lemma *emptyE* [*elim!*]: *a* : { } \implies *P*

<proof>

lemma *empty-subsetI* [*iff*]: { } \subseteq *A*

— One effect is to delete the ASSUMPTION { } \subseteq *A*

<proof>

lemma *equals0I*: (!*y. y* \in *A* \implies *False*) \implies *A* = { }

<proof>

lemma *equals0D*: $A = \{\} \implies a \notin A$
 — Use for reasoning about disjointness: $A \text{ Int } B = \{\}$
 $\langle \text{proof} \rangle$

lemma *ball-empty* [simp]: $\text{Ball } \{\} P = \text{True}$
 $\langle \text{proof} \rangle$

lemma *bex-empty* [simp]: $\text{Bex } \{\} P = \text{False}$
 $\langle \text{proof} \rangle$

lemma *UNIV-not-empty* [iff]: $\text{UNIV} \sim = \{\}$
 $\langle \text{proof} \rangle$

6.3.5 The Powerset operator – Pow

definition *Pow* :: $'a \text{ set} \implies 'a \text{ set set}$ **where**
Pow-def: $\text{Pow } A = \{B. B \leq A\}$

lemma *Pow-iff* [iff]: $(A \in \text{Pow } B) = (A \subseteq B)$
 $\langle \text{proof} \rangle$

lemma *PowI*: $A \subseteq B \implies A \in \text{Pow } B$
 $\langle \text{proof} \rangle$

lemma *PowD*: $A \in \text{Pow } B \implies A \subseteq B$
 $\langle \text{proof} \rangle$

lemma *Pow-bottom*: $\{\} \in \text{Pow } B$
 $\langle \text{proof} \rangle$

lemma *Pow-top*: $A \in \text{Pow } A$
 $\langle \text{proof} \rangle$

6.3.6 Set complement

lemma *Compl-iff* [simp]: $(c \in -A) = (c \notin A)$
 $\langle \text{proof} \rangle$

lemma *ComplI* [intro!]: $(c \in A \implies \text{False}) \implies c \in -A$
 $\langle \text{proof} \rangle$

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!]: $c : -A \implies c \sim : A$
 $\langle \text{proof} \rangle$

lemmas *ComplE* = *ComplD* [elim-format]

lemma *Compl-eq*: $- A = \{x. \sim x : A\}$ $\langle proof \rangle$

6.3.7 Binary union – Un

abbreviation *union* :: 'a set \Rightarrow 'a set \Rightarrow '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*:
 $A \cup B = \{x. x \in A \vee x \in B\}$
 $\langle proof \rangle$

lemma *Un-iff* [*simp*]: $(c : A \text{ Un } B) = (c:A \mid c:B)$
 $\langle proof \rangle$

lemma *UnI1* [*elim?*]: $c:A \implies c : A \text{ Un } B$
 $\langle proof \rangle$

lemma *UnI2* [*elim?*]: $c:B \implies c : A \text{ Un } B$
 $\langle proof \rangle$

Classical introduction rule: no commitment to A vs B .

lemma *UnCI* [*intro!*]: $(c\sim:B \implies c:A) \implies c : A \text{ Un } B$
 $\langle proof \rangle$

lemma *UnE* [*elim!*]: $c : A \text{ Un } B \implies (c:A \implies P) \implies (c:B \implies P) \implies P$
 $\langle proof \rangle$

lemma *insert-def*: $\text{insert } a \ B = \{x. x = a\} \cup B$
 $\langle proof \rangle$

lemma *mono-Un*: $\text{mono } f \implies f \ A \cup f \ B \subseteq f \ (A \cup B)$
 $\langle proof \rangle$

6.3.8 Binary intersection – Int

abbreviation *inter* :: 'a set \Rightarrow 'a set \Rightarrow 'a set (**infixl** Int 70) **where**
op Int \equiv inf

notation (*xsymbols*)
inter (**infixl** \cap 70)

notation (*HTML output*)
 $inter \text{ (infixl } \cap \text{ 70)}$

lemma *Int-def*:
 $A \cap B = \{x. x \in A \wedge x \in B\}$
 $\langle proof \rangle$

lemma *Int-iff [simp]*: $(c : A \text{ Int } B) = (c:A \ \& \ c:B)$
 $\langle proof \rangle$

lemma *IntI [intro!]*: $c:A \implies c:B \implies c : A \text{ Int } B$
 $\langle proof \rangle$

lemma *IntD1*: $c : A \text{ Int } B \implies c:A$
 $\langle proof \rangle$

lemma *IntD2*: $c : A \text{ Int } B \implies c:B$
 $\langle proof \rangle$

lemma *IntE [elim!]*: $c : A \text{ Int } B \implies (c:A \implies c:B \implies P) \implies P$
 $\langle proof \rangle$

lemma *mono-Int*: $mono \ f \implies f \ (A \cap B) \subseteq f \ A \cap f \ B$
 $\langle proof \rangle$

6.3.9 Set difference

lemma *Diff-iff [simp]*: $(c : A - B) = (c:A \ \& \ c \sim B)$
 $\langle proof \rangle$

lemma *DiffI [intro!]*: $c : A \implies c \sim B \implies c : A - B$
 $\langle proof \rangle$

lemma *DiffD1*: $c : A - B \implies c : A$
 $\langle proof \rangle$

lemma *DiffD2*: $c : A - B \implies c : B \implies P$
 $\langle proof \rangle$

lemma *DiffE [elim!]*: $c : A - B \implies (c:A \implies c \sim B \implies P) \implies P$
 $\langle proof \rangle$

lemma *set-diff-eq*: $A - B = \{x. x : A \ \& \ \sim x : B\}$ $\langle proof \rangle$

lemma *Compl-eq-Diff-UNIV*: $-A = (UNIV - A)$
 $\langle proof \rangle$

6.3.10 Augmenting a set – insert

lemma *insert-iff [simp]*: $(a : insert \ b \ A) = (a = b \mid a:A)$

$\langle \text{proof} \rangle$

lemma *insertI1*: $a : \text{insert } a \ B$

$\langle \text{proof} \rangle$

lemma *insertI2*: $a : B \implies a : \text{insert } b \ B$

$\langle \text{proof} \rangle$

lemma *insertE* [*elim!*]: $a : \text{insert } b \ A \implies (a = b \implies P) \implies (a:A \implies P) \implies P$

$\langle \text{proof} \rangle$

lemma *insertCI* [*intro!*]: $(a \sim B \implies a = b) \implies a : \text{insert } b \ B$

— Classical introduction rule.

$\langle \text{proof} \rangle$

lemma *subset-insert-iff*: $(A \subseteq \text{insert } x \ B) = (\text{if } x:A \text{ then } A - \{x\} \subseteq B \text{ else } A \subseteq B)$

$\langle \text{proof} \rangle$

lemma *set-insert*:

assumes $x \in A$

obtains B **where** $A = \text{insert } x \ B$ **and** $x \notin B$

$\langle \text{proof} \rangle$

lemma *insert-ident*: $x \sim A \implies x \sim B \implies (\text{insert } x \ A = \text{insert } x \ B) = (A = B)$

$\langle \text{proof} \rangle$

6.3.11 Singletons, using insert

lemma *singletonI* [*intro!,no-atp*]: $a : \{a\}$

— Redundant? But unlike *insertCI*, it proves the subgoal immediately!

$\langle \text{proof} \rangle$

lemma *singletonD* [*dest!,no-atp*]: $b : \{a\} \implies b = a$

$\langle \text{proof} \rangle$

lemmas *singletonE* = *singletonD* [*elim-format*]

lemma *singleton-iff*: $(b : \{a\}) = (b = a)$

$\langle \text{proof} \rangle$

lemma *singleton-inject* [*dest!*]: $\{a\} = \{b\} \implies a = b$

$\langle \text{proof} \rangle$

lemma *singleton-insert-inj-eq* [*iff,no-atp*]:

$(\{b\} = \text{insert } a \ A) = (a = b \ \& \ A \subseteq \{b\})$

$\langle \text{proof} \rangle$

lemma *singleton-insert-inj-eq'* [iff, no-atp]:
 $(\text{insert } a \ A = \{b\}) = (a = b \ \& \ A \subseteq \{b\})$
 ⟨proof⟩

lemma *subset-singletonD*: $A \subseteq \{x\} ==> A = \{\} \mid A = \{x\}$
 ⟨proof⟩

lemma *singleton-conv* [simp]: $\{x. x = a\} = \{a\}$
 ⟨proof⟩

lemma *singleton-conv2* [simp]: $\{x. a = x\} = \{a\}$
 ⟨proof⟩

lemma *diff-single-insert*: $A - \{x\} \subseteq B ==> x \in A ==> A \subseteq \text{insert } x \ B$
 ⟨proof⟩

lemma *doubleton-eq-iff*: $(\{a, b\} = \{c, d\}) = (a=c \ \& \ b=d \mid a=d \ \& \ b=c)$
 ⟨proof⟩

6.3.12 Image of a set under a function

Frequently b does not have the syntactic form of $f \ x$.

definition *image* :: $('a ==> 'b) ==> 'a \ \text{set} ==> 'b \ \text{set}$ (**infixr** ‘ 90) **where**
image-def [no-atp]: $f \ 'A = \{y. \ \text{EX } x:A. \ y = f(x)\}$

abbreviation

range :: $('a ==> 'b) ==> 'b \ \text{set}$ **where** — of function
range $f == f \ 'UNIV$

lemma *image-eqI* [simp, intro]: $b = f \ x ==> x:A ==> b : f \ 'A$
 ⟨proof⟩

lemma *imageI*: $x : A ==> f \ x : f \ 'A$
 ⟨proof⟩

lemma *rev-image-eqI*: $x:A ==> b = f \ x ==> b : f \ 'A$
 — This version’s more effective when we already have the required x .
 ⟨proof⟩

lemma *imageE* [elim!]:
 $b : (\%x. f \ x) \ 'A ==> (!!x. b = f \ x ==> x:A ==> P) ==> P$
 — The eta-expansion gives variable-name preservation.
 ⟨proof⟩

lemma *image-Un*: $f \ (A \ \text{Un } B) = f \ 'A \ \text{Un } f \ 'B$
 ⟨proof⟩

lemma *image-iff*: $(z : f \ 'A) = (\text{EX } x:A. \ z = f \ x)$

$\langle proof \rangle$

lemma *image-subset-iff*: $(f^*A \subseteq B) = (\forall x \in A. f\ x \in B)$

— This rewrite rule would confuse users if made default.

$\langle proof \rangle$

lemma *subset-image-iff*: $(B \subseteq f^*A) = (EX\ AA. AA \subseteq A \ \& \ B = f^*AA)$

$\langle proof \rangle$

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.

$\langle proof \rangle$

Range of a function – just a translation for image!

lemma *range-eqI*: $b = f\ x ==> b \in range\ f$

$\langle proof \rangle$

lemma *rangeI*: $f\ x \in range\ f$

$\langle proof \rangle$

lemma *rangeE* [*elim?*]: $b \in range\ (\lambda x. f\ x) ==> (!!x. b = f\ x ==> P) ==> P$

$\langle proof \rangle$

6.3.13 Some rules with *if*

Elimination of $\{x. \dots \ \& \ x=t \ \& \ \dots\}$.

lemma *Collect-conv-if*: $\{x. x=a \ \& \ P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$

$\langle proof \rangle$

lemma *Collect-conv-if2*: $\{x. a=x \ \& \ P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$

$\langle proof \rangle$

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

lemma *split-if-eq1*: $((if\ Q\ then\ x\ else\ y) = b) = ((Q \dashrightarrow x = b) \ \& \ (\sim Q \dashrightarrow y = b))$

$\langle proof \rangle$

lemma *split-if-eq2*: $(a = (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a = x) \ \& \ (\sim Q \dashrightarrow a = y))$

$\langle proof \rangle$

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

lemma *split-if-mem1*: $((if\ Q\ then\ x\ else\ y) : b) = ((Q \dashrightarrow x : b) \ \& \ (\sim Q \dashrightarrow y : b))$

$\langle proof \rangle$

lemma *split-if-mem2*: $(a : (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a : x) \ \& \ (\sim Q \dashrightarrow a : y))$
 $\langle proof \rangle$

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*]: $A \subseteq B \implies A \neq B \implies A \subset B$
 $\langle proof \rangle$

lemma *psubsetE* [*elim!,no-atp*]:
 $[|A \subset B; [|A \subseteq B; \sim (B \subseteq A)|] \implies R|] \implies R$
 $\langle proof \rangle$

lemma *psubset-insert-iff*:
 $(A \subset insert\ x\ B) = (if\ x \in B\ then\ A \subset B\ else\ if\ x \in A\ then\ A - \{x\} \subset B\ else\ A \subseteq B)$
 $\langle proof \rangle$

lemma *psubset-eq*: $(A \subset B) = (A \subseteq B \ \& \ A \neq B)$
 $\langle proof \rangle$

lemma *psubset-imp-subset*: $A \subset B \implies A \subseteq B$
 $\langle proof \rangle$

lemma *psubset-trans*: $[|A \subset B; B \subset C|] \implies A \subset C$
 $\langle proof \rangle$

lemma *psubsetD*: $[|A \subset B; c \in A|] \implies c \in B$
 $\langle proof \rangle$

lemma *psubset-subset-trans*: $A \subset B \implies B \subseteq C \implies A \subset C$
 $\langle proof \rangle$

lemma *subset-psubset-trans*: $A \subseteq B \implies B \subset C \implies A \subset C$
 $\langle proof \rangle$

lemma *psubset-imp-ex-mem*: $A \subset B \implies \exists b. b \in (B - A)$
 $\langle proof \rangle$

lemma *atomize-ball*:
 $(!!x. x \in A \implies P\ x) == Trueprop\ (\forall x \in A. P\ x)$
 $\langle proof \rangle$

lemmas [*symmetric, rulify*] = *atomize-ball*
and [*symmetric, defn*] = *atomize-ball*

6.4.2 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq \text{insert } a \ B$
 $\langle \text{proof} \rangle$

lemma *subset-insertI2*: $A \subseteq B \implies A \subseteq \text{insert } b \ B$
 $\langle \text{proof} \rangle$

lemma *subset-insert*: $x \notin A \implies (A \subseteq \text{insert } x \ B) = (A \subseteq B)$
 $\langle \text{proof} \rangle$

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
 $\langle \text{proof} \rangle$

lemma *Un-upper2*: $B \subseteq A \cup B$
 $\langle \text{proof} \rangle$

lemma *Un-least*: $A \subseteq C \implies B \subseteq C \implies A \cup B \subseteq C$
 $\langle \text{proof} \rangle$

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
 $\langle \text{proof} \rangle$

lemma *Int-lower2*: $A \cap B \subseteq B$
 $\langle \text{proof} \rangle$

lemma *Int-greatest*: $C \subseteq A \implies C \subseteq B \implies C \subseteq A \cap B$
 $\langle \text{proof} \rangle$

Set difference.

lemma *Diff-subset*: $A - B \subseteq A$
 $\langle \text{proof} \rangle$

lemma *Diff-subset-conv*: $(A - B \subseteq C) = (A \subseteq B \cup C)$
 $\langle \text{proof} \rangle$

6.4.3 Equalities involving union, intersection, inclusion, etc.

$\{\}$.

lemma *Collect-const [simp]*: $\{s. P\} = (\text{if } P \text{ then } \text{UNIV} \text{ else } \{\})$
 — supersedes *Collect-False-empty*
 $\langle \text{proof} \rangle$

lemma *subset-empty* [simp]: $(A \subseteq \{\}) = (A = \{\})$
 ⟨proof⟩

lemma *not-psubset-empty* [iff]: $\neg (A < \{\})$
 ⟨proof⟩

lemma *Collect-empty-eq* [simp]: $(\text{Collect } P = \{\}) = (\forall x. \neg P x)$
 ⟨proof⟩

lemma *empty-Collect-eq* [simp]: $(\{\} = \text{Collect } P) = (\forall x. \neg P x)$
 ⟨proof⟩

lemma *Collect-neg-eq*: $\{x. \neg P x\} = - \{x. P x\}$
 ⟨proof⟩

lemma *Collect-disj-eq*: $\{x. P x \mid Q x\} = \{x. P x\} \cup \{x. Q x\}$
 ⟨proof⟩

lemma *Collect-imp-eq*: $\{x. P x \longrightarrow Q x\} = -\{x. P x\} \cup \{x. Q x\}$
 ⟨proof⟩

lemma *Collect-conj-eq*: $\{x. P x \ \& \ Q x\} = \{x. P x\} \cap \{x. Q x\}$
 ⟨proof⟩

insert.

lemma *insert-is-Un*: $\text{insert } a \ A = \{a\} \ \text{Un } A$
 — NOT SUITABLE FOR REWRITING since $\{a\} == \text{insert } a \ \{\}$
 ⟨proof⟩

lemma *insert-not-empty* [simp]: $\text{insert } a \ A \neq \{\}$
 ⟨proof⟩

lemmas *empty-not-insert* = *insert-not-empty* [symmetric, standard]
declare *empty-not-insert* [simp]

lemma *insert-absorb*: $a \in A ==> \text{insert } a \ A = A$
 — [simp] causes recursive calls when there are nested inserts
 — with *quadratic* running time
 ⟨proof⟩

lemma *insert-absorb2* [simp]: $\text{insert } x \ (\text{insert } x \ A) = \text{insert } x \ A$
 ⟨proof⟩

lemma *insert-commute*: $\text{insert } x \ (\text{insert } y \ A) = \text{insert } y \ (\text{insert } x \ A)$
 ⟨proof⟩

lemma *insert-subset* [simp]: $(\text{insert } x \ A \subseteq B) = (x \in B \ \& \ A \subseteq B)$
 ⟨proof⟩

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
 $\langle \text{proof} \rangle$

lemma *insert-Collect*: $\text{insert } a \ (\text{Collect } P) = \{u. u \neq a \longrightarrow P \ u\}$
 $\langle \text{proof} \rangle$

lemma *insert-inter-insert* [simp]: $\text{insert } a \ A \cap \text{insert } a \ B = \text{insert } a \ (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *insert-disjoint* [simp, no-atp]:
 $(\text{insert } a \ A \cap B = \{\}) = (a \notin B \wedge A \cap B = \{\})$
 $(\{\} = \text{insert } a \ A \cap B) = (a \notin B \wedge \{\} = A \cap B)$
 $\langle \text{proof} \rangle$

lemma *disjoint-insert* [simp, no-atp]:
 $(B \cap \text{insert } a \ A = \{\}) = (a \notin B \wedge B \cap A = \{\})$
 $(\{\} = A \cap \text{insert } b \ B) = (b \notin A \wedge \{\} = A \cap B)$
 $\langle \text{proof} \rangle$

image.

lemma *image-empty* [simp]: $f' \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *image-insert* [simp]: $f' \text{insert } a \ B = \text{insert } (f \ a) \ (f' B)$
 $\langle \text{proof} \rangle$

lemma *image-constant*: $x \in A \implies (\lambda x. c)' A = \{c\}$
 $\langle \text{proof} \rangle$

lemma *image-constant-conv*: $(\%x. c)' A = (\text{if } A = \{\} \text{ then } \{\} \text{ else } \{c\})$
 $\langle \text{proof} \rangle$

lemma *image-image*: $f' (g' A) = (\lambda x. f (g \ x))' A$
 $\langle \text{proof} \rangle$

lemma *insert-image* [simp]: $x \in A \implies \text{insert } (f \ x) \ (f' A) = f' A$
 $\langle \text{proof} \rangle$

lemma *image-is-empty* [iff]: $(f' A = \{\}) = (A = \{\})$
 $\langle \text{proof} \rangle$

lemma *empty-is-image* [iff]: $(\{\} = f' A) = (A = \{\})$
 $\langle \text{proof} \rangle$

lemma *image-Collect* [no-atp]: $f' \{x. P \ x\} = \{f \ x \mid x. P \ x\}$
 — 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.

<proof>

lemma *if-image-distrib* [simp]:

$(\lambda x. \text{if } P\ x \text{ then } f\ x \text{ else } g\ x) \text{ ` } S$
 $= (f \text{ ` } (S \cap \{x. P\ x\})) \cup (g \text{ ` } (S \cap \{x. \neg P\ x\}))$
<proof>

lemma *image-cong*: $M = N \implies (!x. x \in N \implies f\ x = g\ x) \implies f \text{ ` } M = g \text{ ` } N$

<proof>

range.

lemma *full-SetCompr-eq* [no-atp]: $\{u. \exists x. u = f\ x\} = \text{range } f$

<proof>

lemma *range-composition*: $\text{range } (\lambda x. f\ (g\ x)) = f \text{ ` } \text{range } g$

<proof>

Int

lemma *Int-absorb* [simp]: $A \cap A = A$

<proof>

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$

<proof>

lemma *Int-commute*: $A \cap B = B \cap A$

<proof>

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$

<proof>

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$

<proof>

lemmas *Int-ac* = *Int-assoc* *Int-left-absorb* *Int-commute* *Int-left-commute*

— Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \implies A \cap B = B$

<proof>

lemma *Int-absorb2*: $A \subseteq B \implies A \cap B = A$

<proof>

lemma *Int-empty-left* [simp]: $\{\} \cap B = \{\}$

<proof>

lemma *Int-empty-right* [simp]: $A \cap \{\} = \{\}$

<proof>

lemma *disjoint-eq-subset-Compl*: $(A \cap B = \{\}) = (A \subseteq -B)$
 $\langle proof \rangle$

lemma *disjoint-iff-not-equal*: $(A \cap B = \{\}) = (\forall x \in A. \forall y \in B. x \neq y)$
 $\langle proof \rangle$

lemma *Int-UNIV-left [simp]*: $UNIV \cap B = B$
 $\langle proof \rangle$

lemma *Int-UNIV-right [simp]*: $A \cap UNIV = A$
 $\langle proof \rangle$

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
 $\langle proof \rangle$

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
 $\langle proof \rangle$

lemma *Int-UNIV [simp,no-atp]*: $(A \cap B = UNIV) = (A = UNIV \ \& \ B = UNIV)$
 $\langle proof \rangle$

lemma *Int-subset-iff [simp]*: $(C \subseteq A \cap B) = (C \subseteq A \ \& \ C \subseteq B)$
 $\langle proof \rangle$

lemma *Int-Collect*: $(x \in A \cap \{x. P \ x\}) = (x \in A \ \& \ P \ x)$
 $\langle proof \rangle$

Un.

lemma *Un-absorb [simp]*: $A \cup A = A$
 $\langle proof \rangle$

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
 $\langle proof \rangle$

lemma *Un-commute*: $A \cup B = B \cup A$
 $\langle proof \rangle$

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
 $\langle proof \rangle$

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
 $\langle proof \rangle$

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
 — Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B ==> A \cup B = B$
 $\langle proof \rangle$

lemma *Un-absorb2*: $B \subseteq A \implies A \cup B = A$
 $\langle proof \rangle$

lemma *Un-empty-left [simp]*: $\{\} \cup B = B$
 $\langle proof \rangle$

lemma *Un-empty-right [simp]*: $A \cup \{\} = A$
 $\langle proof \rangle$

lemma *Un-UNIV-left [simp]*: $UNIV \cup B = UNIV$
 $\langle proof \rangle$

lemma *Un-UNIV-right [simp]*: $A \cup UNIV = UNIV$
 $\langle proof \rangle$

lemma *Un-insert-left [simp]*: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
 $\langle proof \rangle$

lemma *Un-insert-right [simp]*: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
 $\langle proof \rangle$

lemma *Int-insert-left*:
 $(insert\ a\ B)\ Int\ C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
 $\langle proof \rangle$

lemma *Int-insert-left-if0 [simp]*:
 $a \notin C \implies (insert\ a\ B)\ Int\ C = B \cap C$
 $\langle proof \rangle$

lemma *Int-insert-left-if1 [simp]*:
 $a \in C \implies (insert\ a\ B)\ Int\ C = insert\ a\ (B \cap C)$
 $\langle proof \rangle$

lemma *Int-insert-right*:
 $A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
 $\langle proof \rangle$

lemma *Int-insert-right-if0 [simp]*:
 $a \notin A \implies A \cap (insert\ a\ B) = A \cap B$
 $\langle proof \rangle$

lemma *Int-insert-right-if1 [simp]*:
 $a \in A \implies A \cap (insert\ a\ B) = insert\ a\ (A \cap B)$
 $\langle proof \rangle$

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
 $\langle proof \rangle$

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
 $\langle \text{proof} \rangle$

lemma *Un-Int-crazy*:
 $(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
 $\langle \text{proof} \rangle$

lemma *subset-Un-eq*: $(A \subseteq B) = (A \cup B = B)$
 $\langle \text{proof} \rangle$

lemma *Un-empty [iff]*: $(A \cup B = \{\}) = (A = \{\} \ \& \ B = \{\})$
 $\langle \text{proof} \rangle$

lemma *Un-subset-iff [simp]*: $(A \cup B \subseteq C) = (A \subseteq C \ \& \ B \subseteq C)$
 $\langle \text{proof} \rangle$

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
 $\langle \text{proof} \rangle$

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
 $\langle \text{proof} \rangle$

Set complement

lemma *Compl-disjoint [simp]*: $A \cap -A = \{\}$
 $\langle \text{proof} \rangle$

lemma *Compl-disjoint2 [simp]*: $-A \cap A = \{\}$
 $\langle \text{proof} \rangle$

lemma *Compl-partition*: $A \cup -A = UNIV$
 $\langle \text{proof} \rangle$

lemma *Compl-partition2*: $-A \cup A = UNIV$
 $\langle \text{proof} \rangle$

lemma *double-complement [simp]*: $-(-A) = (A::'a \text{ set})$
 $\langle \text{proof} \rangle$

lemma *Compl-Un [simp]*: $-(A \cup B) = (-A) \cap (-B)$
 $\langle \text{proof} \rangle$

lemma *Compl-Int [simp]*: $-(A \cap B) = (-A) \cup (-B)$
 $\langle \text{proof} \rangle$

lemma *subset-Compl-self-eq*: $(A \subseteq -A) = (A = \{\})$
 $\langle \text{proof} \rangle$

lemma *Un-Int-assoc-eq*: $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$
 — Halmos, Naive Set Theory, page 16.

$\langle proof \rangle$

lemma *Compl-UNIV-eq* [simp]: $-UNIV = \{\}$
 $\langle proof \rangle$

lemma *Compl-empty-eq* [simp]: $-\{\} = UNIV$
 $\langle proof \rangle$

lemma *Compl-subset-Compl-iff* [iff]: $(-A \subseteq -B) = (B \subseteq A)$
 $\langle proof \rangle$

lemma *Compl-eq-Compl-iff* [iff]: $(-A = -B) = (A = (B::'a\ set))$
 $\langle proof \rangle$

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*: $(\forall x \in A \cup B. P\ x) = ((\forall x \in A. P\ x) \ \& \ (\forall x \in B. P\ x))$
 $\langle proof \rangle$

lemma *bex-Un*: $(\exists x \in A \cup B. P\ x) = ((\exists x \in A. P\ x) \mid (\exists x \in B. P\ x))$
 $\langle proof \rangle$

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
 $\langle proof \rangle$

lemma *Diff-eq-empty-iff* [simp,no-atp]: $(A - B = \{\}) = (A \subseteq B)$
 $\langle proof \rangle$

lemma *Diff-cancel* [simp]: $A - A = \{\}$
 $\langle proof \rangle$

lemma *Diff-idemp* [simp]: $(A - B) - B = A - (B::'a\ set)$
 $\langle proof \rangle$

lemma *Diff-triv*: $A \cap B = \{\} ==> A - B = A$
 $\langle proof \rangle$

lemma *empty-Diff* [simp]: $\{\} - A = \{\}$
 $\langle proof \rangle$

lemma *Diff-empty* [simp]: $A - \{\} = A$
 $\langle proof \rangle$

lemma *Diff-UNIV* [simp]: $A - UNIV = \{\}$
 $\langle proof \rangle$

lemma *Diff-insert0* [simp,no-atp]: $x \notin A \implies A - \text{insert } x B = A - B$
 ⟨proof⟩

lemma *Diff-insert*: $A - \text{insert } a B = A - B - \{a\}$
 — NOT SUITABLE FOR REWRITING since $\{a\} == \text{insert } a 0$
 ⟨proof⟩

lemma *Diff-insert2*: $A - \text{insert } a B = A - \{a\} - B$
 — NOT SUITABLE FOR REWRITING since $\{a\} == \text{insert } a 0$
 ⟨proof⟩

lemma *insert-Diff-if*: $\text{insert } x A - B = (\text{if } x \in B \text{ then } A - B \text{ else } \text{insert } x (A - B))$
 ⟨proof⟩

lemma *insert-Diff1* [simp]: $x \in B \implies \text{insert } x A - B = A - B$
 ⟨proof⟩

lemma *insert-Diff-single*[simp]: $\text{insert } a (A - \{a\}) = \text{insert } a A$
 ⟨proof⟩

lemma *insert-Diff*: $a \in A \implies \text{insert } a (A - \{a\}) = A$
 ⟨proof⟩

lemma *Diff-insert-absorb*: $x \notin A \implies (\text{insert } x A) - \{x\} = A$
 ⟨proof⟩

lemma *Diff-disjoint* [simp]: $A \cap (B - A) = \{\}$
 ⟨proof⟩

lemma *Diff-partition*: $A \subseteq B \implies A \cup (B - A) = B$
 ⟨proof⟩

lemma *double-diff*: $A \subseteq B \implies B \subseteq C \implies B - (C - A) = A$
 ⟨proof⟩

lemma *Un-Diff-cancel* [simp]: $A \cup (B - A) = A \cup B$
 ⟨proof⟩

lemma *Un-Diff-cancel2* [simp]: $(B - A) \cup A = B \cup A$
 ⟨proof⟩

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
 ⟨proof⟩

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$
 ⟨proof⟩

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$

$\langle proof \rangle$

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$
 $\langle proof \rangle$

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
 $\langle proof \rangle$

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
 $\langle proof \rangle$

lemma *Diff-Compl [simp]*: $A - (\neg B) = A \cap B$
 $\langle proof \rangle$

lemma *Compl-Diff-eq [simp]*: $\neg (A - B) = \neg A \cup B$
 $\langle proof \rangle$

Quantification over type *bool*.

lemma *bool-induct*: $P \text{ True} \implies P \text{ False} \implies P x$
 $\langle proof \rangle$

lemma *all-bool-eq*: $(\forall b. P b) \longleftrightarrow P \text{ True} \wedge P \text{ False}$
 $\langle proof \rangle$

lemma *bool-contrapos*: $P x \implies \neg P \text{ False} \implies P \text{ True}$
 $\langle proof \rangle$

lemma *ex-bool-eq*: $(\exists b. P b) \longleftrightarrow P \text{ True} \vee P \text{ False}$
 $\langle proof \rangle$

Pow

lemma *Pow-empty [simp]*: $\text{Pow } \{\} = \{\{\}\}$
 $\langle proof \rangle$

lemma *Pow-insert*: $\text{Pow } (\text{insert } a \ A) = \text{Pow } A \cup (\text{insert } a \ ' \text{Pow } A)$
 $\langle proof \rangle$

lemma *Pow-Compl*: $\text{Pow } (\neg A) = \{-B \mid B. A \in \text{Pow } B\}$
 $\langle proof \rangle$

lemma *Pow-UNIV [simp]*: $\text{Pow } \text{UNIV} = \text{UNIV}$
 $\langle proof \rangle$

lemma *Un-Pow-subset*: $\text{Pow } A \cup \text{Pow } B \subseteq \text{Pow } (A \cup B)$
 $\langle proof \rangle$

lemma *Pow-Int-eq [simp]*: $\text{Pow } (A \cap B) = \text{Pow } A \cap \text{Pow } B$
 $\langle proof \rangle$

Miscellany.

lemma *set-eq-subset*: $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$
 $\langle proof \rangle$

lemma *subset-iff*: $(A \subseteq B) = (\forall t. t \in A \longrightarrow t \in B)$
 $\langle proof \rangle$

lemma *subset-iff-psubset-eq*: $(A \subseteq B) = ((A \subset B) \mid (A = B))$
 $\langle proof \rangle$

lemma *all-not-in-conv* [*simp*]: $(\forall x. x \notin A) = (A = \{\})$
 $\langle proof \rangle$

lemma *ex-in-conv*: $(\exists x. x \in A) = (A \neq \{\})$
 $\langle proof \rangle$

lemma *distinct-lemma*: $f\ x \neq f\ y \implies x \neq y$
 $\langle proof \rangle$

6.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \implies f'A \subseteq f'B$
 $\langle proof \rangle$

lemma *Pow-mono*: $A \subseteq B \implies Pow\ A \subseteq Pow\ B$
 $\langle proof \rangle$

lemma *insert-mono*: $C \subseteq D \implies insert\ a\ C \subseteq insert\ a\ D$
 $\langle proof \rangle$

lemma *Un-mono*: $A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D$
 $\langle proof \rangle$

lemma *Int-mono*: $A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D$
 $\langle proof \rangle$

lemma *Diff-mono*: $A \subseteq C \implies D \subseteq B \implies A - B \subseteq C - D$
 $\langle proof \rangle$

lemma *Compl-anti-mono*: $A \subseteq B \implies -B \subseteq -A$
 $\langle proof \rangle$

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \implies x \in A \longrightarrow x \in B$
 $\langle proof \rangle$

lemma *conj-mono*: $P1 \longrightarrow Q1 \implies P2 \longrightarrow Q2 \implies (P1 \ \& \ P2) \longrightarrow (Q1 \ \& \ Q2)$

$\langle \text{proof} \rangle$

lemma *disj-mono*: $P1 \dashv\dashv Q1 \implies P2 \dashv\dashv Q2 \implies (P1 \mid P2) \dashv\dashv (Q1 \mid Q2)$
 $\langle \text{proof} \rangle$

lemma *imp-mono*: $Q1 \dashv\dashv P1 \implies P2 \dashv\dashv Q2 \implies (P1 \dashv\dashv P2) \dashv\dashv (Q1 \dashv\dashv Q2)$
 $\langle \text{proof} \rangle$

lemma *imp-refl*: $P \dashv\dashv P$ $\langle \text{proof} \rangle$

lemma *not-mono*: $Q \dashv\dashv P \implies \sim P \dashv\dashv \sim Q$
 $\langle \text{proof} \rangle$

lemma *ex-mono*: $(!!x. P x \dashv\dashv Q x) \implies (EX x. P x) \dashv\dashv (EX x. Q x)$
 $\langle \text{proof} \rangle$

lemma *all-mono*: $(!!x. P x \dashv\dashv Q x) \implies (ALL x. P x) \dashv\dashv (ALL x. Q x)$
 $\langle \text{proof} \rangle$

lemma *Collect-mono*: $(!!x. P x \dashv\dashv Q x) \implies \text{Collect } P \subseteq \text{Collect } Q$
 $\langle \text{proof} \rangle$

lemma *Int-Collect-mono*:
 $A \subseteq B \implies (!!x. x \in A \implies P x \dashv\dashv Q x) \implies A \cap \text{Collect } P \subseteq B \cap \text{Collect } Q$
 $\langle \text{proof} \rangle$

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono
ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b \implies c = d \implies b \dashv\dashv d \implies a \dashv\dashv c$
 $\langle \text{proof} \rangle$

6.4.5 Inverse image of a function

definition *vmage* :: $('a \Rightarrow 'b) \Rightarrow 'b \text{ set} \Rightarrow 'a \text{ set}$ (**infixr** $-'$ 90) **where**
 $[code del]: f -' B == \{x. f x : B\}$

lemma *vmage-eq* [*simp*]: $(a : f -' B) = (f a : B)$
 $\langle \text{proof} \rangle$

lemma *vmage-singleton-eq*: $(a : f -' \{b\}) = (f a = b)$
 $\langle \text{proof} \rangle$

lemma *vmageI* [*intro*]: $f a = b \implies b : B \implies a : f -' B$
 $\langle \text{proof} \rangle$

lemma *vimageI2*: $f\ a : A \implies a : f\ -' A$
 $\langle proof \rangle$

lemma *vimageE* [*elim!*]: $a : f\ -' B \implies (!x. f\ a = x \implies x:B \implies P) \implies P$
 $\langle proof \rangle$

lemma *vimageD*: $a : f\ -' A \implies f\ a : A$
 $\langle proof \rangle$

lemma *vimage-empty* [*simp*]: $f\ -' \{\} = \{\}$
 $\langle proof \rangle$

lemma *vimage-Compl*: $f\ -' (-A) = -(f\ -' A)$
 $\langle proof \rangle$

lemma *vimage-Un* [*simp*]: $f\ -' (A\ Un\ B) = (f\ -' A)\ Un\ (f\ -' B)$
 $\langle proof \rangle$

lemma *vimage-Int* [*simp*]: $f\ -' (A\ Int\ B) = (f\ -' A)\ Int\ (f\ -' B)$
 $\langle proof \rangle$

lemma *vimage-Collect-eq* [*simp*]: $f\ -' Collect\ P = \{y. P\ (f\ y)\}$
 $\langle proof \rangle$

lemma *vimage-Collect*: $(!x. P\ (f\ x) = Q\ x) \implies f\ -' (Collect\ P) = Collect\ Q$
 $\langle proof \rangle$

lemma *vimage-insert*: $f\ -' (insert\ a\ B) = (f\ -' \{a\})\ Un\ (f\ -' B)$
 — NOT suitable for rewriting because of the recurrence of $\{a\}$.
 $\langle proof \rangle$

lemma *vimage-Diff*: $f\ -' (A\ -\ B) = (f\ -' A) - (f\ -' B)$
 $\langle proof \rangle$

lemma *vimage-UNIV* [*simp*]: $f\ -' UNIV = UNIV$
 $\langle proof \rangle$

lemma *vimage-mono*: $A \subseteq B \implies f\ -' A \subseteq f\ -' B$
 — monotonicity
 $\langle proof \rangle$

lemma *vimage-image-eq* [*no-atp*]: $f\ -' (f\ ' A) = \{y. EX\ x:A. f\ x = f\ y\}$
 $\langle proof \rangle$

lemma *image-vimage-subset*: $f\ ' (f\ -' A) \leq A$
 $\langle proof \rangle$

lemma *image-vimage-eq* [simp]: $f \text{ ` } (f \text{ - ` } A) = A \text{ Int range } f$
 ⟨proof⟩

lemma *vimage-const* [simp]: $((\lambda x. c) \text{ - ` } A) = (\text{if } c \in A \text{ then UNIV else } \{\})$
 ⟨proof⟩

lemma *vimage-if* [simp]: $((\lambda x. \text{if } x \in B \text{ then } c \text{ else } d) \text{ - ` } A) =$
 $(\text{if } c \in A \text{ then } (\text{if } d \in A \text{ then UNIV else } B)$
 $\text{else if } d \in A \text{ then } \neg B \text{ else } \{\})$
 ⟨proof⟩

lemma *vimage-inter-cong*:
 $(\bigwedge w. w \in S \implies f w = g w) \implies f \text{ - ` } y \cap S = g \text{ - ` } y \cap S$
 ⟨proof⟩

lemma *image-Int-subset*: $f \text{ ` } (A \text{ Int } B) \leq f \text{ ` } A \text{ Int } f \text{ ` } B$
 ⟨proof⟩

lemma *image-diff-subset*: $f \text{ ` } A - f \text{ ` } B \leq f \text{ ` } (A - B)$
 ⟨proof⟩

6.4.6 Getting the Contents of a Singleton Set

definition *contents* :: 'a set \Rightarrow 'a where
 [code del]: *contents* $X = (\text{THE } x. X = \{x\})$

lemma *contents-eq* [simp]: *contents* $\{x\} = x$
 ⟨proof⟩

6.4.7 Least value operator

lemma *Least-mono*:
 $\text{mono } (f :: 'a :: \text{order} \Rightarrow 'b :: \text{order}) \implies \text{EX } x:S. \text{ ALL } y:S. x \leq y$
 $\implies (\text{LEAST } y. y : f \text{ ` } S) = f (\text{LEAST } x. x : S)$
 — Courtesy of Stephan Merz
 ⟨proof⟩

6.5 Misc

Rudimentary code generation

lemma *insert-code* [code]: *insert* $y A x \longleftrightarrow y = x \vee A x$
 ⟨proof⟩

lemma *vimage-code* [code]: $(f \text{ - ` } A) x = A (f x)$
 ⟨proof⟩

Misc theorem and ML bindings

lemmas *equality* $I = \text{subset-antisym}$

$\langle ML \rangle$

end

7 Typedef: HOL type definitions

```

theory Typedef
imports Set
uses
  (Tools/typedef.ML)
  (Tools/typecopy.ML)
  (Tools/typedef-codegen.ML)
begin

 $\langle ML \rangle$ 

locale type-definition =
  fixes Rep and Abs and A
  assumes Rep: Rep  $x \in A$ 
    and Rep-inverse: Abs (Rep  $x$ ) =  $x$ 
    and Abs-inverse:  $y \in A \implies$  Rep (Abs  $y$ ) =  $y$ 
  — This will be axiomatized for each typedef!
begin

lemma Rep-inject:
  (Rep  $x$  = Rep  $y$ ) = ( $x$  =  $y$ )
 $\langle proof \rangle$ 

lemma Abs-inject:
  assumes  $x: x \in A$  and  $y: y \in A$ 
  shows (Abs  $x$  = Abs  $y$ ) = ( $x$  =  $y$ )
 $\langle proof \rangle$ 

lemma Rep-cases [cases set]:
  assumes  $y: y \in A$ 
    and hyp:  $!!x. y = \text{Rep } x \implies P$ 
  shows  $P$ 
 $\langle proof \rangle$ 

lemma Abs-cases [cases type]:
  assumes  $r: !!y. x = \text{Abs } y \implies y \in A \implies P$ 
  shows  $P$ 
 $\langle proof \rangle$ 

lemma Rep-induct [induct set]:
  assumes  $y: y \in A$ 
    and hyp:  $!!x. P (\text{Rep } x)$ 
  shows  $P y$ 

```

<proof>

lemma *Abs-induct* [*induct type*]:
 assumes *r*: $\forall y. y \in A \implies P (Abs\ y)$
 shows $P\ x$
<proof>

lemma *Rep-range*: $range\ Rep = A$
<proof>

lemma *Abs-image*: $Abs\ ` A = UNIV$
<proof>

end

<ML>

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*: $x \in A \implies \sqcap A \sqsubseteq x$
 and *Inf-greatest*: $(\bigwedge x. x \in A \implies z \sqsubseteq x) \implies z \sqsubseteq \sqcap A$
 assumes *Sup-upper*: $x \in A \implies x \sqsubseteq \sqcup A$

and *Sup-least*: $(\bigwedge x. x \in A \implies x \sqsubseteq z) \implies \bigsqcup A \sqsubseteq z$
begin

lemma *dual-complete-lattice*:

class.complete-lattice *Sup Inf* (*op* \geq) (*op* $>$) (*op* \sqcup) (*op* \sqcap) $\top \perp$
 $\langle proof \rangle$

lemma *Inf-Sup*: $\sqcap A = \bigsqcup \{b. \forall a \in A. b \sqsubseteq a\}$
 $\langle proof \rangle$

lemma *Sup-Inf*: $\bigsqcup A = \sqcap \{b. \forall a \in A. a \sqsubseteq b\}$
 $\langle proof \rangle$

lemma *Inf-empty*:

$\sqcap \{\} = \top$
 $\langle proof \rangle$

lemma *Sup-empty*:

$\bigsqcup \{\} = \perp$
 $\langle proof \rangle$

lemma *Inf-insert*: $\sqcap \text{insert } a \ A = a \sqcap \sqcap A$
 $\langle proof \rangle$

lemma *Sup-insert*: $\bigsqcup \text{insert } a \ A = a \sqcup \bigsqcup A$
 $\langle proof \rangle$

lemma *Inf-singleton [simp]*:

$\sqcap \{a\} = a$
 $\langle proof \rangle$

lemma *Sup-singleton [simp]*:

$\bigsqcup \{a\} = a$
 $\langle proof \rangle$

lemma *Inf-binary*:

$\sqcap \{a, b\} = a \sqcap b$
 $\langle proof \rangle$

lemma *Sup-binary*:

$\bigsqcup \{a, b\} = a \sqcup b$
 $\langle proof \rangle$

lemma *Inf-UNIV*:

$\sqcap UNIV = \text{bot}$
 $\langle proof \rangle$

lemma *Sup-UNIV*:

$\bigsqcup UNIV = \text{top}$

$\langle proof \rangle$

lemma *Sup-le-iff*: $Sup\ A \sqsubseteq b \longleftrightarrow (\forall a \in A. a \sqsubseteq b)$
 $\langle proof \rangle$

lemma *le-Inf-iff*: $b \sqsubseteq Inf\ A \longleftrightarrow (\forall a \in A. b \sqsubseteq a)$
 $\langle proof \rangle$

definition *SUPR* :: $'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$ **where**
 $SUPR\ A\ f = \bigsqcup (f\ 'A)$

definition *INFI* :: $'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$ **where**
 $INFI\ A\ f = \bigsqcap (f\ 'A)$

end

syntax

-*SUP1* :: $pttrns \Rightarrow 'b \Rightarrow 'b$ $((3SUP\ -./\ -)\ [0, 10]\ 10)$
 -*SUP* :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b$ $((3SUP\ -./\ -)\ [0, 0, 10]\ 10)$
 -*INF1* :: $pttrns \Rightarrow 'b \Rightarrow 'b$ $((3INF\ -./\ -)\ [0, 10]\ 10)$
 -*INF* :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b$ $((3INF\ -./\ -)\ [0, 0, 10]\ 10)$

translations

$SUP\ x\ y. B == SUP\ x. SUP\ y. B$
 $SUP\ x. B == CONST\ SUPR\ CONST\ UNIV\ (\%x. B)$
 $SUP\ x. B == SUP\ x:CONST\ UNIV. B$
 $SUP\ x:A. B == CONST\ SUPR\ A\ (\%x. B)$
 $INF\ x\ y. B == INF\ x. INF\ y. B$
 $INF\ x. B == CONST\ INFI\ CONST\ UNIV\ (\%x. B)$
 $INF\ x. B == INF\ x:CONST\ UNIV. B$
 $INF\ x:A. B == CONST\ INFI\ A\ (\%x. B)$

$\langle ML \rangle$

context *complete-lattice*

begin

lemma *le-SUPI*: $i : A \Longrightarrow M\ i \sqsubseteq (SUP\ i:A. M\ i)$
 $\langle proof \rangle$

lemma *SUP-leI*: $(\bigwedge i. i : A \Longrightarrow M\ i \sqsubseteq u) \Longrightarrow (SUP\ i:A. M\ i) \sqsubseteq u$
 $\langle proof \rangle$

lemma *INF-leI*: $i : A \Longrightarrow (INF\ i:A. M\ i) \sqsubseteq M\ i$
 $\langle proof \rangle$

lemma *le-INF*: $(\bigwedge i. i : A \Longrightarrow u \sqsubseteq M\ i) \Longrightarrow u \sqsubseteq (INF\ i:A. M\ i)$
 $\langle proof \rangle$

lemma *SUP-le-iff*: $(\text{SUP } i:A. M \ i) \sqsubseteq u \longleftrightarrow (\forall i \in A. M \ i \sqsubseteq u)$
 $\langle \text{proof} \rangle$

lemma *le-INF-iff*: $u \sqsubseteq (\text{INF } i:A. M \ i) \longleftrightarrow (\forall i \in A. u \sqsubseteq M \ i)$
 $\langle \text{proof} \rangle$

lemma *SUP-const[simp]*: $A \neq \{\} \implies (\text{SUP } i:A. M) = M$
 $\langle \text{proof} \rangle$

lemma *INF-const[simp]*: $A \neq \{\} \implies (\text{INF } i:A. M) = M$
 $\langle \text{proof} \rangle$

end

8.3 *bool* and $- \Rightarrow -$ as complete lattice

instantiation *bool* :: *complete-lattice*

begin

definition

Inf-bool-def: $\sqcap A \longleftrightarrow (\forall x \in A. x)$

definition

Sup-bool-def: $\sqcup A \longleftrightarrow (\exists x \in A. x)$

instance $\langle \text{proof} \rangle$

end

lemma *Inf-empty-bool* [simp]:

$\sqcap \{\}$
 $\langle \text{proof} \rangle$

lemma *not-Sup-empty-bool* [simp]:

$\neg \sqcup \{\}$
 $\langle \text{proof} \rangle$

lemma *INF-bool-eq*:

$\text{INF} = \text{Ball}$

$\langle \text{proof} \rangle$

lemma *SUPR-bool-eq*:

$\text{SUPR} = \text{Bex}$

$\langle \text{proof} \rangle$

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 $\langle proof \rangle$

end

lemma *Inf-empty-fun*:

$\sqcap \{\} = (\lambda -. \sqcap \{\})$
 $\langle proof \rangle$

lemma *Sup-empty-fun*:

$\sqcup \{\} = (\lambda -. \sqcup \{\})$
 $\langle proof \rangle$

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*:

$\bigcup A = \{x. \exists B \in A. x \in B\}$
 $\langle proof \rangle$

lemma *Union-iff* [simp, no-atp]:

$A \in \bigcup C \longleftrightarrow (\exists X \in C. A \in X)$
 $\langle proof \rangle$

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.

$\langle proof \rangle$

lemma *UnionE* [elim!]:

$A \in \bigcup C \Longrightarrow (\bigwedge X. A \in X \Longrightarrow X \in C \Longrightarrow R) \Longrightarrow R$
 $\langle proof \rangle$

lemma *Union-upper*: $B \in A \Longrightarrow B \subseteq Union\ A$

$\langle proof \rangle$

lemma *Union-least*: $(!!X. X \in A \Longrightarrow X \subseteq C) \Longrightarrow Union\ A \subseteq C$

$\langle proof \rangle$

lemma *Un-eq-Union*: $A \cup B = \bigcup \{A, B\}$

$\langle proof \rangle$

lemma *Union-empty* [simp]: $Union(\{\}) = \{\}$
 $\langle proof \rangle$

lemma *Union-UNIV* [simp]: $Union\ UNIV = UNIV$
 $\langle proof \rangle$

lemma *Union-insert* [simp]: $Union\ (insert\ a\ B) = a \cup \bigcup B$
 $\langle proof \rangle$

lemma *Union-Un-distrib* [simp]: $\bigcup (A\ Un\ B) = \bigcup A \cup \bigcup B$
 $\langle proof \rangle$

lemma *Union-Int-subset*: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$
 $\langle proof \rangle$

lemma *Union-empty-conv* [simp,no-atp]: $(\bigcup A = \{\}) = (\forall x \in A. x = \{\})$
 $\langle proof \rangle$

lemma *empty-Union-conv* [simp,no-atp]: $(\{\} = \bigcup A) = (\forall x \in A. x = \{\})$
 $\langle proof \rangle$

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) = (\forall B \in C. B \cap A = \{\})$
 $\langle proof \rangle$

lemma *subset-Pow-Union*: $A \subseteq Pow\ (\bigcup A)$
 $\langle proof \rangle$

lemma *Union-Pow-eq* [simp]: $\bigcup (Pow\ A) = A$
 $\langle proof \rangle$

lemma *Union-mono*: $A \subseteq B ==> \bigcup A \subseteq \bigcup B$
 $\langle proof \rangle$

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* ==> 'b set ==> 'b set ((3 \bigcup (00-)/ -) [0, 10] 10)
 -UNION :: *pttrn* ==> 'a set ==> 'b set ==> 'b set ((3 \bigcup (00- \in -)/ -) [0, 0, 10] 10)

translations

$UN\ x\ y.\ B \quad == \quad UN\ x.\ UN\ y.\ B$
 $UN\ x.\ B \quad == \quad CONST\ UNION\ CONST\ UNIV\ (\%x.\ B)$
 $UN\ x.\ B \quad == \quad UN\ x:CONST\ UNIV.\ B$
 $UN\ x:A.\ B \quad == \quad CONST\ UNION\ A\ (\%x.\ B)$

Note the difference between ordinary xsymbol syntax of indexed unions and intersections (e.g. $\bigcup_{a_1 \in A_1} B$) and their L^AT_EX 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.

$\langle ML \rangle$

lemma *UNION-eq-Union-image*:

$(\bigcup_{x \in A} B\ x) = \bigcup (B'A)$
 $\langle proof \rangle$

lemma *Union-def*:

$\bigcup S = (\bigcup_{x \in S} x)$
 $\langle proof \rangle$

lemma *UNION-def [no-atp]*:

$(\bigcup_{x \in A} B\ x) = \{y.\ \exists x \in A.\ y \in B\ x\}$
 $\langle proof \rangle$

lemma *Union-image-eq [simp]*:

$\bigcup (B'A) = (\bigcup_{x \in A} B\ x)$
 $\langle proof \rangle$

lemma *UN-iff [simp]*: $(b: (UN\ x:A.\ B\ x)) = (EX\ x:A.\ b: B\ x)$

$\langle proof \rangle$

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.

$\langle proof \rangle$

lemma *UN-E [elim!]*: $b : (UN\ x:A.\ B\ x) ==> (!!x.\ x:A ==> b: B\ x ==> R)$
 $==> R$

$\langle proof \rangle$

lemma *UN-cong [cong]*:

$A = B ==> (!!x.\ x:B ==> C\ x = D\ x) ==> (UN\ x:A.\ C\ x) = (UN\ x:B.\ D\ x)$

$\langle proof \rangle$

lemma *strong-UN-cong*:

$A = B \implies (!x. x:B \text{ =simp=}> C\ x = D\ x) \implies (UN\ x:A. C\ x) = (UN\ x:B. D\ x)$
 $\langle proof \rangle$

lemma *image-eq-UN*: $f^{\cdot}A = (UN\ x:A. \{f\ x\})$

$\langle proof \rangle$

lemma *UN-upper*: $a \in A \implies B\ a \subseteq (\bigcup_{x \in A} B\ x)$

$\langle proof \rangle$

lemma *UN-least*: $(!x. x \in A \implies B\ x \subseteq C) \implies (\bigcup_{x \in A} B\ x) \subseteq C$

$\langle proof \rangle$

lemma *Collect-bex-eq* [no-atp]: $\{x. \exists y \in A. P\ x\ y\} = (\bigcup_{y \in A} \{x. P\ x\ y\})$

$\langle proof \rangle$

lemma *UN-insert-distrib*: $u \in A \implies (\bigcup_{x \in A} \text{insert}\ a\ (B\ x)) = \text{insert}\ a\ (\bigcup_{x \in A} B\ x)$

$\langle proof \rangle$

lemma *UN-empty* [simp,no-atp]: $(\bigcup_{x \in \{\}} B\ x) = \{\}$

$\langle proof \rangle$

lemma *UN-empty2* [simp]: $(\bigcup_{x \in A} \{\}) = \{\}$

$\langle proof \rangle$

lemma *UN-singleton* [simp]: $(\bigcup_{x \in A} \{x\}) = A$

$\langle proof \rangle$

lemma *UN-absorb*: $k \in I \implies A\ k \cup (\bigcup_{i \in I} A\ i) = (\bigcup_{i \in I} A\ i)$

$\langle proof \rangle$

lemma *UN-insert* [simp]: $(\bigcup_{x \in \text{insert}\ a\ A} B\ x) = B\ a \cup \text{UNION}\ A\ B$

$\langle proof \rangle$

lemma *UN-Un*[simp]: $(\bigcup_{i \in A \cup B} M\ i) = (\bigcup_{i \in A} M\ i) \cup (\bigcup_{i \in B} M\ i)$

$\langle proof \rangle$

lemma *UN-UN-flatten*: $(\bigcup_{x \in (\bigcup_{y \in A} B\ y)} C\ x) = (\bigcup_{y \in A} \bigcup_{x \in B\ y} C\ x)$

$\langle proof \rangle$

lemma *UN-subset-iff*: $((\bigcup_{i \in I} A\ i) \subseteq B) = (\forall i \in I. A\ i \subseteq B)$

$\langle proof \rangle$

lemma *image-Union*: $f^{\cdot} \bigcup S = (\bigcup_{x \in S} f^{\cdot} x)$

$\langle proof \rangle$

lemma *UN-constant* [simp]: $(\bigcup y \in A. c) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } c)$
 ⟨proof⟩

lemma *UN-eq*: $(\bigcup x \in A. B\ x) = \bigcup (\{Y. \exists x \in A. Y = B\ x\})$
 ⟨proof⟩

lemma *UNION-empty-conv*[simp]:
 $(\{\} = (\bigcup x:A. B\ x)) = (\forall x \in A. B\ x = \{\})$
 $((\bigcup x:A. B\ x) = \{\}) = (\forall x \in A. B\ x = \{\})$
 ⟨proof⟩

lemma *Collect-ex-eq* [no-atp]: $\{x. \exists y. P\ x\ y\} = (\bigcup y. \{x. P\ x\ y\})$
 ⟨proof⟩

lemma *ball-UN*: $(\forall z \in \text{UNION } A\ B. P\ z) = (\forall x \in A. \forall z \in B\ x. P\ z)$
 ⟨proof⟩

lemma *bex-UN*: $(\exists z \in \text{UNION } A\ B. P\ z) = (\exists x \in A. \exists z \in B\ x. P\ z)$
 ⟨proof⟩

lemma *Un-eq-UN*: $A \cup B = (\bigcup b. \text{if } b \text{ then } A \text{ else } B)$
 ⟨proof⟩

lemma *UN-bool-eq*: $(\bigcup b::\text{bool}. A\ b) = (A\ \text{True} \cup A\ \text{False})$
 ⟨proof⟩

lemma *UN-Pow-subset*: $(\bigcup x \in A. \text{Pow } (B\ x)) \subseteq \text{Pow } (\bigcup x \in A. B\ x)$
 ⟨proof⟩

lemma *UN-mono*:
 $A \subseteq B \implies (!x. x \in A \implies f\ x \subseteq g\ x) \implies$
 $(\bigcup x \in A. f\ x) \subseteq (\bigcup x \in B. g\ x)$
 ⟨proof⟩

lemma *image-Union*: $f\ -' (\text{Union } A) = (\bigcup X:A. f\ -' X)$
 ⟨proof⟩

lemma *image-UN*: $f\ -' (\bigcup x:A. B\ x) = (\bigcup x:A. f\ -' B\ x)$
 ⟨proof⟩

lemma *image-eq-UN*: $f\ -' B = (\bigcup y: B. f\ -' \{y\})$
 — NOT suitable for rewriting
 ⟨proof⟩

lemma *image-UN*: $(f\ -' (\text{UNION } A\ B)) = (\bigcup x:A. (f\ -' (B\ x)))$
 ⟨proof⟩

8.6 Inter

abbreviation *Inter* :: 'a set set \Rightarrow 'a set **where**

$$\text{Inter } S \equiv \bigcap S$$

notation (*xsymbols*)

$$\text{Inter } (\bigcap - [90] 90)$$

lemma *Inter-eq* [*code del*]:

$$\bigcap A = \{x. \forall B \in A. x \in B\}$$

<proof>

lemma *Inter-iff* [*simp,no-atp*]: $(A : \text{Inter } C) = (\text{ALL } X:C. A:X)$

<proof>

lemma *InterI* [*intro!*]: $(!!X. X:C \Rightarrow A:X) \Rightarrow A : \text{Inter } C$

<proof>

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*]: $A : \text{Inter } C \Rightarrow X:C \Rightarrow A:X$

<proof>

lemma *InterE* [*elim*]: $A : \text{Inter } C \Rightarrow (X \sim C \Rightarrow R) \Rightarrow (A:X \Rightarrow R)$

— “Classical” elimination rule – does not require proving $X \in C$.

<proof>

lemma *Inter-lower*: $B \in A \Rightarrow \text{Inter } A \subseteq B$

<proof>

lemma *Inter-subset*:

$$[!X. X \in A \Rightarrow X \subseteq B; A \sim \{\}] \Rightarrow \bigcap A \subseteq B$$

<proof>

lemma *Inter-greatest*: $(!X. X \in A \Rightarrow C \subseteq X) \Rightarrow C \subseteq \text{Inter } A$

<proof>

lemma *Int-eq-Inter*: $A \cap B = \bigcap \{A, B\}$

<proof>

lemma *Inter-empty* [*simp*]: $\bigcap \{\} = \text{UNIV}$

<proof>

lemma *Inter-UNIV* [*simp*]: $\bigcap \text{UNIV} = \{\}$

<proof>

lemma *Inter-insert* [*simp*]: $\bigcap (\text{insert } a B) = a \cap \bigcap B$

<proof>

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
 $\langle \text{proof} \rangle$

lemma *Inter-Un-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$
 $\langle \text{proof} \rangle$

lemma *Inter-UNIV-conv* [*simp,no-atp*]:
 $(\bigcap A = \text{UNIV}) = (\forall x \in A. x = \text{UNIV})$
 $(\text{UNIV} = \bigcap A) = (\forall x \in A. x = \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *Inter-anti-mono*: $B \subseteq A \implies \bigcap A \subseteq \bigcap B$
 $\langle \text{proof} \rangle$

8.7 Intersections of families

abbreviation *INTER* :: 'a set \Rightarrow ('a \Rightarrow 'b set) \Rightarrow 'b set **where**
INTER \equiv *INFI*

syntax

-*INTER1* :: *pttrns* \Rightarrow 'b set \Rightarrow 'b set $((\exists \text{INT} \text{ -./ -}) [0, 10] 10)$
-*INTER* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set $((\exists \text{INT} \text{ :-./ -}) [0, 0, 10] 10)$

syntax (*xsymbols*)

-*INTER1* :: *pttrns* \Rightarrow 'b set \Rightarrow 'b set $((\exists \bigcap \text{ -./ -}) [0, 10] 10)$
-*INTER* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set $((\exists \bigcap \text{ -}\in\text{ -./ -}) [0, 0, 10] 10)$

syntax (*latex output*)

-*INTER1* :: *pttrns* \Rightarrow 'b set \Rightarrow 'b set $((\exists \bigcap (00\text{ -}) / \text{ -}) [0, 10] 10)$
-*INTER* :: *pttrn* \Rightarrow 'a set \Rightarrow 'b set \Rightarrow 'b set $((\exists \bigcap (00\text{ -}\in\text{ -}) / \text{ -}) [0, 0, 10] 10)$

translations

INT *x y. B* == *INT* *x. INT y. B*
INT *x. B* == *CONST INTER CONST UNIV* (%*x. B*)
INT *x. B* == *INT x:CONST UNIV. B*
INT *x:A. B* == *CONST INTER A* (%*x. B*)

$\langle \text{ML} \rangle$

lemma *INTER-eq-Inter-image*:

$(\bigcap_{x \in A} B \ x) = \bigcap (B \text{' } A)$
 $\langle \text{proof} \rangle$

lemma *Inter-def*:

$\bigcap S = (\bigcap_{x \in S} x)$
 $\langle \text{proof} \rangle$

lemma *INTER-def*:

$$(\bigcap_{x \in A}. B\ x) = \{y. \forall x \in A. y \in B\ x\}$$

<proof>

lemma *Inter-image-eq* [*simp*]:

$$\bigcap (B' A) = (\bigcap_{x \in A}. B\ x)$$

<proof>

lemma *INT-iff* [*simp*]: $(b: (INT\ x:A. B\ x)) = (ALL\ x:A. b: B\ x)$

<proof>

lemma *INT-I* [*intro!*]: $(!!x. x:A ==> b: B\ x) ==> b : (INT\ x:A. B\ x)$

<proof>

lemma *INT-D* [*elim*]: $b : (INT\ x:A. B\ x) ==> a:A ==> b: B\ a$

<proof>

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 $a \in A$.

<proof>

lemma *INT-cong* [*cong*]:

$$A = B ==> (!!x. x:B ==> C\ x = D\ x) ==> (INT\ x:A. C\ x) = (INT\ x:B. D\ x)$$

<proof>

lemma *Collect-ball-eq*: $\{x. \forall y \in A. P\ x\ y\} = (\bigcap_{y \in A}. \{x. P\ x\ y\})$

<proof>

lemma *Collect-all-eq*: $\{x. \forall y. P\ x\ y\} = (\bigcap y. \{x. P\ x\ y\})$

<proof>

lemma *INT-lower*: $a \in A ==> (\bigcap_{x \in A}. B\ x) \subseteq B\ a$

<proof>

lemma *INT-greatest*: $(!!x. x \in A ==> C \subseteq B\ x) ==> C \subseteq (\bigcap_{x \in A}. B\ x)$

<proof>

lemma *INT-empty* [*simp*]: $(\bigcap_{x \in \{\}}. B\ x) = UNIV$

<proof>

lemma *INT-absorb*: $k \in I ==> A\ k \cap (\bigcap_{i \in I}. A\ i) = (\bigcap_{i \in I}. A\ i)$

<proof>

lemma *INT-subset-iff*: $(B \subseteq (\bigcap_{i \in I}. A\ i)) = (\forall i \in I. B \subseteq A\ i)$

<proof>

lemma *INT-insert [simp]*: $(\bigcap x \in \text{insert } a \ A. B \ x) = B \ a \cap \text{INTER } A \ B$
 $\langle \text{proof} \rangle$

lemma *INT-Un*: $(\bigcap i \in A \cup B. M \ i) = (\bigcap i \in A. M \ i) \cap (\bigcap i \in B. M \ i)$
 $\langle \text{proof} \rangle$

lemma *INT-insert-distrib*:
 $u \in A \implies (\bigcap x \in A. \text{insert } a \ (B \ x)) = \text{insert } a \ (\bigcap x \in A. B \ x)$
 $\langle \text{proof} \rangle$

lemma *INT-constant [simp]*: $(\bigcap y \in A. c) = (\text{if } A = \{\} \text{ then } \text{UNIV} \text{ else } c)$
 $\langle \text{proof} \rangle$

lemma *INT-eq*: $(\bigcap x \in A. B \ x) = \bigcap (\{Y. \exists x \in A. Y = B \ x\})$
 — Look: it has an *existential* quantifier
 $\langle \text{proof} \rangle$

lemma *INTER-UNIV-conv[simp]*:
 $(\text{UNIV} = (\text{INT } x:A. B \ x)) = (\forall x \in A. B \ x = \text{UNIV})$
 $((\text{INT } x:A. B \ x) = \text{UNIV}) = (\forall x \in A. B \ x = \text{UNIV})$
 $\langle \text{proof} \rangle$

lemma *INT-bool-eq*: $(\bigcap b::\text{bool}. A \ b) = (A \ \text{True} \cap A \ \text{False})$
 $\langle \text{proof} \rangle$

lemma *Pow-INT-eq*: $\text{Pow } (\bigcap x \in A. B \ x) = (\bigcap x \in A. \text{Pow } (B \ x))$
 $\langle \text{proof} \rangle$

lemma *INT-anti-mono*:
 $B \subseteq A \implies (!x. x \in A \implies f \ x \subseteq g \ x) \implies$
 $(\bigcap x \in A. f \ x) \subseteq (\bigcap x \in A. g \ x)$
 — The last inclusion is POSITIVE!
 $\langle \text{proof} \rangle$

lemma *image-INT*: $f - ' (\text{INT } x:A. B \ x) = (\text{INT } x:A. f - ' B \ x)$
 $\langle \text{proof} \rangle$

8.8 Distributive laws

lemma *Int-Union*: $A \cap \bigcup B = (\bigcup C \in B. A \cap C)$
 $\langle \text{proof} \rangle$

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C \in B. C \cap A)$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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
 $\langle proof \rangle$

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap_{C \in B}. A \cup C)$
 $\langle proof \rangle$

lemma *Int-Inter-image*: $(\bigcap_{x \in C}. A \ x \cap B \ x) = \bigcap (A' C) \cap \bigcap (B' C)$
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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.
 $\langle proof \rangle$

lemma *Un-INT-distrib*: $B \cup (\bigcap_{i \in I}. A \ i) = (\bigcap_{i \in I}. B \cup A \ i)$
 $\langle proof \rangle$

lemma *Int-UN-distrib2*: $(\bigcup_{i \in I}. A \ i) \cap (\bigcup_{j \in J}. B \ j) = (\bigcup_{i \in I}. \bigcup_{j \in J}. A \ i \cap B \ j)$
 $\langle proof \rangle$

lemma *Un-INT-distrib2*: $(\bigcap_{i \in I}. A \ i) \cup (\bigcap_{j \in J}. B \ j) = (\bigcap_{i \in I}. \bigcap_{j \in J}. A \ i \cup B \ j)$
 $\langle proof \rangle$

8.9 Complement

lemma *Compl-UN* [simp]: $-(\bigcup_{x \in A}. B \ x) = (\bigcap_{x \in A}. -B \ x)$
 $\langle proof \rangle$

lemma *Compl-INT* [simp]: $-(\bigcap_{x \in A}. B \ x) = (\bigcup_{x \in A}. -B \ x)$
 $\langle proof \rangle$

8.10 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [simp]:
 $!!a \ B \ C. (\bigcup_{x:C}. \text{insert } a \ (B \ x)) = (\text{if } C=\{\} \text{ then } \{\} \text{ else insert } a \ (\bigcup_{x:C}. B \ x))$
 $!!A \ B \ C. (\bigcup_{x:C}. A \ x \ \text{Un } B) = ((\text{if } C=\{\} \text{ then } \{\} \text{ else } (\bigcup_{x:C}. A \ x) \ \text{Un } B))$
 $!!A \ B \ C. (\bigcup_{x:C}. A \ \text{Un } B \ x) = ((\text{if } C=\{\} \text{ then } \{\} \text{ else } A \ \text{Un } (\bigcup_{x:C}. B \ x)))$

$!!A B C. (UN x:C. A x Int B) = ((UN x:C. A x) Int B)$
 $!!A B C. (UN x:C. A Int B x) = (A Int (UN x:C. B x))$
 $!!A B C. (UN x:C. A x - B) = ((UN x:C. A x) - B)$
 $!!A B C. (UN x:C. A - B x) = (A - (INT x:C. B x))$
 $!!A B. (UN x: Union A. B x) = (UN y:A. UN x:y. B x)$
 $!!A B C. (UN z: UNION A B. C z) = (UN x:A. UN z: B(x). C z)$
 $!!A B f. (UN x:f'A. B x) = (UN a:A. B (f a))$
 $\langle proof \rangle$

lemma *INT-simps* [*simp*]:

$!!A B C. (INT x:C. A x Int B) = (if C=\{\} then UNIV else (INT x:C. A x) Int B)$
 $!!A B C. (INT x:C. A Int B x) = (if C=\{\} then UNIV else A Int (INT x:C. B x))$
 $!!A B C. (INT x:C. A x - B) = (if C=\{\} then UNIV else (INT x:C. A x) - B)$
 $!!A B C. (INT x:C. A - B x) = (if C=\{\} then UNIV else A - (UN x:C. B x))$
 $!!a B C. (INT x:C. insert a (B x)) = insert a (INT x:C. B x)$
 $!!A B C. (INT x:C. A x Un B) = ((INT x:C. A x) Un B)$
 $!!A B C. (INT x:C. A Un B x) = (A Un (INT x:C. B x))$
 $!!A B. (INT x: Union A. B x) = (INT y:A. INT x:y. B x)$
 $!!A B C. (INT z: UNION A B. C z) = (INT x:A. INT z: B(x). C z)$
 $!!A B f. (INT x:f'A. B x) = (INT a:A. B (f a))$
 $\langle proof \rangle$

lemma *ball-simps* [*simp,no-atp*]:

$!!A P Q. (ALL x:A. P x \mid Q) = ((ALL x:A. P x) \mid Q)$
 $!!A P Q. (ALL x:A. P \mid Q x) = (P \mid (ALL x:A. Q x))$
 $!!A P Q. (ALL x:A. P \dashv\rightarrow Q x) = (P \dashv\rightarrow (ALL x:A. Q x))$
 $!!A P Q. (ALL x:A. P x \dashv\rightarrow Q) = ((EX x:A. P x) \dashv\rightarrow Q)$
 $!!P. (ALL x:\{\}. P x) = True$
 $!!P. (ALL x:UNIV. P x) = (ALL x. P x)$
 $!!a B P. (ALL x:insert a B. P x) = (P a \& (ALL x:B. P x))$
 $!!A P. (ALL x:Union A. P x) = (ALL y:A. ALL x:y. P x)$
 $!!A B P. (ALL x: UNION A B. P x) = (ALL a:A. ALL x: B a. P x)$
 $!!P Q. (ALL x:Collect Q. P x) = (ALL x. Q x \dashv\rightarrow P x)$
 $!!A P f. (ALL x:f'A. P x) = (ALL x:A. P (f x))$
 $!!A P. (\sim (ALL x:A. P x)) = (EX x:A. \sim P x)$
 $\langle proof \rangle$

lemma *bex-simps* [*simp,no-atp*]:

$!!A P Q. (EX x:A. P x \& Q) = ((EX x:A. P x) \& Q)$
 $!!A P Q. (EX x:A. P \& Q x) = (P \& (EX x:A. Q x))$
 $!!P. (EX x:\{\}. P x) = False$
 $!!P. (EX x:UNIV. P x) = (EX x. P x)$
 $!!a B P. (EX x:insert a B. P x) = (P(a) \mid (EX x:B. P x))$
 $!!A P. (EX x:Union A. P x) = (EX y:A. EX x:y. P x)$
 $!!A B P. (EX x: UNION A B. P x) = (EX a:A. EX x:B a. P x)$

$!!P \ Q. (EX \ x:Collect \ Q. \ P \ x) = (EX \ x. \ Q \ x \ \& \ P \ x)$
 $!!A \ P \ f. (EX \ x:f'A. \ P \ x) = (EX \ x:A. \ P \ (f \ x))$
 $!!A \ P. (\sim (EX \ x:A. \ P \ x)) = (ALL \ x:A. \ \sim P \ x)$
 $\langle proof \rangle$

lemma *ball-conj-distrib*:

$(ALL \ x:A. \ P \ x \ \& \ Q \ x) = ((ALL \ x:A. \ P \ x) \ \& \ (ALL \ x:A. \ Q \ x))$
 $\langle proof \rangle$

lemma *bex-disj-distrib*:

$(EX \ x:A. \ P \ x \ | \ Q \ x) = ((EX \ x:A. \ P \ x) \ | \ (EX \ x:A. \ Q \ x))$
 $\langle proof \rangle$

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:

$!!a \ B \ C. \ insert \ a \ (UN \ x:C. \ B \ x) = (if \ C=\{\} \ then \ \{a\} \ else \ (UN \ x:C. \ insert \ a \ (B \ x)))$
 $!!A \ B \ C. \ (UN \ x:C. \ A \ x) \ Un \ B = (if \ C=\{\} \ then \ B \ else \ (UN \ x:C. \ A \ x \ Un \ B))$
 $!!A \ B \ C. \ A \ Un \ (UN \ x:C. \ B \ x) = (if \ C=\{\} \ then \ A \ else \ (UN \ x:C. \ A \ Un \ B \ x))$
 $!!A \ B \ C. \ ((UN \ x:C. \ A \ x) \ Int \ B) = (UN \ x:C. \ A \ x \ Int \ B)$
 $!!A \ B \ C. \ (A \ Int \ (UN \ x:C. \ B \ x)) = (UN \ x:C. \ A \ Int \ B \ x)$
 $!!A \ B \ C. \ ((UN \ x:C. \ A \ x) - B) = (UN \ x:C. \ A \ x - B)$
 $!!A \ B \ C. \ (A - (INT \ x:C. \ B \ x)) = (UN \ x:C. \ A - B \ x)$
 $!!A \ B. \ (UN \ y:A. \ UN \ x:y. \ B \ x) = (UN \ x: \ Union \ A. \ B \ x)$
 $!!A \ B \ C. \ (UN \ x:A. \ UN \ z: \ B(x). \ C \ z) = (UN \ z: \ UNION \ A \ B. \ C \ z)$
 $!!A \ B \ f. \ (UN \ a:A. \ B \ (f \ a)) = (UN \ x:f'A. \ B \ x)$
 $\langle proof \rangle$

lemma *INT-extend-simps*:

$!!A \ B \ C. \ (INT \ x:C. \ A \ x) \ Int \ B = (if \ C=\{\} \ then \ B \ else \ (INT \ x:C. \ A \ x \ Int \ B))$
 $!!A \ B \ C. \ A \ Int \ (INT \ x:C. \ B \ x) = (if \ C=\{\} \ then \ A \ else \ (INT \ x:C. \ A \ Int \ B \ x))$
 $!!A \ B \ C. \ (INT \ x:C. \ A \ x) - B = (if \ C=\{\} \ then \ UNIV - B \ else \ (INT \ x:C. \ A \ x - B))$
 $!!A \ B \ C. \ A - (UN \ x:C. \ B \ x) = (if \ C=\{\} \ then \ A \ else \ (INT \ x:C. \ A - B \ x))$
 $!!a \ B \ C. \ insert \ a \ (INT \ x:C. \ B \ x) = (INT \ x:C. \ insert \ a \ (B \ x))$
 $!!A \ B \ C. \ ((INT \ x:C. \ A \ x) \ Un \ B) = (INT \ x:C. \ A \ x \ Un \ B)$
 $!!A \ B \ C. \ A \ Un \ (INT \ x:C. \ B \ x) = (INT \ x:C. \ A \ Un \ B \ x)$
 $!!A \ B. \ (INT \ y:A. \ INT \ x:y. \ B \ x) = (INT \ x: \ Union \ A. \ B \ x)$
 $!!A \ B \ C. \ (INT \ x:A. \ INT \ z: \ B(x). \ C \ z) = (INT \ z: \ UNION \ A \ B. \ C \ z)$
 $!!A \ B \ f. \ (INT \ a:A. \ B \ (f \ a)) = (INT \ x:f'A. \ B \ x)$
 $\langle proof \rangle$

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*: $f\ A \leq A \implies lfp\ f \leq A$

$\langle \text{proof} \rangle$

lemma *lfp-greatest*: $(!!u. f\ u \leq u \implies A \leq u) \implies A \leq \text{lfp}\ f$
 $\langle \text{proof} \rangle$

end

lemma *lfp-lemma2*: $\text{mono}\ f \implies f\ (\text{lfp}\ f) \leq \text{lfp}\ f$
 $\langle \text{proof} \rangle$

lemma *lfp-lemma3*: $\text{mono}\ f \implies \text{lfp}\ f \leq f\ (\text{lfp}\ f)$
 $\langle \text{proof} \rangle$

lemma *lfp-unfold*: $\text{mono}\ f \implies \text{lfp}\ f = f\ (\text{lfp}\ f)$
 $\langle \text{proof} \rangle$

lemma *lfp-const*: $\text{lfp}\ (\lambda x. t) = t$
 $\langle \text{proof} \rangle$

9.3 General induction rules for least fixed points

theorem *lfp-induct*:
assumes *mono*: $\text{mono}\ f$ **and** *ind*: $f\ (\inf\ (\text{lfp}\ f)\ P) \leq P$
shows $\text{lfp}\ f \leq P$
 $\langle \text{proof} \rangle$

lemma *lfp-induct-set*:
assumes *lfp*: $a: \text{lfp}(f)$
and *mono*: $\text{mono}(f)$
and *indhyp*: $!!x. [\mid x: f(\text{lfp}(f)\ \text{Int}\ \{x.\ P(x)\}) \mid] \implies P(x)$
shows $P(a)$
 $\langle \text{proof} \rangle$

lemma *lfp-ordinal-induct*:
fixes $f :: 'a::\text{complete-lattice} \Rightarrow 'a$
assumes *mono*: $\text{mono}\ f$
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 $P\ (\text{lfp}\ f)$
 $\langle \text{proof} \rangle$

lemma *lfp-ordinal-induct-set*:
assumes *mono*: $\text{mono}\ f$
and *P-f*: $!!S. P\ S \implies P(f\ S)$
and *P-Union*: $!!M. !S:M. P\ S \implies P(\text{Union}\ M)$
shows $P(\text{lfp}\ f)$
 $\langle \text{proof} \rangle$

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

lemma *def-lfp-unfold*: $\llbracket h == \text{lfp}(f); \text{mono}(f) \rrbracket ==> h = f(h)$
 $\langle \text{proof} \rangle$

lemma *def-lfp-induct*:
 $\llbracket A == \text{lfp}(f); \text{mono}(f);$
 $f(\inf A P) \leq P$
 $\rrbracket ==> A \leq P$
 $\langle \text{proof} \rangle$

lemma *def-lfp-induct-set*:
 $\llbracket A == \text{lfp}(f); \text{mono}(f); a:A;$
 $!!x. \llbracket x: f(A \text{ Int } \{x. P(x)\}) \rrbracket ==> P(x)$
 $\rrbracket ==> P(a)$
 $\langle \text{proof} \rangle$

lemma *lfp-mono*: $(!!Z. f Z \leq g Z) ==> \text{lfp } f \leq \text{lfp } g$
 $\langle \text{proof} \rangle$

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*: $X \leq f X ==> X \leq \text{gfp } f$
 $\langle \text{proof} \rangle$

lemma *gfp-least*: $(!!u. u \leq f u ==> u \leq X) ==> \text{gfp } f \leq X$
 $\langle \text{proof} \rangle$

lemma *gfp-lemma2*: $\text{mono } f ==> \text{gfp } f \leq f(\text{gfp } f)$
 $\langle \text{proof} \rangle$

lemma *gfp-lemma3*: $\text{mono } f ==> f(\text{gfp } f) \leq \text{gfp } f$
 $\langle \text{proof} \rangle$

lemma *gfp-unfold*: $\text{mono } f ==> \text{gfp } f = f(\text{gfp } f)$
 $\langle \text{proof} \rangle$

9.5 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: $\llbracket a: X; X \subseteq f(X) \rrbracket ==> a : \text{gfp}(f)$
 $\langle \text{proof} \rangle$

lemma *weak-coinduct-image*: $!!X. \llbracket a: X; g'X \subseteq f(g'X) \rrbracket ==> g a : \text{gfp } f$
 $\langle \text{proof} \rangle$

lemma *coinduct-lemma*:
 $\llbracket X \leq f(\sup X(\text{gfp } f)); \text{mono } f \rrbracket ==> \sup X(\text{gfp } f) \leq f(\sup X(\text{gfp } f))$

$\langle \text{proof} \rangle$

strong version, thanks to Coen and Frost

lemma *coinduct-set*: $\llbracket \text{mono}(f); a: X; X \subseteq f(X \text{ Un } \text{gfp}(f)) \rrbracket \implies a : \text{gfp}(f)$
 $\langle \text{proof} \rangle$

lemma *coinduct*: $\llbracket \text{mono}(f); X \leq f(\text{sup } X (\text{gfp } f)) \rrbracket \implies X \leq \text{gfp}(f)$
 $\langle \text{proof} \rangle$

lemma *gfp-fun-UnI2*: $\llbracket \text{mono}(f); a: \text{gfp}(f) \rrbracket \implies a: f(X \text{ Un } \text{gfp}(f))$
 $\langle \text{proof} \rangle$

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*: $\text{mono}(f) \implies \text{mono}(\%x. f(x) \text{ Un } X \text{ Un } B)$
 $\langle \text{proof} \rangle$

lemma *coinduct3-lemma*:
 $\llbracket X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f))); \text{mono}(f) \rrbracket$
 $\implies \text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)) \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f)))$
 $\langle \text{proof} \rangle$

lemma *coinduct3*:
 $\llbracket \text{mono}(f); a: X; X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } \text{gfp}(f))) \rrbracket \implies a : \text{gfp}(f)$
 $\langle \text{proof} \rangle$

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

lemma *def-gfp-unfold*: $\llbracket A == \text{gfp}(f); \text{mono}(f) \rrbracket \implies A = f(A)$
 $\langle \text{proof} \rangle$

lemma *def-coinduct*:
 $\llbracket A == \text{gfp}(f); \text{mono}(f); X \leq f(\text{sup } X A) \rrbracket \implies X \leq A$
 $\langle \text{proof} \rangle$

lemma *def-coinduct-set*:
 $\llbracket A == \text{gfp}(f); \text{mono}(f); a: X; X \subseteq f(X \text{ Un } A) \rrbracket \implies a: A$
 $\langle \text{proof} \rangle$

lemma *def-Collect-coinduct*:
 $\llbracket A == \text{gfp}(\%w. \text{Collect}(P(w))); \text{mono}(\%w. \text{Collect}(P(w)));$
 $a: X; \llbracket !z. z: X \implies P(X \text{ Un } A) z \rrbracket \implies$
 $a : A$
 $\langle \text{proof} \rangle$

lemma *def-coinduct3*:
 $\llbracket A == \text{gfp}(f); \text{mono}(f); a: X; X \subseteq f(\text{lfp}(\%x. f(x) \text{ Un } X \text{ Un } A)) \rrbracket \implies a: A$

<proof>

Monotonicity of *gfp*!

lemma *gfp-mono*: (!*Z*. *f Z* ≤ *g Z*) ==> *gfp f* ≤ *gfp g*
<proof>

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

<ML>

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.

<ML>

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* ==> '*a*' ==> '*b*' ((%-) 10)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* ==> '*a*' ==> '*b*' ((λ-) 10)

<ML>

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* ⟷ (∀ *x*. *f x = g x*)
<proof>

lemma *apply-inverse*:

$f\ x = u \implies (\bigwedge x. P\ x \implies g\ (f\ x) = x) \implies P\ x \implies x = g\ u$
 $\langle proof \rangle$

10.1 The Identity Function *id*

definition

$id :: 'a \Rightarrow 'a$

where

$id = (\lambda x. x)$

lemma *id-apply* [*simp*]: $id\ x = x$

$\langle proof \rangle$

lemma *image-ident* [*simp*]: $(\%x. x)\ 'Y = Y$

$\langle proof \rangle$

lemma *image-id* [*simp*]: $id\ 'Y = Y$

$\langle proof \rangle$

lemma *vimage-ident* [*simp*]: $(\%x. x)\ -'Y = Y$

$\langle proof \rangle$

lemma *vimage-id* [*simp*]: $id\ -'A = A$

$\langle proof \rangle$

10.2 The Composition Operator $f \circ g$

definition

$comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c\ (\text{infixl } o\ 55)$

where

$f\ o\ g = (\lambda x. f\ (g\ x))$

notation (*xsymbols*)

$comp\ (\text{infixl } o\ 55)$

notation (*HTML output*)

$comp\ (\text{infixl } o\ 55)$

compatibility

lemmas *o-def* = *comp-def*

lemma *o-apply* [*simp*]: $(f\ o\ g)\ x = f\ (g\ x)$

$\langle proof \rangle$

lemma *o-assoc*: $f\ o\ (g\ o\ h) = f\ o\ g\ o\ h$

$\langle proof \rangle$

lemma *id-o* [simp]: $id \circ g = g$
 $\langle proof \rangle$

lemma *o-id* [simp]: $f \circ id = f$
 $\langle proof \rangle$

lemma *o-eq-dest*:
 $a \circ b = c \circ d \implies a (b \ v) = c (d \ v)$
 $\langle proof \rangle$

lemma *o-eq-elim*:
 $a \circ b = c \circ d \implies ((\bigwedge v. a (b \ v) = c (d \ v)) \implies R) \implies R$
 $\langle proof \rangle$

lemma *image-compose*: $(f \circ g) \cdot r = f(g \cdot r)$
 $\langle proof \rangle$

lemma *image-compose*: $(g \circ f) \cdot x = f \cdot (g \cdot x)$
 $\langle proof \rangle$

lemma *UN-o*: $UNION \ A \ (g \circ f) = UNION \ (f \cdot A) \ g$
 $\langle proof \rangle$

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*: $(f \ o> \ g) \ x = g \ (f \ x)$
 $\langle proof \rangle$

lemma *fcomp-assoc*: $(f \ o> \ g) \ o> \ h = f \ o> \ (g \ o> \ h)$
 $\langle proof \rangle$

lemma *id-fcomp* [simp]: $id \ o> \ g = g$
 $\langle proof \rangle$

lemma *fcomp-id* [simp]: $f \ o> \ id = f$
 $\langle proof \rangle$

code-const *fcomp*
 (*Eval* **infixl** 1 **#>**)

no-notation *fcomp* (**infixl** *o>* 60)

10.4 Injectivity and Surjectivity

definition

inj-on :: [*'a* => *'b*, *'a set*] => *bool* **where**
 — injective
inj-on *f* *A* == ! *x:A*. ! *y:A*. *f*(*x*)=*f*(*y*) --> *x=y*

A common special case: functions injective over the entire domain type.

abbreviation

inj *f* == *inj-on* *f* *UNIV*

definition

bij-betw :: (*'a* => *'b*) => *'a set* => *'b set* => *bool* **where** — bijective
 [code del]: *bij-betw* *f* *A* *B* <=> *inj-on* *f* *A* & *f* ' *A* = *B*

definition

surj :: (*'a* => *'b*) => *bool* **where**
 — surjective
surj *f* == ! *y*. ? *x*. *y*=*f*(*x*)

definition

bij :: (*'a* => *'b*) => *bool* **where**
 — bijective
bij *f* == *inj* *f* & *surj* *f*

lemma *injI*:

assumes $\bigwedge x y. f\ x = f\ y \implies x = y$
shows *inj* *f*
 <proof>

For Proofs in *Tools/Datatype/datatype-rep-proofs*

lemma *datatype-injI*:

(! *x*. *ALL* *y*. *f*(*x*) = *f*(*y*) --> *x=y*) ==> *inj*(*f*)
 <proof>

theorem *range-ex1-eq*: *inj* *f* ==> *b* : *range* *f* = (*EX*! *x*. *b* = *f* *x*)
 <proof>

lemma *injD*: [| *inj*(*f*); *f*(*x*) = *f*(*y*) |] ==> *x=y*
 <proof>

lemma *inj-on-eq-iff*: *inj-on* *f* *A* ==> *x:A* ==> *y:A* ==> (*f*(*x*) = *f*(*y*)) = (*x=y*)
 <proof>

lemma *inj-eq*: *inj* *f* ==> (*f*(*x*) = *f*(*y*)) = (*x=y*)
 <proof>

lemma *inj-on-id[simp]*: *inj-on* *id* *A*
 <proof>

lemma *inj-on-id2[simp]*: *inj-on* (%*x*. *x*) *A*
 <proof>

lemma *surj-id[simp]*: *surj id*
 $\langle proof \rangle$

lemma *bij-id[simp]*: *bij id*
 $\langle proof \rangle$

lemma *inj-onI*:
 $(!! x y. [\mid x:A; y:A; f(x) = f(y) \mid] \implies x=y) \implies \text{inj-on } f \ A$
 $\langle proof \rangle$

lemma *inj-on-inverseI*: $(!!x. x:A \implies g(f(x)) = x) \implies \text{inj-on } f \ A$
 $\langle proof \rangle$

lemma *inj-onD*: $[\mid \text{inj-on } f \ A; f(x)=f(y); x:A; y:A \mid] \implies x=y$
 $\langle proof \rangle$

lemma *inj-on-iff*: $[\mid \text{inj-on } f \ A; x:A; y:A \mid] \implies (f(x)=f(y)) = (x=y)$
 $\langle proof \rangle$

lemma *comp-inj-on*:
 $[\mid \text{inj-on } f \ A; \text{inj-on } g \ (f'A) \mid] \implies \text{inj-on } (g \circ f) \ A$
 $\langle proof \rangle$

lemma *inj-on-imageI*: $\text{inj-on } (g \circ f) \ A \implies \text{inj-on } g \ (f' A)$
 $\langle proof \rangle$

lemma *inj-on-image-iff*: $[\mid \text{ALL } x:A. \text{ALL } y:A. (g(f x) = g(f y)) = (g x = g y);$
 $\text{inj-on } f \ A \mid] \implies \text{inj-on } g \ (f' A) = \text{inj-on } g \ A$
 $\langle proof \rangle$

lemma *inj-on-contradD*: $[\mid \text{inj-on } f \ A; \sim x=y; x:A; y:A \mid] \implies \sim f(x)=f(y)$
 $\langle proof \rangle$

lemma *inj-singleton*: $\text{inj } (\%s. \{s\})$
 $\langle proof \rangle$

lemma *inj-on-empty[iff]*: $\text{inj-on } f \ \{\}$
 $\langle proof \rangle$

lemma *subset-inj-on*: $[\mid \text{inj-on } f \ B; A \leq B \mid] \implies \text{inj-on } f \ A$
 $\langle proof \rangle$

lemma *inj-on-Un*:
 $\text{inj-on } f \ (A \cup B) =$
 $(\text{inj-on } f \ A \ \& \ \text{inj-on } f \ B \ \& \ f'(A-B) \ \text{Int } f'(B-A) = \{\})$
 $\langle proof \rangle$

lemma *inj-on-insert[iff]*:

inj-on f (insert a A) = (inj-on f A & f a ~: f'(A-{a}))
 ⟨proof⟩

lemma *inj-on-diff*: *inj-on f A ==> inj-on f (A-B)*
 ⟨proof⟩

lemma *surjI*: *(!! x. g(f x) = x) ==> surj g*
 ⟨proof⟩

lemma *surj-range*: *surj f ==> range f = UNIV*
 ⟨proof⟩

lemma *surjD*: *surj f ==> EX x. y = f x*
 ⟨proof⟩

lemma *surjE*: *surj f ==> (!!x. y = f x ==> C) ==> C*
 ⟨proof⟩

lemma *comp-surj*: *[| surj f; surj g |] ==> surj (g o f)*
 ⟨proof⟩

lemma *bijI*: *[| inj f; surj f |] ==> bij f*
 ⟨proof⟩

lemma *bij-is-inj*: *bij f ==> inj f*
 ⟨proof⟩

lemma *bij-is-surj*: *bij f ==> surj f*
 ⟨proof⟩

lemma *bij-betw-imp-inj-on*: *bij-betw f A B ==> inj-on f A*
 ⟨proof⟩

lemma *bij-comp*: *bij f ==> bij g ==> bij (g o f)*
 ⟨proof⟩

lemma *bij-betw-trans*:
bij-betw f A B ==> bij-betw g B C ==> bij-betw (g o f) A C
 ⟨proof⟩

lemma *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g. bij-betw g B A*
 ⟨proof⟩

lemma *surj-image-vimage-eq*: *surj f ==> f ' (f -' A) = A*
 ⟨proof⟩

lemma *inj-vimage-image-eq*: *inj f ==> f -' (f ' A) = A*
 ⟨proof⟩

lemma *vimage-subsetD*: $\text{surj } f \implies f \text{ --' } B \leq A \implies B \leq f \text{ ' } A$
 $\langle \text{proof} \rangle$

lemma *vimage-subsetI*: $\text{inj } f \implies B \leq f \text{ ' } A \implies f \text{ --' } B \leq A$
 $\langle \text{proof} \rangle$

lemma *vimage-subset-eq*: $\text{bij } f \implies (f \text{ --' } B \leq A) = (B \leq f \text{ ' } A)$
 $\langle \text{proof} \rangle$

lemma *inj-on-Un-image-eq-iff*: $\text{inj-on } f \ (A \cup B) \implies f \text{ ' } A = f \text{ ' } B \longleftrightarrow A = B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-Int*:
 $\llbracket \text{inj-on } f \ C; \ A \leq C; \ B \leq C \rrbracket \implies f \text{ ' } (A \text{ Int } B) = f \text{ ' } A \text{ Int } f \text{ ' } B$
 $\langle \text{proof} \rangle$

lemma *inj-on-image-set-diff*:
 $\llbracket \text{inj-on } f \ C; \ A \leq C; \ B \leq C \rrbracket \implies f \text{ ' } (A - B) = f \text{ ' } A - f \text{ ' } B$
 $\langle \text{proof} \rangle$

lemma *image-Int*: $\text{inj } f \implies f \text{ ' } (A \text{ Int } B) = f \text{ ' } A \text{ Int } f \text{ ' } B$
 $\langle \text{proof} \rangle$

lemma *image-set-diff*: $\text{inj } f \implies f \text{ ' } (A - B) = f \text{ ' } A - f \text{ ' } B$
 $\langle \text{proof} \rangle$

lemma *inj-image-mem-iff*: $\text{inj } f \implies (f \ a : f \text{ ' } A) = (a : A)$
 $\langle \text{proof} \rangle$

lemma *inj-image-subset-iff*: $\text{inj } f \implies (f \text{ ' } A \leq f \text{ ' } B) = (A \leq B)$
 $\langle \text{proof} \rangle$

lemma *inj-image-eq-iff*: $\text{inj } f \implies (f \text{ ' } A = f \text{ ' } B) = (A = B)$
 $\langle \text{proof} \rangle$

lemma *image-INT*:
 $\llbracket \text{inj-on } f \ C; \ \text{ALL } x:A. \ B \ x \leq C; \ j:A \rrbracket$
 $\implies f \text{ ' } (\text{INTER } A \ B) = (\text{INT } x:A. \ f \text{ ' } B \ x)$
 $\langle \text{proof} \rangle$

lemma *bij-image-INT*: $\text{bij } f \implies f \text{ ' } (\text{INTER } A \ B) = (\text{INT } x:A. \ f \text{ ' } B \ x)$
 $\langle \text{proof} \rangle$

lemma *surj-Compl-image-subset*: $\text{surj } f \implies \neg(f \text{ ' } A) \leq f \text{ ' } (\neg A)$
 $\langle \text{proof} \rangle$

lemma *inj-image-Compl-subset*: $\text{inj } f \implies f \text{ ' } (\neg A) \leq \neg(f \text{ ' } A)$

<proof>

lemma *bij-image-Compl-eq*: $\text{bij } f \implies f'(-A) = -(f'A)$
<proof>

lemma (*in ordered-ab-group-add*) *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
<proof>

lemma (*in linorder*) *strict-mono-imp-inj-on*: *strict-mono f \implies inj-on f A*
<proof>

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 :: [$'a, 'a$] \Rightarrow *updbind* ((2- :=/ -))
 :: *updbind* \Rightarrow *updbinds* (-)
-updbinds:: [*updbind*, *updbinds*] \Rightarrow *updbinds* (-,/ -)
-Update :: [$'a$, *updbinds*] $\Rightarrow 'a$ (-/'((-)') [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*: $(f(x:=y) = f) = (f x = y)$
<proof>

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*]: $(f(x:=y))z = (\text{if } z=x \text{ then } y \text{ else } f z)$
<proof>

lemma *fun-upd-same*: $(f(x:=y)) x = y$
<proof>

lemma *fun-upd-other*: $z \sim x \implies (f(x:=y)) z = f z$

$\langle proof \rangle$

lemma *fun-upd-upd* [simp]: $f(x:=y, x:=z) = f(x:=z)$
 $\langle proof \rangle$

lemma *fun-upd-twist*: $a \sim c \implies (m(a:=b))(c:=d) = (m(c:=d))(a:=b)$
 $\langle proof \rangle$

lemma *inj-on-fun-updI*: $\llbracket inj-on\ f\ A;\ y \notin f^{\circ}A \rrbracket \implies inj-on\ (f(x:=y))\ A$
 $\langle proof \rangle$

lemma *fun-upd-image*:
 $f(x:=y)^{\circ} A = (if\ x \in A\ then\ insert\ y\ (f^{\circ} (A - \{x\}))\ else\ f^{\circ} A)$
 $\langle proof \rangle$

lemma *fun-upd-comp*: $f \circ (g(x := y)) = (f \circ g)(x := f\ y)$
 $\langle proof \rangle$

10.6 override-on

definition

override-on :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$

where

override-on $f\ g\ A = (\lambda a. if\ a \in A\ then\ g\ a\ else\ f\ a)$

lemma *override-on-emptyset*[simp]: *override-on* $f\ g\ \{\} = f$
 $\langle proof \rangle$

lemma *override-on-apply-notin*[simp]: $a \sim A \implies (override-on\ f\ g\ A)\ a = f\ a$
 $\langle proof \rangle$

lemma *override-on-apply-in*[simp]: $a : A \implies (override-on\ f\ g\ A)\ a = g\ a$
 $\langle proof \rangle$

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* $a\ a\ f = f$
 $\langle proof \rangle$

lemma *swap-commute*: *swap* $a\ b\ f = swap\ b\ a\ f$
 $\langle proof \rangle$

lemma *swap-nilpotent* [simp]: *swap* $a\ b\ (swap\ a\ b\ f) = f$
 $\langle proof \rangle$

lemma *swap-triple*:

assumes $a \neq c$ **and** $b \neq c$

shows $\text{swap } a \ b \ (\text{swap } b \ c \ (\text{swap } a \ b \ f)) = \text{swap } a \ c \ f$

$\langle \text{proof} \rangle$

lemma *comp-swap*: $f \circ \text{swap } a \ b \ g = \text{swap } a \ b \ (f \circ g)$

$\langle \text{proof} \rangle$

lemma *inj-on-imp-inj-on-swap*:

$[[\text{inj-on } f \ A; \ a \in A; \ b \in A]] \implies \text{inj-on } (\text{swap } a \ b \ f) \ A$

$\langle \text{proof} \rangle$

lemma *inj-on-swap-iff* [simp]:

assumes $A: a \in A \ b \in A$ **shows** $\text{inj-on } (\text{swap } a \ b \ f) \ A = \text{inj-on } f \ A$

$\langle \text{proof} \rangle$

lemma *surj-imp-surj-swap*: $\text{surj } f \implies \text{surj } (\text{swap } a \ b \ f)$

$\langle \text{proof} \rangle$

lemma *surj-swap-iff* [simp]: $\text{surj } (\text{swap } a \ b \ f) = \text{surj } f$

$\langle \text{proof} \rangle$

lemma *bij-swap-iff*: $\text{bij } (\text{swap } a \ b \ f) = \text{bij } f$

$\langle \text{proof} \rangle$

hide-const (open) *swap*

10.8 Inversion of injective functions

definition *the-inv-into* :: $'a \text{ set} \implies ('a \implies 'b) \implies ('b \implies 'a)$ **where**
the-inv-into $A \ f == \%x. \text{THE } y. y : A \ \& \ f \ y = x$

lemma *the-inv-into-f-f*:

$[[\text{inj-on } f \ A; \ x : A]] \implies \text{the-inv-into } A \ f \ (f \ x) = x$

$\langle \text{proof} \rangle$

lemma *f-the-inv-into-f*:

$\text{inj-on } f \ A \implies y : f \ A \implies f \ (\text{the-inv-into } A \ f \ y) = y$

$\langle \text{proof} \rangle$

lemma *the-inv-into-into*:

$[[\text{inj-on } f \ A; \ x : f \ A; \ A \leq B]] \implies \text{the-inv-into } A \ f \ x : B$

$\langle \text{proof} \rangle$

lemma *the-inv-into-onto*[simp]:

$\text{inj-on } f \ A \implies \text{the-inv-into } A \ f \ (f \ A) = A$

$\langle \text{proof} \rangle$

lemma *the-inv-into-f-eq*:

$[| \text{inj-on } f \ A; f \ x = y; x : A \ |] \implies \text{the-inv-into } A \ f \ y = x$
 $\langle \text{proof} \rangle$

lemma *the-inv-into-comp*:

$[| \text{inj-on } f \ (g \ 'A); \text{inj-on } g \ A; x : f \ 'g \ 'A \ |] \implies$
 $\text{the-inv-into } A \ (f \ o \ g) \ x = (\text{the-inv-into } A \ g \ o \ \text{the-inv-into } (g \ 'A) \ f) \ x$
 $\langle \text{proof} \rangle$

lemma *inj-on-the-inv-into*:

$\text{inj-on } f \ A \implies \text{inj-on } (\text{the-inv-into } A \ f) \ (f \ 'A)$
 $\langle \text{proof} \rangle$

lemma *bij-betw-the-inv-into*:

$\text{bij-betw } f \ A \ B \implies \text{bij-betw } (\text{the-inv-into } A \ f) \ B \ A$
 $\langle \text{proof} \rangle$

abbreviation *the-inv* :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**

$\text{the-inv } f \equiv \text{the-inv-into } \text{UNIV } f$

lemma *the-inv-f-f*:

assumes *inj f*

shows $\text{the-inv } f \ (f \ x) = x$ $\langle \text{proof} \rangle$

10.9 Proof tool setup

simplifies terms of the form $f(\dots, x := y, \dots, x := z, \dots)$ to $f(\dots, x := z, \dots)$

$\langle ML \rangle$

10.10 Code generator setup

types-code

fun $((- \rightarrow / -))$

attach (*term-of*) $\langle\langle$

fun term-of-fun-type - aT - bT - = *Free* ($\langle \text{function} \rangle$, $aT \rightarrow bT$);

$\rangle\rangle$

attach (*test*) $\langle\langle$

fun gen-fun-type aF aT bG bT i =

let

$\text{val } \text{tab} = \text{Unsynchronized.ref } [];$

fun *mk-upd* $(x, (-, y)) \ t = \text{Const } (\text{Fun.fun-upd},$

$(aT \rightarrow bT) \rightarrow aT \rightarrow bT \rightarrow aT \rightarrow bT) \ \$ \ t \ \$ \ aF \ x \ \$ \ y \ ()$

in

$(\text{fn } x \Rightarrow$

$\text{case } \text{AList.lookup } \text{op} = (!\text{tab}) \ x \ \text{of}$

$\text{NONE} \Rightarrow$

$\text{let } \text{val } p \text{ as } (y, -) = bG \ i$

$\text{in } (\text{tab} := (x, p) :: !\text{tab}; y) \ \text{end}$

$| \text{SOME } (y, -) \Rightarrow y,$

$\text{fn } () \Rightarrow \text{Basics.fold mk-upd } (!\text{tab}) \ (\text{Const } (\text{HOL.undefined}, aT \rightarrow bT)))$

```

    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 <proof>

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  $\longleftrightarrow \text{True}$ 
    <proof>

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>

definition

```

```

  Unity :: unit    ('())
where
  () = Abs-unit True

```

```

lemma unit-eq [no-atp]: u = ()
  ⟨proof⟩

```

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

⟨ML⟩

```

rep-datatype () ⟨proof⟩

```

```

lemma unit-all-eq1: (!x::unit. PROP P x) == PROP P ()
  ⟨proof⟩

```

```

lemma unit-all-eq2: (!x::unit. PROP P) == PROP P
  ⟨proof⟩

```

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
  ⟨proof⟩

```

```

instantiation unit :: default
begin

```

```

definition default = ()

```

```

instance ⟨proof⟩

```

```

end

```

```

lemma [code]:
  eq-class.eq (u::unit) v ⟷ True ⟨proof⟩

```

```

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*. $\exists a b. f = \text{Pair-Rep } (a::'a) (b::'b)$ }
 $\langle \text{proof} \rangle$

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: *Pair* *a b* == *Abs-Prod* (*Pair-Rep* *a b*)

rep-datatype (*prod*) *Pair* $\langle \text{proof} \rangle$

11.3.2 Tuple syntax

global consts
split :: ('*a* \Rightarrow '*b* \Rightarrow '*c*) \Rightarrow '*a* \times '*b* \Rightarrow '*c*

local defs
split-prod-case: *split* == *prod-case*

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminals

tuple-args patterns

syntax

```
-tuple      :: 'a => tuple-args => 'a * 'b      ((1 '(-, -'))
-tuple-arg  :: 'a => tuple-args                  (-)
-tuple-args :: 'a => tuple-args => tuple-args    (-, / -)
-pattern    :: [pttrn, patterns] => pttrn       (('(-, -'))
              :: pttrn => patterns               (-)
-patterns   :: [pttrn, patterns] => patterns    (-, / -)
```

translations

```
(x, y) == CONST Pair x y
-tuple x (-tuple-args y z) == -tuple x (-tuple-arg (-tuple y z))
%(x, y, zs). b == CONST split (%x (y, zs). b)
%(x, y). b == CONST split (%x y. b)
-abs (CONST Pair x y) t => %(x, y). t
— The last rule accommodates tuples in ‘case C ... (x,y) ... =i ...’ The (x,y) is
parsed as ‘Pair x y’ because it is logic, not pttrn
```

$\langle ML \rangle$

11.3.3 Code generator setup

lemma *split-case-cert*:

```
assumes CASE  $\equiv$  split f
shows CASE (a, b)  $\equiv$  f a b
<proof>
```

$\langle ML \rangle$

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 ((-, / -))

⟨*ML*⟩

11.3.4 Fundamental operations and properties

lemma *surj-pair* [*simp*]: $EX\ x\ y. p = (x, y)$
 ⟨*proof*⟩

global consts

fst :: $'a \times 'b \Rightarrow 'a$
snd :: $'a \times 'b \Rightarrow 'b$

local defs

fst-def: *fst p == case p of (a, b) => a*
snd-def: *snd p == case p of (a, b) => b*

lemma *fst-conv* [*simp*, *code*]: $fst\ (a, b) = a$
 ⟨*proof*⟩

lemma *snd-conv* [*simp*, *code*]: $snd\ (a, b) = b$
 ⟨*proof*⟩

code-const *fst* **and** *snd*

(*Haskell* *fst* **and** *snd*)

lemma *prod-case-unfold*: $prod-case = (\%c\ p. c\ (fst\ p)\ (snd\ p))$
 ⟨*proof*⟩

lemma *fst-eqD*: $fst\ (x, y) = a ==> x = a$
 ⟨*proof*⟩

lemma *snd-eqD*: $\text{snd } (x, y) = a \implies y = a$
 $\langle \text{proof} \rangle$

lemma *pair-collapse* [*simp*]: $(\text{fst } p, \text{snd } p) = p$
 $\langle \text{proof} \rangle$

lemmas *surjective-pairing* = *pair-collapse* [*symmetric*]

lemma *Pair-fst-snd-eq*: $s = t \iff \text{fst } s = \text{fst } t \wedge \text{snd } s = \text{snd } t$
 $\langle \text{proof} \rangle$

lemma *prod-eqI* [*intro?*]: $\text{fst } p = \text{fst } q \implies \text{snd } p = \text{snd } q \implies p = q$
 $\langle \text{proof} \rangle$

lemma *split-conv* [*simp*, *code*]: $\text{split } f \ (a, b) = f \ a \ b$
 $\langle \text{proof} \rangle$

lemma *splitI*: $f \ a \ b \implies \text{split } f \ (a, b)$
 $\langle \text{proof} \rangle$

lemma *splitD*: $\text{split } f \ (a, b) \implies f \ a \ b$
 $\langle \text{proof} \rangle$

lemma *split-Pair* [*simp*]: $(\lambda(x, y). (x, y)) = \text{id}$
 $\langle \text{proof} \rangle$

lemma *split-eta*: $(\lambda(x, y). f \ (x, y)) = f$
 — Subsumes the old *split-Pair* when *f* is the identity function.
 $\langle \text{proof} \rangle$

lemma *split-comp*: $\text{split } (f \circ g) \ x = f \ (g \ (\text{fst } x)) \ (\text{snd } x)$
 $\langle \text{proof} \rangle$

lemma *split-twice*: $\text{split } f \ (\text{split } g \ p) = \text{split } (\lambda x \ y. \text{split } f \ (g \ x \ y)) \ p$
 $\langle \text{proof} \rangle$

lemma *The-split*: $\text{The } (\text{split } P) = (\text{THE } xy. P \ (\text{fst } xy) \ (\text{snd } xy))$
 $\langle \text{proof} \rangle$

lemma *split-weak-cong*: $p = q \implies \text{split } c \ p = \text{split } c \ q$
 — Prevents simplification of *c*: much faster
 $\langle \text{proof} \rangle$

lemma *cond-split-eta*: $(!!x \ y. f \ x \ y = g \ (x, y)) \implies (\% (x, y). f \ x \ y) = g$
 $\langle \text{proof} \rangle$

lemma *split-paired-all*: $(!!x. \text{PROP } P \ x) == (!!a \ b. \text{PROP } P \ (a, b))$
 $\langle \text{proof} \rangle$

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*

$\langle ML \rangle$

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

$\langle proof \rangle$

lemma *split-paired-Ex* [*simp*]: $(EX\ x.\ P\ x) = (EX\ a\ b.\ P\ (a, b))$

$\langle proof \rangle$

lemma *split-paired-The*: $(THE\ x.\ P\ x) = (THE\ (a, b).\ P\ (a, b))$

— Can’t be added to simpset: loops!

$\langle proof \rangle$

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*.

$\langle ML \rangle$

lemma *split-beta* [*mono*]: $(\%(x, y).\ P\ x\ y)\ z = P\ (fst\ z)\ (snd\ z)$

$\langle proof \rangle$

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.

$\langle proof \rangle$

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*]: $R\ (split\ c\ p) = (\sim(EX\ x\ y.\ p = (x, y) \ \&\ (\sim R\ (c\ x\ y))))$

$\langle proof \rangle$

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*: $!!p.\ [\![\![a\ b.\ p = (a, b) \implies c\ a\ b]\!]\implies split\ c\ p$

$\langle proof \rangle$

lemma *splitI2'*: $!!p.\ [\![\![a\ b.\ (a, b) = p \implies c\ a\ b\ x]\!]\implies split\ c\ p\ x$

$\langle proof \rangle$

lemma *splitE*: $split\ c\ p \implies (!x\ y.\ p = (x, y) \implies c\ x\ y \implies Q) \implies Q$

$\langle \text{proof} \rangle$

lemma *splitE'*: $\text{split } c \ p \ z \implies (!x \ y. \ p = (x, y) \implies c \ x \ y \ z \implies Q) \implies Q$
 $\langle \text{proof} \rangle$

lemma *splitE2*:
 $[! \ Q \ (\text{split } P \ z); \ !x \ y. \ [z = (x, y); \ Q \ (P \ x \ y)]] \implies R \] \implies R$
 $\langle \text{proof} \rangle$

lemma *splitD'*: $\text{split } R \ (a, b) \ c \implies R \ a \ b \ c$
 $\langle \text{proof} \rangle$

lemma *mem-splitI*: $z: c \ a \ b \implies z: \text{split } c \ (a, b)$
 $\langle \text{proof} \rangle$

lemma *mem-splitI2*: $!p. \ [!a \ b. \ p = (a, b) \implies z: c \ a \ b \] \implies z: \text{split } c \ p$
 $\langle \text{proof} \rangle$

lemma *mem-splitE*:
assumes *major*: $z \in \text{split } c \ p$
and cases: $\bigwedge x \ y. \ p = (x, y) \implies z \in c \ x \ y \implies Q$
shows Q
 $\langle \text{proof} \rangle$

declare *mem-splitI2* [*intro!*] *mem-splitI* [*intro!*] *splitI2'* [*intro!*] *splitI2* [*intro!*] *splitI* [*intro!*]
declare *mem-splitE* [*elim!*] *splitE'* [*elim!*] *splitE* [*elim!*]

$\langle \text{ML} \rangle$

lemma *split-eta-SetCompr* [*simp, no-atp*]: $(\%u. \ EX \ x \ y. \ u = (x, y) \ \& \ P \ (x, y)) = P$
 $\langle \text{proof} \rangle$

lemma *split-eta-SetCompr2* [*simp, no-atp*]: $(\%u. \ EX \ x \ y. \ u = (x, y) \ \& \ P \ x \ y) = \text{split } P$
 $\langle \text{proof} \rangle$

lemma *split-part* [*simp*]: $(\%(a, b). \ P \ \& \ Q \ a \ b) = (\%ab. \ P \ \& \ \text{split } Q \ ab)$
— Allows simplifications of nested splits in case of independent predicates.
 $\langle \text{proof} \rangle$

lemma *split-comp-eq*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$ **and** $g :: 'd \Rightarrow 'a$
shows $(\%u. \ f \ (g \ (\text{fst } u)) \ (\text{snd } u)) = (\text{split } (\%x. \ f \ (g \ x)))$
 $\langle \text{proof} \rangle$

lemma *pair-imageI* [*intro*]: $(a, b) : A \implies f \ a \ b : (\%(a, b). \ f \ a \ b) \ ` \ A$

$\langle \text{proof} \rangle$

lemma *The-split-eq* [simp]: (*THE* (x', y'). $x = x' \ \& \ y = y'$) = (x, y)
 $\langle \text{proof} \rangle$

Setup of internal *split-rule*.

lemmas *prod-caseI* = *prod.cases* [*THEN iffD2*, *standard*]

lemma *prod-caseI2*: $!!p. [\![\![a \ b. \ p = (a, b) \implies c \ a \ b \]\!] \implies \text{prod-case } c \ p$
 $\langle \text{proof} \rangle$

lemma *prod-caseI2'*: $!!p. [\![\![a \ b. \ (a, b) = p \implies c \ a \ b \ x \]\!] \implies \text{prod-case } c \ p \ x$
 $\langle \text{proof} \rangle$

lemma *prod-caseE*: $\text{prod-case } c \ p \implies (!x \ y. \ p = (x, y) \implies c \ x \ y \implies Q)$
 $\implies Q$
 $\langle \text{proof} \rangle$

lemma *prod-caseE'*: $\text{prod-case } c \ p \ z \implies (!x \ y. \ p = (x, y) \implies c \ x \ y \ z \implies Q)$
 $\implies Q$
 $\langle \text{proof} \rangle$

declare *prod-caseI2'* [intro!] *prod-caseI2* [intro!] *prod-caseI* [intro!]

declare *prod-caseE'* [elim!] *prod-caseE* [elim!]

lemma *prod-case-split*:
 $\text{prod-case} = \text{split}$
 $\langle \text{proof} \rangle$

lemma *prod-case-beta*:
 $\text{prod-case } f \ p = f \ (\text{fst } p) \ (\text{snd } p)$
 $\langle \text{proof} \rangle$

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) $a \ b \ c$ **where** $y = (a, b, c)$
 $\langle \text{proof} \rangle$

lemma *prod-induct3* [*case-names fields, induct type*]:
 $(!!a \ b \ c. \ P \ (a, b, c)) \implies P \ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) $a \ b \ c \ d$ **where** $y = (a, b, c, d)$
 $\langle \text{proof} \rangle$

lemma *prod-induct4* [*case-names fields, induct type*]:
 $(!!a \ b \ c \ d. \ P \ (a, b, c, d)) \implies P \ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases5* [*cases type*]:

obtains (*fields*) *a b c d e* **where** $y = (a, b, c, d, e)$
 $\langle \text{proof} \rangle$

lemma *prod-induct5* [*case-names fields, induct type*]:

$(!!a\ b\ c\ d\ e. P\ (a, b, c, d, e)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases6* [*cases type*]:

obtains (*fields*) *a b c d e f* **where** $y = (a, b, c, d, e, f)$
 $\langle \text{proof} \rangle$

lemma *prod-induct6* [*case-names fields, induct type*]:

$(!!a\ b\ c\ d\ e\ f. P\ (a, b, c, d, e, f)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *prod-cases7* [*cases type*]:

obtains (*fields*) *a b c d e f g* **where** $y = (a, b, c, d, e, f, g)$
 $\langle \text{proof} \rangle$

lemma *prod-induct7* [*case-names fields, induct type*]:

$(!!a\ b\ c\ d\ e\ f\ g. P\ (a, b, c, d, e, f, g)) \implies P\ x$
 $\langle \text{proof} \rangle$

lemma *split-def*:

$\text{split} = (\lambda c\ p. c\ (\text{fst}\ p)\ (\text{snd}\ p))$
 $\langle \text{proof} \rangle$

definition *internal-split* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$ **where**
 $\text{internal-split} == \text{split}$

lemma *internal-split-conv*: $\text{internal-split}\ c\ (a, b) = c\ a\ b$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

hide-const *internal-split*

11.3.5 Derived operations

global consts

curry :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local defs

curry-def: $\text{curry} == (\%c\ x\ y. c\ (\text{Pair}\ x\ y))$

lemma *curry-conv* [*simp, code*]: $\text{curry}\ f\ a\ b = f\ (a, b)$
 $\langle \text{proof} \rangle$

lemma *curryI* [*intro!*]: $f \ (a, b) \Longrightarrow \text{curry } f \ a \ b$
 $\langle \text{proof} \rangle$

lemma *curryD* [*dest!*]: $\text{curry } f \ a \ b \Longrightarrow f \ (a, b)$
 $\langle \text{proof} \rangle$

lemma *curryE*: $\text{curry } f \ a \ b \Longrightarrow (f \ (a, b) \Longrightarrow Q) \Longrightarrow Q$
 $\langle \text{proof} \rangle$

lemma *curry-split* [*simp*]: $\text{curry } (\text{split } f) = f$
 $\langle \text{proof} \rangle$

lemma *split-curry* [*simp*]: $\text{split } (\text{curry } f) = f$
 $\langle \text{proof} \rangle$

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→* 60)
where
 $f \ o \rightarrow g = (\lambda x. \text{split } g \ (f \ x))$

lemma *scomp-apply*: $(f \ o \rightarrow g) \ x = \text{split } g \ (f \ x)$
 $\langle \text{proof} \rangle$

lemma *Pair-scomp*: $\text{Pair } x \ o \rightarrow f = f \ x$
 $\langle \text{proof} \rangle$

lemma *scomp-Pair*: $x \ o \rightarrow \text{Pair} = x$
 $\langle \text{proof} \rangle$

lemma *scomp-scomp*: $(f \ o \rightarrow g) \ o \rightarrow h = f \ o \rightarrow (\lambda x. g \ x \ o \rightarrow h)$
 $\langle \text{proof} \rangle$

lemma *scomp-fcomp*: $(f \ o \rightarrow g) \ o > h = f \ o \rightarrow (\lambda x. g \ x \ o > h)$
 $\langle \text{proof} \rangle$

lemma *fcomp-scomp*: $(f \ o > g) \ o \rightarrow h = f \ o > (g \ o \rightarrow h)$
 $\langle \text{proof} \rangle$

code-const *scomp*
 $(\text{Eval } \text{infixl } 3 \ \#-\rightarrow)$

no-notation *fcomp* (**infixl** *o>* 60)

no-notation *scomp* (**infixl** *o→* 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**
 $[\text{code del}]: \text{prod-fun } f \ g = (\lambda (x, y). (f \ x, g \ y))$

lemma *prod-fun [simp, code]: prod-fun f g (a, b) = (f a, g b)*
<proof>

lemma *fst-prod-fun[simp]: fst (prod-fun f g x) = f (fst x)*
<proof>

lemma *snd-prod-fun[simp]: snd (prod-fun f g x) = g (snd x)*
<proof>

lemma *fst-comp-prod-fun[simp]: fst ∘ prod-fun f g = f ∘ fst*
<proof>

lemma *snd-comp-prod-fun[simp]: snd ∘ prod-fun f g = g ∘ snd*
<proof>

lemma *prod-fun-compose:*
prod-fun (f1 o f2) (g1 o g2) = (prod-fun f1 g1 o prod-fun f2 g2)
<proof>

lemma *prod-fun-ident [simp]: prod-fun (%x. x) (%y. y) = (%z. z)*
<proof>

lemma *prod-fun-imageI [intro]: (a, b) : r ==> (f a, g b) : prod-fun f g ‘ r*
<proof>

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
<proof>

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)
<proof>

lemma *apsnd-conv [simp, code]:*
apsnd f (x, y) = (x, f y)
<proof>

lemma *fst-apfst [simp]:*

$$\text{fst } (\text{apfst } f \ x) = f \ (\text{fst } x)$$

<proof>

lemma *fst-apsnd* [simp]:

$$\text{fst } (\text{apsnd } f \ x) = \text{fst } x$$

<proof>

lemma *snd-apfst* [simp]:

$$\text{snd } (\text{apfst } f \ x) = \text{snd } x$$

<proof>

lemma *snd-apsnd* [simp]:

$$\text{snd } (\text{apsnd } f \ x) = f \ (\text{snd } x)$$

<proof>

lemma *apfst-compose*:

$$\text{apfst } f \ (\text{apfst } g \ x) = \text{apfst } (f \circ g) \ x$$

<proof>

lemma *apsnd-compose*:

$$\text{apsnd } f \ (\text{apsnd } g \ x) = \text{apsnd } (f \circ g) \ x$$

<proof>

lemma *apfst-apsnd* [simp]:

$$\text{apfst } f \ (\text{apsnd } g \ x) = (f \ (\text{fst } x), g \ (\text{snd } x))$$

<proof>

lemma *apsnd-apfst* [simp]:

$$\text{apsnd } f \ (\text{apfst } g \ x) = (g \ (\text{fst } x), f \ (\text{snd } x))$$

<proof>

lemma *apfst-id* [simp] :

$$\text{apfst } \text{id} = \text{id}$$

<proof>

lemma *apsnd-id* [simp] :

$$\text{apsnd } \text{id} = \text{id}$$

<proof>

lemma *apfst-eq-conv* [simp]:

$$\text{apfst } f \ x = \text{apfst } g \ x \longleftrightarrow f \ (\text{fst } x) = g \ (\text{fst } x)$$

<proof>

lemma *apsnd-eq-conv* [simp]:

$$\text{apsnd } f \ x = \text{apsnd } g \ x \longleftrightarrow f \ (\text{snd } x) = g \ (\text{snd } x)$$

<proof>

lemma *apsnd-apfst-commute*:

$$\text{apsnd } f \ (\text{apfst } g \ p) = \text{apfst } g \ (\text{apsnd } f \ p)$$

$\langle proof \rangle$

Disjoint union of a family of sets – Sigma.

definition $Sigma :: ['a\ set, 'a \Rightarrow 'b\ set] \Rightarrow ('a \times 'b)\ set$ **where**
Sigma-def: $Sigma\ A\ B == UN\ x:A.\ UN\ y:B\ x.\ \{Pair\ x\ y\}$

abbreviation

$Times :: ['a\ set, 'b\ set] \Rightarrow ('a * 'b)\ set$
(infixr <*> 80) where
 $A\ <*>\ B == Sigma\ A\ (\%-. B)$

notation (*xsymbols*)

$Times$ **(infixr \times 80)**

notation (*HTML output*)

$Times$ **(infixr \times 80)**

syntax

$-Sigma :: [pttrn, 'a\ set, 'b\ set] \Rightarrow ('a * 'b)\ set\ ((3SIGMA\ :-./\ -)\ [0, 0, 10]\ 10)$

translations

$SIGMA\ x:A.\ B == CONST\ Sigma\ A\ (\%x.\ B)$

lemma $SigmaI\ [intro!]: [\ a:A;\ b:B(a)\] \Rightarrow (a,b) : Sigma\ A\ B$
 $\langle proof \rangle$

lemma $SigmaE\ [elim!]:$

$[\ c: Sigma\ A\ B;$
 $!!x\ y. [\ x:A;\ y:B(x);\ c=(x,y)\] \Rightarrow P$
 $]\Rightarrow P$

— The general elimination rule.

$\langle proof \rangle$

Elimination of $(a, b) \in A \times B$ – introduces no eigenvariables.

lemma $SigmaD1: (a, b) : Sigma\ A\ B \Rightarrow a : A$
 $\langle proof \rangle$

lemma $SigmaD2: (a, b) : Sigma\ A\ B \Rightarrow b : B\ a$
 $\langle proof \rangle$

lemma $SigmaE2:$

$[\ (a, b) : Sigma\ A\ B;$
 $[\ a:A;\ b:B(a)\] \Rightarrow P$
 $]\Rightarrow P$

$\langle proof \rangle$

lemma $Sigma-cong:$

$\llbracket A = B; !!x.\ x \in B \Rightarrow C\ x = D\ x \rrbracket$
 $\Rightarrow (SIGMA\ x: A.\ C\ x) = (SIGMA\ x: B.\ D\ x)$

$\langle proof \rangle$

lemma *Sigma-mono*: $[| A \leq C; !!x. x:A ==> B x \leq D x |] ==> \text{Sigma } A B \leq \text{Sigma } C D$
 $\langle \text{proof} \rangle$

lemma *Sigma-empty1* $[simp]$: $\text{Sigma } \{\} B = \{\}$
 $\langle \text{proof} \rangle$

lemma *Sigma-empty2* $[simp]$: $A <*> \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *UNIV-Times-UNIV* $[simp]$: $UNIV <*> UNIV = UNIV$
 $\langle \text{proof} \rangle$

lemma *Compl-Times-UNIV1* $[simp]$: $\neg (UNIV <*> A) = UNIV <*> (\neg A)$
 $\langle \text{proof} \rangle$

lemma *Compl-Times-UNIV2* $[simp]$: $\neg (A <*> UNIV) = (\neg A) <*> UNIV$
 $\langle \text{proof} \rangle$

lemma *mem-Sigma-iff* $[iff]$: $((a,b): \text{Sigma } A B) = (a:A \ \& \ b:B(a))$
 $\langle \text{proof} \rangle$

lemma *Times-subset-cancel2*: $x:C ==> (A <*> C \leq B <*> C) = (A \leq B)$
 $\langle \text{proof} \rangle$

lemma *Times-eq-cancel2*: $x:C ==> (A <*> C = B <*> C) = (A = B)$
 $\langle \text{proof} \rangle$

lemma *SetCompr-Sigma-eq*:
 $\text{Collect } (\text{split } (\%x y. P x \ \& \ Q x y)) = (\text{SIGMA } x:\text{Collect } P. \text{Collect } (Q x))$
 $\langle \text{proof} \rangle$

lemma *Collect-split* $[simp]$: $\{(a,b). P a \ \& \ Q b\} = \text{Collect } P <*> \text{Collect } Q$
 $\langle \text{proof} \rangle$

lemma *UN-Times-distrib*:
 $(UN (a,b):(A <*> B). E a <*> F b) = (UNION A E) <*> (UNION B F)$
 — Suggested by Pierre Chartier
 $\langle \text{proof} \rangle$

lemma *split-paired-Ball-Sigma* $[simp, no-atp]$:
 $(ALL z: \text{Sigma } A B. P z) = (ALL x:A. ALL y: B x. P(x,y))$
 $\langle \text{proof} \rangle$

lemma *split-paired-Bex-Sigma* $[simp, no-atp]$:
 $(EX z: \text{Sigma } A B. P z) = (EX x:A. EX y: B x. P(x,y))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Un-distrib1*: $(\text{SIGMA } i:I \text{ Un } J. C(i)) = (\text{SIGMA } i:I. C(i)) \text{ Un } (\text{SIGMA } j:J. C(j))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Un-distrib2*: $(\text{SIGMA } i:I. A(i) \text{ Un } B(i)) = (\text{SIGMA } i:I. A(i)) \text{ Un } (\text{SIGMA } i:I. B(i))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Int-distrib1*: $(\text{SIGMA } i:I \text{ Int } J. C(i)) = (\text{SIGMA } i:I. C(i)) \text{ Int } (\text{SIGMA } j:J. C(j))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Int-distrib2*: $(\text{SIGMA } i:I. A(i) \text{ Int } B(i)) = (\text{SIGMA } i:I. A(i)) \text{ Int } (\text{SIGMA } i:I. B(i))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Diff-distrib1*: $(\text{SIGMA } i:I - J. C(i)) = (\text{SIGMA } i:I. C(i)) - (\text{SIGMA } j:J. C(j))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Diff-distrib2*: $(\text{SIGMA } i:I. A(i) - B(i)) = (\text{SIGMA } i:I. A(i)) - (\text{SIGMA } i:I. B(i))$
 $\langle \text{proof} \rangle$

lemma *Sigma-Union*: $\text{Sigma } (\text{Union } X) B = (\text{UN } A:X. \text{Sigma } A B)$
 $\langle \text{proof} \rangle$

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A \text{ Un } B) <*> C = (A <*> C) \text{ Un } (B <*> C)$
 $\langle \text{proof} \rangle$

lemma *Times-Int-distrib1*: $(A \text{ Int } B) <*> C = (A <*> C) \text{ Int } (B <*> C)$
 $\langle \text{proof} \rangle$

lemma *Times-Diff-distrib1*: $(A - B) <*> C = (A <*> C) - (B <*> C)$
 $\langle \text{proof} \rangle$

lemma *Times-empty[simp]*: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
 $\langle \text{proof} \rangle$

lemma *fst-image-times[simp]*: $\text{fst } ' (A \times B) = (\text{if } B = \{\} \text{ then } \{\} \text{ else } A)$
 $\langle \text{proof} \rangle$

lemma *snd-image-times[simp]*: $\text{snd } ' (A \times B) = (\text{if } A = \{\} \text{ then } \{\} \text{ else } B)$
 $\langle \text{proof} \rangle$

lemma *insert-times-insert[simp]*:
 $\text{insert } a A \times \text{insert } b B =$

insert (*a,b*) ($A \times \text{insert } b \ B \cup \text{insert } a \ A \times B$)
 ⟨*proof*⟩

lemma *image-Times*: $f -' (A \times B) = ((fst \circ f) -' A) \cap ((snd \circ f) -' B)$
 ⟨*proof*⟩

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* (*prod-fun f g*) ($A \times B$)
 ⟨*proof*⟩

lemma *prod-fun-surj*:
assumes *surj* *f* **and** *surj* *g*
shows *surj* (*prod-fun f g*)
 ⟨*proof*⟩

lemma *prod-fun-surj-on*:
assumes $f -' A = A'$ **and** $g -' B = B'$
shows *prod-fun f g* -' ($A \times B$) = $A' \times B'$
 ⟨*proof*⟩

lemma *swap-inj-on*:
inj-on ($\lambda(i, j). (j, i)$) *A*
 ⟨*proof*⟩

lemma *swap-product*:
 $(\%(i, j). (j, i)) -' (A \times B) = B \times A$
 ⟨*proof*⟩

lemma *image-split-eq-Sigma*:
 $(\lambda x. (f \ x, g \ x)) -' A = \text{Sigma } (f -' A) (\lambda x. g -' (f -' \{x\} \cap A))$
 ⟨*proof*⟩

11.4 Inductively defined sets

⟨*ML*⟩

11.5 Legacy theorem bindings and duplicates

lemma *PairE*:
obtains *x y* **where** $p = (x, y)$
 ⟨*proof*⟩

lemma *Pair-inject*:
assumes $(a, b) = (a', b')$
and $a = a' ==> b = b' ==> R$
shows *R*
 ⟨*proof*⟩

```

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  $\Rightarrow$  bool  $\Rightarrow$  bool where
  Inl-Rep a x y p  $\longleftrightarrow$  x = a  $\wedge$  p

```

```

definition Inr-Rep :: 'b  $\Rightarrow$  'a  $\Rightarrow$  'b  $\Rightarrow$  bool  $\Rightarrow$  bool where
  Inr-Rep b x y p  $\longleftrightarrow$  y = b  $\wedge$   $\neg$  p

```

global

```

typedef (Sum) ('a, 'b) + (infixr + 10) = {f. ( $\exists$  a. f = Inl-Rep (a::'a))  $\vee$  ( $\exists$  b. f
= Inr-Rep (b::'b))}
  <proof>

```

local

```

lemma Inl-RepI: Inl-Rep a  $\in$  Sum
  <proof>

```

```

lemma Inr-RepI: Inr-Rep b  $\in$  Sum
  <proof>

```

```

lemma inj-on-Abs-Sum: A  $\subseteq$  Sum  $\implies$  inj-on Abs-Sum A
  <proof>

```

```

lemma Inl-Rep-inject: inj-on Inl-Rep A
  <proof>

```

```

lemma Inr-Rep-inject: inj-on Inr-Rep A
  <proof>

```

```

lemma Inl-Rep-not-Inr-Rep: Inl-Rep a  $\neq$  Inr-Rep b
  <proof>

```

definition $Inl :: 'a \Rightarrow 'a + 'b$ **where**
 $Inl = Abs-Sum \circ Inl-Rep$

definition $Inr :: 'b \Rightarrow 'a + 'b$ **where**
 $Inr = Abs-Sum \circ Inr-Rep$

lemma $inj-Inl$ [simp]: $inj-on\ Inl\ A$
 $\langle proof \rangle$

lemma $Inl-inject$: $Inl\ x = Inl\ y \implies x = y$
 $\langle proof \rangle$

lemma $inj-Inr$ [simp]: $inj-on\ Inr\ A$
 $\langle proof \rangle$

lemma $Inr-inject$: $Inr\ x = Inr\ y \implies x = y$
 $\langle proof \rangle$

lemma $Inl-not-Inr$: $Inl\ a \neq Inr\ b$
 $\langle proof \rangle$

lemma $Inr-not-Inl$: $Inr\ b \neq Inl\ a$
 $\langle proof \rangle$

lemma $sumE$:
assumes $\bigwedge x::'a. s = Inl\ x \implies P$
and $\bigwedge y::'b. s = Inr\ y \implies P$
shows P
 $\langle proof \rangle$

rep-datatype (sum) $Inl\ Inr$
 $\langle proof \rangle$

12.2 Projections

lemma $sum-case-KK$ [simp]: $sum-case\ (\lambda x. a)\ (\lambda x. a) = (\lambda x. a)$
 $\langle proof \rangle$

lemma $surjective-sum$: $sum-case\ (\lambda x::'a. f\ (Inl\ x))\ (\lambda y::'b. f\ (Inr\ y)) = f$
 $\langle proof \rangle$

lemma $sum-case-inject$:
assumes $a: sum-case\ f1\ f2 = sum-case\ g1\ g2$
assumes $r: f1 = g1 \implies f2 = g2 \implies P$
shows P
 $\langle proof \rangle$

lemma $sum-case-weak-cong$:
 $s = t \implies sum-case\ f\ g\ s = sum-case\ f\ g\ t$

— Prevents simplification of f and g : much faster.
 $\langle proof \rangle$

primrec *Projl* :: $'a + 'b \Rightarrow 'a$ **where**
Projl-Inl: $Projl (Inl\ x) = x$

primrec *Projr* :: $'a + 'b \Rightarrow 'b$ **where**
Projr-Inr: $Projr (Inr\ x) = x$

primrec *Suml* :: $('a \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$ **where**
Suml f ($Inl\ x$) = $f\ x$

primrec *Sumr* :: $('b \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$ **where**
Sumr f ($Inr\ x$) = $f\ x$

lemma *Suml-inject*:
assumes $Suml\ f = Suml\ g$ **shows** $f = g$
 $\langle proof \rangle$

lemma *Sumr-inject*:
assumes $Sumr\ f = Sumr\ g$ **shows** $f = g$
 $\langle proof \rangle$

12.3 The Disjoint Sum of Sets

definition *Plus* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a + 'b)\ set$ (**infixr** $<+>$ 65) **where**
 $A <+> B = Inl\ `A \cup Inr\ `B$

lemma *InlI* [*intro!*]: $a \in A \Longrightarrow Inl\ a \in A <+> B$
 $\langle proof \rangle$

lemma *InrI* [*intro!*]: $b \in B \Longrightarrow Inr\ b \in A <+> B$
 $\langle proof \rangle$

Exhaustion rule for sums, a degenerate form of induction

lemma *PlusE* [*elim!*]:
 $u \in A <+> B \Longrightarrow (\bigwedge x. x \in A \Longrightarrow u = Inl\ x \Longrightarrow P) \Longrightarrow (\bigwedge y. y \in B \Longrightarrow u = Inr\ y \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

lemma *Plus-eq-empty-conv* [*simp*]: $A <+> B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
 $\langle proof \rangle$

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
 $\langle proof \rangle$

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:
   $a * e + (b * e + c) = (a + b) * e + c$ 
  <proof>
```

```
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>
```

```
end
```

```
class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
  assumes distrib:  $(a + b) * c = a * c + b * c$ 
begin
```

```
subclass semiring
  <proof>
```

```
end
```

```
class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero
begin
```

```
subclass semiring-0 <proof>
```

```
end
```

```

class comm-semiring-0-cancel = comm-semiring + cancel-comm-monoid-add
begin

subclass semiring-0-cancel ⟨proof⟩

subclass comm-semiring-0 ⟨proof⟩

end

class zero-neq-one = zero + one +
  assumes zero-neq-one [simp]:  $0 \neq 1$ 
begin

lemma one-neq-zero [simp]:  $1 \neq 0$ 
  ⟨proof⟩

end

class semiring-1 = zero-neq-one + semiring-0 + monoid-mult

Abstract divisibility

class dvd = times
begin

definition dvd :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool (infixl dvd 50) where
  [code del]:  $b \text{ dvd } a \longleftrightarrow (\exists k. a = b * k)$ 

lemma dvdI [intro?]:  $a = b * k \Longrightarrow b \text{ dvd } a$ 
  ⟨proof⟩

lemma dvdE [elim?]:  $b \text{ dvd } a \Longrightarrow (\bigwedge k. a = b * k \Longrightarrow P) \Longrightarrow P$ 
  ⟨proof⟩

end

class comm-semiring-1 = zero-neq-one + comm-semiring-0 + comm-monoid-mult
  + dvd

begin

subclass semiring-1 ⟨proof⟩

lemma dvd-refl [simp]:  $a \text{ dvd } a$ 
  ⟨proof⟩

lemma dvd-trans:
  assumes  $a \text{ dvd } b$  and  $b \text{ dvd } c$ 
  shows  $a \text{ dvd } c$ 

```

$\langle proof \rangle$

lemma *dvd-0-left-iff* [*no-atp*, *simp*]: $0 \text{ dvd } a \longleftrightarrow a = 0$
 $\langle proof \rangle$

lemma *dvd-0-right* [*iff*]: $a \text{ dvd } 0$
 $\langle proof \rangle$

lemma *one-dvd* [*simp*]: $1 \text{ dvd } a$
 $\langle proof \rangle$

lemma *dvd-mult* [*simp*]: $a \text{ dvd } c \implies a \text{ dvd } (b * c)$
 $\langle proof \rangle$

lemma *dvd-mult2* [*simp*]: $a \text{ dvd } b \implies a \text{ dvd } (b * c)$
 $\langle proof \rangle$

lemma *dvd-triv-right* [*simp*]: $a \text{ dvd } b * a$
 $\langle proof \rangle$

lemma *dvd-triv-left* [*simp*]: $a \text{ dvd } a * b$
 $\langle proof \rangle$

lemma *mult-dvd-mono*:
 assumes $a \text{ dvd } b$
 and $c \text{ dvd } d$
 shows $a * c \text{ dvd } b * d$
 $\langle proof \rangle$

lemma *dvd-mult-left*: $a * b \text{ dvd } c \implies a \text{ dvd } c$
 $\langle proof \rangle$

lemma *dvd-mult-right*: $a * b \text{ dvd } c \implies b \text{ dvd } c$
 $\langle proof \rangle$

lemma *dvd-0-left*: $0 \text{ dvd } a \implies a = 0$
 $\langle proof \rangle$

lemma *dvd-add* [*simp*]:
 assumes $a \text{ dvd } b$ and $a \text{ dvd } c$ shows $a \text{ dvd } (b + c)$
 $\langle proof \rangle$

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  $a = 0 \vee b = 0$ 
 $\langle proof \rangle$ 

```

```

end

```

```

class semiring-1-cancel = semiring + cancel-comm-monoid-add
  + zero-neq-one + monoid-mult
begin

```

```

subclass semiring-0-cancel  $\langle proof \rangle$ 

```

```

subclass semiring-1  $\langle proof \rangle$ 

```

```

end

```

```

class comm-semiring-1-cancel = comm-semiring + cancel-comm-monoid-add
  + zero-neq-one + comm-monoid-mult
begin

```

```

subclass semiring-1-cancel  $\langle proof \rangle$ 
subclass comm-semiring-0-cancel  $\langle proof \rangle$ 
subclass comm-semiring-1  $\langle proof \rangle$ 

```

```

end

```

```

class ring = semiring + ab-group-add
begin

```

```

subclass semiring-0-cancel  $\langle proof \rangle$ 

```

Distribution rules

```

lemma minus-mult-left:  $-(a * b) = -a * b$ 
 $\langle proof \rangle$ 

```

```

lemma minus-mult-right:  $-(a * b) = a * -b$ 
 $\langle proof \rangle$ 

```

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]:  $-a * -b = a * b$ 
 $\langle proof \rangle$ 

```

```

lemma minus-mult-commute:  $-a * b = a * -b$ 
 $\langle proof \rangle$ 

```

```

lemma right-diff-distrib[algebra-simps, field-simps]:  $a * (b - c) = a * b - a * c$ 

```

$\langle proof \rangle$

lemma *left-diff-distrib*[*algebra-simps*, *field-simps*]: $(a - b) * c = a * c - b * c$
 $\langle proof \rangle$

lemmas *ring-distrib*[*no-atp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

lemma *eq-add-iff1*:
 $a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
 $\langle proof \rangle$

lemma *eq-add-iff2*:
 $a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
 $\langle proof \rangle$

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* $\langle proof \rangle$
subclass *comm-semiring-0-cancel* $\langle proof \rangle$

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* $\langle proof \rangle$

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* $\langle proof \rangle$
subclass *comm-semiring-1-cancel* $\langle proof \rangle$

lemma *dvd-minus-iff* [*simp*]: $x \text{ dvd } - y \longleftrightarrow x \text{ dvd } y$
 $\langle proof \rangle$

lemma *minus-dvd-iff* [*simp*]: $- x \text{ dvd } y \longleftrightarrow x \text{ dvd } y$
 $\langle proof \rangle$

lemma *dvd-diff* [*simp*]: $x \text{ dvd } y \implies x \text{ dvd } z \implies x \text{ dvd } (y - z)$
 $\langle \text{proof} \rangle$

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
shows $a * b = 0 \iff (a = 0 \vee b = 0)$
 $\langle \text{proof} \rangle$

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*, *no-atp*]:
 $a * c = b * c \iff c = 0 \vee a = b$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-left* [*simp*, *no-atp*]:
 $c * a = c * b \iff c = 0 \vee a = b$
 $\langle \text{proof} \rangle$

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *square-eq-1-iff*:
 $x * x = 1 \iff x = 1 \vee x = -1$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-right1* [*simp*]:
 $c = b * c \iff c = 0 \vee b = 1$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-right2* [*simp*]:
 $a * c = c \iff c = 0 \vee a = 1$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-left1* [*simp*]:
 $c = c * b \iff c = 0 \vee b = 1$
 $\langle \text{proof} \rangle$

lemma *mult-cancel-left2* [*simp*]:
 $c * a = c \iff c = 0 \vee a = 1$
 $\langle \text{proof} \rangle$

end

class *idom* = *comm-ring-1* + *no-zero-divisors*

begin

subclass *ring-1-no-zero-divisors* $\langle \text{proof} \rangle$

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow (a = b \vee a = - b)$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-cancel-right* [*simp*]:
 $a * c \text{ dvd } b * c \longleftrightarrow c = 0 \vee a \text{ dvd } b$
 $\langle \text{proof} \rangle$

lemma *dvd-mult-cancel-left* [*simp*]:
 $c * a \text{ dvd } c * b \longleftrightarrow c = 0 \vee a \text{ dvd } b$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *nonzero-imp-inverse-nonzero*:
 $a \neq 0 \implies \text{inverse } a \neq 0$
 $\langle \text{proof} \rangle$

lemma *inverse-zero-imp-zero*:
 $\text{inverse } a = 0 \implies a = 0$
 $\langle \text{proof} \rangle$

lemma *inverse-unique*:
assumes *ab*: $a * b = 1$
shows $\text{inverse } a = b$
 $\langle \text{proof} \rangle$

lemma *nonzero-inverse-minus-eq*:
 $a \neq 0 \implies \text{inverse } (- a) = - \text{inverse } a$
 $\langle \text{proof} \rangle$

lemma *nonzero-inverse-inverse-eq*:
 $a \neq 0 \implies \text{inverse } (\text{inverse } a) = a$

$\langle \text{proof} \rangle$

lemma *nonzero-inverse-eq-imp-eq*:

assumes $\text{inverse } a = \text{inverse } b$ **and** $a \neq 0$ **and** $b \neq 0$

shows $a = b$

$\langle \text{proof} \rangle$

lemma *inverse-1 [simp]*: $\text{inverse } 1 = 1$

$\langle \text{proof} \rangle$

lemma *nonzero-inverse-mult-distrib*:

assumes $a \neq 0$ **and** $b \neq 0$

shows $\text{inverse } (a * b) = \text{inverse } b * \text{inverse } a$

$\langle \text{proof} \rangle$

lemma *division-ring-inverse-add*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a + \text{inverse } b = \text{inverse } a * (a + b) * \text{inverse } b$

$\langle \text{proof} \rangle$

lemma *division-ring-inverse-diff*:

$a \neq 0 \implies b \neq 0 \implies \text{inverse } a - \text{inverse } b = \text{inverse } a * (b - a) * \text{inverse } b$

$\langle \text{proof} \rangle$

lemma *right-inverse-eq*: $b \neq 0 \implies a / b = 1 \longleftrightarrow a = b$

$\langle \text{proof} \rangle$

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \implies \text{inverse } a = 1 / a$

$\langle \text{proof} \rangle$

lemma *divide-self [simp]*: $a \neq 0 \implies a / a = 1$

$\langle \text{proof} \rangle$

lemma *divide-zero-left [simp]*: $0 / a = 0$

$\langle \text{proof} \rangle$

lemma *inverse-eq-divide*: $\text{inverse } a = 1 / a$

$\langle \text{proof} \rangle$

lemma *add-divide-distrib*: $(a+b) / c = a/c + b/c$

$\langle \text{proof} \rangle$

lemma *divide-1 [simp]*: $a / 1 = a$

$\langle \text{proof} \rangle$

lemma *times-divide-eq-right [simp]*: $a * (b / c) = (a * b) / c$

$\langle \text{proof} \rangle$

lemma *minus-divide-left*: $-(a / b) = (-a) / b$

$\langle \text{proof} \rangle$

lemma *nonzero-minus-divide-right*: $b \neq 0 \implies -(a / b) = a / (-b)$
 ⟨proof⟩

lemma *nonzero-minus-divide-divide*: $b \neq 0 \implies (-a) / (-b) = a / b$
 ⟨proof⟩

lemma *divide-minus-left* [simp, no-atp]: $(-a) / b = -(a / b)$
 ⟨proof⟩

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$
 ⟨proof⟩

lemma *nonzero-eq-divide-eq* [field-simps]: $c \neq 0 \implies a = b / c \longleftrightarrow a * c = b$
 ⟨proof⟩

lemma *nonzero-divide-eq-eq* [field-simps]: $c \neq 0 \implies b / c = a \longleftrightarrow b = a * c$
 ⟨proof⟩

lemma *divide-eq-imp*: $c \neq 0 \implies b = a * c \implies b / c = a$
 ⟨proof⟩

lemma *eq-divide-imp*: $c \neq 0 \implies a * c = b \implies a = b / c$
 ⟨proof⟩

end

class *division-ring-inverse-zero* = *division-ring* +
assumes *inverse-zero* [simp]: $\text{inverse } 0 = 0$
begin

lemma *divide-zero* [simp]:
 $a / 0 = 0$
 ⟨proof⟩

lemma *divide-self-if* [simp]:
 $a / a = (\text{if } a = 0 \text{ then } 0 \text{ else } 1)$
 ⟨proof⟩

lemma *inverse-nonzero-iff-nonzero* [simp]:
 $\text{inverse } a = 0 \longleftrightarrow a = 0$
 ⟨proof⟩

lemma *inverse-minus-eq* [simp]:
 $\text{inverse } (-a) = - \text{inverse } a$
 ⟨proof⟩

lemma *inverse-eq-imp-eq*:
 $\text{inverse } a = \text{inverse } b \implies a = b$

$\langle proof \rangle$

lemma *inverse-eq-iff-eq* [simp]:
 $inverse\ a = inverse\ b \longleftrightarrow a = b$
 $\langle proof \rangle$

lemma *inverse-inverse-eq* [simp]:
 $inverse\ (inverse\ a) = a$
 $\langle proof \rangle$

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*:
 $a \leq b \implies c \leq d \implies 0 \leq b \implies 0 \leq c$
 $\implies a * c \leq b * d$
 $\langle proof \rangle$

lemma *mult-mono'*:
 $a \leq b \implies c \leq d \implies 0 \leq a \implies 0 \leq c$
 $\implies a * c \leq b * d$
 $\langle proof \rangle$

end

class *ordered-cancel-semiring* = *mult-mono* + *ordered-ab-semigroup-add*
+ *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* $\langle proof \rangle$
subclass *ordered-semiring* $\langle proof \rangle$

lemma *mult-nonneg-nonneg*: $0 \leq a \implies 0 \leq b \implies 0 \leq a * b$
 ⟨proof⟩

lemma *mult-nonneg-nonpos*: $0 \leq a \implies b \leq 0 \implies a * b \leq 0$
 ⟨proof⟩

lemma *mult-nonpos-nonneg*: $a \leq 0 \implies 0 \leq b \implies a * b \leq 0$
 ⟨proof⟩

Legacy - use *mult-nonpos-nonneg*

lemma *mult-nonneg-nonpos2*: $0 \leq a \implies b \leq 0 \implies b * a \leq 0$
 ⟨proof⟩

lemma *split-mult-neg-le*: $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \implies a * b \leq 0$
 ⟨proof⟩

end

class *linordered-semiring* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add*
 + *mult-mono*
begin

subclass *ordered-cancel-semiring* ⟨proof⟩

subclass *ordered-comm-monoid-add* ⟨proof⟩

lemma *mult-left-less-imp-less*:
 $c * a < c * b \implies 0 \leq c \implies a < b$
 ⟨proof⟩

lemma *mult-right-less-imp-less*:
 $a * c < b * c \implies 0 \leq c \implies a < b$
 ⟨proof⟩

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⟩

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* $\langle \text{proof} \rangle$

subclass *linordered-semiring*
 $\langle \text{proof} \rangle$

lemma *mult-left-le-imp-le*:
 $c * a \leq c * b \implies 0 < c \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *mult-right-le-imp-le*:
 $a * c \leq b * c \implies 0 < c \implies a \leq b$
 $\langle \text{proof} \rangle$

lemma *mult-pos-pos*: $0 < a \implies 0 < b \implies 0 < a * b$
 $\langle \text{proof} \rangle$

lemma *mult-pos-neg*: $0 < a \implies b < 0 \implies a * b < 0$
 $\langle \text{proof} \rangle$

lemma *mult-neg-pos*: $a < 0 \implies 0 < b \implies a * b < 0$
 $\langle \text{proof} \rangle$

Legacy - use *mult-neg-pos*

lemma *mult-pos-neg2*: $0 < a \implies b < 0 \implies b * a < 0$
 $\langle \text{proof} \rangle$

lemma *zero-less-mult-pos*:
 $0 < a * b \implies 0 < a \implies 0 < b$
 $\langle \text{proof} \rangle$

lemma *zero-less-mult-pos2*:
 $0 < b * a \implies 0 < a \implies 0 < b$
 $\langle \text{proof} \rangle$

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \leq c$
shows $a * c < b * d$
 $\langle \text{proof} \rangle$

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 $a * c < b * d$
 $\langle \text{proof} \rangle$

lemma *mult-less-le-imp-less*:

assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
 $\langle proof \rangle$

lemma *mult-le-less-imp-less*:

assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
 $\langle proof \rangle$

lemma *mult-less-imp-less-left*:

assumes *less*: $c * a < c * b$ **and** *nonneg*: $0 \leq c$
shows $a < b$
 $\langle proof \rangle$

lemma *mult-less-imp-less-right*:

assumes *less*: $a * c < b * c$ **and** *nonneg*: $0 \leq c$
shows $a < b$
 $\langle proof \rangle$

end

class *linordered-semiring-1-strict* = *linordered-semiring-strict* + *semiring-1*
begin

subclass *linordered-semiring-1* $\langle proof \rangle$

lemma *convex-bound-lt*:

assumes $x < a$ $y < a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $u * x + v * y < a$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

end

class *ordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *ordered-ab-semigroup-add* + *mult-mono1*

begin

subclass *ordered-comm-semiring* $\langle \text{proof} \rangle$

subclass *ordered-cancel-semiring* $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

subclass *ordered-cancel-comm-semiring*
 $\langle \text{proof} \rangle$

end

class *ordered-ring* = *ring* + *ordered-cancel-semiring*
begin

subclass *ordered-ab-group-add* $\langle \text{proof} \rangle$

lemma *less-add-iff1*:

$a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
 $\langle \text{proof} \rangle$

lemma *less-add-iff2*:

$a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$
 $\langle \text{proof} \rangle$

lemma *le-add-iff1*:

$a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
 $\langle \text{proof} \rangle$

lemma *le-add-iff2*:

$a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
 $\langle \text{proof} \rangle$

lemma *mult-left-mono-neg*:

$b \leq a \implies c \leq 0 \implies c * a \leq c * b$
 $\langle \text{proof} \rangle$

lemma *mult-right-mono-neg*:

$b \leq a \implies c \leq 0 \implies a * c \leq b * c$
 $\langle \text{proof} \rangle$

lemma *mult-nonpos-nonpos*: $a \leq 0 \implies b \leq 0 \implies 0 \leq a * b$
 $\langle proof \rangle$

lemma *split-mult-pos-le*:
 $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \implies 0 \leq a * b$
 $\langle proof \rangle$

end

class *linordered-ring* = *ring* + *linordered-semiring* + *linordered-ab-group-add* +
abs-if
begin

subclass *ordered-ring* $\langle proof \rangle$

subclass *ordered-ab-group-add-abs*
 $\langle proof \rangle$

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
 $\langle proof \rangle$

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
 $\langle proof \rangle$

end

class *linordered-ring-strict* = *ring* + *linordered-semiring-strict*
+ *ordered-ab-group-add* + *abs-if*
begin

subclass *linordered-ring* $\langle proof \rangle$

lemma *mult-strict-left-mono-neg*: $b < a \implies c < 0 \implies c * a < c * b$
 $\langle proof \rangle$

lemma *mult-strict-right-mono-neg*: $b < a \implies c < 0 \implies a * c < b * c$
 $\langle proof \rangle$

lemma *mult-neg-neg*: $a < 0 \implies b < 0 \implies 0 < a * b$
 $\langle proof \rangle$

subclass *ring-no-zero-divisors*
 $\langle proof \rangle$

lemma *zero-less-mult-iff*:
 $0 < a * b \iff 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
 $\langle proof \rangle$

lemma *zero-le-mult-iff*:

$$0 \leq a * b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$$

<proof>

lemma *mult-less-0-iff*:

$$a * b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$$

<proof>

lemma *mult-le-0-iff*:

$$a * b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$$

<proof>

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*:

$$a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$$

<proof>

lemma *mult-less-cancel-left-disj*:

$$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$$

<proof>

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*:

$$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$$

<proof>

lemma *mult-less-cancel-left*:

$$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$$

<proof>

lemma *mult-le-cancel-right*:

$$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$$

<proof>

lemma *mult-le-cancel-left*:

$$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$$

<proof>

lemma *mult-le-cancel-left-pos*:

$$0 < c \implies c * a \leq c * b \longleftrightarrow a \leq b$$

<proof>

lemma *mult-le-cancel-left-neg*:

$$c < 0 \implies c * a \leq c * b \longleftrightarrow b \leq a$$

<proof>

lemma *mult-less-cancel-left-pos*:

$0 < c \implies c * a < c * b \longleftrightarrow a < b$

<proof>

lemma *mult-less-cancel-left-neg*:

$c < 0 \implies c * a < c * b \longleftrightarrow b < a$

<proof>

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* *<proof>*

subclass *ordered-cancel-comm-semiring* *<proof>*

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 $0 < a \implies b < c \implies b < a + c$

<proof>

lemma *zero-le-one* [*simp*]: $0 \leq 1$

<proof>

lemma *not-one-le-zero* [*simp*]: $\neg 1 \leq 0$

<proof>

lemma *not-one-less-zero* [*simp*]: $\neg 1 < 0$

<proof>

lemma *less-1-mult*:

assumes $1 < m$ **and** $1 < n$

shows $1 < m * n$

<proof>

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* *<proof>*
subclass *linordered-ring-strict* *<proof>*
subclass *ordered-comm-ring* *<proof>*
subclass *idom* *<proof>*

subclass *linordered-semidom*
<proof>

lemma *linorder-neqE-linordered-idom*:
assumes $x \neq y$ **obtains** $x < y \mid y < x$
<proof>

These cancellation simprules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
 $c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
<proof>

lemma *mult-le-cancel-right2*:
 $a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
<proof>

lemma *mult-le-cancel-left1*:
 $c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
<proof>

lemma *mult-le-cancel-left2*:
 $c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
<proof>

lemma *mult-less-cancel-right1*:
 $c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
<proof>

lemma *mult-less-cancel-right2*:
 $a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
<proof>

lemma *mult-less-cancel-left1*:
 $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$

$\langle proof \rangle$

lemma *mult-less-cancel-left2*:

$$c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$$

$\langle proof \rangle$

lemma *sgn-sgn [simp]*:

$$sgn (sgn a) = sgn a$$

$\langle proof \rangle$

lemma *sgn-0-0*:

$$sgn a = 0 \longleftrightarrow a = 0$$

$\langle proof \rangle$

lemma *sgn-1-pos*:

$$sgn a = 1 \longleftrightarrow a > 0$$

$\langle proof \rangle$

lemma *sgn-1-neg*:

$$sgn a = -1 \longleftrightarrow a < 0$$

$\langle proof \rangle$

lemma *sgn-pos [simp]*:

$$0 < a \implies sgn a = 1$$

$\langle proof \rangle$

lemma *sgn-neg [simp]*:

$$a < 0 \implies sgn a = -1$$

$\langle proof \rangle$

lemma *sgn-times*:

$$sgn (a * b) = sgn a * sgn b$$

$\langle proof \rangle$

lemma *abs-sgn*: $|k| = k * sgn k$

$\langle proof \rangle$

lemma *sgn-greater [simp]*:

$$0 < sgn a \longleftrightarrow 0 < a$$

$\langle proof \rangle$

lemma *sgn-less [simp]*:

$$sgn a < 0 \longleftrightarrow a < 0$$

$\langle proof \rangle$

lemma *abs-dvd-iff [simp]*: $|m| \text{ dvd } k \longleftrightarrow m \text{ dvd } k$

$\langle proof \rangle$

lemma *dvd-abs-iff [simp]*: $m \text{ dvd } |k| \longleftrightarrow m \text{ dvd } k$

$\langle proof \rangle$

lemma *dvd-if-abs-eq*:

$|l| = |k| \implies l \text{ dvd } k$

$\langle proof \rangle$

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*: $a < a + 1$

$\langle proof \rangle$

lemma *zero-less-two*: $0 < 1 + 1$

$\langle proof \rangle$

end

context *linordered-idom*

begin

lemma *mult-right-le-one-le*:

$0 \leq x \implies 0 \leq y \implies y \leq 1 \implies x * y \leq x$

$\langle proof \rangle$

lemma *mult-left-le-one-le*:

$0 \leq x \implies 0 \leq y \implies y \leq 1 \implies y * x \leq x$

$\langle proof \rangle$

end

Absolute Value

context *linordered-idom*

begin

lemma *mult-sgn-abs*:

$$\text{sgn } x * |x| = x$$

<proof>

lemma *abs-one [simp]*:

$$|1| = 1$$

<proof>

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>*

lemma *abs-mult*:

$$|a * b| = |a| * |b|$$

<proof>

lemma *abs-mult-self*:

$$|a| * |a| = a * a$$

<proof>

lemma *abs-mult-less*:

$$|a| < c \implies |b| < d \implies |a| * |b| < c * d$$

<proof>

lemma *less-minus-self-iff*:

$$a < -a \longleftrightarrow a < 0$$

<proof>

lemma *abs-less-iff*:

$$|a| < b \longleftrightarrow a < b \wedge -a < b$$

<proof>

lemma *abs-mult-pos*:

$$0 \leq x \implies |y| * x = |y * x|$$

<proof>

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*

$\langle \text{proof} \rangle$

subclass *idom* $\langle \text{proof} \rangle$

There is no slick version using division by zero.

lemma *inverse-add*:

$\llbracket a \neq 0; b \neq 0 \rrbracket$
 $\implies \text{inverse } a + \text{inverse } b = (a + b) * \text{inverse } a * \text{inverse } b$
 $\langle \text{proof} \rangle$

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*, *no-atp*]:

assumes [*simp*]: $b \neq 0$ **and** [*simp*]: $c \neq 0$ **shows** $(c*a)/(c*b) = a/b$
 $\langle \text{proof} \rangle$

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*, *no-atp*]:

$\llbracket b \neq 0; c \neq 0 \rrbracket \implies (a * c) / (b * c) = a / b$
 $\langle \text{proof} \rangle$

lemma *times-divide-eq-left* [*simp*]: $(b / c) * a = (b * a) / c$

$\langle \text{proof} \rangle$

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 $x / y + w / z = (x * z + w * y) / (y * z)$
 $\langle \text{proof} \rangle$

Special Cancellation Simprules for Division

lemma *nonzero-mult-divide-cancel-right* [*simp*, *no-atp*]:

$$b \neq 0 \implies a * b / b = a$$

$\langle \text{proof} \rangle$

lemma *nonzero-mult-divide-cancel-left* [*simp*, *no-atp*]:

$$a \neq 0 \implies a * b / a = b$$

$\langle \text{proof} \rangle$

lemma *nonzero-divide-mult-cancel-right* [*simp*, *no-atp*]:

$$\llbracket a \neq 0; b \neq 0 \rrbracket \implies b / (a * b) = 1 / a$$

$\langle \text{proof} \rangle$

lemma *nonzero-divide-mult-cancel-left* [*simp*, *no-atp*]:

$$\llbracket a \neq 0; b \neq 0 \rrbracket \implies a / (a * b) = 1 / b$$

$\langle \text{proof} \rangle$

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*, *no-atp*]:

$$\llbracket b \neq 0; c \neq 0 \rrbracket \implies (c * a) / (b * c) = a / b$$

$\langle \text{proof} \rangle$

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*, *no-atp*]:

$$\llbracket b \neq 0; c \neq 0 \rrbracket \implies (a * c) / (c * b) = a / b$$

$\langle \text{proof} \rangle$

lemma *add-divide-eq-iff* [*field-simps*]:

$$z \neq 0 \implies x + y / z = (z * x + y) / z$$

$\langle \text{proof} \rangle$

lemma *divide-add-eq-iff* [*field-simps*]:

$$z \neq 0 \implies x / z + y = (x + z * y) / z$$

$\langle \text{proof} \rangle$

lemma *diff-divide-eq-iff* [*field-simps*]:

$$z \neq 0 \implies x - y / z = (z * x - y) / z$$

$\langle \text{proof} \rangle$

lemma *divide-diff-eq-iff* [*field-simps*]:

$$z \neq 0 \implies x / z - y = (x - z * y) / z$$

$\langle \text{proof} \rangle$

lemma *diff-frac-eq*:

$$y \neq 0 \implies z \neq 0 \implies x / y - w / z = (x * z - w * y) / (y * z)$$

$\langle \text{proof} \rangle$

lemma *frac-eq-eq*:

$$y \neq 0 \implies z \neq 0 \implies (x / y = w / z) = (x * z = w * y)$$

$\langle \text{proof} \rangle$

end

class *field-inverse-zero* = *field* +
assumes *field-inverse-zero*: *inverse* 0 = 0
begin

subclass *division-ring-inverse-zero* \langle *proof* \rangle

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
 $\text{inverse } (a * b) = \text{inverse } a * \text{inverse } b$
 \langle *proof* \rangle

lemma *inverse-divide* [*simp*]:
 $\text{inverse } (a / b) = b / a$
 \langle *proof* \rangle

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*:
 $c \neq 0 \implies (c * a) / (c * b) = a / b$
 \langle *proof* \rangle

lemma *mult-divide-mult-cancel-right*:
 $c \neq 0 \implies (a * c) / (b * c) = a / b$
 \langle *proof* \rangle

lemma *divide-divide-eq-right* [*simp*, *no-atp*]:
 $a / (b / c) = (a * c) / b$
 \langle *proof* \rangle

lemma *divide-divide-eq-left* [*simp*, *no-atp*]:
 $(a / b) / c = a / (b * c)$
 \langle *proof* \rangle

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [*simp*, *no-atp*]:
shows $(c * a) / (c * b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a / b)$
 \langle *proof* \rangle

Division and Unary Minus

lemma *minus-divide-right*:
 $-(a / b) = a / -b$
 \langle *proof* \rangle

lemma *divide-minus-right* [*simp*, *no-atp*]:

$$a / - b = - (a / b)$$

<proof>

lemma *minus-divide-divide*:

$$(- a) / (- b) = a / b$$

<proof>

lemma *eq-divide-eq*:

$$a = b / c \longleftrightarrow (\text{if } c \neq 0 \text{ then } a * c = b \text{ else } a = 0)$$

<proof>

lemma *divide-eq-eq*:

$$b / c = a \longleftrightarrow (\text{if } c \neq 0 \text{ then } b = a * c \text{ else } a = 0)$$

<proof>

lemma *inverse-eq-1-iff* [*simp*]:

$$\text{inverse } x = 1 \longleftrightarrow x = 1$$

<proof>

lemma *divide-eq-0-iff* [*simp*, *no-atp*]:

$$a / b = 0 \longleftrightarrow a = 0 \vee b = 0$$

<proof>

lemma *divide-cancel-right* [*simp*, *no-atp*]:

$$a / c = b / c \longleftrightarrow c = 0 \vee a = b$$

<proof>

lemma *divide-cancel-left* [*simp*, *no-atp*]:

$$c / a = c / b \longleftrightarrow c = 0 \vee a = b$$

<proof>

lemma *divide-eq-1-iff* [*simp*, *no-atp*]:

$$a / b = 1 \longleftrightarrow b \neq 0 \wedge a = b$$

<proof>

lemma *one-eq-divide-iff* [*simp*, *no-atp*]:

$$1 = a / b \longleftrightarrow b \neq 0 \wedge a = b$$

<proof>

lemma *times-divide-times-eq*:

$$(x / y) * (z / w) = (x * z) / (y * w)$$

<proof>

lemma *add-frac-num*:

$$y \neq 0 \implies x / y + z = (x + z * y) / y$$

<proof>

lemma *add-num-frac*:

$$y \neq 0 \implies z + x / y = (x + z * y) / y$$

⟨proof⟩

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⟩

lemma *negative-imp-inverse-negative*:

$a < 0 \implies \text{inverse } a < 0$

⟨proof⟩

lemma *inverse-le-imp-le*:

assumes *invle*: $\text{inverse } a \leq \text{inverse } b$ **and** *apos*: $0 < a$

shows $b \leq a$

⟨proof⟩

lemma *inverse-positive-imp-positive*:

assumes *inv-gt-0*: $0 < \text{inverse } a$ **and** *nz*: $a \neq 0$

shows $0 < a$

⟨proof⟩

lemma *inverse-negative-imp-negative*:

assumes *inv-less-0*: $\text{inverse } a < 0$ **and** *nz*: $a \neq 0$

shows $a < 0$

⟨proof⟩

lemma *linordered-field-no-lb*:

$\forall x. \exists y. y < x$

⟨proof⟩

lemma *linordered-field-no-ub*:

$\forall x. \exists y. y > x$

⟨proof⟩

lemma *less-imp-inverse-less*:

assumes *less*: $a < b$ **and** *apos*: $0 < a$

shows $\text{inverse } b < \text{inverse } a$

⟨proof⟩

lemma *inverse-less-imp-less*:

$\text{inverse } a < \text{inverse } b \implies 0 < a \implies b < a$

⟨proof⟩

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp,no-atp*]:

$$0 < a \implies 0 < b \implies \text{inverse } a < \text{inverse } b \longleftrightarrow b < a$$

<proof>

lemma *le-imp-inverse-le*:

$$a \leq b \implies 0 < a \implies \text{inverse } b \leq \text{inverse } a$$

<proof>

lemma *inverse-le-iff-le* [*simp,no-atp*]:

$$0 < a \implies 0 < b \implies \text{inverse } a \leq \text{inverse } b \longleftrightarrow b \leq a$$

<proof>

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:

$$\text{inverse } a \leq \text{inverse } b \implies b < 0 \implies b \leq a$$

<proof>

lemma *less-imp-inverse-less-neg*:

$$a < b \implies b < 0 \implies \text{inverse } b < \text{inverse } a$$

<proof>

lemma *inverse-less-imp-less-neg*:

$$\text{inverse } a < \text{inverse } b \implies b < 0 \implies b < a$$

<proof>

lemma *inverse-less-iff-less-neg* [*simp,no-atp*]:

$$a < 0 \implies b < 0 \implies \text{inverse } a < \text{inverse } b \longleftrightarrow b < a$$

<proof>

lemma *le-imp-inverse-le-neg*:

$$a \leq b \implies b < 0 \implies \text{inverse } b \leq \text{inverse } a$$

<proof>

lemma *inverse-le-iff-le-neg* [*simp,no-atp*]:

$$a < 0 \implies b < 0 \implies \text{inverse } a \leq \text{inverse } b \longleftrightarrow b \leq a$$

<proof>

lemma *one-less-inverse*:

$$0 < a \implies a < 1 \implies 1 < \text{inverse } a$$

<proof>

lemma *one-le-inverse*:

$$0 < a \implies a \leq 1 \implies 1 \leq \text{inverse } a$$

<proof>

lemma *pos-le-divide-eq* [*field-simps*]: $0 < c \implies (a \leq b/c) = (a*c \leq b)$

<proof>

lemma *neg-le-divide-eq* [*field-simps*]: $c < 0 \implies (a \leq b/c) = (b \leq a*c)$
 $\langle \text{proof} \rangle$

lemma *pos-less-divide-eq* [*field-simps*]:
 $0 < c \implies (a < b/c) = (a*c < b)$
 $\langle \text{proof} \rangle$

lemma *neg-less-divide-eq* [*field-simps*]:
 $c < 0 \implies (a < b/c) = (b < a*c)$
 $\langle \text{proof} \rangle$

lemma *pos-divide-less-eq* [*field-simps*]:
 $0 < c \implies (b/c < a) = (b < a*c)$
 $\langle \text{proof} \rangle$

lemma *neg-divide-less-eq* [*field-simps*]:
 $c < 0 \implies (b/c < a) = (a*c < b)$
 $\langle \text{proof} \rangle$

lemma *pos-divide-le-eq* [*field-simps*]: $0 < c \implies (b/c \leq a) = (b \leq a*c)$
 $\langle \text{proof} \rangle$

lemma *neg-divide-le-eq* [*field-simps*]: $c < 0 \implies (b/c \leq a) = (a*c \leq b)$
 $\langle \text{proof} \rangle$

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*:
 $0 < x \implies 0 < y \implies 0 < x / y$
 $\langle \text{proof} \rangle$

lemma *divide-nonneg-pos*:
 $0 \leq x \implies 0 < y \implies 0 \leq x / y$
 $\langle \text{proof} \rangle$

lemma *divide-neg-pos*:
 $x < 0 \implies 0 < y \implies x / y < 0$
 $\langle \text{proof} \rangle$

lemma *divide-nonpos-pos*:

$$x \leq 0 \implies 0 < y \implies x / y \leq 0$$

<proof>

lemma *divide-pos-neg*:

$$0 < x \implies y < 0 \implies x / y < 0$$

<proof>

lemma *divide-nonneg-neg*:

$$0 \leq x \implies y < 0 \implies x / y \leq 0$$

<proof>

lemma *divide-neg-neg*:

$$x < 0 \implies y < 0 \implies 0 < x / y$$

<proof>

lemma *divide-nonpos-neg*:

$$x \leq 0 \implies y < 0 \implies 0 \leq x / y$$

<proof>

lemma *divide-strict-right-mono*:

$$[|a < b; 0 < c|] \implies a / c < b / c$$

<proof>

lemma *divide-strict-right-mono-neg*:

$$[|b < a; c < 0|] \implies a / c < b / c$$

<proof>

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:

$$[|b < a; 0 < c; 0 < a*b|] \implies c / a < c / b$$

<proof>

lemma *divide-left-mono*:

$$[|b \leq a; 0 \leq c; 0 < a*b|] \implies c / a \leq c / b$$

<proof>

lemma *divide-strict-left-mono-neg*:

$$[|a < b; c < 0; 0 < a*b|] \implies c / a < c / b$$

<proof>

lemma *mult-imp-div-pos-le*: $0 < y \implies x \leq z * y \implies$

$$x / y \leq z$$

<proof>

lemma *mult-imp-le-div-pos*: $0 < y \implies z * y \leq x \implies$

$$z \leq x / y$$

<proof>

lemma *mult-imp-div-pos-less*: $0 < y \implies x < z * y \implies$
 $x / y < z$
 $\langle proof \rangle$

lemma *mult-imp-less-div-pos*: $0 < y \implies z * y < x \implies$
 $z < x / y$
 $\langle proof \rangle$

lemma *frac-le*: $0 \leq x \implies$
 $x \leq y \implies 0 < w \implies w \leq z \implies x / z \leq y / w$
 $\langle proof \rangle$

lemma *frac-less*: $0 \leq x \implies$
 $x < y \implies 0 < w \implies w \leq z \implies x / z < y / w$
 $\langle proof \rangle$

lemma *frac-less2*: $0 < x \implies$
 $x \leq y \implies 0 < w \implies w < z \implies x / z < y / w$
 $\langle proof \rangle$

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*: $a < b \implies a < (a+b) / (1+1)$
 $\langle proof \rangle$

lemma *gt-half-sum*: $a < b \implies (a+b)/(1+1) < b$
 $\langle proof \rangle$

subclass *dense-linorder*
 $\langle proof \rangle$

lemma *nonzero-abs-inverse*:
 $a \neq 0 \implies |inverse\ a| = inverse\ |a|$
 $\langle proof \rangle$

lemma *nonzero-abs-divide*:
 $b \neq 0 \implies |a / b| = |a| / |b|$
 $\langle proof \rangle$

lemma *field-le-epsilon*:
assumes $e: \bigwedge e. 0 < e \implies x \leq y + e$
shows $x \leq y$
 $\langle proof \rangle$

end

class *linordered-field-inverse-zero* = *linordered-field* + *field-inverse-zero*

begin

lemma *le-divide-eq*:

$$(a \leq b/c) =$$

$$(if\ 0 < c\ then\ a*c \leq b$$

$$\quad\quad\quad else\ if\ c < 0\ then\ b \leq a*c$$

$$\quad\quad\quad else\ a \leq 0)$$

$\langle proof \rangle$

lemma *inverse-positive-iff-positive* [simp]:

$$(0 < inverse\ a) = (0 < a)$$

$\langle proof \rangle$

lemma *inverse-negative-iff-negative* [simp]:

$$(inverse\ a < 0) = (a < 0)$$

$\langle proof \rangle$

lemma *inverse-nonnegative-iff-nonnegative* [simp]:

$$0 \leq inverse\ a \longleftrightarrow 0 \leq a$$

$\langle proof \rangle$

lemma *inverse-nonpositive-iff-nonpositive* [simp]:

$$inverse\ a \leq 0 \longleftrightarrow a \leq 0$$

$\langle proof \rangle$

lemma *one-less-inverse-iff*:

$$1 < inverse\ x \longleftrightarrow 0 < x \wedge x < 1$$

$\langle proof \rangle$

lemma *one-le-inverse-iff*:

$$1 \leq inverse\ x \longleftrightarrow 0 < x \wedge x \leq 1$$

$\langle proof \rangle$

lemma *inverse-less-1-iff*:

$$inverse\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$$

$\langle proof \rangle$

lemma *inverse-le-1-iff*:

$$inverse\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$$

$\langle proof \rangle$

lemma *divide-le-eq*:

$$(b/c \leq a) =$$

$$(if\ 0 < c\ then\ b \leq a*c$$

$$\quad\quad\quad else\ if\ c < 0\ then\ a*c \leq b$$

$$\quad\quad\quad else\ 0 \leq a)$$

$\langle proof \rangle$

lemma *less-divide-eq*:

$$(a < b/c) =$$

$$(if\ 0 < c\ then\ a*c < b$$

$$\quad\quad\quad else\ if\ c < 0\ then\ b < a*c$$

$$\quad\quad\quad else\ a < 0)$$

$\langle proof \rangle$

lemma *divide-less-eq*:

$$(b/c < a) =$$

$$(if\ 0 < c\ then\ b < a*c$$

$$\quad\quad\quad else\ if\ c < 0\ then\ a*c < b$$

$$\quad\quad\quad else\ 0 < a)$$

$\langle proof \rangle$

Division and Signs

lemma *zero-less-divide-iff*:

$$(0 < a/b) = (0 < a \ \&\ 0 < b \mid a < 0 \ \&\ b < 0)$$

$\langle proof \rangle$

lemma *divide-less-0-iff*:

$$(a/b < 0) =$$

$$(0 < a \ \&\ b < 0 \mid a < 0 \ \&\ 0 < b)$$

$\langle proof \rangle$

lemma *zero-le-divide-iff*:

$$(0 \leq a/b) =$$

$$(0 \leq a \ \&\ 0 \leq b \mid a \leq 0 \ \&\ b \leq 0)$$

$\langle proof \rangle$

lemma *divide-le-0-iff*:

$$(a/b \leq 0) =$$

$$(0 \leq a \ \&\ b \leq 0 \mid a \leq 0 \ \&\ 0 \leq b)$$

$\langle proof \rangle$

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp, no-atp*]:

$$(0 = 1/a) = (a = 0)$$

$\langle proof \rangle$

lemma *one-divide-eq-0-iff* [*simp, no-atp*]:

$$(1/a = 0) = (a = 0)$$

$\langle proof \rangle$

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [*simp, no-atp*]:

$$0 \leq 1/a \iff 0 \leq a$$

$\langle proof \rangle$

lemma *zero-less-divide-1-iff* [*simp, no-atp*]:

$$0 < 1 / a \iff 0 < a$$

<proof>

lemma *divide-le-0-1-iff* [*simp, no-atp*]:

$$1 / a \leq 0 \iff a \leq 0$$

<proof>

lemma *divide-less-0-1-iff* [*simp, no-atp*]:

$$1 / a < 0 \iff a < 0$$

<proof>

lemma *divide-right-mono*:

$$[|a \leq b; 0 \leq c|] \implies a/c \leq b/c$$

<proof>

lemma *divide-right-mono-neg*: $a \leq b$

$$\implies c \leq 0 \implies b / c \leq a / c$$

<proof>

lemma *divide-left-mono-neg*: $a \leq b$

$$\implies c \leq 0 \implies 0 < a * b \implies c / a \leq c / b$$

<proof>

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*no-atp*]:

$$(1 \leq b / a) = ((0 < a \ \& \ a \leq b) \mid (a < 0 \ \& \ b \leq a))$$

<proof>

lemma *divide-le-eq-1* [*no-atp*]:

$$(b / a \leq 1) = ((0 < a \ \& \ b \leq a) \mid (a < 0 \ \& \ a \leq b) \mid a=0)$$

<proof>

lemma *less-divide-eq-1* [*no-atp*]:

$$(1 < b / a) = ((0 < a \ \& \ a < b) \mid (a < 0 \ \& \ b < a))$$

<proof>

lemma *divide-less-eq-1* [*no-atp*]:

$$(b / a < 1) = ((0 < a \ \& \ b < a) \mid (a < 0 \ \& \ a < b) \mid a=0)$$

<proof>

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp, no-atp*]:

$$0 < a \implies (1 \leq b/a) = (a \leq b)$$

<proof>

lemma *le-divide-eq-1-neg* [*simp, no-atp*]:

$$a < 0 \implies (1 \leq b/a) = (b \leq a)$$

<proof>

lemma *divide-le-eq-1-pos* [*simp,no-atp*]:

$$0 < a \implies (b/a \leq 1) = (b \leq a)$$

<proof>

lemma *divide-le-eq-1-neg* [*simp,no-atp*]:

$$a < 0 \implies (b/a \leq 1) = (a \leq b)$$

<proof>

lemma *less-divide-eq-1-pos* [*simp,no-atp*]:

$$0 < a \implies (1 < b/a) = (a < b)$$

<proof>

lemma *less-divide-eq-1-neg* [*simp,no-atp*]:

$$a < 0 \implies (1 < b/a) = (b < a)$$

<proof>

lemma *divide-less-eq-1-pos* [*simp,no-atp*]:

$$0 < a \implies (b/a < 1) = (b < a)$$

<proof>

lemma *divide-less-eq-1-neg* [*simp,no-atp*]:

$$a < 0 \implies b/a < 1 \iff a < b$$

<proof>

lemma *eq-divide-eq-1* [*simp,no-atp*]:

$$(1 = b/a) = ((a \neq 0 \ \& \ a = b))$$

<proof>

lemma *divide-eq-eq-1* [*simp,no-atp*]:

$$(b/a = 1) = ((a \neq 0 \ \& \ a = b))$$

<proof>

lemma *abs-inverse* [*simp*]:

$$|inverse \ a| =$$

$$inverse \ |a|$$

<proof>

lemma *abs-divide* [*simp*]:

$$|a / b| = |a| / |b|$$

<proof>

lemma *abs-div-pos*: $0 < y \implies$

$$|x| / y = |x / y|$$

<proof>

lemma *field-le-mult-one-interval*:

assumes *: $\bigwedge z. \llbracket 0 < z \ ; \ z < 1 \rrbracket \implies z * x \leq y$
shows $x \leq y$

<proof>

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*
 $\langle proof \rangle$

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 $\langle proof \rangle$

end

lemma *Suc-not-Zero*: *Suc* *m* ≠ 0
 $\langle proof \rangle$

lemma *Zero-not-Suc*: 0 ≠ *Suc* *m*
 $\langle proof \rangle$

lemma *Suc-Rep-inject'*: *Suc-Rep* *x* = *Suc-Rep* *y* \longleftrightarrow *x* = *y*
 $\langle proof \rangle$

rep-datatype 0 :: *nat* *Suc*
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *Suc-neq-Zero*: $Suc\ m = 0 \implies R$
 $\langle proof \rangle$

lemma *Zero-neq-Suc*: $0 = Suc\ m \implies R$
 $\langle proof \rangle$

lemma *Suc-inject*: $Suc\ x = Suc\ y \implies x = y$
 $\langle proof \rangle$

lemma *n-not-Suc-n*: $n \neq Suc\ n$
 $\langle proof \rangle$

lemma *Suc-n-not-n*: $Suc\ n \neq n$
 $\langle proof \rangle$

A special form of induction for reasoning about $m < n$ and $m - n$

lemma *diff-induct*: $(!!x. P\ x\ 0) \implies (!!y. P\ 0\ (Suc\ y)) \implies$
 $(!!x\ y. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y)) \implies P\ m\ n$
 $\langle proof \rangle$

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]*: $m + 0 = (m::nat)$
 $\langle proof \rangle$

lemma *add-Suc-right [simp]*: $m + Suc\ n = Suc\ (m + n)$
 $\langle proof \rangle$

declare *add-0 [code]*

lemma *add-Suc-shift [code]*: $Suc\ m + n = m + Suc\ n$
 $\langle proof \rangle$

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]:  $0 - n = (0::nat)$ 
   $\langle proof \rangle$ 

lemma diff-Suc-Suc [simp, code]:  $Suc\ m - Suc\ n = m - n$ 
   $\langle proof \rangle$ 

instance  $\langle proof \rangle$ 

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]:  $(m::nat) * 0 = 0$ 
   $\langle proof \rangle$ 

lemma mult-Suc-right [simp]:  $m * Suc\ n = m + (m * n)$ 
   $\langle proof \rangle$ 

lemma add-mult-distrib:  $(m + n) * k = (m * k) + ((n * k)::nat)$ 
   $\langle proof \rangle$ 

instance  $\langle proof \rangle$ 

end

15.3.1 Addition

lemma nat-add-assoc:  $(m + n) + k = m + ((n + k)::nat)$ 
   $\langle proof \rangle$ 

lemma nat-add-commute:  $m + n = n + (m::nat)$ 
   $\langle proof \rangle$ 

lemma nat-add-left-commute:  $x + (y + z) = y + ((x + z)::nat)$ 

```

$\langle proof \rangle$

lemma *nat-add-left-cancel* [simp]: $(k + m = k + n) = (m = (n::nat))$
 $\langle proof \rangle$

lemma *nat-add-right-cancel* [simp]: $(m + k = n + k) = (m = (n::nat))$
 $\langle proof \rangle$

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [iff]:
fixes $m\ n :: nat$
shows $(m + n = 0) = (m = 0 \ \& \ n = 0)$
 $\langle proof \rangle$

lemma *add-is-1*:
 $(m+n = Suc\ 0) = (m = Suc\ 0 \ \& \ n=0 \mid m=0 \ \& \ n = Suc\ 0)$
 $\langle proof \rangle$

lemma *one-is-add*:
 $(Suc\ 0 = m + n) = (m = Suc\ 0 \ \& \ n = 0 \mid m = 0 \ \& \ n = Suc\ 0)$
 $\langle proof \rangle$

lemma *add-eq-self-zero*:
fixes $m\ n :: nat$
shows $m + n = m \implies n = 0$
 $\langle proof \rangle$

lemma *inj-on-add-nat*[simp]: *inj-on* $(\%n::nat. n+k)\ N$
 $\langle proof \rangle$

15.3.2 Difference

lemma *diff-self-eq-0* [simp]: $(m::nat) - m = 0$
 $\langle proof \rangle$

lemma *diff-diff-left*: $(i::nat) - j - k = i - (j + k)$
 $\langle proof \rangle$

lemma *Suc-diff-diff* [simp]: $(Suc\ m - n) - Suc\ k = m - n - k$
 $\langle proof \rangle$

lemma *diff-commute*: $(i::nat) - j - k = i - k - j$
 $\langle proof \rangle$

lemma *diff-add-inverse*: $(n + m) - n = (m::nat)$
 $\langle proof \rangle$

lemma *diff-add-inverse2*: $(m + n) - n = (m::nat)$
 $\langle proof \rangle$

lemma *diff-cancel*: $(k + m) - (k + n) = m - (n::nat)$
 $\langle proof \rangle$

lemma *diff-cancel2*: $(m + k) - (n + k) = m - (n::nat)$
 $\langle proof \rangle$

lemma *diff-add-0*: $n - (n + m) = (0::nat)$
 $\langle proof \rangle$

lemma *diff-Suc-1* [simp]: $Suc\ n - 1 = n$
 $\langle proof \rangle$

Difference distributes over multiplication

lemma *diff-mult-distrib*: $((m::nat) - n) * k = (m * k) - (n * k)$
 $\langle proof \rangle$

lemma *diff-mult-distrib2*: $k * ((m::nat) - n) = (k * m) - (k * n)$
 $\langle proof \rangle$

15.3.3 Multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
 $\langle proof \rangle$

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
 $\langle proof \rangle$

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
 $\langle proof \rangle$

lemma *mult-is-0* [simp]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
 $\langle proof \rangle$

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

lemma *mult-eq-1-iff* [simp]: $(m * n = Suc\ 0) = (m = Suc\ 0 \ \& \ n = Suc\ 0)$
 $\langle proof \rangle$

lemma *one-eq-mult-iff* [simp,no-atp]: $(Suc\ 0 = m * n) = (m = Suc\ 0 \ \& \ n = Suc\ 0)$
 $\langle proof \rangle$

lemma *nat-mult-eq-1-iff* [simp]: $m * n = (1::nat) \longleftrightarrow m = 1 \ \wedge \ n = 1$
 $\langle proof \rangle$

lemma *nat-1-eq-mult-iff* [simp]: $(1::nat) = m * n \longleftrightarrow m = 1 \ \wedge \ n = 1$
 $\langle proof \rangle$

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
 $\langle proof \rangle$

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
 $\langle proof \rangle$

lemma *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
 $\langle proof \rangle$

15.4 Orders on *nat*

15.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat* **where**
 $(0::nat) \leq n \longleftrightarrow True$
 $\mid Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]
lemma [*code*]: $(0::nat) \leq n \longleftrightarrow True$ $\langle proof \rangle$
lemma *le0* [*iff*]: $0 \leq (n::nat)$ $\langle proof \rangle$

definition *less-nat* **where**
 $less-eq-Suc-le: n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
 $\langle proof \rangle$

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$
 $\langle proof \rangle$

lemma *le-0-eq* [*iff*]: $(n::nat) \leq 0 \longleftrightarrow n = 0$
 $\langle proof \rangle$

lemma *not-less0* [*iff*]: $\neg n < (0::nat)$
 $\langle proof \rangle$

lemma *less-nat-zero-code* [*code*]: $n < (0::nat) \longleftrightarrow False$
 $\langle proof \rangle$

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
 $\langle proof \rangle$

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
 $\langle proof \rangle$

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$

$\langle proof \rangle$

lemma *Suc-leD*: $Suc\ m \leq n \implies m \leq n$
 $\langle proof \rangle$

lemma *less-SucI*: $m < n \implies m < Suc\ n$
 $\langle proof \rangle$

lemma *Suc-lessD*: $Suc\ m < n \implies m < n$
 $\langle proof \rangle$

instance
 $\langle proof \rangle$

end

instantiation *nat* :: *bot*
begin

definition *bot-nat* :: *nat* **where**
bot-nat = 0

instance $\langle proof \rangle$

end

15.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
 $\langle proof \rangle$

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
 $\langle proof \rangle$

15.4.3 Elimination properties

lemma *less-not-refl*: $\sim n < (n::nat)$
 $\langle proof \rangle$

lemma *less-not-refl2*: $n < m \implies m \neq (n::nat)$
 $\langle proof \rangle$

lemma *less-not-refl3*: $(s::nat) < t \implies s \neq t$
 $\langle proof \rangle$

lemma *less-irrefl-nat*: $(n::nat) < n \implies R$
 $\langle proof \rangle$

lemma *less-zeroE*: $(n::nat) < 0 \implies R$
 $\langle proof \rangle$

lemma *less-Suc-eq*: $(m < \text{Suc } n) = (m < n \mid m = n)$
 $\langle \text{proof} \rangle$

lemma *less-Suc0* [iff]: $(n < \text{Suc } 0) = (n = 0)$
 $\langle \text{proof} \rangle$

lemma *less-one* [iff, no-atp]: $(n < (1::\text{nat})) = (n = 0)$
 $\langle \text{proof} \rangle$

lemma *Suc-mono*: $m < n \implies \text{Suc } m < \text{Suc } n$
 $\langle \text{proof} \rangle$

”Less than” is antisymmetric, sort of

lemma *less-antisym*: $\llbracket \neg n < m; n < \text{Suc } m \rrbracket \implies m = n$
 $\langle \text{proof} \rangle$

lemma *nat-neq-iff*: $((m::\text{nat}) \neq n) = (m < n \mid n < m)$
 $\langle \text{proof} \rangle$

lemma *nat-less-cases*: **assumes** *major*: $(m::\text{nat}) < n \implies P \ n \ m$
and *eqCase*: $m = n \implies P \ n \ m$ **and** *lessCase*: $n < m \implies P \ n \ m$
shows $P \ n \ m$
 $\langle \text{proof} \rangle$

15.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \implies \text{Suc } m \neq n \implies \text{Suc } m < n$
 $\langle \text{proof} \rangle$

lemma *lessE*:
assumes *major*: $i < k$
and *p1*: $k = \text{Suc } i \implies P$ **and** *p2*: $\forall j. i < j \implies k = \text{Suc } j \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *less-SucE*: **assumes** *major*: $m < \text{Suc } n$
and *less*: $m < n \implies P$ **and** *eq*: $m = n \implies P$ **shows** P
 $\langle \text{proof} \rangle$

lemma *Suc-lessE*: **assumes** *major*: $\text{Suc } i < k$
and *minor*: $\forall j. i < j \implies k = \text{Suc } j \implies P$ **shows** P
 $\langle \text{proof} \rangle$

lemma *Suc-less-SucD*: $\text{Suc } m < \text{Suc } n \implies m < n$
 $\langle \text{proof} \rangle$

lemma *less-trans-Suc*:
assumes *le*: $i < j$ **shows** $j < k \implies \text{Suc } i < k$

$\langle \text{proof} \rangle$

Can be used with *less-Suc-eq* to get $n = m \vee n < m$

lemma *not-less-eq*: $\neg m < n \longleftrightarrow n < \text{Suc } m$
 $\langle \text{proof} \rangle$

lemma *not-less-eq-eq*: $\neg m \leq n \longleftrightarrow \text{Suc } n \leq m$
 $\langle \text{proof} \rangle$

Properties of ”less than or equal”

lemma *le-imp-less-Suc*: $m \leq n \implies m < \text{Suc } n$
 $\langle \text{proof} \rangle$

lemma *Suc-n-not-le-n*: $\sim \text{Suc } n \leq n$
 $\langle \text{proof} \rangle$

lemma *le-Suc-eq*: $(m \leq \text{Suc } n) = (m \leq n \mid m = \text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *le-SucE*: $m \leq \text{Suc } n \implies (m \leq n \implies R) \implies (m = \text{Suc } n \implies R)$
 $\implies R$
 $\langle \text{proof} \rangle$

lemma *Suc-leI*: $m < n \implies \text{Suc}(m) \leq n$
 $\langle \text{proof} \rangle$

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $\text{Suc } m \leq n \implies m < n$
 $\langle \text{proof} \rangle$

lemma *less-imp-le-nat*: $m < n \implies m \leq (n::\text{nat})$
 $\langle \text{proof} \rangle$

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*: $m < n \mid m = n \implies m \leq (n::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *le-eq-less-or-eq*: $(m \leq (n::\text{nat})) = (m < n \mid m = n)$
 $\langle \text{proof} \rangle$

Useful with *blast*.

lemma *eq-imp-le*: $(m::\text{nat}) = n \implies m \leq n$
 $\langle \text{proof} \rangle$

lemma *le-refl*: $n \leq (n::\text{nat})$

$\langle proof \rangle$

lemma *le-trans*: $[i \leq j; j \leq k] \implies i \leq (k::nat)$
 $\langle proof \rangle$

lemma *le-antisym*: $[m \leq n; n \leq m] \implies m = (n::nat)$
 $\langle proof \rangle$

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \ \& \ m \neq n)$
 $\langle proof \rangle$

lemma *le-neg-implies-less*: $(m::nat) \leq n \implies m \neq n \implies m < n$
 $\langle proof \rangle$

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
 $\langle proof \rangle$

lemmas *linorder-negE-nat* = *linorder-negE* [**where** 'a = nat]

lemma *le-less-Suc-eq*: $m \leq n \implies (n < Suc \ m) = (n = m)$
 $\langle proof \rangle$

lemma *not-less-less-Suc-eq*: $\sim n < m \implies (n < Suc \ m) = (n = m)$
 $\langle proof \rangle$

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*: $(!!n. f \ n == nat-rec \ c \ h \ n) \implies f \ 0 = c$
 $\langle proof \rangle$

lemma *def-nat-rec-Suc*: $(!!n. f \ n == nat-rec \ c \ h \ n) \implies f \ (Suc \ n) = h \ n \ (f \ n)$
 $\langle proof \rangle$

lemma *not0-implies-Suc*: $n \neq 0 \implies \exists m. n = Suc \ m$
 $\langle proof \rangle$

lemma *gr0-implies-Suc*: $n > 0 \implies \exists m. n = Suc \ m$
 $\langle proof \rangle$

lemma *gr-implies-not0*: **fixes** $n :: nat$ **shows** $m < n \implies n \neq 0$
 $\langle proof \rangle$

lemma *neg0-conv[iff]*: **fixes** $n :: nat$ **shows** $(n \neq 0) = (0 < n)$
 $\langle proof \rangle$

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0 \implies False) \implies 0 < n$
 $\langle proof \rangle$

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m. n = \text{Suc } m)$
 $\langle \text{proof} \rangle$

lemma *not-gr0 [iff,no-atp]*: $!!n::\text{nat}. (\sim (0 < n)) = (n = 0)$
 $\langle \text{proof} \rangle$

lemma *Suc-le-D*: $(\text{Suc } n \leq m') ==> (? m. m' = \text{Suc } m)$
 $\langle \text{proof} \rangle$

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $(m < \text{Suc } n) = (m = 0 \mid (\exists j. m = \text{Suc } j \ \& \ j < n))$
 $\langle \text{proof} \rangle$

15.4.5 *min and max*

lemma *mono-Suc*: *mono Suc*
 $\langle \text{proof} \rangle$

lemma *min-0L [simp]*: $\text{min } 0 \ n = (0::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *min-0R [simp]*: $\text{min } n \ 0 = (0::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *min-Suc-Suc [simp]*: $\text{min } (\text{Suc } m) (\text{Suc } n) = \text{Suc } (\text{min } m \ n)$
 $\langle \text{proof} \rangle$

lemma *min-Suc1*:
 $\text{min } (\text{Suc } n) \ m = (\text{case } m \text{ of } 0 => 0 \mid \text{Suc } m' => \text{Suc } (\text{min } n \ m'))$
 $\langle \text{proof} \rangle$

lemma *min-Suc2*:
 $\text{min } m \ (\text{Suc } n) = (\text{case } m \text{ of } 0 => 0 \mid \text{Suc } m' => \text{Suc } (\text{min } m' \ n))$
 $\langle \text{proof} \rangle$

lemma *max-0L [simp]*: $\text{max } 0 \ n = (n::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *max-0R [simp]*: $\text{max } n \ 0 = (n::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *max-Suc-Suc [simp]*: $\text{max } (\text{Suc } m) (\text{Suc } n) = \text{Suc } (\text{max } m \ n)$
 $\langle \text{proof} \rangle$

lemma *max-Suc1*:
 $\text{max } (\text{Suc } n) \ m = (\text{case } m \text{ of } 0 => \text{Suc } n \mid \text{Suc } m' => \text{Suc } (\text{max } n \ m'))$
 $\langle \text{proof} \rangle$

lemma *max-Suc2*:

$\max m \ (Suc\ n) = (case\ m\ of\ 0 ==> Suc\ n \mid Suc\ m' ==> Suc(\max m'\ n))$
 $\langle proof \rangle$

15.4.6 Monotonicity of Addition

lemma *Suc-pred [simp]*: $n > 0 ==> Suc\ (n - Suc\ 0) = n$
 $\langle proof \rangle$

lemma *Suc-diff-1 [simp]*: $0 < n ==> Suc\ (n - 1) = n$
 $\langle proof \rangle$

lemma *nat-add-left-cancel-le [simp]*: $(k + m \leq k + n) = (m \leq (n::nat))$
 $\langle proof \rangle$

lemma *nat-add-left-cancel-less [simp]*: $(k + m < k + n) = (m < (n::nat))$
 $\langle proof \rangle$

lemma *add-gr-0 [iff]*: $!!m::nat. (m + n > 0) = (m > 0 \mid n > 0)$
 $\langle proof \rangle$

strict, in 1st argument

lemma *add-less-mono1*: $i < j ==> i + k < j + (k::nat)$
 $\langle proof \rangle$

strict, in both arguments

lemma *add-less-mono*: $[[i < j; k < l]] ==> i + k < j + (l::nat)$
 $\langle proof \rangle$

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: $m < n ==> (\exists k. n = Suc\ (m + k))$
 $\langle proof \rangle$

strict, in 1st argument; proof is by induction on $k > 0$

lemma *mult-less-mono2*: $(i::nat) < j ==> 0 < k ==> k * i < k * j$
 $\langle proof \rangle$

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat :: linordered-semidom*
 $\langle proof \rangle$

instance *nat :: no-zero-divisors*
 $\langle proof \rangle$

lemma *nat-mult-1*: $(1::nat) * n = n$
 $\langle proof \rangle$

lemma *nat-mult-1-right*: $n * (1::nat) = n$
 $\langle proof \rangle$

15.4.7 Additional theorems about $op \leq$

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder* $\langle proof \rangle$

lemma *Least-Suc*:

$[[P\ n; \sim P\ 0\]] \implies (LEAST\ n.\ P\ n) = Suc\ (LEAST\ m.\ P\ (Suc\ m))$
 $\langle proof \rangle$

lemma *Least-Suc2*:

$[[P\ n; Q\ m; \sim P\ 0; !k.\ P\ (Suc\ k) = Q\ k]] \implies Least\ P = Suc\ (Least\ Q)$
 $\langle proof \rangle$

lemma *ex-least-nat-le*: $\neg P(0) \implies P(n::nat) \implies \exists k \leq n. (\forall i < k. \neg P\ i) \ \&\ P(k)$
 $\langle proof \rangle$

lemma *ex-least-nat-less*: $\neg P(0) \implies P(n::nat) \implies \exists k < n. (\forall i \leq k. \neg P\ i) \ \&\ P(k+1)$
 $\langle proof \rangle$

lemma *nat-less-induct*:

assumes $!!n. \forall m::nat. m < n \dashv\!\!\dashv\!\!\dashv P\ m \implies P\ n$ **shows** $P\ n$
 $\langle proof \rangle$

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 $P\ a$
 $\langle proof \rangle$

old style induction rules:

lemma *measure-induct*:

fixes $f :: 'a \Rightarrow nat$
shows $(\bigwedge x. \forall y. f\ y < f\ x \longrightarrow P\ y \implies P\ x) \implies P\ a$
 $\langle proof \rangle$

lemma *full-nat-induct*:

assumes *step*: $(!!n. (ALL\ m. Suc\ m \leq n \dashv\!\!\dashv\!\!\dashv P\ m) \implies P\ n)$
shows $P\ n$
 $\langle proof \rangle$

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 $P\ i\ j$
 $\langle proof \rangle$

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*:

$\llbracket \text{!}n::\text{nat}. \neg P\ n \implies \exists m < n. \neg P\ m \rrbracket \implies P\ n$
 $\langle \text{proof} \rangle$

lemma *infinite-descent0*[*case-names 0 smaller*]:

$\llbracket P\ 0; \text{!}n. n > 0 \implies \neg P\ n \implies (\exists m::\text{nat}. m < n \wedge \neg P\ m) \rrbracket \implies P\ n$
 $\langle \text{proof} \rangle$

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 $A0: \text{!}x. V\ x = (0::\text{nat}) \implies P\ x$
and $A1: \text{!}x. V\ x > 0 \implies \neg P\ x \implies (\exists y. V\ y < V\ x \wedge \neg P\ y)$
shows $P\ x$
 $\langle \text{proof} \rangle$

Again, without explicit base case:

lemma *infinite-descent-measure*:

assumes $\text{!}x. \neg P\ x \implies \exists y. (V::'a \Rightarrow \text{nat})\ y < V\ x \wedge \neg P\ y$ **shows** $P\ x$
 $\langle \text{proof} \rangle$

A [clumsy] way of lifting $<$ monotonicity to \leq monotonicity

lemma *less-mono-imp-le-mono*:

$\llbracket \text{!}i\ j::\text{nat}. i < j \implies f\ i < f\ j; i \leq j \rrbracket \implies f\ i \leq ((f\ j)::\text{nat})$
 $\langle \text{proof} \rangle$

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j \implies i + k \leq j + (k::\text{nat})$
 $\langle \text{proof} \rangle$

non-strict, in both arguments

lemma *add-le-mono*: $\llbracket i \leq j; k \leq l \rrbracket \implies i + k \leq j + (l::\text{nat})$

$\langle proof \rangle$

lemma *le-add2*: $n \leq ((m + n)::nat)$
 $\langle proof \rangle$

lemma *le-add1*: $n \leq ((n + m)::nat)$
 $\langle proof \rangle$

lemma *less-add-Suc1*: $i < Suc (i + m)$
 $\langle proof \rangle$

lemma *less-add-Suc2*: $i < Suc (m + i)$
 $\langle proof \rangle$

lemma *less-iff-Suc-add*: $(m < n) = (\exists k. n = Suc (m + k))$
 $\langle proof \rangle$

lemma *trans-le-add1*: $(i::nat) \leq j ==> i \leq j + m$
 $\langle proof \rangle$

lemma *trans-le-add2*: $(i::nat) \leq j ==> i \leq m + j$
 $\langle proof \rangle$

lemma *trans-less-add1*: $(i::nat) < j ==> i < j + m$
 $\langle proof \rangle$

lemma *trans-less-add2*: $(i::nat) < j ==> i < m + j$
 $\langle proof \rangle$

lemma *add-lessD1*: $i + j < (k::nat) ==> i < k$
 $\langle proof \rangle$

lemma *not-add-less1* [*iff*]: $\sim (i + j < (i::nat))$
 $\langle proof \rangle$

lemma *not-add-less2* [*iff*]: $\sim (j + i < (i::nat))$
 $\langle proof \rangle$

lemma *add-leD1*: $m + k \leq n ==> m \leq (n::nat)$
 $\langle proof \rangle$

lemma *add-leD2*: $m + k \leq n ==> k \leq (n::nat)$
 $\langle proof \rangle$

lemma *add-leE*: $(m::nat) + k \leq n ==> (m \leq n ==> k \leq n ==> R) ==> R$
 $\langle proof \rangle$

needs !!*k* for *add-ac* to work

lemma *less-add-eq-less*: $!!k::nat. k < l ==> m + l = k + n ==> m < n$

$\langle proof \rangle$

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*: $\sim m < n \implies n + (m - n) = (m::nat)$

$\langle proof \rangle$

lemma *le-add-diff-inverse* [simp]: $n \leq m \implies n + (m - n) = (m::nat)$

$\langle proof \rangle$

lemma *le-add-diff-inverse2* [simp]: $n \leq m \implies (m - n) + n = (m::nat)$

$\langle proof \rangle$

lemma *Suc-diff-le*: $n \leq m \implies \text{Suc } m - n = \text{Suc } (m - n)$

$\langle proof \rangle$

lemma *diff-less-Suc*: $m - n < \text{Suc } m$

$\langle proof \rangle$

lemma *diff-le-self* [simp]: $m - n \leq (m::nat)$

$\langle proof \rangle$

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k. n = m + k)$

$\langle proof \rangle$

lemma *less-imp-diff-less*: $(j::nat) < k \implies j - n < k$

$\langle proof \rangle$

lemma *diff-Suc-less* [simp]: $0 < n \implies n - \text{Suc } i < n$

$\langle proof \rangle$

lemma *diff-add-assoc*: $k \leq (j::nat) \implies (i + j) - k = i + (j - k)$

$\langle proof \rangle$

lemma *diff-add-assoc2*: $k \leq (j::nat) \implies (j + i) - k = (j - k) + i$

$\langle proof \rangle$

lemma *le-imp-diff-is-add*: $i \leq (j::nat) \implies (j - i = k) = (j = k + i)$

$\langle proof \rangle$

lemma *diff-is-0-eq* [simp]: $((m::nat) - n = 0) = (m \leq n)$

$\langle proof \rangle$

lemma *diff-is-0-eq'* [simp]: $m \leq n \implies (m::nat) - n = 0$

$\langle proof \rangle$

lemma *zero-less-diff* [simp]: $(0 < n - (m::nat)) = (m < n)$

$\langle proof \rangle$

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k::nat. 0 < k \ \& \ i + k = j$
 $\langle proof \rangle$

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*:
fixes $n \ m :: nat$
shows $n - m + m = max \ n \ m$
 $\langle proof \rangle$

lemma *nat-diff-split*:
 $P(a - b::nat) = ((a < b \dashrightarrow P \ 0) \ \& \ (ALL \ d. \ a = b + d \dashrightarrow P \ d))$
— elimination of $-$ on *nat*
 $\langle proof \rangle$

lemma *nat-diff-split-asm*:
 $P(a - b::nat) = (\sim (a < b \ \& \ \sim P \ 0 \mid (EX \ d. \ a = b + d \ \& \ \sim P \ d)))$
— elimination of $-$ on *nat* in assumptions
 $\langle proof \rangle$

15.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \leq (j::nat) \implies i * k \leq j * k$
 $\langle proof \rangle$

lemma *mult-le-mono2*: $i \leq (j::nat) \implies k * i \leq k * j$
 $\langle proof \rangle$

\leq monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq (j::nat) \implies k \leq l \implies i * k \leq j * l$
 $\langle proof \rangle$

lemma *mult-less-mono1*: $(i::nat) < j \implies 0 < k \implies i * k < j * k$
 $\langle proof \rangle$

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [simp]: $(0 < (m::nat) * n) = (0 < m \ \& \ 0 < n)$
 $\langle proof \rangle$

lemma *one-le-mult-iff* [simp]: $(Suc \ 0 \leq m * n) = (Suc \ 0 \leq m \ \& \ Suc \ 0 \leq n)$
 $\langle proof \rangle$

lemma *mult-less-cancel2* [simp]: $((m::nat) * k < n * k) = (0 < k \ \& \ m < n)$
 $\langle proof \rangle$

lemma *mult-less-cancel1* [simp]: $(k * (m::nat) < k * n) = (0 < k \ \& \ m < n)$

$\langle proof \rangle$

lemma *mult-le-cancel1* [simp]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
 $\langle proof \rangle$

lemma *mult-le-cancel2* [simp]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
 $\langle proof \rangle$

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
 $\langle proof \rangle$

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
 $\langle proof \rangle$

lemma *le-square*: $m \leq m * (m::nat)$
 $\langle proof \rangle$

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
 $\langle proof \rangle$

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \implies n = 1 \mid m = 0$
 $\langle proof \rangle$

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
 $(inf :: nat \Rightarrow nat \Rightarrow nat) = min$

definition
 $(sup :: nat \Rightarrow nat \Rightarrow nat) = max$

instance $\langle proof \rangle$

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)

$f \hat{\ }^n = f \circ \dots \circ f$, the n -fold composition of f

overloading

funpow == *compow* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$

begin

primrec *funpow* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$ **where**

funpow 0 $f = \text{id}$

| *funpow* (Suc n) $f = f \circ \text{funpow } n f$

end

for code generation

definition *funpow* :: $\text{nat} \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$ **where**

funpow-code-def [code-post]: *funpow* = *compow*

lemmas [code-unfold] = *funpow-code-def* [symmetric]

lemma [code]:

funpow 0 $f = \text{id}$

funpow (Suc n) $f = f \circ \text{funpow } n f$

$\langle \text{proof} \rangle$

hide-const (**open**) *funpow*

lemma *funpow-add*:

$f \hat{\ }^{(m + n)} = f \hat{\ }^m \circ f \hat{\ }^n$

$\langle \text{proof} \rangle$

lemma *funpow-swap1*:

$f ((f \hat{\ }^n) x) = (f \hat{\ }^n) (f x)$

$\langle \text{proof} \rangle$

15.6 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*

begin

primrec

of-nat :: $\text{nat} \Rightarrow 'a$

where

of-nat-0: *of-nat* 0 = 0

| *of-nat-Suc*: *of-nat* (Suc m) = 1 + *of-nat* m

lemma *of-nat-1* [simp]: *of-nat* 1 = 1

$\langle proof \rangle$

lemma *of-nat-add* [*simp*]: $of\text{-}nat\ (m + n) = of\text{-}nat\ m + of\text{-}nat\ n$
 $\langle proof \rangle$

lemma *of-nat-mult*: $of\text{-}nat\ (m * n) = of\text{-}nat\ m * of\text{-}nat\ n$
 $\langle proof \rangle$

primrec *of-nat-aux* :: $('a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$ **where**
 $of\text{-}nat\text{-}aux\ inc\ 0\ i = i$
 $| of\text{-}nat\text{-}aux\ inc\ (Suc\ n)\ i = of\text{-}nat\text{-}aux\ inc\ n\ (inc\ i)$ — tail recursive

lemma *of-nat-code*:
 $of\text{-}nat\ n = of\text{-}nat\text{-}aux\ (\lambda i. i + 1)\ n\ 0$
 $\langle proof \rangle$

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*]: $of\text{-}nat\ m = of\text{-}nat\ n \longleftrightarrow m = n$
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *no-atp*]: $0 = of\text{-}nat\ n \longleftrightarrow 0 = n$
 $\langle proof \rangle$

lemma *of-nat-eq-0-iff* [*simp*, *no-atp*]: $of\text{-}nat\ m = 0 \longleftrightarrow m = 0$
 $\langle proof \rangle$

lemma *inj-of-nat*: $inj\ of\text{-}nat$
 $\langle proof \rangle$

end

context *linordered-semidom*
begin

lemma *zero-le-imp-of-nat*: $0 \leq of\text{-}nat\ m$
 $\langle proof \rangle$

lemma *less-imp-of-nat-less*: $m < n \implies of\text{-}nat\ m < of\text{-}nat\ n$
 $\langle proof \rangle$

lemma *of-nat-less-imp-less*: $of\text{-}nat\ m < of\text{-}nat\ n \implies m < n$
 $\langle proof \rangle$

lemma *of-nat-less-iff* [simp]: $of\text{-}nat\ m < of\text{-}nat\ n \longleftrightarrow m < n$
 $\langle proof \rangle$

lemma *of-nat-le-iff* [simp]: $of\text{-}nat\ m \leq of\text{-}nat\ n \longleftrightarrow m \leq n$
 $\langle proof \rangle$

Every *linordered-semidom* has characteristic zero.

subclass *semiring-char-0*
 $\langle proof \rangle$

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [simp]: $0 \leq of\text{-}nat\ n$
 $\langle proof \rangle$

lemma *of-nat-le-0-iff* [simp, no-atp]: $of\text{-}nat\ m \leq 0 \longleftrightarrow m = 0$
 $\langle proof \rangle$

lemma *of-nat-0-less-iff* [simp]: $0 < of\text{-}nat\ n \longleftrightarrow 0 < n$
 $\langle proof \rangle$

lemma *of-nat-less-0-iff* [simp]: $\neg of\text{-}nat\ m < 0$
 $\langle proof \rangle$

end

context *ring-1*
begin

lemma *of-nat-diff*: $n \leq m \implies of\text{-}nat\ (m - n) = of\text{-}nat\ m - of\text{-}nat\ n$
 $\langle proof \rangle$

end

context *linordered-idom*
begin

lemma *abs-of-nat* [simp]: $|of\text{-}nat\ n| = of\text{-}nat\ n$
 $\langle proof \rangle$

end

lemma *of-nat-id* [simp]: $of\text{-}nat\ n = n$
 $\langle proof \rangle$

lemma *of-nat-eq-id* [simp]: $of\text{-}nat = id$
 $\langle proof \rangle$

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* $n \in \mathbb{N}$

$\langle proof \rangle$

lemma *Nats-0* [*simp*]: $0 \in \mathbb{N}$

$\langle proof \rangle$

lemma *Nats-1* [*simp*]: $1 \in \mathbb{N}$

$\langle proof \rangle$

lemma *Nats-add* [*simp*]: $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a + b \in \mathbb{N}$

$\langle proof \rangle$

lemma *Nats-mult* [*simp*]: $a \in \mathbb{N} \implies b \in \mathbb{N} \implies a * b \in \mathbb{N}$

$\langle proof \rangle$

lemma *Nats-cases* [*cases set: Nats*]:

assumes $x \in \mathbb{N}$

obtains (*of-nat*) n **where** $x = \text{of-nat } n$

$\langle proof \rangle$

lemma *Nats-induct* [*case-names of-nat, induct set: Nats*]:

$x \in \mathbb{N} \implies (\bigwedge n. P (\text{of-nat } n)) \implies P x$

$\langle proof \rangle$

end

15.8 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:

assumes 1: $t = s$ **and** 2: $u = t$

shows $u = s$

$\langle proof \rangle$

$\langle ML \rangle$

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 $f\ n \leq f\ n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-mono-less*:
 assumes *mono*: $!!n. f\ n < f(\text{Suc}\ n)$ and $n < n'$
 shows $f\ n < f\ n'$
 $\langle \text{proof} \rangle$

lemma *lift-Suc-mono-less-iff*:
 $(!!n. f\ n < f(\text{Suc}\ n)) \implies f(n) < f(m) \longleftrightarrow n < m$
 $\langle \text{proof} \rangle$

end

lemma *mono-iff-le-Suc*: $\text{mono}\ f = (\forall n. f\ n \leq f(\text{Suc}\ n))$
 $\langle \text{proof} \rangle$

lemma *mono-nat-linear-lb*:
 $(!!m\ n::\text{nat}. m < n \implies f\ m < f\ n) \implies f(m) + k \leq f(m + k)$
 $\langle \text{proof} \rangle$

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: $[| a < (b::\text{nat}); c \leq a |] \implies a - c < b - c$
 $\langle \text{proof} \rangle$

lemma *less-diff-conv*: $(i < j - k) = (i + k < (j::\text{nat}))$
 $\langle \text{proof} \rangle$

lemma *le-diff-conv*: $(j - k \leq (i::\text{nat})) = (j \leq i + k)$
 $\langle \text{proof} \rangle$

lemma *le-diff-conv2*: $k \leq j \implies (i \leq j - k) = (i + k \leq (j::\text{nat}))$
 $\langle \text{proof} \rangle$

lemma *diff-diff-cancel* [*simp*]: $i \leq (n::\text{nat}) \implies n - (n - i) = i$
 $\langle \text{proof} \rangle$

lemma *le-add-diff*: $k \leq (n::\text{nat}) \implies m \leq n + m - k$
 $\langle \text{proof} \rangle$

lemma *diff-less* [*simp*]: $!!m::\text{nat}. [| 0 < n; 0 < m |] \implies m - n < m$
 $\langle \text{proof} \rangle$

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $[| k \leq m; k \leq (n::nat) |] ==> ((m-k) - (n-k)) = (m-n)$
 $\langle proof \rangle$

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: $[| k \leq m; k \leq (n::nat) |] ==> (m-k = n-k) = (m=n)$
 $\langle proof \rangle$

lemma *less-diff-iff*: $[| k \leq m; k \leq (n::nat) |] ==> (m-k < n-k) = (m < n)$
 $\langle proof \rangle$

lemma *le-diff-iff*: $[| k \leq m; k \leq (n::nat) |] ==> (m-k \leq n-k) = (m \leq n)$
 $\langle proof \rangle$

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat) ==> (m-l) \leq (n-l)$
 $\langle proof \rangle$

lemma *diff-le-mono2*: $m \leq (n::nat) ==> (l-n) \leq (l-m)$
 $\langle proof \rangle$

lemma *diff-less-mono2*: $[| m < (n::nat); m < l |] ==> (l-n) < (l-m)$
 $\langle proof \rangle$

lemma *diffs0-imp-equal*: $!!m::nat. [| m-n = 0; n-m = 0 |] ==> m=n$
 $\langle proof \rangle$

lemma *min-diff*: $\min (m - (i::nat)) (n - i) = \min m n - i$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \rightarrow i - (j - k) = i + (k::nat) - j$
 $\langle proof \rangle$

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j ==> m - Suc (j - k) = m + k - Suc j$
 $\langle proof \rangle$

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j ==> Suc (j - k) - m = Suc j - (k + m)$
 $\langle proof \rangle$

Lemmas for ex/Factorization

lemma *one-less-mult*: $[| Suc 0 < n; Suc 0 < m |] ==> Suc 0 < m*n$
 $\langle proof \rangle$

lemma *n-less-m-mult-n*: $[| \text{Suc } 0 < n; \text{Suc } 0 < m |] \implies n < m * n$
 $\langle \text{proof} \rangle$

lemma *n-less-n-mult-m*: $[| \text{Suc } 0 < n; \text{Suc } 0 < m |] \implies n < n * m$
 $\langle \text{proof} \rangle$

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 $P \ i$
 $\langle \text{proof} \rangle$

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 $P \ i$
 $\langle \text{proof} \rangle$

lemma *zero-induct-lemma*: $P \ k \implies (!!n. P (\text{Suc } n) \implies P \ n) \implies P \ (k - i)$
 $\langle \text{proof} \rangle$

lemma *zero-induct*: $P \ k \implies (!!n. P (\text{Suc } n) \implies P \ n) \implies P \ 0$
 $\langle \text{proof} \rangle$

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*]: $\text{Suc } 0 \text{ dvd } k$
 $\langle \text{proof} \rangle$

lemma *dvd-1-iff-1* [*simp*]: $(m \text{ dvd } \text{Suc } 0) = (m = \text{Suc } 0)$
 $\langle \text{proof} \rangle$

lemma *nat-dvd-1-iff-1* [*simp*]: $m \text{ dvd } (1::\text{nat}) \longleftrightarrow m = 1$
 $\langle \text{proof} \rangle$

lemma *dvd-antisym*: $[| m \text{ dvd } n; n \text{ dvd } m |] \implies m = (n::\text{nat})$
 $\langle \text{proof} \rangle$

op dvd is a partial order

interpretation *dvd*: order op *dvd* $\lambda n\ m :: nat. n\ dvd\ m \wedge \neg\ m\ dvd\ n$
 $\langle proof \rangle$

lemma *dvd-diff-nat[simp]*: $[[\ k\ dvd\ m; k\ dvd\ n\] \implies k\ dvd\ (m - n :: nat)]$
 $\langle proof \rangle$

lemma *dvd-diffD*: $[[\ k\ dvd\ m - n; k\ dvd\ n; n \leq m\] \implies k\ dvd\ (m :: nat)]$
 $\langle proof \rangle$

lemma *dvd-diffD1*: $[[\ k\ dvd\ m - n; k\ dvd\ m; n \leq m\] \implies k\ dvd\ (n :: nat)]$
 $\langle proof \rangle$

lemma *dvd-reduce*: $(k\ dvd\ n + k) = (k\ dvd\ (n :: nat))$
 $\langle proof \rangle$

lemma *dvd-mult-cancel*: $!!k :: nat. [[\ k * m\ dvd\ k * n; 0 < k\] \implies m\ dvd\ n]$
 $\langle proof \rangle$

lemma *dvd-mult-cancel1*: $0 < m \implies (m * n\ dvd\ m) = (n = (1 :: nat))$
 $\langle proof \rangle$

lemma *dvd-mult-cancel2*: $0 < m \implies (n * m\ dvd\ m) = (n = (1 :: nat))$
 $\langle proof \rangle$

lemma *dvd-imp-le*: $[[\ k\ dvd\ n; 0 < n\] \implies k \leq (n :: nat)]$
 $\langle proof \rangle$

lemma *nat-dvd-not-less*:
fixes $m\ n :: nat$
shows $0 < m \implies m < n \implies \neg\ n\ dvd\ m$
 $\langle proof \rangle$

15.10 size of a datatype value

class *size* =
fixes $size :: 'a \Rightarrow nat$ — 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)
  ⟨proof⟩

```

Datatypes will be represented by sets of type *node*

```

types 'a item      = ('a, unit) node set
      ('a, 'b) dtree = ('a, 'b) node set

```

consts

```

  Push      :: [('b + nat), nat => ('b + nat)] => (nat => ('b + nat))

```

```

  Push-Node :: [('b + nat), ('a, 'b) node] => ('a, 'b) node

```

```

  ndepth    :: ('a, 'b) node => nat

```

```

  Atom      :: ('a + nat) => ('a, 'b) dtree

```

```

  Leaf      :: 'a => ('a, 'b) dtree

```

```

  Numb      :: nat => ('a, 'b) dtree

```

```

  Scons     :: [('a, 'b) dtree, ('a, 'b) dtree] => ('a, 'b) dtree

```

```

  In0       :: ('a, 'b) dtree => ('a, 'b) dtree

```

```

  In1       :: ('a, 'b) dtree => ('a, 'b) dtree

```

```

  Lim       :: ('b => ('a, 'b) dtree) => ('a, 'b) dtree

```

```

  ntrunc    :: [nat, ('a, 'b) dtree] => ('a, 'b) dtree

```

```

  uprod     :: [('a, 'b) dtree set, ('a, 'b) dtree set] => ('a, 'b) dtree set

```

```

  usum      :: [('a, 'b) dtree set, ('a, 'b) dtree set] => ('a, 'b) dtree set

```

```

  Split     :: [[('a, 'b) dtree, ('a, 'b) dtree] => 'c, ('a, 'b) dtree] => 'c

```

```

  Case      :: [[('a, 'b) dtree] => 'c, [('a, 'b) dtree] => 'c, ('a, 'b) dtree] => 'c

```

```

  dprod     :: [(((('a, 'b) dtree * ('a, 'b) dtree) set, (('a, 'b) dtree * ('a, 'b) dtree) set]
              => (('a, 'b) dtree * ('a, 'b) dtree) set

```

```

  dsum      :: [(((('a, 'b) dtree * ('a, 'b) dtree) set, (('a, 'b) dtree * ('a, 'b) dtree) set]
              => (('a, 'b) dtree * ('a, 'b) dtree) set

```

defs

Push-Node-def: $Push-Node == (\%n\ x.\ Abs-Node\ (apfst\ (Push\ n)\ (Rep-Node\ x)))$

Push-def: $Push == (\%b\ h.\ nat-case\ b\ h)$

Atom-def: $Atom == (\%x.\ \{Abs-Node((\%k.\ Inr\ 0,\ x))\})$
Scons-def: $Scons\ M\ N == (Push-Node\ (Inr\ 1)\ 'M)\ Un\ (Push-Node\ (Inr\ (Suc\ 1))\ 'N)$

Leaf-def: $Leaf == Atom\ o\ Inl$
Numb-def: $Numb == Atom\ o\ Inr$

In0-def: $In0(M) == Scons\ (Numb\ 0)\ M$
In1-def: $In1(M) == Scons\ (Numb\ 1)\ M$

Lim-def: $Lim\ f == Union\ \{z.\ ?\ x.\ z = Push-Node\ (Inl\ x)\ ' (f\ x)\}$

ndepth-def: $ndepth(n) == (\%(f,x).\ LEAST\ k.\ f\ k = Inr\ 0)\ (Rep-Node\ n)$
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*:

$$\begin{aligned} & \llbracket q = \text{apfst } f \, p; \quad \forall x \, y. \llbracket p = (x, y); \quad q = (f(x), y) \rrbracket \implies R \\ & \rrbracket \implies R \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *Push-inject1*: $\text{Push } i \, f = \text{Push } j \, g \implies i=j$
 $\langle \text{proof} \rangle$

lemma *Push-inject2*: $\text{Push } i \, f = \text{Push } j \, g \implies f=g$
 $\langle \text{proof} \rangle$

lemma *Push-inject*:

$$\llbracket \text{Push } i \, f = \text{Push } j \, g; \quad \llbracket i=j; \quad f=g \rrbracket \implies P \rrbracket \implies P$$

 $\langle \text{proof} \rangle$

lemma *Push-neq-K0*: $\text{Push } (\text{Inr } (\text{Suc } k)) \, f = (\%z. \text{Inr } 0) \implies P$
 $\langle \text{proof} \rangle$

lemmas *Abs-Node-inj* = *Abs-Node-inject* [THEN [2] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: $(\%k. \text{Inr } 0, a) : \text{Node}$
 $\langle \text{proof} \rangle$

lemma *Node-Push-I*: $p : \text{Node} \implies \text{apfst } (\text{Push } i) \, p : \text{Node}$
 $\langle \text{proof} \rangle$

16.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [iff]: $\text{Scons } M \, N \neq \text{Atom}(a)$
 $\langle \text{proof} \rangle$

lemmas *Atom-not-Scons* [iff] = *Scons-not-Atom* [THEN *not-sym*, *standard*]

lemma *inj-Atom*: $\text{inj}(\text{Atom})$
 $\langle \text{proof} \rangle$

lemmas *Atom-inject* = *inj-Atom* [THEN *injD*, *standard*]

lemma *Atom-Atom-eq* [iff]: $(Atom(a)=Atom(b)) = (a=b)$
 $\langle proof \rangle$

lemma *inj-Leaf*: $inj(Leaf)$
 $\langle proof \rangle$

lemmas *Leaf-inject* [dest!] = *inj-Leaf* [THEN injD, standard]

lemma *inj-Numb*: $inj(Numb)$
 $\langle proof \rangle$

lemmas *Numb-inject* [dest!] = *inj-Numb* [THEN injD, standard]

lemma *Push-Node-inject*:

$$[[Push-Node\ i\ m = Push-Node\ j\ n; \ [[\ i=j; \ m=n \]]] ==> P$$

$$[[]] ==> P$$
 $\langle proof \rangle$

lemma *Scons-inject-lemma1*: $Scons\ M\ N <= Scons\ M'\ N' ==> M <= M'$
 $\langle proof \rangle$

lemma *Scons-inject-lemma2*: $Scons\ M\ N <= Scons\ M'\ N' ==> N <= N'$
 $\langle proof \rangle$

lemma *Scons-inject1*: $Scons\ M\ N = Scons\ M'\ N' ==> M = M'$
 $\langle proof \rangle$

lemma *Scons-inject2*: $Scons\ M\ N = Scons\ M'\ N' ==> N = N'$
 $\langle proof \rangle$

lemma *Scons-inject*:

$$[[Scons\ M\ N = Scons\ M'\ N'; \ [[\ M=M'; \ N=N' \]]] ==> P \]] ==> P$$
 $\langle proof \rangle$

lemma *Scons-Scons-eq* [iff]: $(Scons\ M\ N = Scons\ M'\ N') = (M=M' \ \& \ N=N')$
 $\langle proof \rangle$

lemma *Scons-not-Leaf* [iff]: $Scons\ M\ N \neq Leaf(a)$
 $\langle proof \rangle$

lemmas *Leaf-not-Scons* [iff] = *Scons-not-Leaf* [THEN not-sym, standard]

lemma *Scons-not-Numb* [iff]: *Scons* *M N* \neq *Numb*(*k*)
 <proof>

lemmas *Numb-not-Scons* [iff] = *Scons-not-Numb* [THEN not-sym, standard]

lemma *Leaf-not-Numb* [iff]: *Leaf*(*a*) \neq *Numb*(*k*)
 <proof>

lemmas *Numb-not-Leaf* [iff] = *Leaf-not-Numb* [THEN not-sym, standard]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr* 0, *x*)) = 0
 <proof>

lemma *ndepth-Push-Node-aux*:
nat-case (*Inr* (*Suc* *i*)) *f k* = *Inr* 0 \dashrightarrow *Suc*(*LEAST* *x*. *f x* = *Inr* 0) \leq *k*
 <proof>

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc* *i*)) *n*) = *Suc*(*ndepth*(*n*))
 <proof>

lemma *ntrunc-0* [simp]: *ntrunc* 0 *M* = {}
 <proof>

lemma *ntrunc-Atom* [simp]: *ntrunc* (*Suc* *k*) (*Atom* *a*) = *Atom*(*a*)
 <proof>

lemma *ntrunc-Leaf* [simp]: *ntrunc* (*Suc* *k*) (*Leaf* *a*) = *Leaf*(*a*)
 <proof>

lemma *ntrunc-Numb* [simp]: *ntrunc* (*Suc* *k*) (*Numb* *i*) = *Numb*(*i*)
 <proof>

lemma *ntrunc-Scons* [simp]:
ntrunc (*Suc* *k*) (*Scons* *M N*) = *Scons* (*ntrunc* *k M*) (*ntrunc* *k N*)

$\langle proof \rangle$

lemma *ntrunc-one-In0* [simp]: $ntrunc (Suc\ 0) (In0\ M) = \{\}$
 $\langle proof \rangle$

lemma *ntrunc-In0* [simp]: $ntrunc (Suc(Suc\ k)) (In0\ M) = In0\ (ntrunc (Suc\ k)\ M)$
 $\langle proof \rangle$

lemma *ntrunc-one-In1* [simp]: $ntrunc (Suc\ 0) (In1\ M) = \{\}$
 $\langle proof \rangle$

lemma *ntrunc-In1* [simp]: $ntrunc (Suc(Suc\ k)) (In1\ M) = In1\ (ntrunc (Suc\ k)\ M)$
 $\langle proof \rangle$

16.3 Set Constructions

lemma *uprodI* [intro!]: $[\![\ M:A;\ N:B\]\!] ==> Scons\ M\ N : uprod\ A\ B$
 $\langle proof \rangle$

lemma *uprodE* [elim!]:
 $[\![\ c : uprod\ A\ B;$
 $\quad !!x\ y. [\![\ x:A;\ y:B;\ c = Scons\ x\ y\]\!] ==> P$
 $\quad]\!] ==> P$
 $\langle proof \rangle$

lemma *uprodE2*: $[\![\ Scons\ M\ N : uprod\ A\ B;\ [\![\ M:A;\ N:B\]\!] ==> P\]\!] ==> P$
 $\langle proof \rangle$

lemma *usum-In0I* [intro]: $M:A ==> In0(M) : usum\ A\ B$
 $\langle proof \rangle$

lemma *usum-In1I* [intro]: $N:B ==> In1(N) : usum\ A\ B$
 $\langle proof \rangle$

lemma *usumE* [elim!]:
 $[\![\ u : usum\ A\ B;$
 $\quad !!x. [\![\ x:A;\ u=In0(x)\]\!] ==> P;$

$$\begin{aligned} & \text{!!}y. [\text{!} y:B; \ u=In1(y) \text{!}] ==> P \\ & [\text{!}] ==> P \\ & \langle proof \rangle \end{aligned}$$

lemma *In0-not-In1* [iff]: $In0(M) \neq In1(N)$
 $\langle proof \rangle$

lemmas *In1-not-In0* [iff] = *In0-not-In1* [THEN not-sym, standard]

lemma *In0-inject*: $In0(M) = In0(N) ==> M=N$
 $\langle proof \rangle$

lemma *In1-inject*: $In1(M) = In1(N) ==> M=N$
 $\langle proof \rangle$

lemma *In0-eq* [iff]: $(In0\ M = In0\ N) = (M=N)$
 $\langle proof \rangle$

lemma *In1-eq* [iff]: $(In1\ M = In1\ N) = (M=N)$
 $\langle proof \rangle$

lemma *inj-In0*: $inj\ In0$
 $\langle proof \rangle$

lemma *inj-In1*: $inj\ In1$
 $\langle proof \rangle$

lemma *Lim-inject*: $Lim\ f = Lim\ g ==> f = g$
 $\langle proof \rangle$

lemma *ntrunc-subsetI*: $ntrunc\ k\ M \leq M$
 $\langle proof \rangle$

lemma *ntrunc-subsetD*: $(\text{!!}k. ntrunc\ k\ M \leq N) ==> M \leq N$
 $\langle proof \rangle$

lemma *ntrunc-equality*: $(\text{!!}k. ntrunc\ k\ M = ntrunc\ k\ N) ==> M=N$
 $\langle proof \rangle$

lemma *ntrunc-o-equality*:

$\llbracket \text{!!}k. (\text{ntrunc}(k) \circ h1) = (\text{ntrunc}(k) \circ h2) \rrbracket \implies h1=h2$
 $\langle \text{proof} \rangle$

lemma *uprod-mono*: $\llbracket A \leq A'; B \leq B' \rrbracket \implies \text{uprod } A \ B \leq \text{uprod } A' \ B'$
 $\langle \text{proof} \rangle$

lemma *usum-mono*: $\llbracket A \leq A'; B \leq B' \rrbracket \implies \text{usum } A \ B \leq \text{usum } A' \ B'$
 $\langle \text{proof} \rangle$

lemma *Scons-mono*: $\llbracket M \leq M'; N \leq N' \rrbracket \implies \text{Scons } M \ N \leq \text{Scons } M' \ N'$
 $\langle \text{proof} \rangle$

lemma *In0-mono*: $M \leq N \implies \text{In0}(M) \leq \text{In0}(N)$
 $\langle \text{proof} \rangle$

lemma *In1-mono*: $M \leq N \implies \text{In1}(M) \leq \text{In1}(N)$
 $\langle \text{proof} \rangle$

lemma *Split [simp]*: $\text{Split } c \ (\text{Scons } M \ N) = c \ M \ N$
 $\langle \text{proof} \rangle$

lemma *Case-In0 [simp]*: $\text{Case } c \ d \ (\text{In0 } M) = c(M)$
 $\langle \text{proof} \rangle$

lemma *Case-In1 [simp]*: $\text{Case } c \ d \ (\text{In1 } N) = d(N)$
 $\langle \text{proof} \rangle$

lemma *ntrunc-UN1*: $\text{ntrunc } k \ (\text{UN } x. f(x)) = (\text{UN } x. \text{ntrunc } k \ (f \ x))$
 $\langle \text{proof} \rangle$

lemma *Scons-UN1-x*: $\text{Scons } (\text{UN } x. f \ x) \ M = (\text{UN } x. \text{Scons } (f \ x) \ M)$
 $\langle \text{proof} \rangle$

lemma *Scons-UN1-y*: $\text{Scons } M \ (\text{UN } x. f \ x) = (\text{UN } x. \text{Scons } M \ (f \ x))$
 $\langle \text{proof} \rangle$

lemma *In0-UN1*: $\text{In0}(\text{UN } x. f(x)) = (\text{UN } x. \text{In0}(f(x)))$
 $\langle \text{proof} \rangle$

lemma *In1-UN1*: $In1 (UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
 $\langle proof \rangle$

lemma *dprodI* [*intro!*]:
 $[[(M,M'):r; (N,N'):s]] ==> (Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
 $\langle proof \rangle$

lemma *dprodE* [*elim!*]:
 $[[c : dprod\ r\ s;$
 $!!x\ y\ x'\ y'. [[(x,x') : r; (y,y') : s;$
 $c = (Scons\ x\ y,\ Scons\ x'\ y')]] ==> P$
 $]] ==> P$
 $\langle proof \rangle$

lemma *dsum-In0I* [*intro*]: $(M,M'):r ==> (In0(M), In0(M')) : dsum\ r\ s$
 $\langle proof \rangle$

lemma *dsum-In1I* [*intro*]: $(N,N'):s ==> (In1(N), In1(N')) : dsum\ r\ s$
 $\langle proof \rangle$

lemma *dsumE* [*elim!*]:
 $[[w : dsum\ r\ s;$
 $!!x\ x'. [[(x,x') : r; w = (In0(x), In0(x'))]] ==> P;$
 $!!y\ y'. [[(y,y') : s; w = (In1(y), In1(y'))]] ==> P$
 $]] ==> P$
 $\langle proof \rangle$

lemma *dprod-mono*: $[[r<=r'; s<=s']] ==> dprod\ r\ s <= dprod\ r'\ s'$
 $\langle proof \rangle$

lemma *dsum-mono*: $[[r<=r'; s<=s']] ==> dsum\ r\ s <= dsum\ r'\ s'$
 $\langle proof \rangle$

lemma *dprod-Sigma*: $(dprod\ (A\ <*>\ B)\ (C\ <*>\ D)) <= (uprod\ A\ C)\ <*>\ (uprod\ B\ D)$

<proof>

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*)))
<proof>

lemma *dsum-Sigma*: (*dsum* (*A <*> B*) (*C <*> D*)) <= (*usum A C*) <*> (*usum B D*)
<proof>

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*

<ML>

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-update f rec*) = *alpha rec* for distinct fields *alpha* and *beta* of some record *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

$abst :: ('a, 'b, 'c) \text{ tuple-isomorphism} \Rightarrow 'b \times 'c \Rightarrow 'a$ **where**
 $abst \text{ (Tuple-Isomorphism } r \text{ } a) = a$

definition

$iso\text{-tuple}\text{-fst} :: ('a, 'b, 'c) \text{ tuple-isomorphism} \Rightarrow 'a \Rightarrow 'b$ **where**
 $iso\text{-tuple}\text{-fst} \text{ isom} = fst \circ repr \text{ isom}$

definition

$iso\text{-tuple}\text{-snd} :: ('a, 'b, 'c) \text{ tuple-isomorphism} \Rightarrow 'a \Rightarrow 'c$ **where**
 $iso\text{-tuple}\text{-snd} \text{ isom} = snd \circ repr \text{ isom}$

definition

$iso\text{-tuple}\text{-fst}\text{-update} ::$
 $('a, 'b, 'c) \text{ tuple-isomorphism} \Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)$ **where**
 $iso\text{-tuple}\text{-fst}\text{-update} \text{ isom } f = abst \text{ isom} \circ apfst \text{ } f \circ repr \text{ isom}$

definition

$iso\text{-tuple}\text{-snd}\text{-update} ::$
 $('a, 'b, 'c) \text{ tuple-isomorphism} \Rightarrow ('c \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'a)$ **where**
 $iso\text{-tuple}\text{-snd}\text{-update} \text{ isom } f = abst \text{ isom} \circ apsnd \text{ } f \circ repr \text{ isom}$

definition

$iso\text{-tuple}\text{-cons} ::$
 $('a, 'b, 'c) \text{ tuple-isomorphism} \Rightarrow 'b \Rightarrow 'c \Rightarrow 'a$ **where**
 $iso\text{-tuple}\text{-cons} \text{ isom} = \text{curry } (abst \text{ isom})$

17.3 Logical infrastructure for records**definition**

$iso\text{-tuple}\text{-surjective}\text{-proof}\text{-assist} :: 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$ **where**
 $iso\text{-tuple}\text{-surjective}\text{-proof}\text{-assist} \text{ } x \text{ } y \text{ } f \iff f \text{ } x = y$

definition

$iso\text{-tuple}\text{-update}\text{-accessor}\text{-cong}\text{-assist} ::$
 $(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$ **where**
 $iso\text{-tuple}\text{-update}\text{-accessor}\text{-cong}\text{-assist} \text{ } upd \text{ } acc \iff$
 $(\forall f \text{ } v. \text{ } upd \text{ } (\lambda x. \text{ } f \text{ } (acc \text{ } v)) \text{ } v = upd \text{ } f \text{ } v) \wedge (\forall v. \text{ } upd \text{ } id \text{ } v = v)$

definition

$iso\text{-tuple}\text{-update}\text{-accessor}\text{-eq}\text{-assist} ::$
 $(('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow ('b \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$
where
 $iso\text{-tuple}\text{-update}\text{-accessor}\text{-eq}\text{-assist} \text{ } upd \text{ } acc \text{ } v \text{ } f \text{ } v' \text{ } x \iff$
 $upd \text{ } f \text{ } v = v' \wedge acc \text{ } v = x \wedge iso\text{-tuple}\text{-update}\text{-accessor}\text{-cong}\text{-assist} \text{ } upd \text{ } acc$

lemma *update-accessor-congruence-foldE*:

assumes *uac*: $iso\text{-tuple}\text{-update}\text{-accessor}\text{-cong}\text{-assist} \text{ } upd \text{ } acc$
and *r*: $r = r'$ **and** *v*: $acc \text{ } r' = v'$
and *f*: $\bigwedge v. v' = v \implies f \text{ } v = f' \text{ } v$

shows $\text{upd } f \ r = \text{upd } f' \ r'$
 $\langle \text{proof} \rangle$

lemma *update-accessor-congruence-unfoldE*:
 $\text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc} \implies$
 $r = r' \implies \text{acc } r' = v' \implies (\bigwedge v. v = v' \implies f \ v = f' \ v) \implies$
 $\text{upd } f \ r = \text{upd } f' \ r'$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-accessor-cong-assist-id*:
 $\text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc} \implies \text{upd } \text{id} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *update-accessor-noopE*:
assumes $\text{uac}: \text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc}$
and $\text{acc}: f \ (\text{acc } x) = \text{acc } x$
shows $\text{upd } f \ x = x$
 $\langle \text{proof} \rangle$

lemma *update-accessor-noop-compE*:
assumes $\text{uac}: \text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc}$
and $\text{acc}: f \ (\text{acc } x) = \text{acc } x$
shows $\text{upd } (g \circ f) \ x = \text{upd } g \ x$
 $\langle \text{proof} \rangle$

lemma *update-accessor-cong-assist-idI*:
 $\text{iso-tuple-update-accessor-cong-assist } \text{id } \text{id}$
 $\langle \text{proof} \rangle$

lemma *update-accessor-cong-assist-triv*:
 $\text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc} \implies$
 $\text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc}$
 $\langle \text{proof} \rangle$

lemma *update-accessor-accessor-eqE*:
 $\text{iso-tuple-update-accessor-eq-assist } \text{upd } \text{acc } v \ f \ v' \ x \implies \text{acc } v = x$
 $\langle \text{proof} \rangle$

lemma *update-accessor-updator-eqE*:
 $\text{iso-tuple-update-accessor-eq-assist } \text{upd } \text{acc } v \ f \ v' \ x \implies \text{upd } f \ v = v'$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-accessor-eq-assist-idI*:
 $v' = f \ v \implies \text{iso-tuple-update-accessor-eq-assist } \text{id } \text{id } v \ f \ v' \ v$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-accessor-eq-assist-triv*:
 $\text{iso-tuple-update-accessor-eq-assist } \text{upd } \text{acc } v \ f \ v' \ x \implies$
 $\text{iso-tuple-update-accessor-eq-assist } \text{upd } \text{acc } v \ f \ v' \ x$

$\langle \text{proof} \rangle$

lemma *iso-tuple-update-accessor-cong-from-eq*:
 $\text{iso-tuple-update-accessor-eq-assist } \text{upd } \text{acc } v \text{ } f \text{ } v' \text{ } x \implies$
 $\text{iso-tuple-update-accessor-cong-assist } \text{upd } \text{acc}$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-surjective-proof-assistI*:
 $f \text{ } x = y \implies \text{iso-tuple-surjective-proof-assist } x \text{ } y \text{ } f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-surjective-proof-assist-idE*:
 $\text{iso-tuple-surjective-proof-assist } x \text{ } y \text{ } \text{id} \implies x = y$
 $\langle \text{proof} \rangle$

locale *isomorphic-tuple* =
fixes *isom* :: ('a, 'b, 'c) *tuple-isomorphism*
assumes *repr-inv*: $\bigwedge x. \text{abst } \text{isom } (\text{repr } \text{isom } x) = x$
and *abst-inv*: $\bigwedge y. \text{repr } \text{isom } (\text{abst } \text{isom } y) = y$
begin

lemma *repr-inj*: $\text{repr } \text{isom } x = \text{repr } \text{isom } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *abst-inj*: $\text{abst } \text{isom } x = \text{abst } \text{isom } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemmas *simps* = *Let-def repr-inv abst-inv repr-inj abst-inj*

lemma *iso-tuple-access-update-fst-fst*:
 $f \text{ } o \text{ } h \text{ } g = j \text{ } o \text{ } f \implies$
 $(f \text{ } o \text{ } \text{iso-tuple-fst } \text{isom}) \text{ } o \text{ } (\text{iso-tuple-fst-update } \text{isom } o \text{ } h) \text{ } g =$
 $j \text{ } o \text{ } (f \text{ } o \text{ } \text{iso-tuple-fst } \text{isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-access-update-snd-snd*:
 $f \text{ } o \text{ } h \text{ } g = j \text{ } o \text{ } f \implies$
 $(f \text{ } o \text{ } \text{iso-tuple-snd } \text{isom}) \text{ } o \text{ } (\text{iso-tuple-snd-update } \text{isom } o \text{ } h) \text{ } g =$
 $j \text{ } o \text{ } (f \text{ } o \text{ } \text{iso-tuple-snd } \text{isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-access-update-fst-snd*:
 $(f \text{ } o \text{ } \text{iso-tuple-fst } \text{isom}) \text{ } o \text{ } (\text{iso-tuple-snd-update } \text{isom } o \text{ } h) \text{ } g =$
 $\text{id } o \text{ } (f \text{ } o \text{ } \text{iso-tuple-fst } \text{isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-access-update-snd-fst*:
 $(f \text{ } o \text{ } \text{iso-tuple-snd } \text{isom}) \text{ } o \text{ } (\text{iso-tuple-fst-update } \text{isom } o \text{ } h) \text{ } g =$
 $\text{id } o \text{ } (f \text{ } o \text{ } \text{iso-tuple-snd } \text{isom})$

$\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-fst-fst:*

$h \circ f \circ j \circ g = j \circ g \circ h \circ f \implies$
 $(\text{iso-tuple-fst-update isom } o \ h) \circ f \circ (\text{iso-tuple-fst-update isom } o \ j) \circ g =$
 $(\text{iso-tuple-fst-update isom } o \ j) \circ g \circ (\text{iso-tuple-fst-update isom } o \ h) \circ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-snd-snd:*

$h \circ f \circ j \circ g = j \circ g \circ h \circ f \implies$
 $(\text{iso-tuple-snd-update isom } o \ h) \circ f \circ (\text{iso-tuple-snd-update isom } o \ j) \circ g =$
 $(\text{iso-tuple-snd-update isom } o \ j) \circ g \circ (\text{iso-tuple-snd-update isom } o \ h) \circ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-fst-snd:*

$(\text{iso-tuple-snd-update isom } o \ h) \circ f \circ (\text{iso-tuple-fst-update isom } o \ j) \circ g =$
 $(\text{iso-tuple-fst-update isom } o \ j) \circ g \circ (\text{iso-tuple-snd-update isom } o \ h) \circ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-swap-snd-fst:*

$(\text{iso-tuple-fst-update isom } o \ h) \circ f \circ (\text{iso-tuple-snd-update isom } o \ j) \circ g =$
 $(\text{iso-tuple-snd-update isom } o \ j) \circ g \circ (\text{iso-tuple-fst-update isom } o \ h) \circ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-compose-fst-fst:*

$h \circ f \circ j \circ g = k \circ (f \circ g) \implies$
 $(\text{iso-tuple-fst-update isom } o \ h) \circ f \circ (\text{iso-tuple-fst-update isom } o \ j) \circ g =$
 $(\text{iso-tuple-fst-update isom } o \ k) \circ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-update-compose-snd-snd:*

$h \circ f \circ j \circ g = k \circ (f \circ g) \implies$
 $(\text{iso-tuple-snd-update isom } o \ h) \circ f \circ (\text{iso-tuple-snd-update isom } o \ j) \circ g =$
 $(\text{iso-tuple-snd-update isom } o \ k) \circ (f \circ g)$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-surjective-proof-assist-step:*

$\text{iso-tuple-surjective-proof-assist } v \ a \ (\text{iso-tuple-fst isom } o \ f) \implies$
 $\text{iso-tuple-surjective-proof-assist } v \ b \ (\text{iso-tuple-snd isom } o \ f) \implies$
 $\text{iso-tuple-surjective-proof-assist } v \ (\text{iso-tuple-cons isom } a \ b) \ f$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-fst-update-accessor-cong-assist:*

assumes *iso-tuple-update-accessor-cong-assist* $f \ g$
shows *iso-tuple-update-accessor-cong-assist*
 $(\text{iso-tuple-fst-update isom } o \ f) \ (g \circ \text{iso-tuple-fst isom})$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-snd-update-accessor-cong-assist:*

assumes *iso-tuple-update-accessor-cong-assist* $f\ g$
shows *iso-tuple-update-accessor-cong-assist*
 $(\text{iso-tuple-snd-update isom } o\ f)\ (g\ o\ \text{iso-tuple-snd isom})$
 $\langle \text{proof} \rangle$

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*
 $(\text{iso-tuple-fst-update isom } o\ f)\ (g\ o\ \text{iso-tuple-fst isom})$
 $(\text{iso-tuple-cons isom } a\ b)\ u\ (\text{iso-tuple-cons isom } a'\ b)\ v$
 $\langle \text{proof} \rangle$

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*
 $(\text{iso-tuple-snd-update isom } o\ f)\ (g\ o\ \text{iso-tuple-snd isom})$
 $(\text{iso-tuple-cons isom } a\ b)\ u\ (\text{iso-tuple-cons isom } a\ b')\ v$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-cons-conj-eqI*:
 $a = c \wedge b = d \wedge P \longleftrightarrow Q \implies$
 $\text{iso-tuple-cons isom } a\ b = \text{iso-tuple-cons isom } c\ d \wedge P \longleftrightarrow Q$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

definition

tuple-iso-tuple \equiv *Tuple-Isomorphism* *id id*

lemma *tuple-iso-tuple*:

isomorphic-tuple tuple-iso-tuple
 $\langle \text{proof} \rangle$

lemma *refl-conj-eq*: $Q = R \implies P \wedge Q \longleftrightarrow P \wedge R$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-UNIV-I*: $x \in \text{UNIV} \equiv \text{True}$
 $\langle \text{proof} \rangle$

lemma *iso-tuple-True-simp*: $(\text{True} \implies \text{PROP } P) \equiv \text{PROP } P$
 $\langle \text{proof} \rangle$

lemma *prop-subst*: $s = t \implies \text{PROP } P \ t \implies \text{PROP } P \ s$
 $\langle \text{proof} \rangle$

lemma *K-record-comp*: $(\lambda x. \ c) \circ f = (\lambda x. \ c)$
 $\langle \text{proof} \rangle$

lemma *o-eq-dest-lhs*: $a \ o \ b = c \implies a \ (b \ v) = c \ v$
 $\langle \text{proof} \rangle$

lemma *o-eq-id-dest*: $a \ o \ b = \text{id} \ o \ c \implies a \ (b \ v) = c \ v$
 $\langle \text{proof} \rangle$

17.4 Concrete record syntax

nonterminals

ident field-type field-types field fields field-update field-updates

syntax

<i>-constify</i>	$:: \text{ident} \Rightarrow \text{ident}$	$(-)$
<i>-constify</i>	$:: \text{longid} \Rightarrow \text{ident}$	$(-)$
<i>-field-type</i>	$:: \text{ident} \Rightarrow \text{type} \Rightarrow \text{field-type}$	$((2- \ ::/ -))$
	$:: \text{field-type} \Rightarrow \text{field-types}$	$(-)$
<i>-field-types</i>	$:: \text{field-type} \Rightarrow \text{field-types} \Rightarrow \text{field-types}$	$(-, / -)$
<i>-record-type</i>	$:: \text{field-types} \Rightarrow \text{type}$	$((3'(- ')))$
<i>-record-type-scheme</i>	$:: \text{field-types} \Rightarrow \text{type} \Rightarrow \text{type}$	$((3'(-, / (2... \ ::/ -) ')))$
<i>-field</i>	$:: \text{ident} \Rightarrow 'a \Rightarrow \text{field}$	$((2- =/ -))$
	$:: \text{field} \Rightarrow \text{fields}$	$(-)$
<i>-fields</i>	$:: \text{field} \Rightarrow \text{fields} \Rightarrow \text{fields}$	$(-, / -)$
<i>-record</i>	$:: \text{fields} \Rightarrow 'a$	$((3'(- ')))$

```

-record-scheme      :: fields => 'a => 'a          ((3'(| -, (2... =/ -) |'))
-field-update       :: ident => 'a => field-update  ((2- :=/ -))
                   :: field-update => field-updates (-)
-field-updates      :: field-update => field-updates => field-updates (-,/ -)
-record-update      :: 'a => field-updates => 'b     (-/(3'(| - |')) [900, 0] 900)

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

<ML>

```

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* $\langle \text{proof} \rangle$

lemma *power-one* [*simp*]:
 $1 \wedge n = 1$
 $\langle \text{proof} \rangle$

lemma *power-one-right* [*simp*]:
 $a \wedge 1 = a$
 $\langle \text{proof} \rangle$

lemma *power-commutes*:
 $a \wedge n * a = a * a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-Suc2*:
 $a \wedge \text{Suc } n = a \wedge n * a$
 $\langle \text{proof} \rangle$

lemma *power-add*:
 $a \wedge (m + n) = a \wedge m * a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-mult*:
 $a \wedge (m * n) = (a \wedge m) \wedge n$
 $\langle \text{proof} \rangle$

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib*:
 $(a * b) \wedge n = (a \wedge n) * (b \wedge n)$
 $\langle \text{proof} \rangle$

end

context *semiring-1*
begin

lemma *of-nat-power*:
 $\text{of-nat } (m \wedge n) = \text{of-nat } m \wedge n$
 $\langle \text{proof} \rangle$

end

context *comm-semiring-1*
begin

The divides relation

lemma *le-imp-power-dvd*:
 assumes $m \leq n$ **shows** $a \wedge m \text{ dvd } a \wedge n$
 $\langle \text{proof} \rangle$

lemma *power-le-dvd*:
 $a \wedge n \text{ dvd } b \implies m \leq n \implies a \wedge m \text{ dvd } b$
 $\langle \text{proof} \rangle$

lemma *dvd-power-same*:
 $x \text{ dvd } y \implies x \wedge n \text{ dvd } y \wedge n$
 $\langle \text{proof} \rangle$

lemma *dvd-power-le*:
 $x \text{ dvd } y \implies m \geq n \implies x \wedge n \text{ dvd } y \wedge m$
 $\langle \text{proof} \rangle$

lemma *dvd-power [simp]*:
 assumes $n > (0::\text{nat}) \vee x = 1$
 shows $x \text{ dvd } (x \wedge n)$
 $\langle \text{proof} \rangle$

end

context *ring-1*
begin

lemma *power-minus*:
 $(- a) \wedge n = (- 1) \wedge n * a \wedge n$
 $\langle \text{proof} \rangle$

end

context *linordered-semidom*
begin

lemma *zero-less-power [simp]*:
 $0 < a \implies 0 < a \wedge n$
 $\langle \text{proof} \rangle$

lemma *zero-le-power [simp]*:
 $0 \leq a \implies 0 \leq a \wedge n$
 $\langle \text{proof} \rangle$

lemma *one-le-power [simp]*:
 $1 \leq a \implies 1 \leq a \wedge n$

$\langle \text{proof} \rangle$

lemma *power-gt1-lemma*:

assumes *gt1*: $1 < a$

shows $1 < a * a ^ n$

$\langle \text{proof} \rangle$

lemma *power-gt1*:

$1 < a \implies 1 < a ^ \text{Suc } n$

$\langle \text{proof} \rangle$

lemma *one-less-power* [*simp*]:

$1 < a \implies 0 < n \implies 1 < a ^ n$

$\langle \text{proof} \rangle$

lemma *power-le-imp-le-exp*:

assumes *gt1*: $1 < a$

shows $a ^ m \leq a ^ n \implies m \leq n$

$\langle \text{proof} \rangle$

Surely we can strengthen this? It holds for $0 < a < 1$ too.

lemma *power-inject-exp* [*simp*]:

$1 < a \implies a ^ m = a ^ n \longleftrightarrow m = n$

$\langle \text{proof} \rangle$

Can relax the first premise to $(0::'a) < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*:

$1 < a \implies a ^ m < a ^ n \implies m < n$

$\langle \text{proof} \rangle$

lemma *power-mono*:

$a \leq b \implies 0 \leq a \implies a ^ n \leq b ^ n$

$\langle \text{proof} \rangle$

lemma *power-strict-mono* [*rule-format*]:

$a < b \implies 0 \leq a \implies 0 < n \longrightarrow a ^ n < b ^ n$

$\langle \text{proof} \rangle$

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:

$0 < a \implies a < 1 \implies a * a ^ n < a ^ n$

$\langle \text{proof} \rangle$

lemma *power-strict-decreasing* [*rule-format*]:

$n < N \implies 0 < a \implies a < 1 \longrightarrow a ^ N < a ^ n$

$\langle \text{proof} \rangle$

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⟩

lemma *power-Suc-less-one*:

$0 < a \implies a < 1 \implies a \wedge \text{Suc } n < 1$
 ⟨proof⟩

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⟩

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:

$1 < a \implies a \wedge n < a * a \wedge n$
 ⟨proof⟩

lemma *power-strict-increasing* [rule-format]:

$n < N \implies 1 < a \longrightarrow a \wedge n < a \wedge N$
 ⟨proof⟩

lemma *power-increasing-iff* [simp]:

$1 < b \implies b \wedge x \leq b \wedge y \longleftrightarrow x \leq y$
 ⟨proof⟩

lemma *power-strict-increasing-iff* [simp]:

$1 < b \implies b \wedge x < b \wedge y \longleftrightarrow x < y$
 ⟨proof⟩

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⟩

lemma *power-less-imp-less-base*:

assumes *less*: $a \wedge n < b \wedge n$

assumes *nonneg*: $0 \leq b$

shows $a < b$

⟨proof⟩

lemma *power-inject-base*:

$a \wedge \text{Suc } n = b \wedge \text{Suc } n \implies 0 \leq a \implies 0 \leq b \implies a = b$

⟨proof⟩

lemma *power-eq-imp-eq-base*:

$a \wedge n = b \wedge n \implies 0 \leq a \implies 0 \leq b \implies 0 < n \implies a = b$

⟨proof⟩

end

context *linordered-idom*
begin

lemma *power-abs*:
 $abs (a ^ n) = abs a ^ n$
 $\langle proof \rangle$

lemma *abs-power-minus* [*simp*]:
 $abs ((-a) ^ n) = abs (a ^ n)$
 $\langle proof \rangle$

lemma *zero-less-power-abs-iff* [*simp, no-atp*]:
 $0 < abs a ^ n \longleftrightarrow a \neq 0 \vee n = 0$
 $\langle proof \rangle$

lemma *zero-le-power-abs* [*simp*]:
 $0 \leq abs a ^ n$
 $\langle proof \rangle$

end

context *ring-1-no-zero-divisors*
begin

lemma *field-power-not-zero*:
 $a \neq 0 \implies a ^ n \neq 0$
 $\langle proof \rangle$

end

context *division-ring*
begin

FIXME reorient or rename to *nonzero-inverse-power*

lemma *nonzero-power-inverse*:
 $a \neq 0 \implies inverse (a ^ n) = (inverse a) ^ n$
 $\langle proof \rangle$

end

context *field*
begin

lemma *nonzero-power-divide*:
 $b \neq 0 \implies (a / b) ^ n = a ^ n / b ^ n$
 $\langle proof \rangle$

end

lemma *power-0-Suc* [simp]:
 $(0 :: 'a :: \{\text{power}, \text{semiring-0}\}) \wedge \text{Suc } n = 0$
 ⟨proof⟩

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*:
 $0 \wedge n = (\text{if } n = 0 \text{ then } 1 \text{ else } (0 :: 'a :: \{\text{power}, \text{semiring-0}\}))$
 ⟨proof⟩

lemma *power-eq-0-iff* [simp]:
 $a \wedge n = 0 \longleftrightarrow$
 $a = (0 :: 'a :: \{\text{mult-zero}, \text{zero-neq-one}, \text{no-zero-divisors}, \text{power}\}) \wedge n \neq 0$
 ⟨proof⟩

lemma (in *field*) *power-diff*:
assumes *nz*: $a \neq 0$
shows $n \leq m \implies a \wedge (m - n) = a \wedge m / a \wedge n$
 ⟨proof⟩

Perhaps these should be simprules.

lemma *power-inverse*:
fixes $a :: 'a :: \text{division-ring-inverse-zero}$
shows $\text{inverse } (a \wedge n) = \text{inverse } a \wedge n$
 ⟨proof⟩

lemma *power-one-over*:
 $1 / (a :: 'a :: \{\text{field-inverse-zero}, \text{power}\}) \wedge n = (1 / a) \wedge n$
 ⟨proof⟩

lemma *power-divide*:
 $(a / b) \wedge n = (a :: 'a :: \text{field-inverse-zero}) \wedge n / b \wedge n$
 ⟨proof⟩

18.2 Exponentiation for the Natural Numbers

lemma *nat-one-le-power* [simp]:
 $\text{Suc } 0 \leq i \implies \text{Suc } 0 \leq i \wedge n$
 ⟨proof⟩

lemma *nat-zero-less-power-iff* [simp]:
 $x \wedge n > 0 \longleftrightarrow x > (0 :: \text{nat}) \vee n = 0$
 ⟨proof⟩

lemma *nat-power-eq-Suc-0-iff* [simp]:
 $x \wedge m = \text{Suc } 0 \longleftrightarrow m = 0 \vee x = \text{Suc } 0$
 ⟨proof⟩

```

lemma power-Suc-0 [simp]:
  Suc 0 ^ n = Suc 0
  ⟨proof⟩

```

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::nat)
  assumes less: i ^ m < i ^ n
  shows m < n
  ⟨proof⟩

```

```

lemma power-dvd-imp-le:
  i ^ m dvd i ^ n  $\implies$  (1::nat) < i  $\implies$  m  $\leq$  n
  ⟨proof⟩

```

18.3 Code generator tweak

```

lemma power-power-power [code, code-unfold, code-inline del]:
  power = power.power (1::'a::{power}) (op *)
  ⟨proof⟩

```

```

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  $\sim$  = None) = (EX y. x = Some y)
  ⟨proof⟩

```

```

lemma not-Some-eq [iff]: (ALL y. x  $\sim$  = Some y) = (x = None)
  ⟨proof⟩

```

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*
 ⟨proof⟩

lemma *option-caseE*:
 assumes *c*: (*case x of None => P | Some y => Q y*)
 obtains
 (*None*) *x = None and P*
 | (*Some*) *y where x = Some y and Q y*
 ⟨proof⟩

lemma *UNIV-option-conv*: *UNIV = insert None (range Some)*
 ⟨proof⟩

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]: (*ALL x:set A. P x*) ==> *A = Some x ==> P x*
 ⟨proof⟩

⟨ML⟩

lemma *elem-set* [iff]: (*x : set xo*) = (*xo = Some x*)
 ⟨proof⟩

lemma *set-empty-eq* [simp]: (*set xo = {}*) = (*xo = None*)
 ⟨proof⟩

definition *map* :: ('*a* => '*b*) => '*a* option => '*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*
 ⟨proof⟩

lemma *option-map-Some* [simp, code]: *map f (Some x) = Some (f x)*
 ⟨proof⟩

lemma *option-map-is-None* [iff]:
 (*map f opt = None*) = (*opt = None*)
 ⟨proof⟩

lemma *option-map-eq-Some* [*iff*]:
 $(\text{map } f \text{ } xo = \text{Some } y) = (EX \ z. \text{ } xo = \text{Some } z \ \& \ f \ z = y)$
 ⟨*proof*⟩

lemma *option-map-comp*:
 $\text{map } f \ (\text{map } g \ \text{opt}) = \text{map } (f \ o \ g) \ \text{opt}$
 ⟨*proof*⟩

lemma *option-map-o-sum-case* [*simp*]:
 $\text{map } f \ o \ \text{sum-case } g \ h = \text{sum-case } (\text{map } f \ o \ g) \ (\text{map } f \ o \ h)$
 ⟨*proof*⟩

hide-const (**open**) *set map*

19.0.2 Code generator setup

definition *is-none* :: 'a option \Rightarrow bool **where**
 [*code-post*]: *is-none* $x \longleftrightarrow x = \text{None}$

lemma *is-none-code* [*code*]:
shows *is-none* None \longleftrightarrow True
and *is-none* (Some x) \longleftrightarrow False
 ⟨*proof*⟩

lemma *is-none-none*:
 $\text{is-none } x \longleftrightarrow x = \text{None}$
 ⟨*proof*⟩

lemma [*code-unfold*]:
 $\text{eq-class.eq } x \ \text{None} \longleftrightarrow \text{is-none } x$
 ⟨*proof*⟩

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  $\Rightarrow$  'a option  $\Rightarrow$  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  $\Rightarrow$  bool
  where
    emptyI [simp, intro!]: finite {}
    | insertI [simp, intro!]: finite A  $\Rightarrow$  finite (insert a A)

lemma ex-new-if-finite: — does not depend on def of finite at all
  assumes  $\neg$  finite (UNIV :: 'a set) and finite A
  shows  $\exists a::'a. a \notin A$ 
  <proof>

lemma finite-induct [case-names empty insert, induct set: finite]:
  finite F  $\Rightarrow$ 
    P {}  $\Rightarrow$  (!x F. finite F  $\Rightarrow$  x  $\notin$  F  $\Rightarrow$  P F  $\Rightarrow$  P (insert x F))  $\Rightarrow$ 
    P F
  — Discharging x  $\notin$  F entails extra work.
  <proof>

lemma finite-ne-induct [case-names singleton insert, consumes 2]:
assumes fin: finite F shows F  $\neq$  {}  $\Rightarrow$ 
  [  $\bigwedge x. P\{x\};$ 
     $\bigwedge x F. [finite F; F \neq \{x\}; x \notin F; P F] \Rightarrow P (insert x F)$  ]
   $\Rightarrow$  P F
  <proof>

lemma finite-subset-induct [consumes 2, case-names empty insert]:

```

assumes *finite F* **and** $F \subseteq A$
and *empty: P {}*
and *insert: !!a F. finite F ==> a ∈ A ==> a ∉ F ==> P F ==> P (insert a F)*
shows $P F$
 ⟨*proof*⟩

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*:
 $finite A \implies ALL x:A. (EX y. P x y) \implies EX f. ALL x:A. P x (f x)$
 ⟨*proof*⟩

Finite sets are the images of initial segments of natural numbers:

lemma *finite-imp-nat-seg-image-inj-on*:
assumes *fin: finite A*
shows $\exists (n::nat) f. A = f \text{ ‘ } \{i. i < n\} \ \& \ inj\text{-on } f \ \{i. i < n\}$
 ⟨*proof*⟩

lemma *nat-seg-image-imp-finite*:
 $!!f A. A = f \text{ ‘ } \{i::nat. i < n\} \implies finite A$
 ⟨*proof*⟩

lemma *finite-conv-nat-seg-image*:
 $finite A = (\exists (n::nat) f. A = f \text{ ‘ } \{i::nat. i < n\})$
 ⟨*proof*⟩

lemma *finite-imp-inj-to-nat-seg*:
assumes *finite A*
shows $EX f n::nat. f \text{ ‘ } A = \{i. i < n\} \ \& \ inj\text{-on } f \ A$
 ⟨*proof*⟩

lemma *finite-Collect-less-nat[iff]*: $finite\{n::nat. n < k\}$
 ⟨*proof*⟩

Finiteness and set theoretic constructions

lemma *finite-UnI*: $finite F \implies finite G \implies finite (F \text{ Un } G)$
 ⟨*proof*⟩

lemma *finite-subset*: $A \subseteq B \implies finite B \implies finite A$
 — Every subset of a finite set is finite.
 ⟨*proof*⟩

lemma *rev-finite-subset*: $finite B \implies A \subseteq B \implies finite A$
 ⟨*proof*⟩

lemma *finite-Un [iff]*: $finite (F \text{ Un } G) = (finite F \ \& \ finite G)$
 ⟨*proof*⟩

lemma *finite-Collect-disjI[simp]*:

$\text{finite}\{x. P\ x \mid Q\ x\} = (\text{finite}\{x. P\ x\} \ \& \ \text{finite}\{x. Q\ x\})$
 $\langle \text{proof} \rangle$

lemma *finite-Int* [*simp*, *intro*]: $\text{finite}\ F \mid \text{finite}\ G \implies \text{finite}\ (F\ \text{Int}\ G)$
 — The converse obviously fails.
 $\langle \text{proof} \rangle$

lemma *finite-Collect-conjI* [*simp*, *intro*]:
 $\text{finite}\{x. P\ x\} \mid \text{finite}\{x. Q\ x\} \implies \text{finite}\{x. P\ x \ \& \ Q\ x\}$
 — The converse obviously fails.
 $\langle \text{proof} \rangle$

lemma *finite-Collect-le-nat*[*iff*]: $\text{finite}\{n::\text{nat}. n \leq k\}$
 $\langle \text{proof} \rangle$

lemma *finite-insert* [*simp*]: $\text{finite}\ (\text{insert}\ a\ A) = \text{finite}\ A$
 $\langle \text{proof} \rangle$

lemma *finite-Union*[*simp*, *intro*]:
 $\llbracket \text{finite}\ A; !!M. M \in A \implies \text{finite}\ M \rrbracket \implies \text{finite}(\bigcup A)$
 $\langle \text{proof} \rangle$

lemma *finite-Inter*[*intro*]: $\text{EX}\ A:M. \text{finite}(A) \implies \text{finite}(\text{Inter}\ M)$
 $\langle \text{proof} \rangle$

lemma *finite-INT*[*intro*]: $\text{EX}\ x:I. \text{finite}(A\ x) \implies \text{finite}(\text{INT}\ x:I. A\ x)$
 $\langle \text{proof} \rangle$

lemma *finite-empty-induct*:
 assumes $\text{finite}\ A$
 and $P\ A$
 and $!!a\ A. \text{finite}\ A \implies a:A \implies P\ A \implies P\ (A - \{a\})$
 shows $P\ \{\}$
 $\langle \text{proof} \rangle$

lemma *finite-Diff* [*simp*]: $\text{finite}\ A \implies \text{finite}\ (A - B)$
 $\langle \text{proof} \rangle$

lemma *finite-Diff2* [*simp*]:
 assumes $\text{finite}\ B$ shows $\text{finite}\ (A - B) = \text{finite}\ A$
 $\langle \text{proof} \rangle$

lemma *finite-compl*[*simp*]:
 $\text{finite}(A::'a\ \text{set}) \implies \text{finite}(-A) = \text{finite}(\text{UNIV}::'a\ \text{set})$
 $\langle \text{proof} \rangle$

lemma *finite-Collect-not*[*simp*]:
 $\text{finite}\{x::'a. P\ x\} \implies \text{finite}\{x. \sim P\ x\} = \text{finite}(\text{UNIV}::'a\ \text{set})$
 $\langle \text{proof} \rangle$

lemma *finite-Diff-insert [iff]*: $\text{finite } (A - \text{insert } a \ B) = \text{finite } (A - B)$
 ⟨proof⟩

Image and Inverse Image over Finite Sets

lemma *finite-imageI[simp]*: $\text{finite } F \implies \text{finite } (h \ ` \ F)$
 — The image of a finite set is finite.
 ⟨proof⟩

lemma *finite-image-set [simp]*:
 $\text{finite } \{x. P \ x\} \implies \text{finite } \{f \ x \mid x. P \ x\}$
 ⟨proof⟩

lemma *finite-surj*: $\text{finite } A \implies B \leq f \ ` \ A \implies \text{finite } B$
 ⟨proof⟩

lemma *finite-range-imageI*:
 $\text{finite } (\text{range } g) \implies \text{finite } (\text{range } (\%x. f \ (g \ x)))$
 ⟨proof⟩

lemma *finite-imageD*: $\text{finite } (f \ ` \ A) \implies \text{inj-on } f \ A \implies \text{finite } A$
 ⟨proof⟩

lemma *inj-vimage-singleton*: $\text{inj } f \implies f \ ` \ \{a\} \subseteq \{THE \ x. f \ x = a\}$
 — The inverse image of a singleton under an injective function is included in a singleton.
 ⟨proof⟩

lemma *finite-vimageI*: $[[\text{finite } F; \text{inj } h]] \implies \text{finite } (h \ ` \ F)$
 — The inverse image of a finite set under an injective function is finite.
 ⟨proof⟩

lemma *finite-vimageD*:
assumes *fin*: $\text{finite } (h \ ` \ F)$ **and** *surj*: $\text{surj } h$
shows $\text{finite } F$
 ⟨proof⟩

lemma *finite-vimage-iff*: $\text{bij } h \implies \text{finite } (h \ ` \ F) \longleftrightarrow \text{finite } F$
 ⟨proof⟩

The finite UNION of finite sets

lemma *finite-UN-I*: $\text{finite } A \implies (!a. a:A \implies \text{finite } (B \ a)) \implies \text{finite } (\text{UN } a:A. B \ a)$
 ⟨proof⟩

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*]:
 $\text{finite } A \implies \text{finite } (\text{UNION } A \ B) = (\text{ALL } x:A. \text{finite } (B \ x))$
 <proof>

lemma *finite-Collect-bex*[*simp*]: $\text{finite } A \implies$
 $\text{finite}\{x. \text{EX } y:A. \ Q \ x \ y\} = (\text{ALL } y:A. \text{finite}\{x. \ Q \ x \ y\})$
 <proof>

lemma *finite-Collect-bounded-ex*[*simp*]: $\text{finite}\{y. \ P \ y\} \implies$
 $\text{finite}\{x. \text{EX } y. \ P \ y \ \& \ Q \ x \ y\} = (\text{ALL } y. \ P \ y \longrightarrow \text{finite}\{x. \ Q \ x \ y\})$
 <proof>

lemma *finite-Plus*: $[\text{finite } A; \text{finite } B] \implies \text{finite } (A \ <+> B)$
 <proof>

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>

lemma *finite-Plus-iff*[*simp*]: $\text{finite } (A \ <+> B) \longleftrightarrow \text{finite } A \ \wedge \ \text{finite } B$
 <proof>

lemma *finite-Plus-UNIV-iff*[*simp*]:
 $\text{finite } (\text{UNIV} :: ('a + 'b) \text{ set}) =$
 $(\text{finite } (\text{UNIV} :: 'a \text{ set}) \ \& \ \text{finite } (\text{UNIV} :: 'b \text{ set}))$
 <proof>

Sigma of finite sets

lemma *finite-SigmaI* [*simp*]:
 $\text{finite } A \implies (!a. \ a:A \implies \text{finite } (B \ a)) \implies \text{finite } (\text{SIGMA } a:A. \ B \ a)$
 <proof>

lemma *finite-cartesian-product*: $[\text{finite } A; \text{finite } B] \implies$
 $\text{finite } (A \ <*> B)$
 <proof>

lemma *finite-Prod-UNIV*:
 $\text{finite } (\text{UNIV} :: 'a \text{ set}) \implies \text{finite } (\text{UNIV} :: 'b \text{ set}) \implies \text{finite } (\text{UNIV} :: ('a * 'b) \text{ set})$
 <proof>

lemma *finite-cartesian-productD1*:
 $[\text{finite } (A \ <*> B); B \neq \{\}] \implies \text{finite } A$
 <proof>

lemma *finite-cartesian-productD2*:

$\llbracket \text{finite } (A < * > B); A \neq \{\} \rrbracket \implies \text{finite } B$
 $\langle \text{proof} \rangle$

The powerset of a finite set

lemma *finite-Pow-iff* [iff]: $\text{finite } (\text{Pow } A) = \text{finite } A$
 $\langle \text{proof} \rangle$

lemma *finite-Collect-subsets*[simp,intro]: $\text{finite } A \implies \text{finite } \{B. B \subseteq A\}$
 $\langle \text{proof} \rangle$

lemma *finite-UnionD*: $\text{finite}(\bigcup A) \implies \text{finite } A$
 $\langle \text{proof} \rangle$

lemma *finite-subset-image*:
 assumes *finite B*
 shows $B \subseteq f^{-1} A \implies \exists C \subseteq A. \text{finite } C \wedge B = f^{-1} C$
 $\langle \text{proof} \rangle$

20.2 Class *finite*

class *finite* =
 assumes *finite-UNIV*: $\text{finite } (\text{UNIV} :: 'a \text{ set})$
begin

lemma *finite* [simp]: $\text{finite } (A :: 'a \text{ set})$
 $\langle \text{proof} \rangle$

end

lemma *UNIV-unit* [no-atp]:
 $\text{UNIV} = \{()\}$ $\langle \text{proof} \rangle$

instance *unit* :: *finite* $\langle \text{proof} \rangle$

lemma *UNIV-bool* [no-atp]:
 $\text{UNIV} = \{\text{False}, \text{True}\}$ $\langle \text{proof} \rangle$

instance *bool* :: *finite* $\langle \text{proof} \rangle$

instance $*$:: (*finite*, *finite*) *finite* $\langle \text{proof} \rangle$

lemma *finite-option-UNIV* [simp]:
 $\text{finite } (\text{UNIV} :: 'a \text{ option set}) = \text{finite } (\text{UNIV} :: 'a \text{ set})$
 $\langle \text{proof} \rangle$

instance *option* :: (*finite*) *finite* $\langle \text{proof} \rangle$

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f* *x*})
 ⟨*proof*⟩

instance *fun* :: (*finite*, *finite*) *finite*
 ⟨*proof*⟩

instance + :: (*finite*, *finite*) *finite* ⟨*proof*⟩

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 \Rightarrow 'b \Rightarrow '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* \circ *f* *y* = *f* *y* \circ *f* *x*
 ⟨*proof*⟩

end

inductive *fold-graph* :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a set \Rightarrow 'b \Rightarrow bool
for *f* :: 'a \Rightarrow 'b \Rightarrow 'b **and** *z* :: 'b **where**
 emptyI [*intro*]: *fold-graph* *f* *z* {} *z* |
 insertI [*intro*]: *x* \notin *A* \Longrightarrow *fold-graph* *f* *z* *A* *y*
 \Longrightarrow *fold-graph* *f* *z* (*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 set \Rightarrow 'b **where**
 [*code del*]: *fold* *f* *z* *A* = (*THE* *y*. *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 *fold-graph* *f* *z* *A* (*f* *x* *y*)
 ⟨*proof*⟩

lemma *fold-graph-imp-finite*: *fold-graph* *f* *z* *A* *x* \Longrightarrow *finite* *A*
 ⟨*proof*⟩

lemma *finite-imp-fold-graph*: *finite* *A* \Longrightarrow $\exists x$. *fold-graph* *f* *z* *A* *x*
 ⟨*proof*⟩

20.3.1 From *fold-graph* to *fold***context** *fun-left-comm***begin****lemma** *fold-graph-insertE-aux*:
$$\text{fold-graph } f \ z \ A \ y \implies a \in A \implies \exists y'. y = f \ a \ y' \wedge \text{fold-graph } f \ z \ (A - \{a\}) \ y'$$

<proof>

lemma *fold-graph-insertE*:

assumes *fold-graph* $f \ z \ (\text{insert } x \ A) \ v$ **and** $x \notin A$
obtains y **where** $v = f \ x \ y$ **and** *fold-graph* $f \ z \ A \ y$
<proof>

lemma *fold-graph-determ*:
$$\text{fold-graph } f \ z \ A \ x \implies \text{fold-graph } f \ z \ A \ y \implies y = x$$

<proof>

lemma *fold-equality*:
$$\text{fold-graph } f \ z \ A \ y \implies \text{fold } f \ z \ A = y$$

<proof>

lemma *fold-graph-fold*: *finite* $A \implies \text{fold-graph } f \ z \ A \ (\text{fold } f \ z \ A)$ *<proof>*The base case for *fold*:**lemma** (**in** $-$) *fold-empty* [*simp*]: *fold* $f \ z \ \{\} = z$ *<proof>*The various recursion equations for *fold*:**lemma** *fold-insert* [*simp*]:
$$\text{finite } A \implies x \notin A \implies \text{fold } f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ A)$$

<proof>

lemma *fold-fun-comm*:
$$\text{finite } A \implies f \ x \ (\text{fold } f \ z \ A) = \text{fold } f \ (f \ x \ z) \ A$$

<proof>

lemma *fold-insert2*:
$$\text{finite } A \implies x \notin A \implies \text{fold } f \ z \ (\text{insert } x \ A) = \text{fold } f \ (f \ x \ z) \ A$$

<proof>

lemma *fold-rec*:**assumes** *finite* A **and** $x \in A$ **shows** *fold* $f \ z \ A = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$ *<proof>***lemma** *fold-insert-remove*:**assumes** *finite* A **shows** *fold* $f \ z \ (\text{insert } x \ A) = f \ x \ (\text{fold } f \ z \ (A - \{x\}))$

<proof>

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* : $f\ x\ o\ f\ x = f\ x$
<proof>

lemma *fold-insert-idem*:
assumes *fin*: *finite A*
shows $fold\ f\ z\ (insert\ x\ A) = f\ x\ (fold\ f\ z\ A)$
<proof>

declare *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

lemma *fold-insert-idem2*:
 $finite\ A \implies fold\ f\ z\ (insert\ x\ A) = fold\ f\ (f\ x\ z)\ A$
<proof>

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>

lemma (**in** *fun-left-comm-idem*) *fun-left-comm-idem-apply*:
 $fun-left-comm-idem\ (\lambda x. f\ (g\ x))$
<proof>

lemma *fun-left-comm-idem-insert*:
 $fun-left-comm-idem\ insert$
<proof>

lemma *fun-left-comm-idem-remove*:
 $fun-left-comm-idem\ (\lambda x\ A. A - \{x\})$
<proof>

lemma (**in** *semilattice-inf*) *fun-left-comm-idem-inf*:
 $fun-left-comm-idem\ inf$
<proof>

lemma (**in** *semilattice-sup*) *fun-left-comm-idem-sup*:

fun-left-comm-idem sup
 $\langle \text{proof} \rangle$

lemma *union-fold-insert*:
 assumes *finite A*
 shows $A \cup B = \text{fold insert } B \ A$
 $\langle \text{proof} \rangle$

lemma *minus-fold-remove*:
 assumes *finite A*
 shows $B - A = \text{fold } (\lambda x \ A. \ A - \{x\}) \ B \ A$
 $\langle \text{proof} \rangle$

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:
 assumes *finite A*
 shows $\text{inf } B \ (\text{Inf } A) = \text{fold inf } B \ A$
 $\langle \text{proof} \rangle$

lemma *sup-Sup-fold-sup*:
 assumes *finite A*
 shows $\text{sup } B \ (\text{Sup } A) = \text{fold sup } B \ A$
 $\langle \text{proof} \rangle$

lemma *Inf-fold-inf*:
 assumes *finite A*
 shows $\text{Inf } A = \text{fold inf top } A$
 $\langle \text{proof} \rangle$

lemma *Sup-fold-sup*:
 assumes *finite A*
 shows $\text{Sup } A = \text{fold sup bot } A$
 $\langle \text{proof} \rangle$

lemma *inf-INF-fold-inf*:
 assumes *finite A*
 shows $\text{inf } B \ (\text{INF } A \ f) = \text{fold } (\lambda A. \ \text{inf } (f \ A)) \ B \ A \ (\text{is } ?\text{inf} = ?\text{fold})$
 $\langle \text{proof} \rangle$

lemma *sup-SUPR-fold-sup*:
 assumes *finite A*
 shows $\text{sup } B \ (\text{SUPR } A \ f) = \text{fold } (\lambda A. \ \text{sup } (f \ A)) \ B \ A \ (\text{is } ?\text{sup} = ?\text{fold})$
 $\langle \text{proof} \rangle$

lemma *INF-fold-inf*:
 assumes *finite A*
 shows $\text{INF } A \ f = \text{fold } (\lambda A. \ \text{inf } (f \ A)) \ \text{top } A$

$\langle proof \rangle$

lemma *SUPR-fold-sup*:

assumes *finite A*

shows *SUPR A f = fold ($\lambda A. sup (f A)$) bot A*

$\langle proof \rangle$

end

20.4 The derived combinator *fold-image*

definition *fold-image* :: $('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a \text{ set} \Rightarrow 'b$

where *fold-image f g = fold ($\%x y. f (g x) y$)*

lemma *fold-image-empty[simp]*: *fold-image f g z {} = z*

$\langle proof \rangle$

context *ab-semigroup-mult*

begin

lemma *fold-image-insert[simp]*:

assumes *finite A* **and** $a \notin A$

shows *fold-image times g z (insert a A) = g a * (fold-image times g z A)*

$\langle proof \rangle$

lemma *fold-image-reindex*:

assumes *fin: finite A*

shows *inj-on h A \implies fold-image times g z ($h^{\ast}A$) = fold-image times ($g \circ h$) z A*

$\langle proof \rangle$

lemma *fold-image-cong*:

finite A \implies

($\! \! \! \lambda x. x:A \implies g x = h x$) \implies fold-image times g z A = fold-image times h z A

$\langle proof \rangle$

end

context *comm-monoid-mult*

begin

lemma *fold-image-1*:

*finite S \implies ($\forall x \in S. f x = 1$) \implies fold-image op * f 1 S = 1*

$\langle proof \rangle$

lemma *fold-image-Un-Int*:

$finite\ A ==> finite\ B ==>$
 $fold-image\ times\ g\ 1\ A * fold-image\ times\ g\ 1\ B =$
 $fold-image\ times\ g\ 1\ (A\ Un\ B) * fold-image\ times\ g\ 1\ (A\ Int\ B)$
 <proof>

lemma *fold-image-Un-one*:
assumes fS : $finite\ S$ **and** fT : $finite\ T$
and $I0$: $\forall x \in S \cap T. f\ x = 1$
shows $fold-image\ (op\ *)\ f\ 1\ (S \cup T) = fold-image\ (op\ *)\ f\ 1\ S * fold-image\ (op\ *)\ f\ 1\ T$
 <proof>

corollary *fold-Un-disjoint*:
 $finite\ A ==> finite\ B ==> A\ Int\ B = \{\} ==>$
 $fold-image\ times\ g\ 1\ (A\ Un\ B) =$
 $fold-image\ times\ g\ 1\ A * fold-image\ times\ g\ 1\ B$
 <proof>

lemma *fold-image-UN-disjoint*:
 $\llbracket finite\ I; ALL\ i:I. finite\ (A\ i);$
 $ALL\ i:I. ALL\ j:I. i \neq j \longrightarrow A\ i\ Int\ A\ j = \{\} \rrbracket$
 $\implies fold-image\ times\ g\ 1\ (UNION\ I\ A) =$
 $fold-image\ times\ (\%i. fold-image\ times\ g\ 1\ (A\ i))\ 1\ I$
 <proof>

lemma *fold-image-Sigma*: $finite\ A ==> ALL\ x:A. finite\ (B\ x) ==>$
 $fold-image\ times\ (\%x. fold-image\ times\ (g\ x)\ 1\ (B\ x))\ 1\ A =$
 $fold-image\ times\ (split\ g)\ 1\ (SIGMA\ x:A. B\ x)$
 <proof>

lemma *fold-image-distrib*: $finite\ A \implies$
 $fold-image\ times\ (\%x. g\ x * h\ x)\ 1\ A =$
 $fold-image\ times\ g\ 1\ A * fold-image\ times\ h\ 1\ A$
 <proof>

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 $R\ (fold-image\ (op\ *)\ h\ e\ S)\ (fold-image\ (op\ *)\ g\ e\ S)$
 <proof>

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>

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*
 $\langle proof \rangle$
end

20.5 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow bool$
for *f* :: $'a \Rightarrow 'a \Rightarrow 'a$
where
fold1Set-insertI [*intro*]:
 $\llbracket fold-graph\ f\ a\ A\ x; a \notin A \rrbracket \implies fold1Set\ f\ (insert\ a\ A)\ x$
definition *fold1* :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a$ **where**
fold1 *f A* == *THE* *x. fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x $\implies A \neq \{\}$
 $\langle proof \rangle$

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*)
 $\langle proof \rangle$

lemma *fold1-singleton* [*simp*]: *fold1 f* $\{a\}$ = *a*
 $\langle proof \rangle$

lemma *finite-nonempty-imp-fold1Set*:
 $\llbracket finite\ A; A \neq \{\} \rrbracket \implies EX\ x. fold1Set\ f\ A\ x$
 $\langle proof \rangle$

First, some lemmas about *fold-graph*.

context *ab-semigroup-mult*
begin

lemma *fun-left-comm*: *fun-left-comm*(*op* *)
 $\langle proof \rangle$

lemma *fold-graph-insert-swap*:

assumes *fold*: *fold-graph times* $(b::'a) A y$ **and** $b \notin A$

shows *fold-graph times* $z (\text{insert } b A) (z * y)$

<proof>

lemma *fold-graph-permute-diff*:

assumes *fold*: *fold-graph times* $b A x$

shows $!!a. [a \in A; b \notin A] \implies \text{fold-graph times } a (\text{insert } b (A - \{a\})) x$

<proof>

lemma *fold1-eq-fold*:

assumes *finite* A $a \notin A$ **shows** *fold1 times* $(\text{insert } a A) = \text{fold times } a A$

<proof>

lemma *nonempty-iff*: $(A \neq \{\}) = (\exists x B. A = \text{insert } x B \ \& \ x \notin B)$

<proof>

lemma *fold1-insert*:

assumes *nonempty*: $A \neq \{\}$ **and** A : *finite* A $x \notin A$

shows *fold1 times* $(\text{insert } x A) = x * \text{fold1 times } A$

<proof>

end

context *ab-semigroup-idem-mult*

begin

lemma *fun-left-comm-idem*: *fun-left-comm-idem*(*op* *)

<proof>

lemma *fold1-insert-idem* [*simp*]:

assumes *nonempty*: $A \neq \{\}$ **and** A : *finite* A

shows *fold1 times* $(\text{insert } x A) = x * \text{fold1 times } A$

<proof>

lemma *hom-fold1-commute*:

assumes *hom*: $!!x y. h (x * y) = h x * h y$

and N : *finite* N $N \neq \{\}$ **shows** $h (\text{fold1 times } N) = \text{fold1 times } (h \text{ ‘ } N)$

<proof>

lemma *fold1-eq-fold-idem*:

assumes *finite* A

shows *fold1 times* $(\text{insert } a A) = \text{fold times } a A$

<proof>

end

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: $g = \text{fold1 } f \implies g \{a\} = a$

<proof>

lemma (in *ab-semigroup-mult*) *fold1-insert-def*:

$\llbracket g = \text{fold1 times}; \text{finite } A; x \notin A; A \neq \{\} \rrbracket \implies g (\text{insert } x A) = x * g A$
<proof>

lemma (in *ab-semigroup-idem-mult*) *fold1-insert-idem-def*:

$\llbracket g = \text{fold1 times}; \text{finite } A; A \neq \{\} \rrbracket \implies g (\text{insert } x A) = x * g A$
<proof>

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* $\implies B \neq \{\} \implies A \text{ Int } B = \{\} \implies$

$\text{fold1 times } (A \text{ Un } B) = \text{fold1 times } A * \text{fold1 times } B$

<proof>

lemma *fold1-in*:

assumes *A*: *finite (A) A* $\neq \{\}$ **and** *elem*: $\bigwedge x y. x * y \in \{x, y\}$

shows *fold1 times A* $\in A$

<proof>

end

lemma (in *ab-semigroup-idem-mult*) *fold1-Un2*:

assumes *A*: *finite A A* $\neq \{\}$

shows *finite B* $\implies B \neq \{\} \implies$

$\text{fold1 times } (A \text{ Un } B) = \text{fold1 times } A * \text{fold1 times } B$

<proof>

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*: *finite A* $\implies F A s = \text{fold } f s A$

begin

lemma *empty* [*simp*]:

$F \{\} = id$

$\langle proof \rangle$

lemma *insert* [*simp*]:

assumes *finite* A **and** $x \notin A$

shows $F (\text{insert } x A) = F A \circ f x$

$\langle proof \rangle$

lemma *remove*:

assumes *finite* A **and** $x \in A$

shows $F A = F (A - \{x\}) \circ f x$

$\langle proof \rangle$

lemma *insert-remove*:

assumes *finite* A

shows $F (\text{insert } x A) = F (A - \{x\}) \circ f x$

$\langle proof \rangle$

lemma *commute-left-comp*:

$f y \circ (f x \circ g) = f x \circ (f y \circ g)$

$\langle proof \rangle$

lemma *commute-comp'*:

assumes *finite* A

shows $f x \circ F A = F A \circ f x$

$\langle proof \rangle$

lemma *commute-left-comp'*:

assumes *finite* A

shows $f x \circ (F A \circ g) = F A \circ (f x \circ g)$

$\langle proof \rangle$

lemma *commute-comp''*:

assumes *finite* A **and** *finite* B

shows $F B \circ F A = F A \circ F B$

$\langle proof \rangle$

lemma *commute-left-comp''*:

assumes *finite* A **and** *finite* B

shows $F B \circ (F A \circ g) = F A \circ (F B \circ g)$

$\langle proof \rangle$

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 $F (A \cup B) \circ F (A \cap B) = F A \circ F B$
 $\langle proof \rangle$

lemma *union*:
assumes *finite A and finite B*
and $A \cap B = \{\}$
shows $F (A \cup B) = F A \circ F B$
 $\langle proof \rangle$

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*:
 $f x \circ (f x \circ g) = f x \circ g$
 $\langle proof \rangle$

lemma *in-comp-idem*:
assumes *finite A and* $x \in A$
shows $F A \circ f x = F A$
 $\langle proof \rangle$

lemma *subset-comp-idem*:
assumes *finite A and* $B \subseteq A$
shows $F A \circ F B = F A$
 $\langle proof \rangle$

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite A*
shows $F (\text{insert } x A) = F A \circ f x$
 $\langle proof \rangle$

lemma *union-idem*:
assumes *finite A and finite B*
shows $F (A \cup B) = F A \circ F B$
 $\langle proof \rangle$

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*]:
 $F \ \{\} = 1$
 $\langle \text{proof} \rangle$

lemma *insert* [*simp*]:
assumes *finite A* **and** $x \notin A$
shows $F (\text{insert } x \ A) = g \ x * F \ A$
 $\langle \text{proof} \rangle$

lemma *remove*:
assumes *finite A* **and** $x \in A$
shows $F A = g \ x * F (A - \{x\})$
 $\langle \text{proof} \rangle$

lemma *insert-remove*:
assumes *finite A*
shows $F (\text{insert } x \ A) = g \ x * F (A - \{x\})$
 $\langle \text{proof} \rangle$

lemma *neutral*:
assumes *finite A* **and** $\forall x \in A. \ g \ x = 1$
shows $F A = 1$
 $\langle \text{proof} \rangle$

lemma *union-inter*:
assumes *finite A* **and** *finite B*
shows $F (A \cup B) * F (A \cap B) = F A * F B$
 $\langle \text{proof} \rangle$

corollary *union-inter-neutral*:
assumes *finite A* **and** *finite B*
and *I0*: $\forall x \in A \cap B. \ g \ x = 1$
shows $F (A \cup B) = F A * F B$
 $\langle \text{proof} \rangle$

corollary *union-disjoint*:
assumes *finite A* **and** *finite B*
assumes $A \cap B = \{\}$
shows $F (A \cup B) = F A * F B$
 $\langle \text{proof} \rangle$

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*: $\bigwedge g. \text{finite } A \implies F \ g \ A = \text{fold-image } f \ g \ 1 \ A$

sublocale *folding-image* < *folding-image-simple* *op* * 1 *g* *F* *g* *<proof>*

context *folding-image*
begin

lemma *reindex*:
assumes *finite* *A* **and** *inj-on* *h* *A*
shows $F \ g \ (h \text{ ` } A) = F \ (g \circ h) \ A$
<proof>

lemma *cong*:
assumes *finite* *A* **and** $\bigwedge x. x \in A \implies g \ x = h \ x$
shows $F \ g \ A = F \ h \ A$
<proof>

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 $F \ g \ (\text{UNION } I \ A) = F \ (F \ g \circ A) \ I$
<proof>

lemma *distrib*:
assumes *finite* *A*
shows $F \ (\lambda x. g \ x * h \ x) \ A = F \ g \ A * F \ h \ A$
<proof>

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 $R \ (F \ h \ S) \ (F \ g \ S)$
<proof>

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>

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 $F \ j \ S = F \ g \ T$

$\langle proof \rangle$

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* $\langle proof \rangle$

context *folding-image-simple-idem*
 begin

lemma *in-idem*:
 assumes *finite* *A* and $x \in A$
 shows $g \ x * F \ A = F \ A$
 $\langle proof \rangle$

lemma *subset-idem*:
 assumes *finite* *A* and $B \subseteq A$
 shows $F \ B * F \ A = F \ A$
 $\langle proof \rangle$

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
 assumes *finite* *A*
 shows $F \ (\text{insert } x \ A) = g \ x * F \ A$
 $\langle proof \rangle$

lemma *union-idem*:
 assumes *finite* *A* and *finite* *B*
 shows $F \ (A \cup B) = F \ A * F \ B$
 $\langle proof \rangle$

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* $\langle proof \rangle$

20.6.7 The neutral-less case

locale *folding-one* = *abel-semigroup* +

fixes $F :: 'a \text{ set} \Rightarrow 'a$
assumes $eq\text{-fold}$: $finite\ A \Longrightarrow F\ A = fold1\ f\ A$
begin

lemma *singleton* [*simp*]:
 $F\ \{x\} = x$
 $\langle proof \rangle$

lemma *eq-fold'*:
assumes $finite\ A$ **and** $x \notin A$
shows $F\ (insert\ x\ A) = fold\ (op\ *)\ x\ A$
 $\langle proof \rangle$

lemma *insert* [*simp*]:
assumes $finite\ A$ **and** $x \notin A$ **and** $A \neq \{\}$
shows $F\ (insert\ x\ A) = x * F\ A$
 $\langle proof \rangle$
thm *fold.commute-comp'* [*of B b, simplified expand-fun-eq, simplified*]
 $\langle proof \rangle$

lemma *remove*:
assumes $finite\ A$ **and** $x \in A$
shows $F\ A = (if\ A - \{x\} = \{\} \text{ then } x \text{ else } x * F\ (A - \{x\}))$
 $\langle proof \rangle$

lemma *insert-remove*:
assumes $finite\ A$
shows $F\ (insert\ x\ A) = (if\ A - \{x\} = \{\} \text{ then } x \text{ else } x * F\ (A - \{x\}))$
 $\langle proof \rangle$

lemma *union-disjoint*:
assumes $finite\ A\ A \neq \{\}$ **and** $finite\ B\ B \neq \{\}$ **and** $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
 $\langle proof \rangle$

lemma *union-inter*:
assumes $finite\ A$ **and** $finite\ B$ **and** $A \cap B \neq \{\}$
shows $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$
 $\langle proof \rangle$

lemma *closed*:
assumes $finite\ A\ A \neq \{\}$ **and** $elem$: $\bigwedge x\ y. x * y \in \{x, y\}$
shows $F\ A \in A$
 $\langle proof \rangle$

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* $\langle \text{proof} \rangle$

context *folding-one-idem*
begin

lemma *in-idem*:
assumes *finite* *A* **and** $x \in A$
shows $x * F A = F A$
 $\langle \text{proof} \rangle$

lemma *subset-idem*:
assumes *finite* *A* $B \neq \{\}$ **and** $B \subseteq A$
shows $F B * F A = F A$
 $\langle \text{proof} \rangle$

lemma *eq-fold-idem'*:
assumes *finite* *A*
shows $F (\text{insert } a A) = \text{fold } (op *) a A$
 $\langle \text{proof} \rangle$

lemma *insert-idem* [*simp*]:
assumes *finite* *A* **and** $A \neq \{\}$
shows $F (\text{insert } x A) = x * F A$
 $\langle \text{proof} \rangle$

lemma *union-idem*:
assumes *finite* *A* $A \neq \{\}$ **and** *finite* *B* $B \neq \{\}$
shows $F (A \cup B) = F A * F B$
 $\langle \text{proof} \rangle$

lemma *hom-commute*:
assumes *hom*: $\bigwedge x y. h (x * y) = h x * h y$
and *N*: *finite* *N* $N \neq \{\}$ **shows** $h (F N) = F (h ` N)$
 $\langle \text{proof} \rangle$

end

notation *times* (**infixl** * 70)

notation *Groups.one* (1)

20.7 Finite cardinality

This definition, although traditional, is ugly to work with: $\text{card } A == \text{LEAST } n. \text{ EX } f. A = \{f i \mid i. i < n\}$. 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 $\lambda x. 1$ *card* <proof>

lemma *card-infinite* [simp]:

\neg finite A \Longrightarrow *card* A = 0

<proof>

lemma *card-empty*:

card {} = 0

<proof>

lemma *card-insert-disjoint*:

finite A \Longrightarrow $x \notin A \Longrightarrow$ *card* (insert x A) = Suc (*card* A)

<proof>

lemma *card-insert-if*:

finite A \Longrightarrow *card* (insert x A) = (if $x \in A$ then *card* A else Suc (*card* A))

<proof>

lemma *card-ge-0-finite*:

card A > 0 \Longrightarrow finite A

<proof>

lemma *card-0-eq* [simp, no-atp]:

finite A \Longrightarrow *card* A = 0 \longleftrightarrow A = {}

<proof>

lemma *finite-UNIV-card-ge-0*:

finite (UNIV :: 'a set) \Longrightarrow *card* (UNIV :: 'a set) > 0

<proof>

lemma *card-eq-0-iff*:

card A = 0 \longleftrightarrow A = {} $\vee \neg$ finite A

<proof>

lemma *card-gt-0-iff*:

0 < *card* A \longleftrightarrow A \neq {} \wedge finite A

<proof>

lemma *card-Suc-Diff1*: finite A \Longrightarrow $x: A \Longrightarrow$ Suc (*card* (A - {x})) = *card* A

<proof>

lemma *card-Diff-singleton*:

finite A \Longrightarrow $x: A \Longrightarrow$ *card* (A - {x}) = *card* A - 1

<proof>

lemma *card-Diff-singleton-if*:

finite A ==> card (A - {x}) = (if x : A then card A - 1 else card A)
<proof>

lemma *card-Diff-insert[simp]*:

assumes *finite A and a:A and a ~: B*
shows *card(A - insert a B) = card(A - B) - 1*
<proof>

lemma *card-insert*: *finite A ==> card (insert x A) = Suc (card (A - {x}))*
<proof>

lemma *card-insert-le*: *finite A ==> card A <= card (insert x A)*
<proof>

lemma *card-mono*:

assumes *finite B and A ⊆ B*
shows *card A ≤ card B*
<proof>

lemma *card-seteq*: *finite B ==> (!A. A <= B ==> card B <= card A ==> A = B)*
<proof>

lemma *psubset-card-mono*: *finite B ==> A < B ==> card A < card B*
<proof>

lemma *card-Un-Int*: *finite A ==> finite B*
==> card A + card B = card (A Un B) + card (A Int B)
<proof>

lemma *card-Un-disjoint*: *finite A ==> finite B*
==> A Int B = {} ==> card (A Un B) = card A + card B
<proof>

lemma *card-Diff-subset*:

assumes *finite B and B ⊆ A*
shows *card (A - B) = card A - card B*
<proof>

lemma *card-Diff-subset-Int*:

assumes *AB: finite (A ∩ B)* **shows** *card (A - B) = card A - card (A ∩ B)*
<proof>

lemma *card-Diff1-less*: *finite A ==> x: A ==> card (A - {x}) < card A*
<proof>

lemma *card-Diff2-less*:

finite A ==> x: A ==> y: A ==> card (A - {x} - {y}) < card A

<proof>

lemma *card-Diff1-le*: *finite* $A \implies \text{card } (A - \{x\}) \leq \text{card } A$
<proof>

lemma *card-psubset*: *finite* $B \implies A \subseteq B \implies \text{card } A < \text{card } B \implies A < B$
<proof>

lemma *insert-partition*:

$$\llbracket x \notin F; \forall c1 \in \text{insert } x F. \forall c2 \in \text{insert } x F. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\} \rrbracket$$

$$\implies x \cap \bigcup F = \{\}$$
<proof>

lemma *finite-psubset-induct*[*consumes 1, case-names psubset*]:
assumes *fin*: *finite* A
and *major*: $\bigwedge A. \text{finite } A \implies (\bigwedge B. B \subset A \implies P B) \implies P A$
shows $P A$
<proof>

main cardinality theorem

lemma *card-partition* [*rule-format*]:

$$\text{finite } C \implies$$

$$\text{finite } (\bigcup C) \longrightarrow$$

$$(\forall c \in C. \text{card } c = k) \longrightarrow$$

$$(\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}) \longrightarrow$$

$$k * \text{card}(C) = \text{card } (\bigcup C)$$
<proof>

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>

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes $\text{card } A = \text{Suc } k$
shows $\exists b B. A = \text{insert } b B \ \& \ b \notin B \ \& \ \text{card } B = k \ \& \ (k=0 \longrightarrow B=\{\})$
<proof>

lemma *card-Suc-eq*:
 $(\text{card } A = \text{Suc } k) =$
 $(\exists b B. A = \text{insert } b B \ \& \ b \notin B \ \& \ \text{card } B = k \ \& \ (k=0 \longrightarrow B=\{\}))$
<proof>

lemma *finite-fun-UNIVD2*:
assumes *fin*: *finite* (*UNIV* :: ('a \Rightarrow 'b) set)
shows *finite* (*UNIV* :: 'b set)
<proof>

lemma *card-UNIV-unit*: $\text{card } (\text{UNIV} :: \text{unit set}) = 1$
 ⟨proof⟩

20.7.1 Cardinality of image

lemma *card-image-le*: $\text{finite } A \implies \text{card } (f \text{ ‘ } A) \leq \text{card } A$
 ⟨proof⟩

lemma *card-image*:
 assumes *inj-on* f A
 shows $\text{card } (f \text{ ‘ } A) = \text{card } A$
 ⟨proof⟩

lemma *bij-betw-same-card*: $\text{bij-betw } f \text{ } A \text{ } B \implies \text{card } A = \text{card } B$
 ⟨proof⟩

lemma *endo-inj-surj*: $\text{finite } A \implies f \text{ ‘ } A \subseteq A \implies \text{inj-on } f \text{ } A \implies f \text{ ‘ } A = A$
 ⟨proof⟩

lemma *eq-card-imp-inj-on*:
 $[\text{finite } A; \text{card}(f \text{ ‘ } A) = \text{card } A] \implies \text{inj-on } f \text{ } A$
 ⟨proof⟩

lemma *inj-on-iff-eq-card*:
 $\text{finite } A \implies \text{inj-on } f \text{ } A = (\text{card}(f \text{ ‘ } A) = \text{card } A)$
 ⟨proof⟩

lemma *card-inj-on-le*:
 $[\text{inj-on } f \text{ } A; f \text{ ‘ } A \subseteq B; \text{finite } B] \implies \text{card } A \leq \text{card } B$
 ⟨proof⟩

lemma *card-bij-eq*:
 $[\text{inj-on } f \text{ } A; f \text{ ‘ } A \subseteq B; \text{inj-on } g \text{ } B; g \text{ ‘ } B \subseteq A; \text{finite } A; \text{finite } B] \implies \text{card } A = \text{card } B$
 ⟨proof⟩

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⟩

lemma *card-Plus-conv-if*:
 $\text{card } (A <+> B) = (\text{if } \text{finite } A \wedge \text{finite } B \text{ then } \text{card } A + \text{card } B \text{ else } 0)$
 ⟨proof⟩

20.7.3 Cardinality of the Powerset

lemma *card-Pow*: $\text{finite } A \implies \text{card } (\text{Pow } A) = \text{Suc } (\text{Suc } 0) ^ \text{card } A$
 <proof>

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:
 $\text{finite } (\text{Union } C) \implies$
 $\text{ALL } c : C. k \text{ dvd card } c \implies$
 $(\text{ALL } c1 : C. \text{ALL } c2 : C. c1 \neq c2 \longrightarrow c1 \text{ Int } c2 = \{\}) \implies$
 $k \text{ dvd card } (\text{Union } C)$
 <proof>

20.7.4 Relating injectivity and surjectivity

lemma *finite-surj-inj*: $\text{finite}(A) \implies A \leq f^*A \implies \text{inj-on } f \ A$
 <proof>

lemma *finite-UNIV-surj-inj*: **fixes** $f :: 'a \Rightarrow 'a$
shows $\text{finite}(\text{UNIV} :: 'a \text{ set}) \implies \text{surj } f \implies \text{inj } f$
 <proof>

lemma *finite-UNIV-inj-surj*: **fixes** $f :: 'a \Rightarrow 'a$
shows $\text{finite}(\text{UNIV} :: 'a \text{ set}) \implies \text{inj } f \implies \text{surj } f$
 <proof>

corollary *infinite-UNIV-nat[iff]*: $\sim \text{finite}(\text{UNIV} :: \text{nat set})$
 <proof>

lemma *infinite-UNIV-char-0[no-atp]*:
 $\neg \text{finite } (\text{UNIV} :: 'a :: \text{semiring-char-0 set})$
 <proof>

end

21 Relation: Relations

theory *Relation*
imports *Datatype Finite-Set*
begin

21.1 Definitions

definition
 $\text{converse} :: ('a * 'b) \text{ set} \Rightarrow ('b * 'a) \text{ set}$
 $((\text{--}^{\wedge} - 1) [1000] 999) \text{ where}$
 $\text{r}^{\wedge} - 1 == \{(y, x). (x, y) : r\}$

notation (*xsymbols*)

converse $((-^1) [1000] 999)$

definition

rel-comp $:: [('a * 'b) \text{ set}, ('b * 'c) \text{ set}] \Rightarrow ('a * 'c) \text{ set}$

(**infixr** 0 75) **where**

$r \text{ O } s == \{(x,z). \text{EX } y. (x, y) : r \ \& \ (y, z) : s\}$

definition

Image $:: [('a * 'b) \text{ set}, 'a \text{ set}] \Rightarrow 'b \text{ set}$

(**infixl** “ 90) **where**

$r \text{ “ } s == \{y. \text{EX } x:s. (x,y):r\}$

definition

Id $:: ('a * 'a) \text{ set}$ **where** — the identity relation

$\text{Id} == \{p. \text{EX } x. p = (x,x)\}$

definition

Id-on $:: 'a \text{ set} \Rightarrow ('a * 'a) \text{ set}$ **where** — diagonal: identity over a set

$\text{Id-on } A == \bigcup_{x \in A}. \{(x,x)\}$

definition

Domain $:: ('a * 'b) \text{ set} \Rightarrow 'a \text{ set}$ **where**

$\text{Domain } r == \{x. \text{EX } y. (x,y):r\}$

definition

Range $:: ('a * 'b) \text{ set} \Rightarrow 'b \text{ set}$ **where**

$\text{Range } r == \text{Domain}(r^{-1})$

definition

Field $:: ('a * 'a) \text{ set} \Rightarrow 'a \text{ set}$ **where**

$\text{Field } r == \text{Domain } r \cup \text{Range } r$

definition

refl-on $:: ['a \text{ set}, ('a * 'a) \text{ set}] \Rightarrow \text{bool}$ **where** — reflexivity over a set

$\text{refl-on } A \ r == r \subseteq A \times A \ \& \ (\text{ALL } x: A. (x,x) : r)$

abbreviation

refl $:: ('a * 'a) \text{ set} \Rightarrow \text{bool}$ **where** — reflexivity over a type

$\text{refl} == \text{refl-on } \text{UNIV}$

definition

sym $:: ('a * 'a) \text{ set} \Rightarrow \text{bool}$ **where** — symmetry predicate

$\text{sym } r == \text{ALL } x \ y. (x,y):r \longrightarrow (y,x):r$

definition

antisym $:: ('a * 'a) \text{ set} \Rightarrow \text{bool}$ **where** — antisymmetry predicate

$\text{antisym } r == \text{ALL } x \ y. (x,y):r \longrightarrow (y,x):r \longrightarrow x=y$

definition

$trans :: ('a * 'a) set \Rightarrow bool$ **where** — transitivity predicate
 $trans\ r == (ALL\ x\ y\ z.\ (x,y):r \longrightarrow (y,z):r \longrightarrow (x,z):r)$

definition

$irrefl :: ('a * 'a) set \Rightarrow bool$ **where**
 $irrefl\ r \equiv \forall x.\ (x,x) \notin r$

definition

$total-on :: 'a set \Rightarrow ('a * 'a) set \Rightarrow bool$ **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

$single-valued :: ('a * 'b) set \Rightarrow bool$ **where**
 $single-valued\ r == ALL\ x\ y.\ (x,y):r \longrightarrow (ALL\ z.\ (x,z):r \longrightarrow y=z)$

definition

$inv-image :: ('b * 'b) set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a * 'a) set$ **where**
 $inv-image\ r\ f == \{(x, y).\ (f\ x, f\ y) : r\}$

21.2 The identity relation

lemma IdI [intro]: $(a, a) : Id$
 $\langle proof \rangle$

lemma IdE [elim!]: $p : Id \Longrightarrow (!x.\ p = (x, x) \Longrightarrow P) \Longrightarrow P$
 $\langle proof \rangle$

lemma $pair-in-Id-conv$ [iff]: $((a, b) : Id) = (a = b)$
 $\langle proof \rangle$

lemma $refl-Id$: $refl\ Id$
 $\langle proof \rangle$

lemma $antisym-Id$: $antisym\ Id$
 — A strange result, since Id is also symmetric.
 $\langle proof \rangle$

lemma $sym-Id$: $sym\ Id$
 $\langle proof \rangle$

lemma $trans-Id$: $trans\ Id$
 $\langle proof \rangle$

21.3 Diagonal: identity over a set

lemma $Id-on-empty$ [simp]: $Id-on\ \{\} = \{\}$

$\langle proof \rangle$

lemma *Id-on-eqI*: $a = b \implies a : A \implies (a, b) : Id-on\ A$
 $\langle proof \rangle$

lemma *Id-onI* [*intro!,no-atp*]: $a : A \implies (a, a) : Id-on\ A$
 $\langle proof \rangle$

lemma *Id-onE* [*elim!*]:
 $c : Id-on\ A \implies (!x. x : A \implies c = (x, x) \implies P) \implies P$
 — The general elimination rule.
 $\langle proof \rangle$

lemma *Id-on-iff*: $((x, y) : Id-on\ A) = (x = y \ \& \ x : A)$
 $\langle proof \rangle$

lemma *Id-on-subset-Times*: $Id-on\ A \subseteq A \times A$
 $\langle proof \rangle$

21.4 Composition of two relations

lemma *rel-compI* [*intro*]:
 $(a, b) : r \implies (b, c) : s \implies (a, c) : r\ O\ s$
 $\langle proof \rangle$

lemma *rel-compE* [*elim!*]: $xz : r\ O\ s \implies$
 $(!x\ y\ z. xz = (x, z) \implies (x, y) : r \implies (y, z) : s \implies P) \implies P$
 $\langle proof \rangle$

lemma *rel-compEpair*:
 $(a, c) : r\ O\ s \implies (!y. (a, y) : r \implies (y, c) : s \implies P) \implies P$
 $\langle proof \rangle$

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
 $\langle proof \rangle$

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
 $\langle proof \rangle$

lemma *rel-comp-empty1* [*simp*]: $\{\} \ O\ R = \{\}$
 $\langle proof \rangle$

lemma *rel-comp-empty2* [*simp*]: $R\ O\ \{\} = \{\}$
 $\langle proof \rangle$

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
 $\langle proof \rangle$

lemma *trans-O-subset*: $trans\ r \implies r\ O\ r \subseteq r$

$\langle \text{proof} \rangle$

lemma *rel-comp-mono*: $r' \subseteq r \implies s' \subseteq s \implies (r' \ O \ s') \subseteq (r \ O \ s)$
 $\langle \text{proof} \rangle$

lemma *rel-comp-subset-Sigma*:
 $r \subseteq A \times B \implies s \subseteq B \times C \implies (r \ O \ s) \subseteq A \times C$
 $\langle \text{proof} \rangle$

lemma *rel-comp-distrib[simp]*: $R \ O \ (S \cup T) = (R \ O \ S) \cup (R \ O \ T)$
 $\langle \text{proof} \rangle$

lemma *rel-comp-distrib2[simp]*: $(S \cup T) \ O \ R = (S \ O \ R) \cup (T \ O \ R)$
 $\langle \text{proof} \rangle$

lemma *rel-comp-UNION-distrib*: $s \ O \ \text{UNION } I \ r = \text{UNION } I \ (\%i. s \ O \ r \ i)$
 $\langle \text{proof} \rangle$

lemma *rel-comp-UNION-distrib2*: $\text{UNION } I \ r \ O \ s = \text{UNION } I \ (\%i. r \ i \ O \ s)$
 $\langle \text{proof} \rangle$

21.5 Reflexivity

lemma *refl-onI*: $r \subseteq A \times A \implies (!x. x : A \implies (x, x) : r) \implies \text{refl-on } A \ r$
 $\langle \text{proof} \rangle$

lemma *refl-onD*: $\text{refl-on } A \ r \implies a : A \implies (a, a) : r$
 $\langle \text{proof} \rangle$

lemma *refl-onD1*: $\text{refl-on } A \ r \implies (x, y) : r \implies x : A$
 $\langle \text{proof} \rangle$

lemma *refl-onD2*: $\text{refl-on } A \ r \implies (x, y) : r \implies y : A$
 $\langle \text{proof} \rangle$

lemma *refl-on-Int*: $\text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cap B) \ (r \cap s)$
 $\langle \text{proof} \rangle$

lemma *refl-on-Un*: $\text{refl-on } A \ r \implies \text{refl-on } B \ s \implies \text{refl-on } (A \cup B) \ (r \cup s)$
 $\langle \text{proof} \rangle$

lemma *refl-on-INTER*:
 $\text{ALL } x:S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\text{INTER } S \ A) \ (\text{INTER } S \ r)$
 $\langle \text{proof} \rangle$

lemma *refl-on-UNION*:
 $\text{ALL } x:S. \text{refl-on } (A \ x) \ (r \ x) \implies \text{refl-on } (\text{UNION } S \ A) \ (\text{UNION } S \ r)$
 $\langle \text{proof} \rangle$

lemma *refl-on-empty*[simp]: *refl-on* {} {}
 $\langle proof \rangle$

lemma *refl-on-Id-on*: *refl-on* A (*Id-on* A)
 $\langle proof \rangle$

21.6 Antisymmetry

lemma *antisymI*:
 $(!!x\ y. (x, y) : r ==> (y, x) : r ==> x=y) ==> antisym\ r$
 $\langle proof \rangle$

lemma *antisymD*: *antisym* r ==> (a, b) : r ==> (b, a) : r ==> a = b
 $\langle proof \rangle$

lemma *antisym-subset*: $r \subseteq s ==> antisym\ s ==> antisym\ r$
 $\langle proof \rangle$

lemma *antisym-empty* [simp]: *antisym* {}
 $\langle proof \rangle$

lemma *antisym-Id-on* [simp]: *antisym* (*Id-on* A)
 $\langle proof \rangle$

21.7 Symmetry

lemma *symI*: $(!!a\ b. (a, b) : r ==> (b, a) : r) ==> sym\ r$
 $\langle proof \rangle$

lemma *symD*: *sym* r ==> (a, b) : r ==> (b, a) : r
 $\langle proof \rangle$

lemma *sym-Int*: *sym* r ==> *sym* s ==> *sym* (r \cap s)
 $\langle proof \rangle$

lemma *sym-Un*: *sym* r ==> *sym* s ==> *sym* (r \cup s)
 $\langle proof \rangle$

lemma *sym-INTER*: $ALL\ x:S. sym\ (r\ x) ==> sym\ (INTER\ S\ r)$
 $\langle proof \rangle$

lemma *sym-UNION*: $ALL\ x:S. sym\ (r\ x) ==> sym\ (UNION\ S\ r)$
 $\langle proof \rangle$

lemma *sym-Id-on* [simp]: *sym* (*Id-on* A)
 $\langle proof \rangle$

21.8 Transitivity

lemma *transI*:

$(!!x\ y\ z. (x, y) : r ==> (y, z) : r ==> (x, z) : r) ==> \text{trans } r$
 $\langle \text{proof} \rangle$

lemma *transD*: $\text{trans } r ==> (a, b) : r ==> (b, c) : r ==> (a, c) : r$
 $\langle \text{proof} \rangle$

lemma *trans-Int*: $\text{trans } r ==> \text{trans } s ==> \text{trans } (r \cap s)$
 $\langle \text{proof} \rangle$

lemma *trans-INTER*: $\text{ALL } x:S. \text{trans } (r\ x) ==> \text{trans } (\text{INTER } S\ r)$
 $\langle \text{proof} \rangle$

lemma *trans-Id-on* [simp]: $\text{trans } (\text{Id-on } A)$
 $\langle \text{proof} \rangle$

lemma *trans-diff-Id*: $\text{trans } r \implies \text{antisym } r \implies \text{trans } (r - \text{Id})$
 $\langle \text{proof} \rangle$

21.9 Irreflexivity

lemma *irrefl-diff-Id*[simp]: $\text{irrefl}(r - \text{Id})$
 $\langle \text{proof} \rangle$

21.10 Totality

lemma *total-on-empty*[simp]: $\text{total-on } \{\} r$
 $\langle \text{proof} \rangle$

lemma *total-on-diff-Id*[simp]: $\text{total-on } A\ (r - \text{Id}) = \text{total-on } A\ r$
 $\langle \text{proof} \rangle$

21.11 Converse

lemma *converse-iff* [iff]: $((a, b) : r^{-1}) = ((b, a) : r)$
 $\langle \text{proof} \rangle$

lemma *converseI*[sym]: $(a, b) : r ==> (b, a) : r^{-1}$
 $\langle \text{proof} \rangle$

lemma *converseD*[sym]: $(a, b) : r^{-1} ==> (b, a) : r$
 $\langle \text{proof} \rangle$

lemma *converseE* [elim!]:
 $yx : r^{-1} ==> (!!x\ y. yx = (y, x) ==> (x, y) : r ==> P) ==> P$
 — More general than *converseD*, as it “splits” the member of the relation.
 $\langle \text{proof} \rangle$

lemma *converse-converse* [simp]: $(r^{-1})^{-1} = r$
 $\langle \text{proof} \rangle$

lemma *converse-rel-comp*: $(r \ O \ s)^{\wedge-1} = s^{\wedge-1} \ O \ r^{\wedge-1}$
 $\langle proof \rangle$

lemma *converse-Int*: $(r \cap s)^{\wedge-1} = r^{\wedge-1} \cap s^{\wedge-1}$
 $\langle proof \rangle$

lemma *converse-Un*: $(r \cup s)^{\wedge-1} = r^{\wedge-1} \cup s^{\wedge-1}$
 $\langle proof \rangle$

lemma *converse-INTER*: $(INTER \ S \ r)^{\wedge-1} = (INT \ x:S. (r \ x)^{\wedge-1})$
 $\langle proof \rangle$

lemma *converse-UNION*: $(UNION \ S \ r)^{\wedge-1} = (UN \ x:S. (r \ x)^{\wedge-1})$
 $\langle proof \rangle$

lemma *converse-Id* [simp]: $Id^{\wedge-1} = Id$
 $\langle proof \rangle$

lemma *converse-Id-on* [simp]: $(Id-on \ A)^{\wedge-1} = Id-on \ A$
 $\langle proof \rangle$

lemma *refl-on-converse* [simp]: $refl-on \ A \ (converse \ r) = refl-on \ A \ r$
 $\langle proof \rangle$

lemma *sym-converse* [simp]: $sym \ (converse \ r) = sym \ r$
 $\langle proof \rangle$

lemma *antisym-converse* [simp]: $antisym \ (converse \ r) = antisym \ r$
 $\langle proof \rangle$

lemma *trans-converse* [simp]: $trans \ (converse \ r) = trans \ r$
 $\langle proof \rangle$

lemma *sym-conv-converse-eq*: $sym \ r = (r^{\wedge-1} = r)$
 $\langle proof \rangle$

lemma *sym-Un-converse*: $sym \ (r \cup r^{\wedge-1})$
 $\langle proof \rangle$

lemma *sym-Int-converse*: $sym \ (r \cap r^{\wedge-1})$
 $\langle proof \rangle$

lemma *total-on-converse*[simp]: $total-on \ A \ (r^{\wedge-1}) = total-on \ A \ r$
 $\langle proof \rangle$

21.12 Domain

declare *Domain-def* [no-atp]

lemma *Domain-iff*: $(a : \text{Domain } r) = (EX\ y. (a, y) : r)$
 $\langle \text{proof} \rangle$

lemma *DomainI* [intro]: $(a, b) : r ==> a : \text{Domain } r$
 $\langle \text{proof} \rangle$

lemma *DomainE* [elim!]:
 $a : \text{Domain } r ==> (!y. (a, y) : r ==> P) ==> P$
 $\langle \text{proof} \rangle$

lemma *Domain-empty* [simp]: $\text{Domain } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *Domain-empty-iff*: $\text{Domain } r = \{\} \longleftrightarrow r = \{\}$
 $\langle \text{proof} \rangle$

lemma *Domain-insert*: $\text{Domain } (\text{insert } (a, b) \ r) = \text{insert } a \ (\text{Domain } r)$
 $\langle \text{proof} \rangle$

lemma *Domain-Id* [simp]: $\text{Domain } \text{Id} = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *Domain-Id-on* [simp]: $\text{Domain } (\text{Id-on } A) = A$
 $\langle \text{proof} \rangle$

lemma *Domain-Un-eq*: $\text{Domain}(A \cup B) = \text{Domain}(A) \cup \text{Domain}(B)$
 $\langle \text{proof} \rangle$

lemma *Domain-Int-subset*: $\text{Domain}(A \cap B) \subseteq \text{Domain}(A) \cap \text{Domain}(B)$
 $\langle \text{proof} \rangle$

lemma *Domain-Diff-subset*: $\text{Domain}(A) - \text{Domain}(B) \subseteq \text{Domain}(A - B)$
 $\langle \text{proof} \rangle$

lemma *Domain-Union*: $\text{Domain } (\text{Union } S) = (\bigcup A \in S. \text{Domain } A)$
 $\langle \text{proof} \rangle$

lemma *Domain-converse*[simp]: $\text{Domain}(r^{-1}) = \text{Range } r$
 $\langle \text{proof} \rangle$

lemma *Domain-mono*: $r \subseteq s ==> \text{Domain } r \subseteq \text{Domain } s$
 $\langle \text{proof} \rangle$

lemma *fst-eq-Domain*: $\text{fst } ' R = \text{Domain } R$
 $\langle \text{proof} \rangle$

lemma *Domain-dprod* [simp]: $\text{Domain } (\text{dprod } r \ s) = \text{uprod } (\text{Domain } r) \ (\text{Domain } s)$
 $\langle \text{proof} \rangle$

lemma *Domain-dsum* [simp]: $\text{Domain } (dsum\ r\ s) = usum\ (\text{Domain } r)\ (\text{Domain } s)$
 $\langle proof \rangle$

21.13 Range

lemma *Range-iff*: $(a : \text{Range } r) = (EX\ y. (y, a) : r)$
 $\langle proof \rangle$

lemma *RangeI* [intro]: $(a, b) : r ==> b : \text{Range } r$
 $\langle proof \rangle$

lemma *RangeE* [elim!]: $b : \text{Range } r ==> (!x. (x, b) : r ==> P) ==> P$
 $\langle proof \rangle$

lemma *Range-empty* [simp]: $\text{Range } \{\} = \{\}$
 $\langle proof \rangle$

lemma *Range-empty-iff*: $\text{Range } r = \{\} \longleftrightarrow r = \{\}$
 $\langle proof \rangle$

lemma *Range-insert*: $\text{Range } (\text{insert } (a, b) r) = \text{insert } b\ (\text{Range } r)$
 $\langle proof \rangle$

lemma *Range-Id* [simp]: $\text{Range } Id = UNIV$
 $\langle proof \rangle$

lemma *Range-Id-on* [simp]: $\text{Range } (Id\text{-on } A) = A$
 $\langle proof \rangle$

lemma *Range-Un-eq*: $\text{Range}(A \cup B) = \text{Range}(A) \cup \text{Range}(B)$
 $\langle proof \rangle$

lemma *Range-Int-subset*: $\text{Range}(A \cap B) \subseteq \text{Range}(A) \cap \text{Range}(B)$
 $\langle proof \rangle$

lemma *Range-Diff-subset*: $\text{Range}(A) - \text{Range}(B) \subseteq \text{Range}(A - B)$
 $\langle proof \rangle$

lemma *Range-Union*: $\text{Range } (\text{Union } S) = (\bigcup A \in S. \text{Range } A)$
 $\langle proof \rangle$

lemma *Range-converse*[simp]: $\text{Range}(r^{-1}) = \text{Domain } r$
 $\langle proof \rangle$

lemma *snd-eq-Range*: $\text{snd } ` R = \text{Range } R$
 $\langle proof \rangle$

21.14 Field

lemma *mono-Field*: $r \subseteq s \implies \text{Field } r \subseteq \text{Field } s$

<proof>

lemma *Field-empty[simp]*: $\text{Field } \{\} = \{\}$

<proof>

lemma *Field-insert[simp]*: $\text{Field } (\text{insert } (a,b) \ r) = \{a,b\} \cup \text{Field } r$

<proof>

lemma *Field-Un[simp]*: $\text{Field } (r \cup s) = \text{Field } r \cup \text{Field } s$

<proof>

lemma *Field-Union[simp]*: $\text{Field } (\bigcup R) = \bigcup (\text{Field } ` R)$

<proof>

lemma *Field-converse[simp]*: $\text{Field } (r^{-1}) = \text{Field } r$

<proof>

21.15 Image of a set under a relation

declare *Image-def* [no-atp]

lemma *Image-iff*: $(b : r `` A) = (\exists x : A. (x, b) : r)$

<proof>

lemma *Image-singleton*: $r `` \{a\} = \{b. (a, b) : r\}$

<proof>

lemma *Image-singleton-iff* [iff]: $(b : r `` \{a\}) = ((a, b) : r)$

<proof>

lemma *ImageI* [intro,no-atp]: $(a, b) : r \implies a : A \implies b : r `` A$

<proof>

lemma *ImageE* [elim!]:

$b : r `` A \implies (!x. (x, b) : r \implies x : A \implies P) \implies P$

<proof>

lemma *rev-ImageI*: $a : A \implies (a, b) : r \implies b : r `` A$

— This version’s more effective when we already have the required a

<proof>

lemma *Image-empty* [simp]: $R `` \{\} = \{\}$

<proof>

lemma *Image-Id* [simp]: $\text{Id} `` A = A$

<proof>

lemma *Image-Id-on* [*simp*]: $Id-on\ A \text{ “ } B = A \cap B$
 $\langle proof \rangle$

lemma *Image-Int-subset*: $R \text{ “ } (A \cap B) \subseteq R \text{ “ } A \cap R \text{ “ } B$
 $\langle proof \rangle$

lemma *Image-Int-eq*:
 $single-valued\ (converse\ R) ==> R \text{ “ } (A \cap B) = R \text{ “ } A \cap R \text{ “ } B$
 $\langle proof \rangle$

lemma *Image-Un*: $R \text{ “ } (A \cup B) = R \text{ “ } A \cup R \text{ “ } B$
 $\langle proof \rangle$

lemma *Un-Image*: $(R \cup S) \text{ “ } A = R \text{ “ } A \cup S \text{ “ } A$
 $\langle proof \rangle$

lemma *Image-subset*: $r \subseteq A \times B ==> r \text{ “ } C \subseteq B$
 $\langle proof \rangle$

lemma *Image-eq-UN*: $r \text{ “ } B = (\bigcup y \in B. r \text{ “ } \{y\})$
 — NOT suitable for rewriting
 $\langle proof \rangle$

lemma *Image-mono*: $r' \subseteq r ==> A' \subseteq A ==> (r' \text{ “ } A') \subseteq (r \text{ “ } A)$
 $\langle proof \rangle$

lemma *Image-UN*: $(r \text{ “ } (UNION\ A\ B)) = (\bigcup x \in A. r \text{ “ } (B\ x))$
 $\langle proof \rangle$

lemma *Image-INT-subset*: $(r \text{ “ } INTER\ A\ B) \subseteq (\bigcap x \in A. r \text{ “ } (B\ x))$
 $\langle proof \rangle$

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
 $[single-valued\ (r^{-1}); A \neq \{\}] ==> r \text{ “ } INTER\ A\ B = (\bigcap x \in A. r \text{ “ } B\ x)$
 $\langle proof \rangle$

lemma *Image-subset-eq*: $(r \text{ “ } A \subseteq B) = (A \subseteq -((r^{-1}) \text{ “ } (-B)))$
 $\langle proof \rangle$

21.16 Single valued relations

lemma *single-valuedI*:
 $ALL\ x\ y. (x, y) : r --> (ALL\ z. (x, z) : r --> y = z) ==> single-valued\ r$
 $\langle proof \rangle$

lemma *single-valuedD*:
 $single-valued\ r ==> (x, y) : r ==> (x, z) : r ==> y = z$
 $\langle proof \rangle$

lemma *single-valued-rel-comp*:

single-valued $r \implies \text{single-valued } s \implies \text{single-valued } (r \circ s)$
 $\langle \text{proof} \rangle$

lemma *single-valued-subset*:

$r \subseteq s \implies \text{single-valued } s \implies \text{single-valued } r$
 $\langle \text{proof} \rangle$

lemma *single-valued-Id* [simp]: *single-valued* Id

$\langle \text{proof} \rangle$

lemma *single-valued-Id-on* [simp]: *single-valued* ($\text{Id-on } A$)

$\langle \text{proof} \rangle$

21.17 Graphs given by *Collect*

lemma *Domain-Collect-split* [simp]: $\text{Domain}\{(x,y). P\ x\ y\} = \{x. \exists y. P\ x\ y\}$

$\langle \text{proof} \rangle$

lemma *Range-Collect-split* [simp]: $\text{Range}\{(x,y). P\ x\ y\} = \{y. \exists x. P\ x\ y\}$

$\langle \text{proof} \rangle$

lemma *Image-Collect-split* [simp]: $\{(x,y). P\ x\ y\} \text{ “} A = \{y. \exists x:A. P\ x\ y\}$

$\langle \text{proof} \rangle$

21.18 Inverse image

lemma *sym-inv-image*: $\text{sym } r \implies \text{sym } (\text{inv-image } r\ f)$

$\langle \text{proof} \rangle$

lemma *trans-inv-image*: $\text{trans } r \implies \text{trans } (\text{inv-image } r\ f)$

$\langle \text{proof} \rangle$

lemma *in-inv-image*[simp]: $((x,y) : \text{inv-image } r\ f) = ((f\ x, f\ y) : r)$

$\langle \text{proof} \rangle$

lemma *converse-inv-image*[simp]: $(\text{inv-image } R\ f)^{-1} = \text{inv-image } (R^{-1})\ f$

$\langle \text{proof} \rangle$

21.19 Finiteness

lemma *finite-converse* [iff]: $\text{finite } (r^{-1}) = \text{finite } r$

$\langle \text{proof} \rangle$

lemma *finite-Domain*: $\text{finite } r \implies \text{finite } (\text{Domain } r)$

$\langle \text{proof} \rangle$

lemma *finite-Range*: $\text{finite } r \implies \text{finite } (\text{Range } r)$

$\langle \text{proof} \rangle$

lemma *finite-Field*: $\text{finite } r \implies \text{finite } (\text{Field } r)$
 — A finite relation has a finite field (= $\text{domain} \cup \text{range}$).
 $\langle \text{proof} \rangle$

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*: $(\bigwedge x y. (x, y) \in r \implies (x, y) \in s) \implies r \subseteq s$
 $\langle \text{proof} \rangle$

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 $P \leq Q$
 $\langle \text{proof} \rangle$

lemma *predicate1D* [*Pure.dest?*, *dest?*]:
 $P \leq Q \implies P x \implies Q x$
 $\langle \text{proof} \rangle$

lemma *rev-predicate1D*:

$P\ x ==> P\ <= Q ==> Q\ x$
 $\langle proof \rangle$

lemma *predicate2I* [*Pure.intro!*, *intro!*]:
 assumes $PQ: \bigwedge x\ y. P\ x\ y \implies Q\ x\ y$
 shows $P \leq Q$
 $\langle proof \rangle$

lemma *predicate2D* [*Pure.dest*, *dest*]:
 $P \leq Q \implies P\ x\ y \implies Q\ x\ y$
 $\langle proof \rangle$

lemma *rev-predicate2D*:
 $P\ x\ y ==> P\ <= Q ==> Q\ x\ y$
 $\langle proof \rangle$

22.1.1 Equality

lemma *pred-equals-eq*: $((\lambda x. x \in R) = (\lambda x. x \in S)) = (R = S)$
 $\langle proof \rangle$

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y. (x, y) \in R) = (\lambda x\ y. (x, y) \in S)) = (R = S)$
 $\langle proof \rangle$

22.1.2 Order relation

lemma *pred-subset-eq*: $((\lambda x. x \in R) <= (\lambda x. x \in S)) = (R <= S)$
 $\langle proof \rangle$

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y. (x, y) \in R) <= (\lambda x\ y. (x, y) \in S)) = (R <= S)$
 $\langle proof \rangle$

22.1.3 Top and bottom elements

lemma *top1I* [*intro!*]: *top* x
 $\langle proof \rangle$

lemma *top2I* [*intro!*]: *top* $x\ y$
 $\langle proof \rangle$

lemma *bot1E* [*elim!*]: *bot* $x \implies P$
 $\langle proof \rangle$

lemma *bot2E* [*elim!*]: *bot* $x\ y \implies P$
 $\langle proof \rangle$

lemma *bot-empty-eq*: *bot* $= (\lambda x. x \in \{\})$
 $\langle proof \rangle$

lemma *bot-empty-eq2*: $\text{bot} = (\lambda x y. (x, y) \in \{\})$
 $\langle \text{proof} \rangle$

22.1.4 Binary union

lemma *sup1I1* [*elim?*]: $A x \implies \text{sup } A B x$
 $\langle \text{proof} \rangle$

lemma *sup2I1* [*elim?*]: $A x y \implies \text{sup } A B x y$
 $\langle \text{proof} \rangle$

lemma *sup1I2* [*elim?*]: $B x \implies \text{sup } A B x$
 $\langle \text{proof} \rangle$

lemma *sup2I2* [*elim?*]: $B x y \implies \text{sup } A B x y$
 $\langle \text{proof} \rangle$

lemma *sup1E* [*elim!*]: $\text{sup } A B x \implies (A x \implies P) \implies (B x \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *sup2E* [*elim!*]: $\text{sup } A B x y \implies (A x y \implies P) \implies (B x y \implies P) \implies P$
 $\langle \text{proof} \rangle$

Classical introduction rule: no commitment to A vs B .

lemma *sup1CI* [*intro!*]: $(\sim B x \implies A x) \implies \text{sup } A B x$
 $\langle \text{proof} \rangle$

lemma *sup2CI* [*intro!*]: $(\sim B x y \implies A x y) \implies \text{sup } A B x y$
 $\langle \text{proof} \rangle$

lemma *sup-Un-eq*: $\text{sup } (\lambda x. x \in R) (\lambda x. x \in S) = (\lambda x. x \in R \cup S)$
 $\langle \text{proof} \rangle$

lemma *sup-Un-eq2* [*pred-set-conv*]: $\text{sup } (\lambda x y. (x, y) \in R) (\lambda x y. (x, y) \in S) = (\lambda x y. (x, y) \in R \cup S)$
 $\langle \text{proof} \rangle$

22.1.5 Binary intersection

lemma *inf1I* [*intro!*]: $A x \implies B x \implies \text{inf } A B x$
 $\langle \text{proof} \rangle$

lemma *inf2I* [*intro!*]: $A x y \implies B x y \implies \text{inf } A B x y$
 $\langle \text{proof} \rangle$

lemma *inf1E* [*elim!*]: $\text{inf } A B x \implies (A x \implies B x \implies P) \implies P$

$\langle proof \rangle$

lemma *inf2E* [*elim!*]: $\inf A B x y \implies (A x y \implies B x y \implies P) \implies P$
 $\langle proof \rangle$

lemma *inf1D1*: $\inf A B x \implies A x$
 $\langle proof \rangle$

lemma *inf2D1*: $\inf A B x y \implies A x y$
 $\langle proof \rangle$

lemma *inf1D2*: $\inf A B x \implies B x$
 $\langle proof \rangle$

lemma *inf2D2*: $\inf A B x y \implies B x y$
 $\langle proof \rangle$

lemma *inf-Int-eq*: $\inf (\lambda x. x \in R) (\lambda x. x \in S) = (\lambda x. x \in R \cap S)$
 $\langle proof \rangle$

lemma *inf-Int-eq2* [*pred-set-conv*]: $\inf (\lambda x y. (x, y) \in R) (\lambda x y. (x, y) \in S) =$
 $(\lambda x y. (x, y) \in R \cap S)$
 $\langle proof \rangle$

22.1.6 Unions of families

lemma *SUP1-iff*: $(\text{SUP } x:A. B x) b = (\text{EX } x:A. B x b)$
 $\langle proof \rangle$

lemma *SUP2-iff*: $(\text{SUP } x:A. B x) b c = (\text{EX } x:A. B x b c)$
 $\langle proof \rangle$

lemma *SUP1-I* [*intro*]: $a : A \implies B a b \implies (\text{SUP } x:A. B x) b$
 $\langle proof \rangle$

lemma *SUP2-I* [*intro*]: $a : A \implies B a b c \implies (\text{SUP } x:A. B x) b c$
 $\langle proof \rangle$

lemma *SUP1-E* [*elim!*]: $(\text{SUP } x:A. B x) b \implies (!x. x : A \implies B x b \implies R) \implies R$
 $\langle proof \rangle$

lemma *SUP2-E* [*elim!*]: $(\text{SUP } x:A. B x) b c \implies (!x. x : A \implies B x b c \implies R) \implies R$
 $\langle proof \rangle$

lemma *SUP-UN-eq*: $(\text{SUP } i. (\lambda x. x \in r i)) = (\lambda x. x \in (\text{UN } i. r i))$
 $\langle proof \rangle$

lemma *SUP-UN-eq2*: $(\text{SUP } i. (\lambda x y. (x, y) \in r \ i)) = (\lambda x y. (x, y) \in (\text{UN } i. r \ i))$
 $\langle \text{proof} \rangle$

22.1.7 Intersections of families

lemma *INF1-iff*: $(\text{INF } x:A. B \ x) \ b = (\text{ALL } x:A. B \ x \ b)$
 $\langle \text{proof} \rangle$

lemma *INF2-iff*: $(\text{INF } x:A. B \ x) \ b \ c = (\text{ALL } x:A. B \ x \ b \ c)$
 $\langle \text{proof} \rangle$

lemma *INF1-I* [intro!]: $(!!x. x : A ==> B \ x \ b) ==> (\text{INF } x:A. B \ x) \ b$
 $\langle \text{proof} \rangle$

lemma *INF2-I* [intro!]: $(!!x. x : A ==> B \ x \ b \ c) ==> (\text{INF } x:A. B \ x) \ b \ c$
 $\langle \text{proof} \rangle$

lemma *INF1-D* [elim]: $(\text{INF } x:A. B \ x) \ b ==> a : A ==> B \ a \ b$
 $\langle \text{proof} \rangle$

lemma *INF2-D* [elim]: $(\text{INF } x:A. B \ x) \ b \ c ==> a : A ==> B \ a \ b \ c$
 $\langle \text{proof} \rangle$

lemma *INF1-E* [elim]: $(\text{INF } x:A. B \ x) \ b ==> (B \ a \ b ==> R) ==> (a \sim: A ==> R) ==> R$
 $\langle \text{proof} \rangle$

lemma *INF2-E* [elim]: $(\text{INF } x:A. B \ x) \ b \ c ==> (B \ a \ b \ c ==> R) ==> (a \sim: A ==> R) ==> R$
 $\langle \text{proof} \rangle$

lemma *INF-INT-eq*: $(\text{INF } i. (\lambda x. x \in r \ i)) = (\lambda x. x \in (\text{INT } i. r \ i))$
 $\langle \text{proof} \rangle$

lemma *INF-INT-eq2*: $(\text{INF } i. (\lambda x y. (x, y) \in r \ i)) = (\lambda x y. (x, y) \in (\text{INT } i. r \ i))$
 $\langle \text{proof} \rangle$

22.2 Predicates as relations

22.2.1 Composition

inductive

pred-comp :: $['a ==> 'b ==> \text{bool}, 'b ==> 'c ==> \text{bool}] ==> 'a ==> 'c ==> \text{bool}$
 (infixr OO 75)

for $r :: 'a ==> 'b ==> \text{bool}$ and $s :: 'b ==> 'c ==> \text{bool}$

where

pred-compI [intro]: $r \ a \ b ==> s \ b \ c ==> (r \ OO \ s) \ a \ c$

inductive-cases *pred-compE* [elim!]: $(r \ OO \ s) \ a \ c$

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
 $((\lambda x y. (x, y) \in r) \text{ OO } (\lambda x y. (x, y) \in s)) = (\lambda x y. (x, y) \in r \text{ O } s))$
 $\langle \text{proof} \rangle$

22.2.2 Converse

inductive

conversep :: ('a => 'b => bool) => 'b => 'a => bool
 $((-\hat{\text{---}}1) [1000] 1000)$
for *r* :: 'a => 'b => bool

where

conversepI: $r \text{ a } b \implies r \hat{\text{---}}1 \text{ b } a$

notation (*xsymbols*)

conversep $((-\hat{\text{---}}1) [1000] 1000)$

lemma *conversepD*:

assumes *ab*: $r \hat{\text{---}}1 \text{ a } b$

shows $r \text{ b } a$ $\langle \text{proof} \rangle$

lemma *conversep-iff* [*iff*]: $r \hat{\text{---}}1 \text{ a } b = r \text{ b } a$
 $\langle \text{proof} \rangle$

lemma *conversep-converse-eq* [*pred-set-conv*]:
 $(\lambda x y. (x, y) \in r) \hat{\text{---}}1 = (\lambda x y. (x, y) \in r \hat{\text{---}}1)$
 $\langle \text{proof} \rangle$

lemma *conversep-conversep* [*simp*]: $(r \hat{\text{---}}1) \hat{\text{---}}1 = r$
 $\langle \text{proof} \rangle$

lemma *converse-pred-comp*: $(r \text{ OO } s) \hat{\text{---}}1 = s \hat{\text{---}}1 \text{ OO } r \hat{\text{---}}1$
 $\langle \text{proof} \rangle$

lemma *converse-meet*: $(\inf r \text{ s}) \hat{\text{---}}1 = \inf r \hat{\text{---}}1 \text{ s} \hat{\text{---}}1$
 $\langle \text{proof} \rangle$

lemma *converse-join*: $(\sup r \text{ s}) \hat{\text{---}}1 = \sup r \hat{\text{---}}1 \text{ s} \hat{\text{---}}1$
 $\langle \text{proof} \rangle$

lemma *conversep-noteq* [*simp*]: $(op \sim) \hat{\text{---}}1 = op \sim$
 $\langle \text{proof} \rangle$

lemma *conversep-eq* [*simp*]: $(op =) \hat{\text{---}}1 = op =$
 $\langle \text{proof} \rangle$

22.2.3 Domain

inductive

DomainP :: ('a => 'b => bool) => 'a => bool
for *r* :: 'a => 'b => bool

where

DomainPI [intro]: $r \ a \ b \implies \text{DomainP } r \ a$

inductive-cases *DomainPE* [elim!]: $\text{DomainP } r \ a$

lemma *DomainP-Domain-eq* [pred-set-conv]: $\text{DomainP } (\lambda x \ y. (x, y) \in r) = (\lambda x. x \in \text{Domain } r)$
 ⟨proof⟩

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]: $r \ a \ b \implies \text{RangeP } r \ b$

inductive-cases *RangePE* [elim!]: $\text{RangeP } r \ b$

lemma *RangeP-Range-eq* [pred-set-conv]: $\text{RangeP } (\lambda x \ y. (x, y) \in r) = (\lambda x. x \in \text{Range } r)$
 ⟨proof⟩

22.2.5 Inverse image

definition

inv-imagep :: $('b \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$ **where**
inv-imagep $r \ f == \%x \ y. r \ (f \ x) \ (f \ y)$

lemma [pred-set-conv]: $\text{inv-imagep } (\lambda x \ y. (x, y) \in r) \ f = (\lambda x \ y. (x, y) \in \text{inv-image } r \ f)$
 ⟨proof⟩

lemma *in-inv-imagep* [simp]: $\text{inv-imagep } r \ f \ x \ y = r \ (f \ x) \ (f \ y)$
 ⟨proof⟩

22.2.6 Powerset

definition *Powp* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \ \text{set} \Rightarrow \text{bool}$ **where**

Powp $A == \lambda B. \forall x \in B. A \ x$

lemma *Powp-Pow-eq* [pred-set-conv]: $\text{Powp } (\lambda x. x \in A) = (\lambda x. x \in \text{Pow } A)$
 ⟨proof⟩

lemmas *Powp-mono* [mono] = *Pow-mono* [to-pred pred-subset-eq]

22.2.7 Properties of relations

abbreviation *antisymP* :: $('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**

antisymP $r == \text{antisym } \{(x, y). r \ x \ y\}$

abbreviation $\text{transP} :: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{transP } r == \text{trans } \{(x, y). r \ x \ y\}$

abbreviation $\text{single-valuedP} :: ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow \text{bool}$ **where**
 $\text{single-valuedP } r == \text{single-valued } \{(x, y). r \ x \ y\}$

22.3 Predicates as enumerations

22.3.1 The type of predicate enumerations (a monad)

datatype $'a \text{ pred} = \text{Pred } 'a \Rightarrow \text{bool}$

primrec $\text{eval} :: 'a \text{ pred} \Rightarrow 'a \Rightarrow \text{bool}$ **where**
 $\text{eval-pred: eval } (\text{Pred } f) = f$

lemma Pred-eval [simp] :
 $\text{Pred } (\text{eval } x) = x$
 $\langle \text{proof} \rangle$

lemma $\text{eval-inject: eval } x = \text{eval } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

definition $\text{single} :: 'a \Rightarrow 'a \text{ pred}$ **where**
 $\text{single } x = \text{Pred } ((\text{op } =) \ x)$

definition $\text{bind} :: 'a \text{ pred} \Rightarrow ('a \Rightarrow 'b \text{ pred}) \Rightarrow 'b \text{ pred}$ (**infixl** $\gg=$ 70) **where**
 $P \gg= f = \text{Pred } (\lambda x. (\exists y. \text{eval } P \ y \wedge \text{eval } (f \ y) \ x))$

instantiation $\text{pred} :: (\text{type}) \ \{ \text{complete-lattice}, \text{boolean-algebra} \}$
begin

definition
 $P \leq Q \longleftrightarrow \text{eval } P \leq \text{eval } Q$

definition
 $P < Q \longleftrightarrow \text{eval } P < \text{eval } Q$

definition
 $\perp = \text{Pred } \perp$

definition
 $\top = \text{Pred } \top$

definition
 $P \sqcap Q = \text{Pred } (\text{eval } P \sqcap \text{eval } Q)$

definition
 $P \sqcup Q = \text{Pred } (\text{eval } P \sqcup \text{eval } Q)$

definition

[code del]: $\prod A = \text{Pred } (\text{INFI } A \text{ eval})$

definition

[code del]: $\sqcup A = \text{Pred } (\text{SUPR } A \text{ eval})$

definition

$- P = \text{Pred } (- \text{ eval } P)$

definition

$P - Q = \text{Pred } (\text{eval } P - \text{eval } Q)$

instance $\langle \text{proof} \rangle$

end

lemma *bind-bind*:

$(P \gg Q) \gg R = P \gg (\lambda x. Q \ x \gg R)$
 $\langle \text{proof} \rangle$

lemma *bind-single*:

$P \gg \text{single} = P$
 $\langle \text{proof} \rangle$

lemma *single-bind*:

$\text{single } x \gg P = P \ x$
 $\langle \text{proof} \rangle$

lemma *bottom-bind*:

$\perp \gg P = \perp$
 $\langle \text{proof} \rangle$

lemma *sup-bind*:

$(P \sqcup Q) \gg R = P \gg R \sqcup Q \gg R$
 $\langle \text{proof} \rangle$

lemma *Sup-bind*: $(\sqcup A \gg f) = \sqcup ((\lambda x. x \gg f) \text{ ‘ } A)$

$\langle \text{proof} \rangle$

lemma *pred-iffI*:

assumes $\bigwedge x. \text{eval } A \ x \implies \text{eval } B \ x$
and $\bigwedge x. \text{eval } B \ x \implies \text{eval } A \ x$
shows $A = B$

$\langle \text{proof} \rangle$

lemma *singleI*: $\text{eval } (\text{single } x) \ x$

$\langle \text{proof} \rangle$

lemma *singleI-unit*: $\text{eval } (\text{single } ()) \ x$

$\langle \text{proof} \rangle$

lemma *singleE*: $\text{eval } (\text{single } x) \ y \implies (y = x \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *singleE'*: $\text{eval } (\text{single } x) \ y \implies (x = y \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *bindI*: $\text{eval } P \ x \implies \text{eval } (Q \ x) \ y \implies \text{eval } (P \gg= Q) \ y$
 $\langle \text{proof} \rangle$

lemma *bindE*: $\text{eval } (R \gg= Q) \ y \implies (\bigwedge x. \text{eval } R \ x \implies \text{eval } (Q \ x) \ y \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *botE*: $\text{eval } \perp \ x \implies P$
 $\langle \text{proof} \rangle$

lemma *supI1*: $\text{eval } A \ x \implies \text{eval } (A \sqcup B) \ x$
 $\langle \text{proof} \rangle$

lemma *supI2*: $\text{eval } B \ x \implies \text{eval } (A \sqcup B) \ x$
 $\langle \text{proof} \rangle$

lemma *supE*: $\text{eval } (A \sqcup B) \ x \implies (\text{eval } A \ x \implies P) \implies (\text{eval } B \ x \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *single-not-bot* [*simp*]:
 $\text{single } x \neq \perp$
 $\langle \text{proof} \rangle$

lemma *not-bot*:
assumes $A \neq \perp$
obtains x **where** $\text{eval } A \ x$
 $\langle \text{proof} \rangle$

22.3.2 Emptiness check and definite choice

definition *is-empty* :: $'a \text{ pred} \Rightarrow \text{bool}$ **where**
 $\text{is-empty } A \longleftrightarrow A = \perp$

lemma *is-empty-bot*:
 $\text{is-empty } \perp$
 $\langle \text{proof} \rangle$

lemma *not-is-empty-single*:
 $\neg \text{is-empty } (\text{single } x)$
 $\langle \text{proof} \rangle$

lemma *is-empty-sup*:

is-empty ($A \sqcup B$) \longleftrightarrow *is-empty* $A \wedge$ *is-empty* B

\langle *proof* \rangle

definition *singleton* :: (*unit* \Rightarrow 'a) \Rightarrow 'a *pred* \Rightarrow 'a **where**

singleton default $A =$ (*if* $\exists!x.$ *eval* $A\ x$ *then* *THE* $x.$ *eval* $A\ x$ *else* *default* ())

lemma *singleton-eqI*:

$\exists!x.$ *eval* $A\ x \Longrightarrow$ *eval* $A\ x \Longrightarrow$ *singleton default* $A = x$

\langle *proof* \rangle

lemma *eval-singletonI*:

$\exists!x.$ *eval* $A\ x \Longrightarrow$ *eval* A (*singleton default* A)

\langle *proof* \rangle

lemma *single-singleton*:

$\exists!x.$ *eval* $A\ x \Longrightarrow$ *single* (*singleton default* A) = A

\langle *proof* \rangle

lemma *singleton-undefinedI*:

$\neg (\exists!x.$ *eval* $A\ x) \Longrightarrow$ *singleton default* $A =$ *default* ()

\langle *proof* \rangle

lemma *singleton-bot*:

singleton default $\perp =$ *default* ()

\langle *proof* \rangle

lemma *singleton-single*:

singleton default (*single* x) = x

\langle *proof* \rangle

lemma *singleton-sup-single-single*:

singleton default (*single* $x \sqcup$ *single* y) = (*if* $x = y$ *then* x *else* *default* ())

\langle *proof* \rangle

lemma *singleton-sup-aux*:

singleton default ($A \sqcup B$) = (*if* $A = \perp$ *then* *singleton default* B

else if $B = \perp$ *then* *singleton default* A

else *singleton default*

(*single* (*singleton default* A) \sqcup *single* (*singleton default* B)))

\langle *proof* \rangle

lemma *singleton-sup*:

singleton default ($A \sqcup B$) = (*if* $A = \perp$ *then* *singleton default* B

else if $B = \perp$ *then* *singleton default* A

else if *singleton default* $A =$ *singleton default* B *then* *singleton default* A *else* *default*

\langle *proof* \rangle

\langle *proof* \rangle

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* \Longrightarrow *eval* (*if-pred* *P*) ()

<proof>

lemma *if-predE*: *eval* (*if-pred* *b*) *x* \Longrightarrow (*b* \Longrightarrow *x* = () \Longrightarrow *P*) \Longrightarrow *P*

<proof>

lemma *not-predI*: \neg *P* \Longrightarrow *eval* (*not-pred* (*Pred* ($\lambda u.$ *P*))) ()

<proof>

lemma *not-predI'*: \neg *eval* *P* () \Longrightarrow *eval* (*not-pred* *P*) ()

<proof>

lemma *not-predE*: *eval* (*not-pred* (*Pred* ($\lambda u.$ *P*))) *x* \Longrightarrow (\neg *P* \Longrightarrow *thesis*) \Longrightarrow *thesis*

<proof>

lemma *not-predE'*: *eval* (*not-pred* *P*) *x* \Longrightarrow (\neg *eval* *P* *x* \Longrightarrow *thesis*) \Longrightarrow *thesis*

<proof>

lemma *f* () = *False* \vee *f* () = *True*

<proof>

lemma *closure-of-bool-cases*:

assumes (*f* :: *unit* \Rightarrow *bool*) = (%*u.* *False*) \Longrightarrow *P f*

assumes *f* = (%*u.* *True*) \Longrightarrow *P f*

shows *P f*

<proof>

lemma *unit-pred-cases*:

assumes *P* \perp

assumes *P* (*single* ())

shows *P* *Q*

<proof>

lemma *holds-if-pred*:

holds (*if-pred* *b*) = *b*

<proof>

lemma *if-pred-holds*:

if-pred (*holds* *P*) = *P*

$\langle \text{proof} \rangle$

lemma *is-empty-holds*:

is-empty $P \iff \neg \text{holds } P$

$\langle \text{proof} \rangle$

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* $P xq)$ = $P \sqcup \text{pred-of-seq } xq$

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 $x \iff \text{False}$

| *member (Insert* $y P)$ $x \iff x = y \vee \text{eval } P x$

| *member (Join* $P xq)$ $x \iff \text{eval } P x \vee \text{member } xq x$

lemma *eval-member*:

member xq = *eval* (*pred-of-seq* xq)

$\langle \text{proof} \rangle$

lemma *eval-code* [*code*]: *eval* (*Seq f*) = *member* ($f \ ()$)

$\langle \text{proof} \rangle$

lemma *single-code* [*code*]:

single $x = \text{Seq } (\lambda u. \text{Insert } x \perp)$

$\langle \text{proof} \rangle$

primrec *apply* :: (*'a* \Rightarrow *'b Predicate.pred*) \Rightarrow *'a seq* \Rightarrow *'b seq* **where**

apply f Empty = *Empty*

| *apply f (Insert* $x P)$ = *Join* ($f x$) (*Join* ($P \gg= f$) *Empty*)

| *apply f (Join* $P xq)$ = *Join* ($P \gg= f$) (*apply f* xq)

lemma *apply-bind*:

pred-of-seq (*apply f* xq) = *pred-of-seq* $xq \gg= f$

$\langle \text{proof} \rangle$

lemma *bind-code* [*code*]:

Seq g $\gg= f = \text{Seq } (\lambda u. \text{apply } f (g \ ()))$

$\langle \text{proof} \rangle$

lemma *bot-set-code* [code]:

$\perp = \text{Seq } (\lambda u. \text{Empty})$

$\langle \text{proof} \rangle$

primrec *adjunct* :: 'a pred \Rightarrow 'a seq \Rightarrow 'a 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*:

$\text{pred-of-seq } (\text{adjunct } P \ xq) = P \sqcup \text{pred-of-seq } xq$

$\langle \text{proof} \rangle$

lemma *sup-code* [code]:

$\text{Seq } f \sqcup \text{Seq } g = \text{Seq } (\lambda u. \text{case } f \ ())$

$\text{of Empty} \Rightarrow g \ ()$

$| \text{Insert } x \ P \Rightarrow \text{Insert } x \ (P \sqcup \text{Seq } g)$

$| \text{Join } P \ xq \Rightarrow \text{adjunct } (\text{Seq } g) (\text{Join } P \ xq)$

$\langle \text{proof} \rangle$

primrec *contained* :: 'a seq \Rightarrow 'a pred \Rightarrow bool **where**

$\text{contained Empty } Q \longleftrightarrow \text{True}$

$| \text{contained } (\text{Insert } x \ P) \ Q \longleftrightarrow \text{eval } Q \ x \wedge P \leq Q$

$| \text{contained } (\text{Join } P \ xq) \ Q \longleftrightarrow P \leq Q \wedge \text{contained } xq \ Q$

lemma *single-less-eq-eval*:

$\text{single } x \leq P \longleftrightarrow \text{eval } P \ x$

$\langle \text{proof} \rangle$

lemma *contained-less-eq*:

$\text{contained } xq \ Q \longleftrightarrow \text{pred-of-seq } xq \leq Q$

$\langle \text{proof} \rangle$

lemma *less-eq-pred-code* [code]:

$\text{Seq } f \leq Q = (\text{case } f \ ())$

$\text{of Empty} \Rightarrow \text{True}$

$| \text{Insert } x \ P \Rightarrow \text{eval } Q \ x \wedge P \leq Q$

$| \text{Join } P \ xq \Rightarrow P \leq Q \wedge \text{contained } xq \ Q$

$\langle \text{proof} \rangle$

lemma *eq-pred-code* [code]:

fixes $P \ Q :: 'a \text{ pred}$

shows $\text{eq-class.eq } P \ Q \longleftrightarrow P \leq Q \wedge Q \leq P$

$\langle \text{proof} \rangle$

lemma [code]:

$\text{pred-case } f \ P = f \ (\text{eval } P)$

$\langle \text{proof} \rangle$

lemma *[code]*:

pred-rec $f\ P = f\ (\text{eval } P)$

<proof>

inductive *eq* :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ **where** *eq* $x\ x$

lemma *eq-is-eq*: $\text{eq } x\ y \equiv (x = y)$

<proof>

definition *map* :: $('a \Rightarrow 'b) \Rightarrow 'a\ \text{pred} \Rightarrow 'b\ \text{pred}$ **where**

map $f\ P = P \gg= (\text{single } o\ f)$

primrec *null* :: $'a\ \text{seq} \Rightarrow \text{bool}$ **where**

null *Empty* $\longleftrightarrow \text{True}$

| *null* (*Insert* $x\ P$) $\longleftrightarrow \text{False}$

| *null* (*Join* $P\ xq$) $\longleftrightarrow \text{is-empty } P \wedge \text{null } xq$

lemma *null-is-empty*:

null $xq \longleftrightarrow \text{is-empty } (\text{pred-of-seq } xq)$

<proof>

lemma *is-empty-code* *[code]*:

is-empty (*Seq* f) $\longleftrightarrow \text{null } (f\ ())$

<proof>

primrec *the-only* :: $(\text{unit} \Rightarrow 'a) \Rightarrow 'a\ \text{seq} \Rightarrow 'a$ **where**

[code del]: *the-only* *dfault* *Empty* = *dfault* $()$

| *the-only* *dfault* (*Insert* $x\ P$) = (if *is-empty* P then x else let $y = \text{singleton } \text{dfault } P$ 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 $x = \text{singleton } \text{dfault } P$; $y = \text{the-only } \text{dfault } xq$ in

if $x = y$ then x else *dfault* $()$)

lemma *the-only-singleton*:

the-only *dfault* $xq = \text{singleton } \text{dfault } (\text{pred-of-seq } xq)$

<proof>

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 $y = \text{singleton } \text{dfault } P$ 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 $x = \text{singleton } \text{dfault } P$; $y = \text{the-only } \text{dfault } xq$ in

if $x = y$ then x else *dfault* $()$)

<proof>

definition *not-unique* :: 'a pred => 'a

where

[code del]: *not-unique* A = (THE x. eval A x)

definition *the* :: 'a pred => 'a

where

[code del]: *the* A = (THE x. eval A x)

lemma *the-eq*[code]: *the* A = singleton ($\lambda x.$ *not-unique* A) A
 <proof>

code-abort *not-unique*

code-reflect *Predicate*

datatypes *pred* = Seq **and** *seq* = Empty | Insert | Join

functions *map*

<ML>

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

$rtrancl :: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \quad ((-^*) [1000] 999)$
for $r :: ('a \times 'a) \text{ set}$

where

$rtrancl\text{-}refl \text{ [intro!, Pure.intro!, simp]: } (a, a) : r^*$
 $| \text{ } rtrancl\text{-}into\text{-}rtrancl \text{ [Pure.intro]: } (a, b) : r^* \Rightarrow (b, c) : r \Rightarrow (a, c) : r^*$

inductive-set

$trancl :: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \quad ((-^+) [1000] 999)$
for $r :: ('a \times 'a) \text{ set}$

where

$r\text{-}into\text{-}trancl \text{ [intro, Pure.intro]: } (a, b) : r \Rightarrow (a, b) : r^+$
 $| \text{ } trancl\text{-}into\text{-}trancl \text{ [Pure.intro]: } (a, b) : r^+ \Rightarrow (b, c) : r \Rightarrow (a, c) : r^+$

declare $rtrancl\text{-}def \text{ [nitpick-def del]}$

$rtranclp\text{-}def \text{ [nitpick-def del]}$
 $trancl\text{-}def \text{ [nitpick-def del]}$
 $tranclp\text{-}def \text{ [nitpick-def del]}$

notation

$rtranclp \quad ((-^{**}) [1000] 1000) \text{ and}$
 $tranclp \quad ((-^{++}) [1000] 1000)$

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

$reflcl :: ('a \times 'a) \text{ set} \Rightarrow ('a \times 'a) \text{ set} \quad ((-^{\hat{=}}) [1000] 999) \text{ where}$
 $r^{\hat{=}} == r \cup Id$

notation (*xsymbols*)

$rtranclp \quad ((-^{**}) [1000] 1000) \text{ and}$
 $tranclp \quad ((-^{++}) [1000] 1000) \text{ and}$
 $reflclp \quad ((-^{\hat{=}}) [1000] 1000) \text{ and}$
 $rtrancl \quad ((-^*) [1000] 999) \text{ and}$
 $trancl \quad ((-^+) [1000] 999) \text{ and}$
 $reflcl \quad ((-^=) [1000] 999)$

notation (*HTML output*)

$rtranclp \quad ((-^{**}) [1000] 1000) \text{ and}$
 $tranclp \quad ((-^{++}) [1000] 1000) \text{ and}$
 $reflclp \quad ((-^{\hat{=}}) [1000] 1000) \text{ and}$
 $rtrancl \quad ((-^*) [1000] 999) \text{ and}$
 $trancl \quad ((-^+) [1000] 999) \text{ and}$
 $reflcl \quad ((-^=) [1000] 999)$

23.1 Reflexive closure

lemma *refl-reflcl[simp]*: $\text{refl}(r^{\hat{=}})$
 $\langle \text{proof} \rangle$

lemma *antisym-reflcl[simp]*: $\text{antisym}(r^{\hat{=}}) = \text{antisym } r$
 $\langle \text{proof} \rangle$

lemma *trans-reflclI[simp]*: $\text{trans } r \implies \text{trans}(r^{\hat{=}})$
 $\langle \text{proof} \rangle$

23.2 Reflexive-transitive closure

lemma *reflcl-set-eq [pred-set-conv]*: $(\text{sup } (\lambda x y. (x, y) \in r) \text{ op } =) = (\lambda x y. (x, y) \in r \cup \text{Id})$
 $\langle \text{proof} \rangle$

lemma *r-into-rtrancl [intro]*: $!!p. p \in r \implies p \in r^{\hat{*}}$
 — *rtrancl* of *r* contains *r*
 $\langle \text{proof} \rangle$

lemma *r-into-rtranclp [intro]*: $r \ x \ y \implies r^{\hat{*}*} \ x \ y$
 — *rtrancl* of *r* contains *r*
 $\langle \text{proof} \rangle$

lemma *rtranclp-mono*: $r \leq s \implies r^{\hat{*}*} \leq s^{\hat{*}*}$
 — monotonicity of *rtrancl*
 $\langle \text{proof} \rangle$

lemmas *rtrancl-mono = rtranclp-mono [to-set]*

theorem *rtranclp-induct [consumes 1, case-names base step, induct set: rtranclp]*:
 assumes *a*: $r^{\hat{*}*} \ a \ b$
 and cases: $P \ a \ !!y \ z. [\mid r^{\hat{*}*} \ a \ y; r \ y \ z; P \ y] \implies P \ z$
 shows $P \ b$ $\langle \text{proof} \rangle$

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*: $\text{refl } (r^{\hat{*}})$
 $\langle \text{proof} \rangle$

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* (r^*)
 $\langle proof \rangle$

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD, standard*]

lemma *rtranclp-trans*:
 assumes $xy: r^{**} x y$
 and $yz: r^{**} y z$
 shows $r^{**} x z$ $\langle proof \rangle$

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
 $\langle proof \rangle$

lemma *rtrancl-Int-subset*: $[Id \subseteq s; (r^* \cap s) O r \subseteq s] \implies r^* \subseteq s$
 $\langle proof \rangle$

lemma *converse-rtranclp-into-rtranclp*:
 $r a b \implies r^{**} b c \implies r^{**} a c$
 $\langle proof \rangle$

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [*to-set*]

More r^* equations and inclusions.

lemma *rtranclp-idemp* [*simp*]: $(r^{**})^{**} = r^{**}$
 $\langle proof \rangle$

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

lemma *rtrancl-idemp-self-comp* [*simp*]: $R^* O R^* = R^*$
 $\langle proof \rangle$

lemma *rtrancl-subset-rtrancl*: $r \subseteq s^* \implies r^* \subseteq s^*$
 $\langle proof \rangle$

lemma *rtranclp-subset*: $R \leq S \implies S \leq R^{**} \implies S^{**} = R^{**}$
 $\langle proof \rangle$

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(\sup (R^{**}) (S^{**}))^{**} = (\sup R S)^{**}$
 $\langle proof \rangle$

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflcl* [simp]: $(R^{\hat{}} ==)^{\hat{}}_{**} = R^{\hat{}}_{**}$
 ⟨proof⟩

lemmas *rtrancl-reflcl* [simp] = *rtranclp-reflcl* [to-set]

lemma *rtrancl-r-diff-Id*: $(r - Id)^{\hat{}}_* = r^{\hat{}}_*$
 ⟨proof⟩

lemma *rtranclp-r-diff-Id*: $(\inf r \text{ op } \sim)^{\hat{}}_{**} = r^{\hat{}}_{**}$
 ⟨proof⟩

theorem *rtranclp-converseD*:
 assumes $r: (r^{\hat{}} - 1)^{\hat{}}_{**} x y$
 shows $r^{\hat{}}_{**} y x$
 ⟨proof⟩

lemmas *rtrancl-converseD* = *rtranclp-converseD* [to-set]

theorem *rtranclp-converseI*:
 assumes $r^{\hat{}}_{**} y x$
 shows $(r^{\hat{}} - 1)^{\hat{}}_{**} x y$
 ⟨proof⟩

lemmas *rtrancl-converseI* = *rtranclp-converseI* [to-set]

lemma *rtrancl-converse*: $(r^{\hat{}} - 1)^{\hat{}}_* = (r^{\hat{}}_*)^{\hat{}} - 1$
 ⟨proof⟩

lemma *sym-rtrancl*: $\text{sym } r ==> \text{sym } (r^{\hat{}}_*)$
 ⟨proof⟩

theorem *converse-rtranclp-induct* [consumes 1, case-names base step]:
 assumes major: $r^{\hat{}}_{**} a b$
 and cases: $P b !!y z. [| r y z; r^{\hat{}}_{**} z b; P z |] ==> P y$
 shows $P a$
 ⟨proof⟩

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: $r^{\hat{}}_{**} x z$

and *cases*: $x=z \implies P$
 $!!y. [\mid r\ x\ y; r^{\wedge**}\ y\ z\ \mid] \implies P$
shows P
 $\langle proof \rangle$

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 $r-comp-rtrancl-eq$: $r\ O\ r^{\wedge*} = r^{\wedge*}\ O\ r$
 $\langle proof \rangle$

lemma $rtrancl-unfold$: $r^{\wedge*} = Id\ Un\ r^{\wedge*}\ O\ r$
 $\langle proof \rangle$

lemma $rtrancl-Un-separatorE$:
 $(a,b) : (P \cup Q)^{\wedge*} \implies \forall x\ y. (a,x) : P^{\wedge*} \longrightarrow (x,y) : Q \longrightarrow x=y \implies (a,b) : P^{\wedge*}$
 $\langle proof \rangle$

lemma $rtrancl-Un-separator-converseE$:
 $(a,b) : (P \cup Q)^{\wedge*} \implies \forall x\ y. (x,b) : P^{\wedge*} \longrightarrow (y,x) : Q \longrightarrow y=x \implies (a,b) : P^{\wedge*}$
 $\langle proof \rangle$

lemma $Image-closed-trancl$:
assumes $r\ \text{“}\ X \subseteq X\ \text{”}$ **shows** $r^*\ \text{“}\ X = X\ \text{”}$
 $\langle proof \rangle$

23.3 Transitive closure

lemma $trancl-mono$: $!!p. p \in r^{\wedge+} \implies r \subseteq s \implies p \in s^{\wedge+}$
 $\langle proof \rangle$

lemma $r-into-trancl'$: $!!p. p : r \implies p : r^{\wedge+}$
 $\langle proof \rangle$

Conversions between $trancl$ and $rtrancl$.

lemma $tranclp-into-rtranclp$: $r^{\wedge++}\ a\ b \implies r^{\wedge**}\ a\ b$
 $\langle proof \rangle$

lemmas $trancl-into-rtrancl = tranclp-into-rtranclp\ [to-set]$

lemma $rtranclp-into-tranclp1$: **assumes** $r: r^{\wedge**}\ a\ b$
shows $!!c. r\ b\ c \implies r^{\wedge++}\ a\ c$ $\langle proof \rangle$

lemmas $rtrancl-into-trancl1 = rtranclp-into-tranclp1\ [to-set]$

lemma $rtranclp-into-tranclp2$: $[\mid r\ a\ b; r^{\wedge**}\ b\ c\ \mid] \implies r^{\wedge++}\ a\ c$

— intro rule from r and $rtranc1$
 $\langle proof \rangle$

lemmas $rtranc1\text{-into-tranc12} = rtranc1p\text{-into-tranc1p2}$ [to-set]

Nice induction rule for $tranc1$

lemma $tranc1p\text{-induct}$ [consumes 1, case-names base step, induct pred: $tranc1p$]:

assumes $a: r^{++} a b$
and cases: $!!y. r a y \implies P y$
 $!!y z. r^{++} a y \implies r y z \implies P y \implies P z$
shows $P b$ $\langle proof \rangle$

lemmas $tranc1\text{-induct}$ [induct set: $tranc1$] = $tranc1p\text{-induct}$ [to-set]

lemmas $tranc1p\text{-induct2} =$
 $tranc1p\text{-induct}$ [of - (ax, ay) (bx, by) , split-rule,
consumes 1, case-names base step]

lemmas $tranc1\text{-induct2} =$
 $tranc1\text{-induct}$ [of (ax, ay) (bx, by) , split-format (complete),
consumes 1, case-names base step]

lemma $tranc1p\text{-trans-induct}$:
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
 $\langle proof \rangle$

lemmas $tranc1\text{-trans-induct} = tranc1p\text{-trans-induct}$ [to-set]

lemma $tranc1E$ [cases set: $tranc1$]:
assumes $(a, b) : r^{++}$
obtains
 $(base) (a, b) : r$
 $| (step) c \text{ where } (a, c) : r^{++} \text{ and } (c, b) : r$
 $\langle proof \rangle$

lemma $tranc1\text{-Int-subset}$: $[r \subseteq s; (r^{++} \cap s) O r \subseteq s] \implies r^{++} \subseteq s$
 $\langle proof \rangle$

lemma $tranc1\text{-unfold}$: $r^{++} = r \cup r^{++} O r$
 $\langle proof \rangle$

Transitivity of r^{+}

lemma $trans\text{-tranc1}$ [simp]: $trans (r^{++})$
 $\langle proof \rangle$

lemmas *trancl-trans* = *trans-trancl* [*THEN transD, standard*]

lemma *tranclp-trans*:

assumes $xy: r^{\hat{}}++ x y$

and $yz: r^{\hat{}}++ y z$

shows $r^{\hat{}}++ x z$ *<proof>*

lemma *trancl-id* [*simp*]: $\text{trans } r \implies r^{\hat{}}+ = r$
<proof>

lemma *rtranclp-tranclp-tranclp*:

assumes $r^{\hat{}}** x y$

shows $!!z. r^{\hat{}}++ y z \implies r^{\hat{}}++ x z$ *<proof>*

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r a b \implies r^{\hat{}}++ b c \implies r^{\hat{}}++ a c$
<proof>

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:

$(\text{insert } (y, x) r)^{\hat{}}+ = r^{\hat{}}+ \cup \{(a, b). (a, y) \in r^{\hat{}}* \wedge (x, b) \in r^{\hat{}}*\}$

— primitive recursion for *trancl* over finite relations

<proof>

lemma *tranclp-converseI*: $(r^{\hat{}}++)^{\hat{}}--1 x y \implies (r^{\hat{}}--1)^{\hat{}}++ x y$
<proof>

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*: $(r^{\hat{}}--1)^{\hat{}}++ x y \implies (r^{\hat{}}++)^{\hat{}}--1 x y$
<proof>

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r^{\hat{}}--1)^{\hat{}}++ = (r^{\hat{}}++)^{\hat{}}--1$
<proof>

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: $\text{sym } r \implies \text{sym } (r^{\hat{}}+)$
<proof>

lemma *converse-tranclp-induct* [*consumes 1, case-names base step*]:

assumes *major*: $r^{\hat{}}++ a b$

and *cases*: $!!y. r y b \implies P(y)$

$!!y z. [r y z; r^{\hat{}}++ z b; P(z)] \implies P(y)$

shows $P a$

$\langle proof \rangle$

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [to-set]

lemma *tranclpD*: $R^+ \vdash x \ y \implies \exists z. R \ x \ z \wedge R^{**} \ z \ y$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemmas *converse-tranclE* = *converse-tranclpE* [to-set]

lemma *tranclD2*:
 $(x, y) \in R^+ \implies \exists z. (x, z) \in R^* \wedge (z, y) \in R$
 $\langle proof \rangle$

lemma *irrefl-tranclI*: $r^{-1} \cap r^* = \{\}$ $\implies (x, x) \notin r^+$
 $\langle proof \rangle$

lemma *irrefl-trancl-rD*: $\forall x. (x, x) \notin r^+ \implies (x, y) \in r \implies x \neq y$
 $\langle proof \rangle$

lemma *trancl-subset-Sigma-aux*:
 $(a, b) \in r^* \implies r \subseteq A \times A \implies a = b \vee a \in A$
 $\langle proof \rangle$

lemma *trancl-subset-Sigma*: $r \subseteq A \times A \implies r^+ \subseteq A \times A$
 $\langle proof \rangle$

lemma *reflcl-tranclp* [simp]: $(r^+)^+ = r^{**}$
 $\langle proof \rangle$

lemmas *reflcl-trancl* [simp] = *reflcl-tranclp* [to-set]

lemma *trancl-reflcl* [simp]: $(r^+)^+ = r^*$
 $\langle proof \rangle$

lemma *trancl-empty* [simp]: $\{\}^+ = \{\}$
 $\langle proof \rangle$

lemma *rtrancl-empty* [simp]: $\{\}^* = Id$
 $\langle proof \rangle$

lemma *rtranclpD*: $R^* a b \implies a = b \vee a \neq b \wedge R^+ a b$
 ⟨proof⟩

lemmas *rtranclD* = *rtranclpD* [to-set]

lemma *rtrancl-eq-or-trancl*:
 $(x, y) \in R^* = (x = y \vee x \neq y \wedge (x, y) \in R^+)$
 ⟨proof⟩

lemma *trancl-unfold-right*: $R^+ = R^* \circ r$
 ⟨proof⟩

lemma *trancl-unfold-left*: $R^+ = r \circ R^*$
 ⟨proof⟩

Simplifying nested closures

lemma *rtrancl-trancl-absorb[simp]*: $(R^*)^+ = R^*$
 ⟨proof⟩

lemma *trancl-rtrancl-absorb[simp]*: $(R^+)^* = R^*$
 ⟨proof⟩

lemma *rtrancl-reflcl-absorb[simp]*: $(R^*)^+ = R^*$
 ⟨proof⟩

Domain and *Range*

lemma *Domain-rtrancl* [simp]: $\text{Domain } (R^*) = \text{UNIV}$
 ⟨proof⟩

lemma *Range-rtrancl* [simp]: $\text{Range } (R^*) = \text{UNIV}$
 ⟨proof⟩

lemma *rtrancl-Un-subset*: $(R^* \cup S^*) \subseteq (R \cup S)^*$
 ⟨proof⟩

lemma *in-rtrancl-UnI*: $x \in R^* \vee x \in S^* \implies x \in (R \cup S)^*$
 ⟨proof⟩

lemma *trancl-domain* [simp]: $\text{Domain } (R^+) = \text{Domain } r$
 ⟨proof⟩

lemma *trancl-range* [simp]: $\text{Range } (R^+) = \text{Range } r$
 ⟨proof⟩

lemma *Not-Domain-rtrancl*:
 $x \sim: \text{Domain } R \implies ((x, y) : R^*) = (x = y)$
 ⟨proof⟩

lemma *trancl-subset-Field2*: $R^+ \leq \text{Field } r \times \text{Field } r$

<proof>

lemma *finite-trancl*: *finite* (r^+) = *finite* r
<proof>

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
 $\llbracket \text{single-valued } r; (x,y) \in r^*; (x,z) \in r^* \rrbracket$
 $\implies (y,z) \in r^* \vee (z,y) \in r^*$
<proof>

lemma *r-r-into-trancl*: $(a, b) \in R \implies (b, c) \in R \implies (a, c) \in R^+$
<proof>

lemma *trancl-into-trancl* [rule-format]:
 $(a, b) \in r^+ \implies (b, c) \in r \dashrightarrow (a, c) \in r^+$
<proof>

lemma *tranclp-rtranclp-tranclp*:
 $r^{++} a b \implies r^{**} b c \implies r^{++} a c$
<proof>

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

$R^{\wedge n} = R \circ \dots \circ R$, 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) $R = (R^{\wedge n}) \circ R$

end

lemma *rel-pow-1* [*simp*]:

fixes $R :: ('a \times 'a) \text{ set}$

shows $R^{\wedge} 1 = R$

<proof>

lemma *rel-pow-0-I*:

$(x, x) \in R^{\wedge} 0$

<proof>

lemma *rel-pow-Suc-I*:

$(x, y) \in R^{\wedge} n \implies (y, z) \in R \implies (x, z) \in R^{\wedge} \text{Suc } n$

<proof>

lemma *rel-pow-Suc-I2*:

$(x, y) \in R \implies (y, z) \in R^{\wedge} n \implies (x, z) \in R^{\wedge} \text{Suc } n$

<proof>

lemma *rel-pow-0-E*:

$(x, y) \in R^{\wedge} 0 \implies (x = y \implies P) \implies P$

<proof>

lemma *rel-pow-Suc-E*:

$(x, z) \in R^{\wedge} \text{Suc } n \implies (\bigwedge y. (x, y) \in R^{\wedge} n \implies (y, z) \in R \implies P) \implies P$

<proof>

lemma *rel-pow-E*:

$(x, z) \in R^{\wedge} n \implies (n = 0 \implies x = z \implies P)$

$\implies (\bigwedge y m. n = \text{Suc } m \implies (x, y) \in R^{\wedge} m \implies (y, z) \in R \implies P)$

$\implies P$

<proof>

lemma *rel-pow-Suc-D2*:

$(x, z) \in R^{\wedge} \text{Suc } n \implies (\exists y. (x, y) \in R \wedge (y, z) \in R^{\wedge} n)$

<proof>

lemma *rel-pow-Suc-E2*:

$(x, z) \in R^{\wedge} \text{Suc } n \implies (\bigwedge y. (x, y) \in R \implies (y, z) \in R^{\wedge} n \implies P) \implies P$

<proof>

lemma *rel-pow-Suc-D2'*:

$\forall x y z. (x, y) \in R^{\wedge} n \wedge (y, z) \in R \longrightarrow (\exists w. (x, w) \in R \wedge (w, z) \in R^{\wedge} n)$

<proof>

lemma *rel-pow-E2*:

$(x, z) \in R^{\wedge} n \implies (n = 0 \implies x = z \implies P)$

$\implies (\bigwedge y m. n = \text{Suc } m \implies (x, y) \in R \implies (y, z) \in R^{\wedge} m \implies P)$

$\implies P$

$\langle proof \rangle$

lemma *rel-pow-add*: $R^{\wedge\wedge} (m+n) = R^{\wedge\wedge m} \circ R^{\wedge\wedge n}$
 $\langle proof \rangle$

lemma *rel-pow-commute*: $R \circ R^{\wedge\wedge n} = R^{\wedge\wedge n} \circ R$
 $\langle proof \rangle$

lemma *rtranc1-imp-UN-rel-pow*:
 assumes $p \in R^{\wedge*}$
 shows $p \in (\bigcup n. R^{\wedge\wedge n})$
 $\langle proof \rangle$

lemma *rel-pow-imp-rtranc1*:
 assumes $p \in R^{\wedge\wedge n}$
 shows $p \in R^{\wedge*}$
 $\langle proof \rangle$

lemma *rtranc1-is-UN-rel-pow*:
 $R^{\wedge*} = (\bigcup n. R^{\wedge\wedge n})$
 $\langle proof \rangle$

lemma *rtranc1-power*:
 $p \in R^{\wedge*} \longleftrightarrow (\exists n. p \in R^{\wedge\wedge n})$
 $\langle proof \rangle$

lemma *tranc1-power*:
 $p \in R^{\wedge+} \longleftrightarrow (\exists n > 0. p \in R^{\wedge\wedge n})$
 $\langle proof \rangle$

lemma *rtranc1-imp-rel-pow*:
 $p \in R^{\wedge*} \implies \exists n. p \in R^{\wedge\wedge n}$
 $\langle proof \rangle$

lemma *single-valued-rel-pow*:
 fixes $R :: ('a * 'a) \text{ set}$
 shows *single-valued* $R \implies \text{single-valued } (R^{\wedge\wedge n})$
 $\langle proof \rangle$

23.5 Setup of transitivity reasoner

$\langle ML \rangle$

Optional methods.

$\langle ML \rangle$

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
  <proof>

```

```

lemma wfUNIVI:
  (!!P x. (ALL y. (ALL y. (y,x) : r --> P(y)) --> P(x)) ==> P(x)) ==>
  wf(r)
  <proof>

```

```

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 ⊆ A <*> B;
    !!x P. [|∀x. (∀y. (y,x) : r --> P y) --> P x; x : A; x : B |] ==> P x |]
  ==> wf r
  <proof>

```

```

lemma wf-induct:
  [| wf(r);
    !!x. [| ALL y. (y,x): r --> P(y) |] ==> P(x)
    |] ==> P(a)
  <proof>

```

```

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) ~: r
  <proof>

```

lemma *wf-asym*:

assumes $wf\ r\ (a, x) \in r$

obtains $(x, a) \notin r$

$\langle proof \rangle$

lemma *wf-not-refl [simp]*: $wf\ r \implies (a, a) \sim: r$

$\langle proof \rangle$

lemma *wf-irrefl*: **assumes** $wf\ r$ **obtains** $(a, a) \notin r$

$\langle proof \rangle$

lemma *wf-wellorderI*:

assumes $wf: wf\ \{(x::'a::ord, y). x < y\}$

assumes $lin: OFCLASS('a::ord, linorder-class)$

shows $OFCLASS('a::ord, wellorder-class)$

$\langle proof \rangle$

lemma (**in** *wellorder*) *wf*:

$wf\ \{(x, y). x < y\}$

$\langle proof \rangle$

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 $A = \{\}$

$\langle proof \rangle$

lemma *wfI-pf*:

assumes $a: \bigwedge A. A \subseteq R \text{ “ } A \implies A = \{\}$

shows $wf\ R$

$\langle proof \rangle$

Minimal-element characterization of well-foundedness

lemma *wfE-min*:

assumes $wf: wf\ R$ **and** $Q: x \in Q$

obtains z **where** $z \in Q \wedge y. (y, z) \in R \implies y \notin Q$

$\langle proof \rangle$

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 $wf\ R$

$\langle proof \rangle$

lemma *wf-eq-minimal*: $wf\ r = (\forall Q\ x. x \in Q \longrightarrow (\exists z \in Q. \forall y. (y, z) \in r \longrightarrow y \notin Q))$

$\langle proof \rangle$

lemmas $wfP\text{-}eq\text{-}minimal = wf\text{-}eq\text{-}minimal$ $[to\text{-}pred]$

Well-foundedness of transitive closure

lemma $wf\text{-}trancl$:

assumes $wf\ r$

shows $wf\ (r^+)$

$\langle proof \rangle$

lemmas $wfP\text{-}trancl = wf\text{-}trancl$ $[to\text{-}pred]$

lemma $wf\text{-}converse\text{-}trancl$: $wf\ (r^-1) ==> wf\ ((r^+)^-1)$

$\langle proof \rangle$

Well-foundedness of subsets

lemma $wf\text{-}subset$: $[| wf\ (r); p \leq r |] ==> wf\ (p)$

$\langle proof \rangle$

lemmas $wfP\text{-}subset = wf\text{-}subset$ $[to\text{-}pred]$

Well-foundedness of the empty relation

lemma $wf\text{-}empty$ $[iff]$: $wf\ \{\}$

$\langle proof \rangle$

lemma $wfP\text{-}empty$ $[iff]$:

$wfP\ (\lambda x\ y. False)$

$\langle proof \rangle$

lemma $wf\text{-}Int1$: $wf\ r ==> wf\ (r\ Int\ r')$

$\langle proof \rangle$

lemma $wf\text{-}Int2$: $wf\ r ==> wf\ (r'\ Int\ r)$

$\langle proof \rangle$

Exponentiation

lemma $wf\text{-}exp$:

assumes $wf\ (R^{\wedge n})$

shows $wf\ R$

$\langle proof \rangle$

Well-foundedness of insert

lemma $wf\text{-}insert$ $[iff]$: $wf\ (insert\ (y,x)\ r) = (wf\ (r) \ \&\ (x,y) \sim: r^*)$

$\langle proof \rangle$

Well-foundedness of image

lemma $wf\text{-}prod\text{-}fun\text{-}image$: $[| wf\ r; inj\ f |] ==> wf\ (prod\text{-}fun\ f\ f\ 'r)$

$\langle proof \rangle$

24.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:

assumes $wf\ R\ wf\ S$
assumes $R\ O\ S \subseteq R$
shows $wf\ (R \cup S)$

<proof>

Well-foundedness of indexed union with disjoint domains and ranges

lemma *wf-UN*: $[[\ ALL\ i:I.\ wf\ (r\ i);$

$\ ALL\ i:I.\ ALL\ j:I.\ r\ i \sim = r\ j \longrightarrow Domain(r\ i)\ Int\ Range(r\ j) = \{\}$

$]] \implies wf\ (UN\ i:I.\ r\ i)$

<proof>

lemma *wfP-SUP*:

$\forall i.\ wfP\ (r\ i) \implies \forall i\ j.\ r\ i \neq r\ j \longrightarrow \inf\ (DomainP\ (r\ i))\ (RangeP\ (r\ j)) = bot$
 $\implies wfP\ (SUPR\ UNIV\ r)$

<proof>

lemma *wf-Union*:

$[[\ ALL\ r:R.\ wf\ r;$

$\ ALL\ r:R.\ ALL\ s:R.\ r \sim = s \longrightarrow Domain\ r\ Int\ Range\ s = \{\}$

$]] \implies wf\ (Union\ R)$

<proof>

lemma *wf-Un*:

$[[\ wf\ r; wf\ s; Domain\ r\ Int\ Range\ s = \{\}] \implies wf\ (r\ Un\ s)$

<proof>

lemma *wf-union-merge*:

$wf\ (R \cup S) = wf\ (R\ O\ R \cup S\ O\ R \cup S)$ (**is** $wf\ ?A = wf\ ?B$)

<proof>

lemma *wf-comp-self*: $wf\ R = wf\ (R\ O\ R)$ — special case

<proof>

24.4 Acyclic relations

definition *acyclic* :: $('a * 'a)\ set \Rightarrow bool$ **where**

$acyclic\ r == !x.\ (x,x) \sim: r^{\wedge+}$

abbreviation *acyclicP* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**

$acyclicP\ r == acyclic\ \{(x,y).\ r\ x\ y\}$

lemma *acyclicI*: $ALL\ x.\ (x,x) \sim: r^{\wedge+} \implies acyclic\ r$

<proof>

lemma *wf-acyclic*: $wf\ r \implies acyclic\ r$

<proof>

lemmas $wfP\text{-}acyclicP = wf\text{-}acyclic$ $[to\text{-}pred]$

lemma $acyclic\text{-}insert$ $[iff]$:
 $acyclic(insert\ (y,x)\ r) = (acyclic\ r \ \&\ (x,y) \sim : r^{\wedge*})$
 $\langle proof \rangle$

lemma $acyclic\text{-}converse$ $[iff]$: $acyclic(r^{\wedge-1}) = acyclic\ r$
 $\langle proof \rangle$

lemmas $acyclicP\text{-}converse$ $[iff] = acyclic\text{-}converse$ $[to\text{-}pred]$

lemma $acyclic\text{-}impl\text{-}antisym\text{-}rtrancl$: $acyclic\ r ==> antisym(r^{\wedge*})$
 $\langle proof \rangle$

lemma $acyclic\text{-}subset$: $[| acyclic\ s; r \leq s |] ==> acyclic\ r$
 $\langle proof \rangle$

Wellfoundedness of finite acyclic relations

lemma $finite\text{-}acyclic\text{-}wf$ $[rule\text{-}format]$: $finite\ r ==> acyclic\ r --> wf\ r$
 $\langle proof \rangle$

lemma $finite\text{-}acyclic\text{-}wf\text{-}converse$: $[|finite\ r; acyclic\ r|] ==> wf\ (r^{\wedge-1})$
 $\langle proof \rangle$

lemma $wf\text{-}iff\text{-}acyclic\text{-}if\text{-}finite$: $finite\ r ==> wf\ r = acyclic\ r$
 $\langle proof \rangle$

24.5 nat is well-founded

lemma $less\text{-}nat\text{-}rel$: $op < = (\lambda m\ n. n = Suc\ m)^{\wedge++}$
 $\langle proof \rangle$

definition

$pred\text{-}nat :: (nat * nat) \text{ set } \mathbf{where}$
 $pred\text{-}nat = \{(m, n). n = Suc\ m\}$

definition

$less\text{-}than :: (nat * nat) \text{ set } \mathbf{where}$
 $less\text{-}than = pred\text{-}nat^{\wedge+}$

lemma $less\text{-}eq$: $(m, n) \in pred\text{-}nat^{\wedge+} \longleftrightarrow m < n$
 $\langle proof \rangle$

lemma $pred\text{-}nat\text{-}trancl\text{-}eq\text{-}le$:

$(m, n) \in pred\text{-}nat^{\wedge*} \longleftrightarrow m \leq n$
 $\langle proof \rangle$

lemma *wf-pred-nat*: *wf pred-nat*
 ⟨*proof*⟩

lemma *wf-less-than* [*iff*]: *wf less-than*
 ⟨*proof*⟩

lemma *trans-less-than* [*iff*]: *trans less-than*
 ⟨*proof*⟩

lemma *less-than-iff* [*iff*]: $((x,y): \text{less-than}) = (x < y)$
 ⟨*proof*⟩

lemma *wf-less*: *wf* $\{(x, y::\text{nat}). x < y\}$
 ⟨*proof*⟩

24.6 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [?].

inductive-set

acc :: (*'a* * *'a*) *set* ==> *'a set*
for *r* :: (*'a* * *'a*) *set*
where
accI: $(!!y. (y, x) : r ==> y : \text{acc } r) ==> x : \text{acc } r$

abbreviation

termip :: (*'a* ==> *'a* ==> *bool*) ==> *'a* ==> *bool* **where**
termip *r* == *accp* (r^{-1-1})

abbreviation

termi :: (*'a* * *'a*) *set* ==> *'a set* **where**
termi *r* == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

Induction rules

theorem *accp-induct*:

assumes *major*: *accp r a*
assumes *hyp*: $!!x. \text{accp } r \ x ==> \forall y. r \ y \ x --> P \ y ==> P \ x$
shows *P a*
 ⟨*proof*⟩

theorems *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b ==> r a b ==> accp r a*
 ⟨*proof*⟩

lemma *not-accp-down*:

assumes *na*: $\neg \text{accp } R \ x$

obtains z **where** $R\ z\ x$ **and** $\neg\ accp\ R\ z$
 $\langle proof \rangle$

lemma *accp-downwards-aux*: $r^{**}\ b\ a \implies accp\ r\ a \dashv\dashv accp\ r\ b$
 $\langle proof \rangle$

theorem *accp-downwards*: $accp\ r\ a \implies r^{**}\ b\ a \implies accp\ r\ b$
 $\langle proof \rangle$

theorem *accp-wfPI*: $\forall x. accp\ r\ x \implies wfP\ r$
 $\langle proof \rangle$

theorem *accp-wfPD*: $wfP\ r \implies accp\ r\ x$
 $\langle proof \rangle$

theorem *wfP-accp-iff*: $wfP\ r = (\forall x. accp\ r\ x)$
 $\langle proof \rangle$

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes *sub*: $R1 \leq R2$
shows $accp\ R2 \leq accp\ R1$
 $\langle proof \rangle$

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: $D \leq accp\ R$
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 $P\ x$
 $\langle proof \rangle$

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-wfI* = *accp-wfPI* [*to-set*]

lemmas $acc\text{-}wfD = accp\text{-}wfPD$ $[to\text{-}set]$

lemmas $wf\text{-}acc\text{-}iff = wfP\text{-}accp\text{-}iff$ $[to\text{-}set]$

lemmas $acc\text{-}subset = accp\text{-}subset$ $[to\text{-}set\ pred\text{-}subset\text{-}eq]$

lemmas $acc\text{-}subset\text{-}induct = accp\text{-}subset\text{-}induct$ $[to\text{-}set\ pred\text{-}subset\text{-}eq]$

24.7 Tools for building wellfounded relations

Inverse Image

lemma $wf\text{-}inv\text{-}image$ $[simp,intro!]$: $wf(r) \implies wf(inv\text{-}image\ r\ (f::'a \Rightarrow 'b))$
 $\langle proof \rangle$

Measure functions into nat

definition $measure :: ('a \Rightarrow nat) \Rightarrow ('a * 'a) set$
where $measure == inv\text{-}image\ less\text{-}than$

lemma $in\text{-}measure[simp]$: $((x,y) : measure\ f) = (f\ x < f\ y)$
 $\langle proof \rangle$

lemma $wf\text{-}measure$ $[iff]$: $wf\ (measure\ f)$
 $\langle proof \rangle$

Lexicographic combinations

definition
 $lex\text{-}prod :: [('a * 'a) set, ('b * 'b) set] \Rightarrow (('a * 'b) * ('a * 'b)) set$
 $(\mathbf{infixr}\ <*lex*>\ 80)$

where

$ra\ <*lex*>\ rb == \{((a,b),(a',b')).\ (a,a') : ra \mid a=a' \ \&\ (b,b') : rb\}$

lemma $wf\text{-}lex\text{-}prod$ $[intro!]$: $[| wf(ra); wf(rb) |] \implies wf(ra\ <*lex*>\ rb)$
 $\langle proof \rangle$

lemma $in\text{-}lex\text{-}prod[simp]$:
 $((a,b),(a',b')) : r\ <*lex*>\ s = ((a,a') : r \vee (a = a' \wedge (b, b') : s))$
 $\langle proof \rangle$

$op\ <*lex*>$ preserves transitivity

lemma $trans\text{-}lex\text{-}prod$ $[intro!]$:
 $[| trans\ R1; trans\ R2 |] \implies trans\ (R1\ <*lex*>\ R2)$
 $\langle proof \rangle$

lexicographic combinations with measure functions

definition
 $mlex\text{-}prod :: ('a \Rightarrow nat) \Rightarrow ('a \times 'a) set \Rightarrow ('a \times 'a) set\ (\mathbf{infixr}\ <*mlex*>\ 80)$
where
 $f\ <*mlex*>\ R = inv\text{-}image\ (less\text{-}than\ <*lex*>\ R)\ (\%x.\ (f\ x,\ x))$

lemma *wf-mlex*: $wf\ R \implies wf\ (f\ <{*mlex*}\ R)$
 $\langle proof \rangle$

lemma *mlex-less*: $f\ x < f\ y \implies (x, y) \in f\ <{*mlex*}\ R$
 $\langle proof \rangle$

lemma *mlex-leq*: $f\ x \leq f\ y \implies (x, y) \in R \implies (x, y) \in f\ <{*mlex*}\ R$
 $\langle proof \rangle$

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]*: $wf(finite-psubset)$
 $\langle proof \rangle$

lemma *trans-finite-psubset*: $trans\ finite-psubset$
 $\langle proof \rangle$

lemma *in-finite-psubset[simp]*: $(A, B) \in finite-psubset = (A < B \ \& \ finite\ B)$
 $\langle proof \rangle$

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 $wf\ (max-ext\ r)$
 $\langle proof \rangle$

lemma *max-ext-additive*:
 $(A, B) \in max-ext\ R \implies (C, D) \in max-ext\ R \implies$
 $(A \cup C, B \cup D) \in max-ext\ R$
 $\langle proof \rangle$

definition

min-ext :: $('a \times 'a)\ set \Rightarrow ('a\ set \times 'a\ set)\ set$

where

[code del]: $min-ext\ r = \{(X, Y) \mid X\ Y. X \neq \{\} \wedge (\forall y \in Y. (\exists x \in X. (x, y) \in r))\}$

lemma *min-ext-wf*:
assumes *wf* $wf\ r$

shows $wf \ (min-ext \ r)$
 $\langle proof \rangle$

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*: $[| \ ALL \ i. \ (f \ (Suc \ i), \ f \ i) : r^* \ |] \implies (f \ (i+k), \ f \ i) : r^*$
 $\langle proof \rangle$

lemma *wf-weak-decr-stable*:
assumes *as*: $\ALL \ i. \ (f \ (Suc \ i), \ f \ i) : r^* \ wf \ (r^+)$
shows $EX \ i. \ \ALL \ k. \ f \ (i+k) = f \ i$
 $\langle proof \rangle$

lemma *weak-decr-stable*:
 $\ALL \ i. \ f \ (Suc \ i) <= ((f \ i)::nat) \implies EX \ i. \ \ALL \ k. \ f \ (i+k) = f \ i$
 $\langle proof \rangle$

24.9 size of a datatype value

$\langle ML \rangle$

lemma *size-bool* [*code*]:
 $size \ (b::bool) = 0 \ \langle proof \rangle$

lemma *nat-size* [*simp*, *code*]: $size \ (n::nat) = n$
 $\langle proof \rangle$

declare *prod.size* [*no-atp*]

lemma [*code*]:
 $size \ (P :: 'a \ Predicate.pred) = 0 \ \langle proof \rangle$

lemma [*code*]:
 $pred-size \ f \ P = 0 \ \langle proof \rangle$

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

$THE\text{-}default :: 'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
 $THE\text{-}default\ d\ P = (if\ (\exists!x. P\ x)\ then\ (THE\ x. P\ x)\ else\ d)$

lemma $THE\text{-}defaultI'$: $\exists!x. P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
 $\langle proof \rangle$

lemma $THE\text{-}default1\text{-}equality$:

$\llbracket \exists!x. P\ x; P\ a \rrbracket \Longrightarrow THE\text{-}default\ d\ P = a$
 $\langle proof \rangle$

lemma $THE\text{-}default\text{-}none$:

$\neg(\exists!x. P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
 $\langle proof \rangle$

lemma $fundef\text{-}ex1\text{-}existence$:

assumes $f\text{-}def$: $f == (\lambda x::'a. THE\text{-}default\ (d\ x)\ (\lambda y. G\ x\ y))$
assumes $ex1$: $\exists!y. G\ x\ y$
shows $G\ x\ (f\ x)$
 $\langle proof \rangle$

lemma $fundef\text{-}ex1\text{-}uniqueness$:

assumes $f\text{-}def$: $f == (\lambda x::'a. THE\text{-}default\ (d\ x)\ (\lambda y. G\ x\ y))$
assumes $ex1$: $\exists!y. G\ x\ y$
assumes elm : $G\ x\ (h\ x)$
shows $h\ x = f\ x$

$\langle proof \rangle$

lemma *fundef-ex1-iff*:

assumes *f-def*: $f == (\lambda x::'a. \text{THE-default } (d \ x) (\lambda y. G \ x \ y))$

assumes *ex1*: $\exists! y. G \ x \ y$

shows $(G \ x \ y) = (f \ x = y)$

$\langle proof \rangle$

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 $f \ x = d \ x$

$\langle proof \rangle$

definition *in-rel-def*[*simp*]:

in-rel $R \ x \ y == (x, y) \in R$

lemma *wf-in-rel*:

wf $R \implies wfP \ (in-rel \ R)$

$\langle proof \rangle$

$\langle ML \rangle$

25.2 Measure Functions

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$

where *is-measure-trivial*: *is-measure* f

$\langle ML \rangle$

lemma *measure-size*[*measure-function*]: *is-measure* *size*

$\langle proof \rangle$

lemma *measure-fst*[*measure-function*]: *is-measure* $f \implies is-measure \ (\lambda p. f \ (fst \ p))$

$\langle proof \rangle$

lemma *measure-snd*[*measure-function*]: *is-measure* $f \implies is-measure \ (\lambda p. f \ (snd \ p))$

$\langle proof \rangle$

$\langle ML \rangle$

25.3 Congruence Rules

lemma *let-cong* [*fundef-cong*]:

$M = N \implies (\bigwedge x. x = N \implies f \ x = g \ x) \implies Let \ M \ f = Let \ N \ g$

$\langle proof \rangle$

lemmas [*fundef-cong*] =

if-cong *image-cong* *INT-cong* *UN-cong*

bex-cong ball-cong imp-cong

lemma *split-cong* [*fundef-cong*]:

$(\bigwedge x y. (x, y) = q \implies f x y = g x y) \implies p = q$
 $\implies \text{split } f p = \text{split } g q$
 $\langle \text{proof} \rangle$

lemma *comp-cong* [*fundef-cong*]:

$f (g x) = f' (g' x') \implies (f \circ g) x = (f' \circ g') x'$
 $\langle \text{proof} \rangle$

25.4 Simp rules for termination proofs

lemma *termination-basic-simps*[*termination-simp*]:

$x < (y::\text{nat}) \implies x < y + z$
 $x < z \implies x < y + z$
 $x \leq y \implies x \leq y + (z::\text{nat})$
 $x \leq z \implies x \leq y + (z::\text{nat})$
 $x < y \implies x \leq (y::\text{nat})$
 $\langle \text{proof} \rangle$

declare *le-imp-less-Suc*[*termination-simp*]

lemma *prod-size-simp*[*termination-simp*]:

$\text{prod-size } f g p = f (\text{fst } p) + g (\text{snd } p) + \text{Suc } 0$
 $\langle \text{proof} \rangle$

25.5 Decomposition

lemma *less-by-empty*:

$A = \{\} \implies A \subseteq B$

and *union-comp-emptyL*:

$\llbracket A \circ C = \{\}; B \circ C = \{\} \rrbracket \implies (A \cup B) \circ C = \{\}$

and *union-comp-emptyR*:

$\llbracket A \circ B = \{\}; A \circ C = \{\} \rrbracket \implies A \circ (B \cup C) = \{\}$

and *wf-no-loop*:

$R \circ R = \{\} \implies \text{wf } R$

$\langle \text{proof} \rangle$

25.6 Reduction Pairs

definition

reduction-pair $P = (\text{wf } (\text{fst } P) \wedge \text{fst } P \circ \text{snd } P \subseteq \text{fst } P)$

lemma *reduction-pairI*[*intro*]: $\text{wf } R \implies R \circ S \subseteq R \implies \text{reduction-pair } (R, S)$

$\langle \text{proof} \rangle$

lemma *reduction-pair-lemma*:

assumes *rp*: *reduction-pair* P

assumes $R \subseteq \text{fst } P$

assumes $S \subseteq \text{snd } P$
assumes $\text{wf } S$
shows $\text{wf } (R \cup S)$
 $\langle \text{proof} \rangle$

definition

$\text{rp-inv-image} = (\lambda(R, S) f. (\text{inv-image } R f, \text{inv-image } S f))$

lemma rp-inv-image-rp :

$\text{reduction-pair } P \implies \text{reduction-pair } (\text{rp-inv-image } P f)$
 $\langle \text{proof} \rangle$

25.7 Concrete orders for SCNP termination proofs

definition $\text{pair-less} = \text{less-than } <*\text{lex}*> \text{less-than}$

definition $[\text{code del}]: \text{pair-leq} = \text{pair-less}^\wedge =$

definition $\text{max-strict} = \text{max-ext pair-less}$

definition $[\text{code del}]: \text{max-weak} = \text{max-ext pair-leq} \cup \{(\{\}, \{\})\}$

definition $[\text{code del}]: \text{min-strict} = \text{min-ext pair-less}$

definition $[\text{code del}]: \text{min-weak} = \text{min-ext pair-leq} \cup \{(\{\}, \{\})\}$

lemma $\text{wf-pair-less}[\text{simp}]: \text{wf pair-less}$

$\langle \text{proof} \rangle$

Introduction rules for $\text{pair-less}/\text{pair-leq}$

lemma $\text{pair-leqI1}: a < b \implies ((a, s), (b, t)) \in \text{pair-leq}$

and $\text{pair-leqI2}: a \leq b \implies s \leq t \implies ((a, s), (b, t)) \in \text{pair-leq}$

and $\text{pair-lessI1}: a < b \implies ((a, s), (b, t)) \in \text{pair-less}$

and $\text{pair-lessI2}: a \leq b \implies s < t \implies ((a, s), (b, t)) \in \text{pair-less}$

$\langle \text{proof} \rangle$

Introduction rules for max

lemma $\text{smax-emptyI}:$

$\text{finite } Y \implies Y \neq \{\} \implies (\{\}, Y) \in \text{max-strict}$

and $\text{smax-insertI}:$

$\llbracket y \in Y; (x, y) \in \text{pair-less}; (X, Y) \in \text{max-strict} \rrbracket \implies (\text{insert } x X, Y) \in \text{max-strict}$

and $\text{wmax-emptyI}:$

$\text{finite } X \implies (\{\}, X) \in \text{max-weak}$

and $\text{wmax-insertI}:$

$\llbracket y \in YS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{max-weak} \rrbracket \implies (\text{insert } x XS, YS) \in \text{max-weak}$

$\langle \text{proof} \rangle$

Introduction rules for min

lemma $\text{smin-emptyI}:$

$X \neq \{\} \implies (X, \{\}) \in \text{min-strict}$

and $\text{smin-insertI}:$

$\llbracket x \in XS; (x, y) \in \text{pair-less}; (XS, YS) \in \text{min-strict} \rrbracket \implies (XS, \text{insert } y YS) \in \text{min-strict}$

and *wmin-emptyI*:
 $(X, \{\}) \in \text{min-weak}$
and *wmin-insertI*:
 $\llbracket x \in XS; (x, y) \in \text{pair-leq}; (XS, YS) \in \text{min-weak} \rrbracket \implies (XS, \text{insert } y \text{ } YS) \in \text{min-weak}$
 $\langle \text{proof} \rangle$

Reduction Pairs

lemma *max-ext-compat*:
assumes $R \ O \ S \subseteq R$
shows $\text{max-ext } R \ O \ (\text{max-ext } S \cup \{(\{\}, \{\})\}) \subseteq \text{max-ext } R$
 $\langle \text{proof} \rangle$

lemma *max-rpair-set*: *reduction-pair* (*max-strict*, *max-weak*)
 $\langle \text{proof} \rangle$

lemma *min-ext-compat*:
assumes $R \ O \ S \subseteq R$
shows $\text{min-ext } R \ O \ (\text{min-ext } S \cup \{(\{\}, \{\})\}) \subseteq \text{min-ext } R$
 $\langle \text{proof} \rangle$

lemma *min-rpair-set*: *reduction-pair* (*min-strict*, *min-weak*)
 $\langle \text{proof} \rangle$

25.8 Tool setup

$\langle ML \rangle$

end

26 Extraction: Program extraction for HOL

theory *Extraction*
imports *Option*
uses *Tools/rewrite-hol-proof.ML*
begin

26.1 Setup

$\langle ML \rangle$

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*))

typeof *P* \equiv *Type* (*TYPE*('P)) \implies *typeof* *Q* \equiv *Type* (*TYPE*('Q)) \implies
typeof (*P* \longrightarrow *Q*) \equiv *Type* (*TYPE*('P \Rightarrow 'Q))

($\lambda x. \text{typeof } (P\ x)$) \equiv ($\lambda x. \text{Type } (\text{TYPE}(\text{Null}))$) \implies
typeof ($\forall x. P\ x$) \equiv *Type* (*TYPE*(*Null*))

($\lambda x. \text{typeof } (P\ x)$) \equiv ($\lambda x. \text{Type } (\text{TYPE}('P))$) \implies
typeof ($\forall x::'a. P\ x$) \equiv *Type* (*TYPE*('a \Rightarrow 'P))

($\lambda x. \text{typeof } (P\ x)$) \equiv ($\lambda x. \text{Type } (\text{TYPE}(\text{Null}))$) \implies
typeof ($\exists x::'a. P\ x$) \equiv *Type* (*TYPE*('a))

($\lambda x. \text{typeof } (P\ x)$) \equiv ($\lambda x. \text{Type } (\text{TYPE}('P))$) \implies
typeof ($\exists x::'a. P\ x$) \equiv *Type* (*TYPE*('a \times 'P))

typeof *P* \equiv *Type* (*TYPE*(*Null*)) \implies *typeof* *Q* \equiv *Type* (*TYPE*(*Null*)) \implies
typeof (*P* \vee *Q*) \equiv *Type* (*TYPE*(*sumbool*))

typeof *P* \equiv *Type* (*TYPE*(*Null*)) \implies *typeof* *Q* \equiv *Type* (*TYPE*('Q)) \implies
typeof (*P* \vee *Q*) \equiv *Type* (*TYPE*('Q option))

typeof *P* \equiv *Type* (*TYPE*('P)) \implies *typeof* *Q* \equiv *Type* (*TYPE*(*Null*)) \implies
typeof (*P* \vee *Q*) \equiv *Type* (*TYPE*('P option))

typeof *P* \equiv *Type* (*TYPE*('P)) \implies *typeof* *Q* \equiv *Type* (*TYPE*('Q)) \implies
typeof (*P* \vee *Q*) \equiv *Type* (*TYPE*('P + 'Q))

typeof *P* \equiv *Type* (*TYPE*(*Null*)) \implies *typeof* *Q* \equiv *Type* (*TYPE*('Q)) \implies
typeof (*P* \wedge *Q*) \equiv *Type* (*TYPE*('Q))

typeof *P* \equiv *Type* (*TYPE*('P)) \implies *typeof* *Q* \equiv *Type* (*TYPE*(*Null*)) \implies

$$\text{typeof } (P \wedge Q) \equiv \text{Type } (\text{TYPE}('P))$$

$$\begin{aligned} \text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) &\implies \text{typeof } Q \equiv \text{Type } (\text{TYPE}('Q)) \implies \\ \text{typeof } (P \wedge Q) &\equiv \text{Type } (\text{TYPE}('P \times 'Q)) \end{aligned}$$

$$\text{typeof } (P = Q) \equiv \text{typeof } ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$$

$$\text{typeof } (x \in P) \equiv \text{typeof } P$$

26.3 Realizability

realizability

$$(\text{realizes } t \text{ (Trueprop } P)) \equiv (\text{Trueprop } (\text{realizes } t \text{ } P))$$

$$\begin{aligned} (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) &\implies \\ (\text{realizes } t \text{ } (P \longrightarrow Q)) &\equiv (\text{realizes } \text{Null } P \longrightarrow \text{realizes } t \text{ } Q) \end{aligned}$$

$$\begin{aligned} (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}('P))) &\implies \\ (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) &\implies \\ (\text{realizes } t \text{ } (P \longrightarrow Q)) &\equiv (\forall x::'P. \text{realizes } x \text{ } P \longrightarrow \text{realizes } \text{Null } Q) \end{aligned}$$

$$(\text{realizes } t \text{ } (P \longrightarrow Q)) \equiv (\forall x. \text{realizes } x \text{ } P \longrightarrow \text{realizes } (t \text{ } x) \text{ } Q)$$

$$\begin{aligned} (\lambda x. \text{typeof } (P \text{ } x)) &\equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\ (\text{realizes } t \text{ } (\forall x. P \text{ } x)) &\equiv (\forall x. \text{realizes } \text{Null } (P \text{ } x)) \end{aligned}$$

$$(\text{realizes } t \text{ } (\forall x. P \text{ } x)) \equiv (\forall x. \text{realizes } (t \text{ } x) \text{ } (P \text{ } x))$$

$$\begin{aligned} (\lambda x. \text{typeof } (P \text{ } x)) &\equiv (\lambda x. \text{Type } (\text{TYPE}(\text{Null}))) \implies \\ (\text{realizes } t \text{ } (\exists x. P \text{ } x)) &\equiv (\text{realizes } \text{Null } (P \text{ } t)) \end{aligned}$$

$$(\text{realizes } t \text{ } (\exists x. P \text{ } x)) \equiv (\text{realizes } (\text{snd } t) \text{ } (P \text{ } (\text{fst } t)))$$

$$\begin{aligned} (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) &\implies \\ (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) &\implies \\ (\text{realizes } t \text{ } (P \vee Q)) &\equiv \\ (\text{case } t \text{ of Left} \Rightarrow \text{realizes } \text{Null } P \mid \text{Right} \Rightarrow \text{realizes } \text{Null } Q) \end{aligned}$$

$$\begin{aligned} (\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) &\implies \\ (\text{realizes } t \text{ } (P \vee Q)) &\equiv \\ (\text{case } t \text{ of None} \Rightarrow \text{realizes } \text{Null } P \mid \text{Some } q \Rightarrow \text{realizes } q \text{ } Q) \end{aligned}$$

$$\begin{aligned} (\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) &\implies \\ (\text{realizes } t \text{ } (P \vee Q)) &\equiv \\ (\text{case } t \text{ of None} \Rightarrow \text{realizes } \text{Null } Q \mid \text{Some } p \Rightarrow \text{realizes } p \text{ } P) \end{aligned}$$

$$\begin{aligned} (\text{realizes } t \text{ } (P \vee Q)) &\equiv \\ (\text{case } t \text{ of Inl } p \Rightarrow \text{realizes } p \text{ } P \mid \text{Inr } q \Rightarrow \text{realizes } q \text{ } Q) \end{aligned}$$

$$\begin{aligned}
&(\text{typeof } P) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
&\quad (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } \text{Null } P \wedge \text{realizes } t \ Q) \\
\\
&(\text{typeof } Q) \equiv (\text{Type } (\text{TYPE}(\text{Null}))) \implies \\
&\quad (\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } t \ P \wedge \text{realizes } \text{Null } Q) \\
\\
&(\text{realizes } t \ (P \wedge Q)) \equiv (\text{realizes } (\text{fst } t) \ P \wedge \text{realizes } (\text{snd } t) \ Q) \\
\\
&\text{typeof } P \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
&\quad \text{realizes } t \ (\neg P) \equiv \neg \text{realizes } \text{Null } P \\
\\
&\text{typeof } P \equiv \text{Type } (\text{TYPE}('P)) \implies \\
&\quad \text{realizes } t \ (\neg P) \equiv (\forall x::'P. \neg \text{realizes } x \ P) \\
\\
&\text{typeof } (P::\text{bool}) \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
&\text{typeof } Q \equiv \text{Type } (\text{TYPE}(\text{Null})) \implies \\
&\quad \text{realizes } t \ (P = Q) \equiv \text{realizes } \text{Null } P = \text{realizes } \text{Null } Q \\
\\
&(\text{realizes } t \ (P = Q)) \equiv (\text{realizes } t \ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))
\end{aligned}$$

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 $r1$: $\bigwedge p. P \ p \implies R \ (f \ p)$ **and** $r2$: $\bigwedge q. Q \ q \implies R \ (g \ q)$
shows $R \ (\text{case } x \text{ of } \text{Inl } p \Rightarrow f \ p \mid \text{Inr } q \Rightarrow g \ q)$
 $\langle \text{proof} \rangle$

theorem *disjE-realizer2*:

assumes r : $\text{case } x \text{ of } \text{None} \Rightarrow P \mid \text{Some } q \Rightarrow Q \ q$
and $r1$: $P \implies R \ f$ **and** $r2$: $\bigwedge q. Q \ q \implies R \ (g \ q)$
shows $R \ (\text{case } x \text{ of } \text{None} \Rightarrow f \mid \text{Some } q \Rightarrow g \ q)$
 $\langle \text{proof} \rangle$

theorem *disjE-realizer3*:

assumes r : $\text{case } x \text{ of } \text{Left} \Rightarrow P \mid \text{Right} \Rightarrow Q$
and $r1$: $P \implies R \ f$ **and** $r2$: $Q \implies R \ g$
shows $R \ (\text{case } x \text{ of } \text{Left} \Rightarrow f \mid \text{Right} \Rightarrow g)$
 $\langle \text{proof} \rangle$

theorem *conjI-realizer*:

$P \ p \implies Q \ q \implies P \ (\text{fst } (p, q)) \wedge Q \ (\text{snd } (p, q))$
 $\langle \text{proof} \rangle$

theorem *exI-realizer*:

$P \ y \ x \implies P \ (\text{snd } (x, y)) \ (\text{fst } (x, y)) \ \langle \text{proof} \rangle$

theorem *exE-realizer*: $P \ (\text{snd } p) \ (\text{fst } p) \implies$

$(\bigwedge x \ y. P \ y \ x \implies Q \ (f \ x \ y)) \implies Q \ (\text{let } (x, y) = p \text{ in } f \ x \ y)$

$\langle proof \rangle$

theorem *exE-realizer'*: $P \text{ (snd } p) \text{ (fst } p) \implies$
 $(\bigwedge x y. P y x \implies Q) \implies Q \langle proof \rangle$

realizers

impI (P, Q): $\lambda pq. pq$
 $\Lambda (c: -) (d: -) P Q pq (h: -). \text{allI} \cdot \cdot \cdot c \cdot (\Lambda x. \text{impI} \cdot \cdot \cdot \cdot (h \cdot x))$

impI (P): *Null*
 $\Lambda (c: -) P Q (h: -). \text{allI} \cdot \cdot \cdot c \cdot (\Lambda x. \text{impI} \cdot \cdot \cdot \cdot (h \cdot x))$

impI (Q): $\lambda q. q \Lambda (c: -) P Q q. \text{impI} \cdot \cdot \cdot -$

impI: *Null impI*

mp (P, Q): $\lambda pq. pq$
 $\Lambda (c: -) (d: -) P Q pq (h: -) p. \text{mp} \cdot \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot c \cdot h)$

mp (P): *Null*
 $\Lambda (c: -) P Q (h: -) p. \text{mp} \cdot \cdot \cdot \cdot (\text{spec} \cdot \cdot \cdot p \cdot c \cdot h)$

mp (Q): $\lambda q. q \Lambda (c: -) P Q q. \text{mp} \cdot \cdot \cdot -$

mp: *Null mp*

allI (P): $\lambda p. p \Lambda (c: -) P (d: -) p. \text{allI} \cdot \cdot \cdot d$

allI: *Null allI*

spec (P): $\lambda x p. p x \Lambda (c: -) P x (d: -) p. \text{spec} \cdot \cdot \cdot x \cdot d$

spec: *Null spec*

exI (P): $\lambda x p. (x, p) \Lambda (c: -) P x (d: -) p. \text{exI-realizer} \cdot P \cdot p \cdot x \cdot c \cdot d$

exI: $\lambda x. x \Lambda P x (c: -) (h: -). h$

exE (P, Q): $\lambda p pq. \text{let } (x, y) = p \text{ in } pq x y$
 $\Lambda (c: -) (d: -) P Q (e: -) p (h: -) pq. \text{exE-realizer} \cdot P \cdot p \cdot Q \cdot pq \cdot c \cdot e \cdot d \cdot h$

exE (P): *Null*
 $\Lambda (c: -) P Q (d: -) p. \text{exE-realizer}' \cdot \cdot \cdot \cdot \cdot c \cdot d$

exE (Q): $\lambda x pq. pq x$
 $\Lambda (c: -) P Q (d: -) x (h1: -) pq (h2: -). h2 \cdot x \cdot h1$

exE: *Null*
 $\Lambda P Q (c: -) x (h1: -) (h2: -). h2 \cdot x \cdot h1$

conjI (*P*, *Q*): *Pair*
 $\Lambda (c: -) (d: -) P Q p (h: -) q. \text{conjI-realizer} \cdot P \cdot p \cdot Q \cdot q \cdot c \cdot d \cdot h$

conjI (*P*): $\lambda p. p$
 $\Lambda (c: -) P Q p. \text{conjI} \cdot - \cdot - \cdot -$

conjI (*Q*): $\lambda q. q$
 $\Lambda (c: -) P Q (h: -) q. \text{conjI} \cdot - \cdot - \cdot - \cdot h$

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*
 $\Lambda (c: -) (d: -) P Q pq. \text{conjunct1} \cdot - \cdot - \cdot -$

conjunct1 (*P*): $\lambda p. p$
 $\Lambda (c: -) P Q p. \text{conjunct1} \cdot - \cdot - \cdot -$

conjunct1 (*Q*): *Null*
 $\Lambda (c: -) P Q q. \text{conjunct1} \cdot - \cdot - \cdot -$

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
 $\Lambda (c: -) (d: -) P Q pq. \text{conjunct2} \cdot - \cdot - \cdot -$

conjunct2 (*P*): *Null*
 $\Lambda (c: -) P Q p. \text{conjunct2} \cdot - \cdot - \cdot -$

conjunct2 (*Q*): $\lambda p. p$
 $\Lambda (c: -) P Q p. \text{conjunct2} \cdot - \cdot - \cdot -$

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
 $\Lambda (c: -) (d: -) P Q p. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{sum.cases-1} \cdot P \cdot - \cdot p \cdot \text{arity-type-bool} \cdot c \cdot d)$

disjI1 (*P*): *Some*
 $\Lambda (c: -) P Q p. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{option.cases-2} \cdot - \cdot - \cdot P \cdot p \cdot \text{arity-type-bool} \cdot c)$

disjI1 (*Q*): *None*
 $\Lambda (c: -) P Q. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{option.cases-1} \cdot - \cdot - \cdot - \cdot \text{arity-type-bool} \cdot c)$

disjI1: *Left*
 $\Lambda P Q. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{sumbool.cases-1} \cdot - \cdot - \cdot - \cdot \text{arity-type-bool})$

disjI2 (*P*, *Q*): *Inr*
 $\Lambda (d: -) (c: -) Q P q. \text{iffD2} \cdot - \cdot - \cdot - \cdot (\text{sum.cases-2} \cdot - \cdot - \cdot Q \cdot q \cdot \text{arity-type-bool} \cdot c)$

$c \cdot d$)

$disjI2 (P): None$

$\Lambda (c: -) Q P. iffD2 \cdot \cdot \cdot \cdot (option.cases-1 \cdot \cdot \cdot \cdot arity-type-bool \cdot c)$

$disjI2 (Q): Some$

$\Lambda (c: -) Q P q. iffD2 \cdot \cdot \cdot \cdot (option.cases-2 \cdot \cdot \cdot Q \cdot q \cdot arity-type-bool \cdot c)$

$disjI2: Right$

$\Lambda Q P. iffD2 \cdot \cdot \cdot \cdot (sumbool.cases-2 \cdot \cdot \cdot \cdot arity-type-bool)$

$disjE (P, Q, R): \lambda pq pr qr.$

$(case pq of Inl p \Rightarrow pr p \mid Inr q \Rightarrow qr q)$

$\Lambda (c: -) (d: -) (e: -) P Q R pq (h1: -) pr (h2: -) qr.$

$disjE-realizer \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot e \cdot h1 \cdot h2$

$disjE (Q, R): \lambda pq pr qr.$

$(case pq of None \Rightarrow pr \mid Some q \Rightarrow qr q)$

$\Lambda (c: -) (d: -) P Q R pq (h1: -) pr (h2: -) qr.$

$disjE-realizer2 \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot d \cdot h1 \cdot h2$

$disjE (P, R): \lambda pq pr qr.$

$(case pq of None \Rightarrow qr \mid Some p \Rightarrow pr p)$

$\Lambda (c: -) (d: -) P Q R pq (h1: -) pr (h2: -) qr (h3: -).$

$disjE-realizer2 \cdot \cdot \cdot \cdot pq \cdot R \cdot qr \cdot pr \cdot c \cdot d \cdot h1 \cdot h3 \cdot h2$

$disjE (R): \lambda pq pr qr.$

$(case pq of Left \Rightarrow pr \mid Right \Rightarrow qr)$

$\Lambda (c: -) P Q R pq (h1: -) pr (h2: -) qr.$

$disjE-realizer3 \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \cdot c \cdot h1 \cdot h2$

$disjE (P, Q): Null$

$\Lambda (c: -) (d: -) P Q R pq. disjE-realizer \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot d \cdot$
 $arity-type-bool$

$disjE (Q): Null$

$\Lambda (c: -) P Q R pq. disjE-realizer2 \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot arity-type-bool$

$disjE (P): Null$

$\Lambda (c: -) P Q R pq (h1: -) (h2: -) (h3: -).$

$disjE-realizer2 \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot c \cdot arity-type-bool \cdot h1 \cdot h3 \cdot h2$

$disjE: Null$

$\Lambda P Q R pq. disjE-realizer3 \cdot \cdot \cdot \cdot pq \cdot (\lambda x. R) \cdot \cdot \cdot \cdot arity-type-bool$

$FalseE (P): default$

$\Lambda (c: -) P. FalseE \cdot \cdot$

$FalseE: Null FalseE$

$notI (P): Null$
 $\Lambda (c: -) P (h: -). allI \cdot - \cdot c \cdot (\Lambda x. notI \cdot - \cdot (h \cdot x))$

$notI: Null notI$

$notE (P, R): \lambda p. default$
 $\Lambda (c: -) (d: -) P R (h: -) p. notE \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot c \cdot h)$

$notE (P): Null$
 $\Lambda (c: -) P R (h: -) p. notE \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot c \cdot h)$

$notE (R): default$
 $\Lambda (c: -) P R. notE \cdot - \cdot -$

$notE: Null notE$

$subst (P): \lambda s \ t \ ps. ps$
 $\Lambda (c: -) s \ t \ P (d: -) (h: -) ps. subst \cdot s \cdot t \cdot P \ ps \cdot d \cdot h$

$subst: Null subst$

$iffD1 (P, Q): fst$
 $\Lambda (d: -) (c: -) Q P pq (h: -) p.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot d \cdot (conjunct1 \cdot - \cdot - \cdot h))$

$iffD1 (P): \lambda p. p$
 $\Lambda (c: -) Q P p (h: -). mp \cdot - \cdot - \cdot - \cdot (conjunct1 \cdot - \cdot - \cdot h)$

$iffD1 (Q): Null$
 $\Lambda (c: -) Q P q1 (h: -) q2.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot c \cdot (conjunct1 \cdot - \cdot - \cdot h))$

$iffD1: Null iffD1$

$iffD2 (P, Q): snd$
 $\Lambda (c: -) (d: -) P Q pq (h: -) q.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot q \cdot d \cdot (conjunct2 \cdot - \cdot - \cdot h))$

$iffD2 (P): \lambda p. p$
 $\Lambda (c: -) P Q p (h: -). mp \cdot - \cdot - \cdot - \cdot (conjunct2 \cdot - \cdot - \cdot h)$

$iffD2 (Q): Null$
 $\Lambda (c: -) P Q q1 (h: -) q2.$
 $mp \cdot - \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot c \cdot (conjunct2 \cdot - \cdot - \cdot h))$

$iffD2: Null iffD2$

$iffI (P, Q): Pair$

```

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*.

$\langle ML \rangle$

```
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 set  $\Rightarrow$  'a
  assumes F-eq: F g A = (if finite A then fold-image (op *) g 1 A else 1)

```

```

sublocale comm-monoid-big < folding-image <proof>

```

```

context comm-monoid-big
begin

```

```

lemma infinite [simp]:
   $\neg$  finite A  $\Longrightarrow$  F g A = 1
  <proof>

```

```

end

```

for ad-hoc proofs for *fold-image*

```

lemma (in comm-monoid-add) comm-monoid-mult:
  class.comm-monoid-mult (op +) 0
  <proof>

```

```

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 set  $\Rightarrow$  'a where
  setsum f A = (if finite A then fold-image (op +) f 0 A else 0)

```

```

sublocale comm-monoid-add < setsum!: comm-monoid-big op + 0 setsum <proof>

```

abbreviation

```

  Setsum ( $\sum$  - [1000] 999) where
   $\sum A ==$  setsum (%x. x) A

```

Now: lot's of fancy syntax. First, *setsum* ($\lambda x. e$) *A* is written $\sum_{x \in A}. e$.

syntax

```

  -setsum :: pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b::comm-monoid-add (( $\sum$  -.-) [0,
  51, 10] 10)

```

syntax (*xsymbols*)

```

  -setsum :: pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b::comm-monoid-add (( $\sum$  -.-) [0,
  51, 10] 10)

```

syntax (*HTML output*)

```

  -setsum :: pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'b  $\Rightarrow$  'b::comm-monoid-add (( $\sum$  -.-) [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

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\mathcal{S}SUM - | / - / -) [0,0,10] 10)$

syntax (*xsymbols*)

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\mathcal{S}\sum - | (-). / -) [0,0,10] 10)$

syntax (*HTML output*)

$-qsetsum :: pttrn \Rightarrow bool \Rightarrow 'a \Rightarrow 'a \ ((\mathcal{S}\sum - | (-). / -) [0,0,10] 10)$

translations

$SUM x | P. t \Rightarrow CONST setsum (\%x. t) \{x. P\}$

$\sum x | P. t \Rightarrow CONST setsum (\%x. t) \{x. P\}$

$\langle ML \rangle$

lemma *setsum-empty*:

$setsum f \{\} = 0$

$\langle proof \rangle$

lemma *setsum-insert*:

$finite F \Rightarrow a \notin F \Rightarrow setsum f (insert a F) = f a + setsum f F$

$\langle proof \rangle$

lemma *setsum-infinite*:

$\sim finite A \Rightarrow setsum f A = 0$

$\langle proof \rangle$

lemma (*in comm-monoid-add*) *setsum-reindex*:

assumes *inj-on* $f B$ **shows** $setsum h (f ' B) = setsum (h \circ f) B$

$\langle proof \rangle$

lemma (*in comm-monoid-add*) *setsum-reindex-id*:

inj-on $f B \Rightarrow setsum f B = setsum id (f ' B)$

$\langle proof \rangle$

lemma (*in comm-monoid-add*) *setsum-reindex-nonzero*:

assumes fS : *finite* S

and nz : $\bigwedge x y. x \in S \Rightarrow y \in S \Rightarrow x \neq y \Rightarrow f x = f y \Rightarrow h (f x) = 0$

shows $setsum h (f ' S) = setsum (h \circ f) S$

$\langle proof \rangle$

lemma (*in comm-monoid-add*) *setsum-cong*:

$A = B \Rightarrow (!x. x:B \Rightarrow f x = g x) \Rightarrow setsum f A = setsum g B$

$\langle proof \rangle$

lemma (*in comm-monoid-add*) *strong-setsum-cong* [*cong*]:

$A = B \Rightarrow (!x. x:B \Rightarrow f x = g x)$

$\Rightarrow setsum (\%x. f x) A = setsum (\%x. g x) B$

$\langle proof \rangle$

lemma (in *comm-monoid-add*) *setsum-cong2*: $\llbracket \bigwedge x. x \in A \implies f\ x = g\ x \rrbracket \implies \text{setsum } f\ A = \text{setsum } g\ A$
 ⟨proof⟩

lemma (in *comm-monoid-add*) *setsum-reindex-cong*:
 $\llbracket \text{inj-on } f\ A; B = f\ ' A; !!a. a:A \implies g\ a = h\ (f\ a) \rrbracket$
 $\implies \text{setsum } h\ B = \text{setsum } g\ A$
 ⟨proof⟩

lemma (in *comm-monoid-add*) *setsum-0[simp]*: $\text{setsum } (\%i. 0)\ A = 0$
 ⟨proof⟩

lemma (in *comm-monoid-add*) *setsum-0'*: $ALL\ a:A. f\ a = 0 \implies \text{setsum } f\ A = 0$
 ⟨proof⟩

lemma (in *comm-monoid-add*) *setsum-Un-Int*: $\text{finite } A \implies \text{finite } B \implies$
 $\text{setsum } g\ (A\ \text{Un } B) + \text{setsum } g\ (A\ \text{Int } B) = \text{setsum } g\ A + \text{setsum } g\ B$
 — The reversed orientation looks more natural, but LOOPS as a simp rule!
 ⟨proof⟩

lemma (in *comm-monoid-add*) *setsum-Un-disjoint*: $\text{finite } A \implies \text{finite } B$
 $\implies A\ \text{Int } B = \{\} \implies \text{setsum } g\ (A\ \text{Un } B) = \text{setsum } g\ A + \text{setsum } g\ B$
 ⟨proof⟩

lemma *setsum-mono-zero-left*:
 assumes fT : $\text{finite } T$ and ST : $S \subseteq T$
 and z : $\forall i \in T - S. f\ i = 0$
 shows $\text{setsum } f\ S = \text{setsum } f\ T$
 ⟨proof⟩

lemma *setsum-mono-zero-right*:
 $\text{finite } T \implies S \subseteq T \implies \forall i \in T - S. f\ i = 0 \implies \text{setsum } f\ T = \text{setsum } f\ S$
 ⟨proof⟩

lemma *setsum-mono-zero-cong-left*:
 assumes fT : $\text{finite } T$ 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⟩

lemma *setsum-mono-zero-cong-right*:
 assumes fT : $\text{finite } T$ 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 $\text{setsum } f\ T = \text{setsum } g\ S$
 ⟨proof⟩

lemma *setsum-delta*:

assumes fS : *finite* S

shows $\text{setsum } (\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } 0) \ S = (\text{if } a \in S \text{ then } b \ a \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma *setsum-delta'*:

assumes fS : *finite* S **shows**

$\text{setsum } (\lambda k. \text{if } a = k \text{ then } b \ k \text{ else } 0) \ S =$
 $(\text{if } a \in S \text{ then } b \ a \text{ else } 0)$

$\langle \text{proof} \rangle$

lemma *setsum-restrict-set*:

assumes fA : *finite* A

shows $\text{setsum } f \ (A \cap B) = \text{setsum } (\lambda x. \text{if } x \in B \text{ then } f \ x \text{ else } 0) \ A$

$\langle \text{proof} \rangle$

lemma *setsum-cases*:

assumes fA : *finite* A

shows $\text{setsum } (\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x) \ A =$

$\text{setsum } f \ (A \cap \{x. P \ x\}) + \text{setsum } g \ (A \cap - \{x. P \ x\})$

$\langle \text{proof} \rangle$

lemma (*in comm-monoid-add*) *setsum-UN-disjoint*:

assumes *finite* I **and** $\text{ALL } i:I. \text{finite } (A \ i)$

and $\text{ALL } i:I. \text{ALL } j:I. i \neq j \longrightarrow A \ i \ \text{Int } A \ j = \{\}$

shows $\text{setsum } f \ (\text{UNION } I \ A) = (\sum i \in I. \text{setsum } f \ (A \ i))$

$\langle \text{proof} \rangle$

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*:

$\llbracket (\text{ALL } A:C. \text{finite } A);$

$(\text{ALL } A:C. \text{ALL } B:C. A \neq B \longrightarrow A \ \text{Int } B = \{\}) \rrbracket$

$\implies \text{setsum } f \ (\text{Union } C) = \text{setsum } (\text{setsum } f) \ C$

$\langle \text{proof} \rangle$

lemma (*in comm-monoid-add*) *setsum-Sigma*:

assumes *finite* A **and** $\text{ALL } x:A. \text{finite } (B \ x)$

shows $(\sum x \in A. (\sum y \in B \ x. f \ x \ y)) = (\sum (x,y) \in (\text{SIGMA } x:A. B \ x). f \ x \ y)$

$\langle \text{proof} \rangle$

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-cartesian-product*:

$(\sum x \in A. (\sum y \in B. f \ x \ y)) = (\sum (x,y) \in A \lt * \gt B. f \ x \ y)$

$\langle \text{proof} \rangle$

lemma (in *comm-monoid-add*) *setsum-addf*: $\text{setsum } (\%x. f\ x + g\ x)\ A = (\text{setsum } f\ A + \text{setsum } g\ A)$
 ⟨proof⟩

28.2.1 Properties in more restricted classes of structures

lemma *setsum-SucD*: $\text{setsum } f\ A = \text{Suc } n \implies \exists x:A. 0 < f\ x$
 ⟨proof⟩

lemma *setsum-eq-0-iff* [simp]:
 $\text{finite } F \implies (\text{setsum } f\ F = 0) = (\forall a:F. f\ a = (0::nat))$
 ⟨proof⟩

lemma *setsum-eq-Suc0-iff*: $\text{finite } A \implies$
 $(\text{setsum } f\ A = \text{Suc } 0) = (\exists a:A. f\ a = \text{Suc } 0 \ \& \ (\forall b:A. a \neq b \implies f\ b = 0))$
 ⟨proof⟩

lemmas *setsum-eq-1-iff* = *setsum-eq-Suc0-iff* [simplified One-nat-def[symmetric]]

lemma *setsum-Un-nat*: $\text{finite } A \implies \text{finite } B \implies$
 $(\text{setsum } f\ (A\ \text{Un } B) :: nat) = \text{setsum } f\ A + \text{setsum } f\ B - \text{setsum } f\ (A\ \text{Int } B)$
 — For the natural numbers, we have subtraction.
 ⟨proof⟩

lemma *setsum-Un*: $\text{finite } A \implies \text{finite } B \implies$
 $(\text{setsum } f\ (A\ \text{Un } B) :: 'a :: ab-group-add) =$
 $\text{setsum } f\ A + \text{setsum } f\ B - \text{setsum } f\ (A\ \text{Int } B)$
 ⟨proof⟩

lemma (in *comm-monoid-add*) *setsum-eq-general-reverses*:
assumes *fS*: $\text{finite } S$ **and** *fT*: $\text{finite } T$
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⟩

lemma (in *comm-monoid-add*) *setsum-Un-zero*:
assumes *fS*: $\text{finite } S$ **and** *fT*: $\text{finite } T$
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⟩

lemma *setsum-UNION-zero*:
assumes *fS*: $\text{finite } S$ **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 $\text{setsum } f\ (\bigcup S) = \text{setsum } (\lambda T. \text{setsum } f\ T)\ S$
 ⟨proof⟩

lemma *setsum-diff1-nat*: $(\text{setsum } f \ (A - \{a\}) :: \text{nat}) =$
 $(\text{if } a:A \text{ then } \text{setsum } f \ A - f \ a \text{ else } \text{setsum } f \ A)$
 $\langle \text{proof} \rangle$

lemma *setsum-diff1*: $\text{finite } A \implies$
 $(\text{setsum } f \ (A - \{a\}) :: ('a::\text{ab-group-add})) =$
 $(\text{if } a:A \text{ then } \text{setsum } f \ A - f \ a \text{ else } \text{setsum } f \ A)$
 $\langle \text{proof} \rangle$

lemma *setsum-diff1 '[rule-format]*:
 $\text{finite } A \implies a \in A \longrightarrow (\sum x \in A. f \ x) = f \ a + (\sum x \in (A - \{a\}). f \ x)$
 $\langle \text{proof} \rangle$

lemma *setsum-diff1-ring*: **assumes** $\text{finite } A \ a \in A$
shows $\text{setsum } f \ (A - \{a\}) = \text{setsum } f \ A - (f \ a :: 'a::\text{ring})$
 $\langle \text{proof} \rangle$

lemma *setsum-diff-nat*:
assumes $\text{finite } B \text{ and } B \subseteq A$
shows $(\text{setsum } f \ (A - B) :: \text{nat}) = (\text{setsum } f \ A) - (\text{setsum } f \ B)$
 $\langle \text{proof} \rangle$

lemma *setsum-diff*:
assumes $le: \text{finite } A \ B \subseteq A$
shows $\text{setsum } f \ (A - B) = \text{setsum } f \ A - ((\text{setsum } f \ B) :: ('a::\text{ab-group-add}))$
 $\langle \text{proof} \rangle$

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 $(\sum i \in K. f \ i) \leq (\sum i \in K. g \ i)$
 $\langle \text{proof} \rangle$

lemma *setsum-strict-mono*:
fixes $f :: 'a \Rightarrow 'b::\{\text{ordered-cancel-ab-semigroup-add}, \text{comm-monoid-add}\}$
assumes $\text{finite } A \ A \neq \{\}$
and $!!x. x:A \implies f \ x < g \ x$
shows $\text{setsum } f \ A < \text{setsum } g \ A$
 $\langle \text{proof} \rangle$

lemma *setsum-negf*:
 $\text{setsum } (\%x. - (f \ x) :: 'a::\text{ab-group-add}) \ A = - \text{setsum } f \ A$
 $\langle \text{proof} \rangle$

lemma *setsum-subtractf*:
 $\text{setsum } (\%x. ((f \ x) :: 'a::\text{ab-group-add}) - g \ x) \ A =$
 $\text{setsum } f \ A - \text{setsum } g \ A$
 $\langle \text{proof} \rangle$

lemma *setsum-nonneg*:

assumes *nn*: $\forall x \in A. (0 :: 'a :: \{\text{ordered-ab-semigroup-add}, \text{comm-monoid-add}\}) \leq f\ x$
shows $0 \leq \text{setsum } f\ A$
 $\langle \text{proof} \rangle$

lemma *setsum-nonpos*:

assumes *np*: $\forall x \in A. f\ x \leq (0 :: 'a :: \{\text{ordered-ab-semigroup-add}, \text{comm-monoid-add}\})$
shows $\text{setsum } f\ A \leq 0$
 $\langle \text{proof} \rangle$

lemma *setsum-nonneg-leq-bound*:

fixes $f :: 'a \Rightarrow 'b :: \{\text{ordered-ab-group-add}\}$
assumes *finite s* $\bigwedge i. i \in s \implies f\ i \geq 0$ $(\sum i \in s. f\ i) = B$ $i \in s$
shows $f\ i \leq B$
 $\langle \text{proof} \rangle$

lemma *setsum-nonneg-0*:

fixes $f :: 'a \Rightarrow 'b :: \{\text{ordered-ab-group-add}\}$
assumes *finite s* **and** *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 $f\ i = 0$
 $\langle \text{proof} \rangle$

lemma *setsum-mono2*:

fixes $f :: 'a \Rightarrow 'b :: \text{ordered-comm-monoid-add}$
assumes *fin*: *finite B* **and** *sub*: $A \subseteq B$ **and** *nn*: $\bigwedge b. b \in B - A \implies 0 \leq f\ b$
shows $\text{setsum } f\ A \leq \text{setsum } f\ B$
 $\langle \text{proof} \rangle$

lemma *setsum-mono3*: *finite B ==> A <= B ==>*

ALL x: B - A.
 $0 \leq ((f\ x) :: 'a :: \{\text{comm-monoid-add}, \text{ordered-ab-semigroup-add}\}) \implies$
 $\text{setsum } f\ A \leq \text{setsum } f\ B$
 $\langle \text{proof} \rangle$

lemma *setsum-right-distrib*:

fixes $f :: 'a \Rightarrow ('b :: \text{semiring-0})$
shows $r * \text{setsum } f\ A = \text{setsum } (\%n. r * f\ n)\ A$
 $\langle \text{proof} \rangle$

lemma *setsum-left-distrib*:

$\text{setsum } f\ A * (r :: 'a :: \text{semiring-0}) = (\sum n \in A. f\ n * r)$
 $\langle \text{proof} \rangle$

lemma *setsum-divide-distrib*:

$\text{setsum } f\ A / (r :: 'a :: \text{field}) = (\sum n \in A. f\ n / r)$
 $\langle \text{proof} \rangle$

lemma *setsum-abs[iff]*:
fixes $f :: 'a \Rightarrow ('b::\text{ordered-ab-group-add-abs})$
shows $\text{abs} (\text{setsum } f \ A) \leq \text{setsum } (\%i. \text{abs}(f \ i)) \ A$
 $\langle \text{proof} \rangle$

lemma *setsum-abs-ge-zero[iff]*:
fixes $f :: 'a \Rightarrow ('b::\text{ordered-ab-group-add-abs})$
shows $0 \leq \text{setsum } (\%i. \text{abs}(f \ i)) \ A$
 $\langle \text{proof} \rangle$

lemma *abs-setsum-abs[simp]*:
fixes $f :: 'a \Rightarrow ('b::\text{ordered-ab-group-add-abs})$
shows $\text{abs} (\sum_{a \in A. \text{abs}(f \ a)}) = (\sum_{a \in A. \text{abs}(f \ a)})$
 $\langle \text{proof} \rangle$

lemma *setsum-Plus*:
fixes $A :: 'a \text{ set}$ **and** $B :: 'b \text{ set}$
assumes *fin*: $\text{finite } A \ \text{finite } B$
shows $\text{setsum } f \ (A <+> B) = \text{setsum } (f \circ \text{Inl}) \ A + \text{setsum } (f \circ \text{Inr}) \ B$
 $\langle \text{proof} \rangle$

Commuting outer and inner summation

lemma *setsum-commute*:
 $(\sum_{i \in A. \sum_{j \in B. f \ i \ j}) = (\sum_{j \in B. \sum_{i \in A. f \ i \ j})$
 $\langle \text{proof} \rangle$

lemma *setsum-product*:
fixes $f :: 'a \Rightarrow ('b::\text{semiring-0})$
shows $\text{setsum } f \ A * \text{setsum } g \ B = (\sum_{i \in A. \sum_{j \in B. f \ i * g \ j})$
 $\langle \text{proof} \rangle$

lemma *setsum-mult-setsum-if-inj*:
fixes $f :: 'a \Rightarrow ('b::\text{semiring-0})$
shows $\text{inj-on } (\% (a,b). f \ a * g \ b) \ (A \times B) \implies$
 $\text{setsum } f \ A * \text{setsum } g \ B = \text{setsum } \text{id} \ \{f \ a * g \ b \mid a \in A \ \& \ b \in B\}$
 $\langle \text{proof} \rangle$

lemma *setsum-constant [simp]*: $(\sum x \in A. y) = \text{of-nat}(\text{card } A) * y$
 $\langle \text{proof} \rangle$

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i. i \in A \implies f \ i \leq (K :: 'a::\{\text{semiring-1}, \text{ordered-ab-semigroup-add}\})$
shows $\text{setsum } f \ A \leq \text{of-nat}(\text{card } A) * K$
 $\langle \text{proof} \rangle$

28.2.2 Cardinality as special case of *setsum*

lemma *card-eq-setsum*:

$card\ A = setsum\ (\lambda x. 1)\ A$
 $\langle proof \rangle$

lemma *card-UN-disjoint*:

$finite\ I ==> (ALL\ i:I. finite\ (A\ i)) ==>$
 $(ALL\ i:I. ALL\ j:I. i \neq j \longrightarrow A\ i\ Int\ A\ j = \{\})$
 $==> card\ (UNION\ I\ A) = (\sum\ i \in I. card(A\ i))$
 $\langle proof \rangle$

lemma *card-Union-disjoint*:

$finite\ C ==> (ALL\ A:C. finite\ A) ==>$
 $(ALL\ A:C. ALL\ B:C. A \neq B \longrightarrow A\ Int\ B = \{\})$
 $==> card\ (Union\ C) = setsum\ card\ C$
 $\langle proof \rangle$

The image of a finite set can be expressed using *fold-image*.

lemma *image-eq-fold-image*:

$finite\ A ==> f\ ` A = fold-image\ (op\ Un)\ (\%x. \{f\ x\})\ \{\}\ A$
 $\langle proof \rangle$

28.2.3 Cardinality of products

lemma *card-SigmaI [simp]*:

$\llbracket finite\ A; ALL\ a:A. finite\ (B\ a) \rrbracket$
 $\implies card\ (SIGMA\ x: A. B\ x) = (\sum\ a \in A. card\ (B\ a))$
 $\langle proof \rangle$

lemma *card-cartesian-product*: $card\ (A\ <*>\ B) = card(A) * card(B)$
 $\langle proof \rangle$

lemma *card-cartesian-product-singleton*: $card(\{x\}\ <*>\ A) = card(A)$
 $\langle proof \rangle$

28.3 Generalized product over a set

definition (*in comm-monoid-mult*) *setprod* :: $('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'a$ **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* $\langle proof \rangle$

abbreviation

$Setprod\ (\prod - [1000]\ 999)$ **where**
 $\prod A == setprod\ (\%x. x)\ A$

syntax

$-setprod :: ptnr \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b::comm-monoid-mult\ ((3PROD\ :-.\ -)$
 $[0, 51, 10]\ 10)$

syntax (*xsymbols*)
 $\text{-setprod} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-mult} \ ((\exists \prod - \in -. -) [0, 51, 10] 10)$
syntax (*HTML output*)
 $\text{-setprod} :: \text{pttrn} \Rightarrow 'a \text{ set} \Rightarrow 'b \Rightarrow 'b::\text{comm-monoid-mult} \ ((\exists \prod - \in -. -) [0, 51, 10] 10)$

translations — Beware of argument permutation!

$\text{PROD } i:A. b == \text{CONST setprod } (\%i. b) A$
 $\prod i \in A. b == \text{CONST setprod } (\%i. b) A$

Instead of $\prod x \in \{x. P\}. e$ we introduce the shorter $\prod x | P. e$.

syntax
 $\text{-qsetprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ ((\exists \text{PROD} - | / -. / -) [0, 0, 10] 10)$
syntax (*xsymbols*)
 $\text{-qsetprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ ((\exists \prod - | (-). / -) [0, 0, 10] 10)$
syntax (*HTML output*)
 $\text{-qsetprod} :: \text{pttrn} \Rightarrow \text{bool} \Rightarrow 'a \Rightarrow 'a \ ((\exists \prod - | (-). / -) [0, 0, 10] 10)$

translations

$\text{PROD } x | P. t \Rightarrow \text{CONST setprod } (\%x. t) \{x. P\}$
 $\prod x | P. t \Rightarrow \text{CONST setprod } (\%x. t) \{x. P\}$

lemma *setprod-empty*: $\text{setprod } f \ \{\} = 1$
 $\langle \text{proof} \rangle$

lemma *setprod-insert*: $[\text{finite } A; a \notin A] \Rightarrow$
 $\text{setprod } f \ (\text{insert } a \ A) = f \ a * \text{setprod } f \ A$
 $\langle \text{proof} \rangle$

lemma *setprod-infinite*: $\sim \text{finite } A \Rightarrow \text{setprod } f \ A = 1$
 $\langle \text{proof} \rangle$

lemma *setprod-reindex*:
 $\text{inj-on } f \ B \Rightarrow \text{setprod } h \ (f \ ' B) = \text{setprod } (h \circ f) \ B$
 $\langle \text{proof} \rangle$

lemma *setprod-reindex-id*: $\text{inj-on } f \ B \Rightarrow \text{setprod } f \ B = \text{setprod id } (f \ ' B)$
 $\langle \text{proof} \rangle$

lemma *setprod-cong*:
 $A = B \Rightarrow (!x. x:B \Rightarrow f \ x = g \ x) \Rightarrow \text{setprod } f \ A = \text{setprod } g \ B$
 $\langle \text{proof} \rangle$

lemma *strong-setprod-cong*[*cong*]:
 $A = B \Rightarrow (!x. x:B = \text{simp} \Rightarrow f \ x = g \ x) \Rightarrow \text{setprod } f \ A = \text{setprod } g \ B$
 $\langle \text{proof} \rangle$

lemma *setprod-reindex-cong*: $\text{inj-on } f \ A \Rightarrow$

$B = f \circ A \implies g = h \circ f \implies \text{setprod } h \ B = \text{setprod } g \ A$
 $\langle \text{proof} \rangle$

lemma *strong-setprod-reindex-cong*: **assumes** i : *inj-on* $f \ A$
and B : $B = f \circ A$ **and** eq : $\bigwedge x. x \in A \implies g \ x = (h \circ f) \ x$
shows $\text{setprod } h \ B = \text{setprod } g \ A$
 $\langle \text{proof} \rangle$

lemma *setprod-Un-one*:
assumes fS : *finite* S **and** fT : *finite* T
and $I0$: $\forall x \in S \cap T. f \ x = 1$
shows $\text{setprod } f \ (S \cup T) = \text{setprod } f \ S \ * \ \text{setprod } f \ T$
 $\langle \text{proof} \rangle$

lemma *setprod-1*: $\text{setprod } (\%i. 1) \ A = 1$
 $\langle \text{proof} \rangle$

lemma *setprod-1'*: $\text{ALL } a:F. f \ a = 1 \implies \text{setprod } f \ F = 1$
 $\langle \text{proof} \rangle$

lemma *setprod-Un-Int*: *finite* $A \implies \text{finite } B$
 $\implies \text{setprod } g \ (A \cup B) = \text{setprod } g \ (A \cap B) = \text{setprod } g \ A \ * \ \text{setprod } g \ B$
 $\langle \text{proof} \rangle$

lemma *setprod-Un-disjoint*: *finite* $A \implies \text{finite } B$
 $\implies A \cap B = \{\} \implies \text{setprod } g \ (A \cup B) = \text{setprod } g \ A \ * \ \text{setprod } g \ B$
 $\langle \text{proof} \rangle$

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 $\text{setprod } f \ S = \text{setprod } f \ T$
 $\langle \text{proof} \rangle$

lemmas *setprod-mono-one-right* = *setprod-mono-one-left* [THEN *sym*]

lemma *setprod-delta*:
assumes fS : *finite* S
shows $\text{setprod } (\lambda k. \text{if } k=a \text{ then } b \ k \text{ else } 1) \ S = (\text{if } a \in S \text{ then } b \ a \text{ else } 1)$
 $\langle \text{proof} \rangle$

lemma *setprod-delta'*:
assumes fS : *finite* S **shows**
 $\text{setprod } (\lambda k. \text{if } a = k \text{ then } b \ k \text{ else } 1) \ S =$
 $(\text{if } a \in S \text{ then } b \ a \text{ else } 1)$
 $\langle \text{proof} \rangle$

lemma *setprod-UN-disjoint*:

$$\begin{aligned} & \text{finite } I \implies (\text{ALL } i:I. \text{finite } (A \ i)) \implies \\ & (\text{ALL } i:I. \text{ALL } j:I. i \neq j \implies A \ i \text{ Int } A \ j = \{\}) \implies \\ & \text{setprod } f \ (\text{UNION } I \ A) = \text{setprod } (\%i. \text{setprod } f \ (A \ i)) \ I \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setprod-Union-disjoint*:

$$\begin{aligned} & [| (\text{ALL } A:C. \text{finite } A); \\ & (\text{ALL } A:C. \text{ALL } B:C. A \neq B \implies A \text{ Int } B = \{\}) |] \\ & \implies \text{setprod } f \ (\text{Union } C) = \text{setprod } (\text{setprod } f) \ C \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setprod-Sigma*: $\text{finite } A \implies \text{ALL } x:A. \text{finite } (B \ x) \implies$

$$\begin{aligned} & (\prod_{x \in A}. (\prod_{y \in B \ x}. f \ x \ y)) = \\ & (\prod_{(x,y) \in (\text{SIGMA } x:A. B \ x)}. f \ x \ y) \\ & \langle \text{proof} \rangle \end{aligned}$$

Here we can eliminate the finiteness assumptions, by cases.

lemma *setprod-cartesian-product*:

$$(\prod_{x \in A}. (\prod_{y \in B}. f \ x \ y)) = (\prod_{(x,y) \in (A \ltimes B)}. f \ x \ y)$$

$\langle \text{proof} \rangle$

lemma *setprod-timesf*:

$$\text{setprod } (\%x. f \ x * g \ x) \ A = (\text{setprod } f \ A * \text{setprod } g \ A)$$

$\langle \text{proof} \rangle$

28.3.1 Properties in more restricted classes of structures

lemma *setprod-eq-1-iff* [simp]:

$$\text{finite } F \implies (\text{setprod } f \ F = 1) = (\text{ALL } a:F. f \ a = (1::\text{nat}))$$

$\langle \text{proof} \rangle$

lemma *setprod-zero*:

$$\text{finite } A \implies \text{EX } x: A. f \ x = (0::'a::\text{comm-semiring-1}) \implies \text{setprod } f \ A = 0$$

$\langle \text{proof} \rangle$

lemma *setprod-nonneg* [rule-format]:

$$(\text{ALL } x: A. (0::'a::\text{linordered-semidom}) \leq f \ x) \implies 0 \leq \text{setprod } f \ A$$

$\langle \text{proof} \rangle$

lemma *setprod-pos* [rule-format]: $(\text{ALL } x: A. (0::'a::\text{linordered-semidom}) < f \ x)$

$$\implies 0 < \text{setprod } f \ A$$

$\langle \text{proof} \rangle$

lemma *setprod-zero-iff* [simp]: $\text{finite } A \implies$

$$\begin{aligned} & (\text{setprod } f \ A = (0::'a::\{\text{comm-semiring-1}, \text{no-zero-divisors}\})) = \\ & (\text{EX } x: A. f \ x = 0) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *setprod-pos-nat*:

$finite\ S \implies (ALL\ x : S. f\ x > (0::nat)) \implies setprod\ f\ S > 0$
 $\langle proof \rangle$

lemma *setprod-pos-nat-iff* [simp]:

$finite\ S \implies (setprod\ f\ S > 0) = (ALL\ x : S. f\ x > (0::nat))$
 $\langle proof \rangle$

lemma *setprod-Un*: $finite\ A \implies finite\ B \implies (ALL\ x: A\ Int\ B. f\ x \neq 0) \implies$

$(setprod\ f\ (A\ Un\ B) :: 'a :: \{field\})$
 $= setprod\ f\ A * setprod\ f\ B / setprod\ f\ (A\ Int\ B)$
 $\langle proof \rangle$

lemma *setprod-diff1*: $finite\ A \implies f\ a \neq 0 \implies$

$(setprod\ f\ (A - \{a\}) :: 'a :: \{field\}) =$
 $(if\ a:A\ then\ setprod\ f\ A / f\ a\ else\ setprod\ f\ A)$
 $\langle proof \rangle$

lemma *setprod-inversef*:

fixes $f :: 'b \Rightarrow 'a :: field-inverse-zero$
shows $finite\ A \implies setprod\ (inverse \circ f)\ A = inverse\ (setprod\ f\ A)$
 $\langle proof \rangle$

lemma *setprod-dividef*:

fixes $f :: 'b \Rightarrow 'a :: field-inverse-zero$
shows $finite\ A$
 $\implies setprod\ (\%x. f\ x / g\ x)\ A = setprod\ f\ A / setprod\ g\ A$
 $\langle proof \rangle$

lemma *setprod-dvd-setprod* [rule-format]:

$(ALL\ x : A. f\ x\ dvd\ g\ x) \longrightarrow setprod\ f\ A\ dvd\ setprod\ g\ A$
 $\langle proof \rangle$

lemma *setprod-dvd-setprod-subset*:

$finite\ B \implies A \leq B \implies setprod\ f\ A\ dvd\ setprod\ f\ B$
 $\langle proof \rangle$

lemma *setprod-dvd-setprod-subset2*:

$finite\ B \implies A \leq B \implies ALL\ x : A. (f\ x :: 'a :: comm-semiring-1)\ dvd\ g\ x \implies$
 $setprod\ f\ A\ dvd\ setprod\ g\ B$
 $\langle proof \rangle$

lemma *dvd-setprod*: $finite\ A \implies i:A \implies$

$(f\ i :: 'a :: comm-semiring-1)\ dvd\ setprod\ f\ A$
 $\langle proof \rangle$

lemma *dvd-setsum* [rule-format]: $(ALL\ i : A. d\ dvd\ f\ i) \longrightarrow$

$(d :: 'a :: comm-semiring-1)\ dvd\ (SUM\ x : A. f\ x)$
 $\langle proof \rangle$

lemma *setprod-mono*:

fixes $f :: 'a \Rightarrow 'b :: \text{linordered-semidom}$

assumes $\forall i \in A. 0 \leq f\ i \wedge f\ i \leq g\ i$

shows $\text{setprod } f\ A \leq \text{setprod } g\ A$

$\langle \text{proof} \rangle$

lemma *abs-setprod*:

fixes $f :: 'a \Rightarrow 'b :: \{\text{linordered-field}, \text{abs}\}$

shows $\text{abs } (\text{setprod } f\ A) = \text{setprod } (\lambda x. \text{abs } (f\ x))\ A$

$\langle \text{proof} \rangle$

lemma *setprod-constant*: $\text{finite } A \implies (\prod x \in A. (y :: 'a :: \{\text{comm-monoid-mult}\}))$
 $= y^{\text{card } A}$

$\langle \text{proof} \rangle$

lemma *setprod-gen-delta*:

assumes $fS: \text{finite } S$

shows $\text{setprod } (\lambda k. \text{if } k=a \text{ then } b \text{ else } c)\ S = (\text{if } a \in S \text{ then } (b\ a :: 'a :: \{\text{comm-monoid-mult}\})$
 $*\ c^{\text{card } S - 1} \text{ else } c^{\text{card } S})$

$\langle \text{proof} \rangle$

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 $F :: 'a \text{ set} \Rightarrow 'a$

assumes $F\text{-eq}: \text{finite } A \implies F\ A = \text{fold1 } (op\ *)\ A$

sublocale *semilattice-big* < *folding-one-idem* $\langle \text{proof} \rangle$

notation *times* (**infixl** * 70)

notation *Groups.one* (1)

context *lattice*

begin

definition *Inf-fin* :: $'a \text{ set} \Rightarrow 'a$ ($\prod_{fin} \text{ [900] } 900$) **where**

$\text{Inf-fin} = \text{fold1 } \text{inf}$

definition *Sup-fin* :: $'a \text{ set} \Rightarrow 'a$ ($\sqcup_{fin} \text{ [900] } 900$) **where**

$\text{Sup-fin} = \text{fold1 } \text{sup}$

end

sublocale *lattice* < *Inf-fin*!: *semilattice-big inf Inf-fin* $\langle \text{proof} \rangle$

sublocale *lattice* < *Sup-fin*!: *semilattice-big sup Sup-fin* $\langle \text{proof} \rangle$

context *semilattice-inf*
begin

lemma *ab-semigroup-idem-mult-inf*:
class.ab-semigroup-idem-mult inf
 $\langle \text{proof} \rangle$

lemma *fold-inf-insert[simp]*: *finite A* \implies *fold inf b (insert a A) = inf a (fold inf b A)*
 $\langle \text{proof} \rangle$

lemma *inf-le-fold-inf*: *finite A* \implies *ALL a:A. b \leq a \implies inf b c \leq fold inf c A*
 $\langle \text{proof} \rangle$

lemma *fold-inf-le-inf*: *finite A* \implies *a \in A \implies fold inf b A \leq inf a b*
 $\langle \text{proof} \rangle$

lemma *below-fold1-iff*:
assumes finite A A \neq {}
shows x \leq fold1 inf A \longleftrightarrow ($\forall a \in A. x \leq a$)
 $\langle \text{proof} \rangle$

lemma *fold1-belowI*:
assumes finite A
and a \in A
shows fold1 inf A \leq a
 $\langle \text{proof} \rangle$

end

context *semilattice-sup*
begin

lemma *ab-semigroup-idem-mult-sup*: *class.ab-semigroup-idem-mult sup*
 $\langle \text{proof} \rangle$

lemma *fold-sup-insert[simp]*: *finite A* \implies *fold sup b (insert a A) = sup a (fold sup b A)*
 $\langle \text{proof} \rangle$

lemma *fold-sup-le-sup*: *finite A* \implies *ALL a:A. a \leq b \implies fold sup c A \leq sup b c*
 $\langle \text{proof} \rangle$

lemma *sup-le-fold-sup*: *finite A* \implies *a \in A \implies sup a b \leq fold sup b A*
 $\langle \text{proof} \rangle$

end

context *lattice*

begin

lemma *Inf-le-Sup* [*simp*]: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \bigcap_{fin} A \leq \bigcup_{fin} A$
 $\langle \text{proof} \rangle$

lemma *sup-Inf-absorb* [*simp*]:
 $\text{finite } A \implies a \in A \implies \text{sup } a \ (\bigcap_{fin} A) = a$
 $\langle \text{proof} \rangle$

lemma *inf-Sup-absorb* [*simp*]:
 $\text{finite } A \implies a \in A \implies \text{inf } a \ (\bigcup_{fin} A) = a$
 $\langle \text{proof} \rangle$

end

context *distrib-lattice*

begin

lemma *sup-Inf1-distrib*:
assumes *finite A*
and $A \neq \{\}$
shows $\text{sup } x \ (\bigcap_{fin} A) = \bigcap_{fin} \{\text{sup } x \ a \mid a. a \in A\}$
 $\langle \text{proof} \rangle$

lemma *sup-Inf2-distrib*:
assumes *A: finite A A ≠ {} and B: finite B B ≠ {}*
shows $\text{sup} \ (\bigcap_{fin} A) \ (\bigcap_{fin} B) = \bigcap_{fin} \{\text{sup } a \ b \mid a \ b. a \in A \wedge b \in B\}$
 $\langle \text{proof} \rangle$

lemma *inf-Sup1-distrib*:
assumes *finite A and A ≠ {}*
shows $\text{inf } x \ (\bigcup_{fin} A) = \bigcup_{fin} \{\text{inf } x \ a \mid a. a \in A\}$
 $\langle \text{proof} \rangle$

lemma *inf-Sup2-distrib*:
assumes *A: finite A A ≠ {} and B: finite B B ≠ {}*
shows $\text{inf} \ (\bigcup_{fin} A) \ (\bigcup_{fin} B) = \bigcup_{fin} \{\text{inf } a \ b \mid a \ b. a \in A \wedge b \in B\}$
 $\langle \text{proof} \rangle$

end

context *complete-lattice*

begin

lemma *Inf-fin-Inf*:
assumes *finite A and A ≠ {}*
shows $\bigcap_{fin} A = \text{Inf } A$

$\langle \text{proof} \rangle$

lemma *Sup-fin-Sup*:

assumes *finite* A and $A \neq \{\}$

shows $\bigsqcup_{\text{fin}} A = \text{Sup } A$

$\langle \text{proof} \rangle$

end

28.5 Versions of *min* and *max* on non-empty sets

definition (*in linorder*) $\text{Min} :: 'a \text{ set} \Rightarrow 'a$ **where**

$\text{Min} = \text{fold1 } \text{min}$

definition (*in linorder*) $\text{Max} :: 'a \text{ set} \Rightarrow 'a$ **where**

$\text{Max} = \text{fold1 } \text{max}$

sublocale *linorder* < $\text{Min}!$: *semilattice-big min* Min $\langle \text{proof} \rangle$

sublocale *linorder* < $\text{Max}!$: *semilattice-big max* Max $\langle \text{proof} \rangle$

context *linorder*

begin

lemmas $\text{Min-singleton} = \text{Min.singleton}$

lemmas $\text{Max-singleton} = \text{Max.singleton}$

lemma *Min-insert*:

assumes *finite* A and $A \neq \{\}$

shows $\text{Min } (\text{insert } x \ A) = \text{min } x \ (\text{Min } A)$

$\langle \text{proof} \rangle$

lemma *Max-insert*:

assumes *finite* A and $A \neq \{\}$

shows $\text{Max } (\text{insert } x \ A) = \text{max } x \ (\text{Max } A)$

$\langle \text{proof} \rangle$

lemma *Min-Un*:

assumes *finite* A and $A \neq \{\}$ and *finite* B and $B \neq \{\}$

shows $\text{Min } (A \cup B) = \text{min } (\text{Min } A) \ (\text{Min } B)$

$\langle \text{proof} \rangle$

lemma *Max-Un*:

assumes *finite* A and $A \neq \{\}$ and *finite* B and $B \neq \{\}$

shows $\text{Max } (A \cup B) = \text{max } (\text{Max } A) \ (\text{Max } B)$

$\langle \text{proof} \rangle$

lemma *hom-Min-commute*:

assumes $\bigwedge x \ y. \ h \ (\text{min } x \ y) = \text{min } (h \ x) \ (h \ y)$

and *finite* N **and** $N \neq \{\}$
shows $h \text{ (Min } N) = \text{Min } (h \text{ ‘ } N)$
 $\langle \text{proof} \rangle$

lemma *hom-Max-commute*:
assumes $\bigwedge x y. h \text{ (max } x y) = \text{max } (h x) (h y)$
and *finite* N **and** $N \neq \{\}$
shows $h \text{ (Max } N) = \text{Max } (h \text{ ‘ } N)$
 $\langle \text{proof} \rangle$

lemma *ab-semigroup-idem-mult-min*:
class.ab-semigroup-idem-mult min
 $\langle \text{proof} \rangle$

lemma *ab-semigroup-idem-mult-max*:
class.ab-semigroup-idem-mult max
 $\langle \text{proof} \rangle$

lemma *max-lattice*:
class.semilattice-inf (op \geq) (op $>$) max
 $\langle \text{proof} \rangle$

lemma *dual-max*:
ord.max (op \geq) = min
 $\langle \text{proof} \rangle$

lemma *dual-min*:
ord.min (op \geq) = max
 $\langle \text{proof} \rangle$

lemma *strict-below-fold1-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $x < \text{fold1 min } A \longleftrightarrow (\forall a \in A. x < a)$
 $\langle \text{proof} \rangle$

lemma *fold1-below-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $\text{fold1 min } A \leq x \longleftrightarrow (\exists a \in A. a \leq x)$
 $\langle \text{proof} \rangle$

lemma *fold1-strict-below-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $\text{fold1 min } A < x \longleftrightarrow (\exists a \in A. a < x)$
 $\langle \text{proof} \rangle$

lemma *fold1-antimono*:
assumes $A \neq \{\}$ **and** $A \subseteq B$ **and** *finite* B
shows $\text{fold1 min } B \leq \text{fold1 min } A$
 $\langle \text{proof} \rangle$

lemma *Min-in* [*simp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $\text{Min } A \in A$
 $\langle \text{proof} \rangle$

lemma *Max-in* [*simp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $\text{Max } A \in A$
 $\langle \text{proof} \rangle$

lemma *Min-le* [*simp*]:
 assumes *finite A* and $x \in A$
 shows $\text{Min } A \leq x$
 $\langle \text{proof} \rangle$

lemma *Max-ge* [*simp*]:
 assumes *finite A* and $x \in A$
 shows $x \leq \text{Max } A$
 $\langle \text{proof} \rangle$

lemma *Min-ge-iff* [*simp*, *no-atp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $x \leq \text{Min } A \longleftrightarrow (\forall a \in A. x \leq a)$
 $\langle \text{proof} \rangle$

lemma *Max-le-iff* [*simp*, *no-atp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $\text{Max } A \leq x \longleftrightarrow (\forall a \in A. a \leq x)$
 $\langle \text{proof} \rangle$

lemma *Min-gr-iff* [*simp*, *no-atp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $x < \text{Min } A \longleftrightarrow (\forall a \in A. x < a)$
 $\langle \text{proof} \rangle$

lemma *Max-less-iff* [*simp*, *no-atp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $\text{Max } A < x \longleftrightarrow (\forall a \in A. a < x)$
 $\langle \text{proof} \rangle$

lemma *Min-le-iff* [*no-atp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $\text{Min } A \leq x \longleftrightarrow (\exists a \in A. a \leq x)$
 $\langle \text{proof} \rangle$

lemma *Max-ge-iff* [*no-atp*]:
 assumes *finite A* and $A \neq \{\}$
 shows $x \leq \text{Max } A \longleftrightarrow (\exists a \in A. x \leq a)$

$\langle \text{proof} \rangle$

lemma *Min-less-iff* [no-atp]:
 assumes *finite* A and $A \neq \{\}$
 shows $\text{Min } A < x \iff (\exists a \in A. a < x)$
 $\langle \text{proof} \rangle$

lemma *Max-gr-iff* [no-atp]:
 assumes *finite* A and $A \neq \{\}$
 shows $x < \text{Max } A \iff (\exists a \in A. x < a)$
 $\langle \text{proof} \rangle$

lemma *Min-eqI*:
 assumes *finite* A
 assumes $\bigwedge y. y \in A \implies y \geq x$
 and $x \in A$
 shows $\text{Min } A = x$
 $\langle \text{proof} \rangle$

lemma *Max-eqI*:
 assumes *finite* A
 assumes $\bigwedge y. y \in A \implies y \leq x$
 and $x \in A$
 shows $\text{Max } A = x$
 $\langle \text{proof} \rangle$

lemma *Min-antimono*:
 assumes $M \subseteq N$ and $M \neq \{\}$ and *finite* N
 shows $\text{Min } N \leq \text{Min } M$
 $\langle \text{proof} \rangle$

lemma *Max-mono*:
 assumes $M \subseteq N$ and $M \neq \{\}$ and *finite* N
 shows $\text{Max } M \leq \text{Max } N$
 $\langle \text{proof} \rangle$

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 $P \ A$
 $\langle \text{proof} \rangle$

lemma *finite-linorder-min-induct*[consumes 1, case-names empty insert]:
 $\llbracket \text{finite } A; P \ \{\}; \bigwedge b \ A. \llbracket \text{finite } A; \forall a \in A. b < a; P \ A \rrbracket \implies P \ (\text{insert } b \ A) \rrbracket \implies P \ A$
 $\langle \text{proof} \rangle$

end

context *linordered-ab-semigroup-add*
begin

lemma *add-Min-commute*:
 fixes k
 assumes *finite* N and $N \neq \{\}$
 shows $k + \text{Min } N = \text{Min } \{k + m \mid m. m \in N\}$
 $\langle \text{proof} \rangle$

lemma *add-Max-commute*:
 fixes k
 assumes *finite* N and $N \neq \{\}$
 shows $k + \text{Max } N = \text{Max } \{k + m \mid m. m \in N\}$
 $\langle \text{proof} \rangle$

end

context *linordered-ab-group-add*
begin

lemma *minus-Max-eq-Min* [*simp*]:
 $\text{finite } S \implies S \neq \{\} \implies - (\text{Max } S) = \text{Min } (\text{uminus } S)$
 $\langle \text{proof} \rangle$

lemma *minus-Min-eq-Max* [*simp*]:
 $\text{finite } S \implies S \neq \{\} \implies - (\text{Min } S) = \text{Max } (\text{uminus } S)$
 $\langle \text{proof} \rangle$

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: $\text{equiv } A \ r \implies r^{-1} \circ r = r$.

lemma *sym-trans-comp-subset*:

$\text{sym } r \implies \text{trans } r \implies r^{-1} \circ r \subseteq r$
 $\langle \text{proof} \rangle$

lemma *refl-on-comp-subset*: $\text{refl-on } A \ r \implies r \subseteq r^{-1} \circ r$
 $\langle \text{proof} \rangle$

lemma *equiv-comp-eq*: $\text{equiv } A \ r \implies r^{-1} \circ r = r$
 $\langle \text{proof} \rangle$

Second half.

lemma *comp-equivI*:

$r^{-1} \circ r = r \implies \text{Domain } r = A \implies \text{equiv } A \ r$
 $\langle \text{proof} \rangle$

29.2 Equivalence classes

lemma *equiv-class-subset*:

$\text{equiv } A \ r \implies (a, b) \in r \implies r^{\prime\prime}\{a\} \subseteq r^{\prime\prime}\{b\}$
 — lemma for the next result
 $\langle \text{proof} \rangle$

theorem *equiv-class-eq*: $\text{equiv } A \ r \implies (a, b) \in r \implies r^{\prime\prime}\{a\} = r^{\prime\prime}\{b\}$
 $\langle \text{proof} \rangle$

lemma *equiv-class-self*: $\text{equiv } A \ r \implies a \in A \implies a \in r^{\prime\prime}\{a\}$
 $\langle \text{proof} \rangle$

lemma *subset-equiv-class*:

$\text{equiv } A \ r \implies r^{\prime\prime}\{b\} \subseteq r^{\prime\prime}\{a\} \implies b \in A \implies (a, b) \in r$
 — lemma for the next result
 $\langle \text{proof} \rangle$

lemma *eq-equiv-class*:

$r^{\prime\prime}\{a\} = r^{\prime\prime}\{b\} \implies \text{equiv } A \ r \implies b \in A \implies (a, b) \in r$
 $\langle \text{proof} \rangle$

lemma *equiv-class-nondisjoint*:

$\text{equiv } A \ r \implies x \in (r^{\prime\prime}\{a\} \cap r^{\prime\prime}\{b\}) \implies (a, b) \in r$
 $\langle \text{proof} \rangle$

lemma *equiv-type*: $\text{equiv } A \ r \implies r \subseteq A \times A$
 $\langle \text{proof} \rangle$

theorem *equiv-class-eq-iff*:

$\text{equiv } A \ r \implies ((x, y) \in r) = (r^{\prime\prime}\{x\} = r^{\prime\prime}\{y\} \ \& \ x \in A \ \& \ y \in A)$
 $\langle \text{proof} \rangle$

theorem *eq-equiv-class-iff*:

equiv A r ==> x ∈ A ==> y ∈ A ==> (r“{x} = r“{y}) = ((x, y) ∈ r)
<proof>

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 ∈ A ==> r“{x} ∈ A//r*
<proof>

lemma *quotientE*:

X ∈ A//r ==> (!x. X = r“{x} ==> x ∈ A ==> P) ==> P
<proof>

lemma *Union-quotient*: *equiv A r ==> Union (A//r) = A*
<proof>

lemma *quotient-disj*:

equiv A r ==> X ∈ A//r ==> Y ∈ A//r ==> X = Y | (X \cap Y = {})
<proof>

lemma *quotient-eqI*:

[|equiv A r; X ∈ A//r; Y ∈ A//r; x ∈ X; y ∈ Y; (x,y) ∈ r|] ==> X = Y
<proof>

lemma *quotient-eq-iff*:

[|equiv A r; X ∈ A//r; Y ∈ A//r; x ∈ X; y ∈ Y|] ==> (X = Y) = ((x,y) ∈ r)
<proof>

lemma *eq-equiv-class-iff2*:

[|equiv A r; x ∈ A; y ∈ A|] ==> ({x}//r = {y}//r) = ((x,y) : r)
<proof>

lemma *quotient-empty [simp]*: *{ }//r = { }*
<proof>

lemma *quotient-is-empty [iff]*: *(A//r = { }) = (A = { })*
<proof>

lemma *quotient-is-empty2 [iff]*: *({ } = A//r) = (A = { })*
<proof>

lemma *singleton-quotient*: *{x}//r = {r“{x}}*
<proof>

lemma *quotient-diff1*:

$\llbracket \text{inj-on } (\%a. \{a\}/r) \ A; \ a \in A \rrbracket \implies (A - \{a\})/r = A/r - \{a\}/r$
 $\langle \text{proof} \rangle$

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 :: $('a \implies 'b) \implies ('a * 'a) \text{ set} \implies \text{bool}$

(**infixr** *respects* 80) **where**

f respects r == *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*

$\langle \text{proof} \rangle$

lemma *UN-equiv-class*:

equiv A r $\implies f\ \text{respects } r \implies a \in A$

$\implies (\bigcup x \in r^{-1}\{a\}. f\ x) = f\ a$

— Conversion rule

$\langle \text{proof} \rangle$

lemma *UN-equiv-class-type*:

equiv A r $\implies f\ \text{respects } r \implies X \in A/r \implies$

$(!!x. x \in A \implies f\ x \in B) \implies (\bigcup x \in X. f\ x) \in B$

$\langle \text{proof} \rangle$

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 $\implies f\ \text{respects } r \implies$

$(\bigcup x \in X. f\ x) = (\bigcup y \in Y. f\ y) \implies X \in A/r \implies Y \in A/r$

$\implies (!!x\ y. x \in A \implies y \in A \implies f\ x = f\ y \implies (x, y) \in r)$

$\implies X = Y$

$\langle \text{proof} \rangle$

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*:

$$(y1, z1) \in r1 ==> (y2, z2) \in r2 ==> f\ y1\ y2 = f\ z1\ z2$$

Abbreviation for the common case where the relations are identical

abbreviation

RESPECTS2:: [*'a* ==> *'a* ==> *'b*, (*'a* * *'a*) set] ==> *bool*
 (**infixr** *respects2* 80) **where**
f respects2 r == *congruent2 r r f*

lemma *congruent2-implies-congruent*:

equiv A r1 ==> congruent2 r1 r2 f ==> a ∈ A ==> congruent r2 (f a)
<proof>

lemma *congruent2-implies-congruent-UN*:

equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a ∈ A2 ==>
congruent r1 (λx1. ∪ x2 ∈ r2“{a}. f x1 x2)
<proof>

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
<proof>

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
<proof>

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*
<proof>

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.
<proof>

lemma *congruent2-commuteI*:

assumes *equivA*: *equiv A r*

and *commute*: $!!y\ z. y \in A \implies z \in A \implies f\ y\ z = f\ z\ y$
and *cong*t: $!!y\ z\ w. w \in A \implies (y, z) \in r \implies f\ w\ y = f\ w\ z$
shows *f respects2* *r*
 <proof>

29.6 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*: $finite\ A \implies r \subseteq A \times A \implies finite\ (A/r)$
 — recall *equiv* *?A* *?r* $\implies ?r \subseteq ?A \times ?A$
 <proof>

lemma *finite-equiv-class*:
 $finite\ A \implies r \subseteq A \times A \implies X \in A/r \implies finite\ X$
 <proof>

lemma *equiv-imp-dvd-card*:
 $finite\ A \implies equiv\ A\ r \implies \forall X \in A/r. k\ dvd\ card\ X$
 $\implies k\ dvd\ card\ A$
 <proof>

lemma *card-quotient-disjoint*:
 $\llbracket finite\ A; inj-on\ (\lambda x. \{x\} // r)\ A \rrbracket \implies card(A/r) = card\ A$
 <proof>

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* :: $((nat \times nat) \times (nat \times nat))\ set$ **where**
 [code del]: $intrel = \{((x, y), (u, v)) \mid x\ y\ u\ v. x + v = u + y\}$

typedef (*Integ*)
 $int = UNIV // intrel$
 <proof>

instantiation *int* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *ord*, *abs*, *sgn*}
begin

definition

Zero-int-def [code del]: $0 = \text{Abs-Integ } (\text{intrel } “ \{(0, 0)\} ”)$

definition

One-int-def [code del]: $1 = \text{Abs-Integ } (\text{intrel } “ \{(1, 0)\} ”)$

definition

add-int-def [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 [code del]:
 $-z = \text{Abs-Integ } (\bigcup (x, y) \in \text{Rep-Integ } z. \text{intrel } “ \{(y, x)\} ”)$

definition

diff-int-def [code del]: $z - w = z + (-w :: \text{int})$

definition

mult-int-def [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 [code del]:
 $z \leq w \longleftrightarrow (\exists x \ y \ u \ v. x+v \leq u+y \wedge (x, y) \in \text{Rep-Integ } z \wedge (u, v) \in \text{Rep-Integ } w)$

definition

less-int-def [code del]: $(z :: \text{int}) < w \longleftrightarrow z \leq w \wedge z \neq w$

definition

zabs-def: $|i :: \text{int}| = (\text{if } i < 0 \text{ then } -i \text{ else } i)$

definition

zsgn-def: $\text{sgn } (i :: \text{int}) = (\text{if } i=0 \text{ then } 0 \text{ else if } 0 < i \text{ then } 1 \text{ else } -1)$

instance $\langle \text{proof} \rangle$

end

30.2 Construction of the Integers

lemma *intrel-iff* [simp]: $((x, y), (u, v)) \in \text{intrel} = (x+v = u+y)$
 $\langle \text{proof} \rangle$

lemma *equiv-intrel*: *equiv UNIV intrel*
 ⟨*proof*⟩

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*]: $\text{intrel} \text{ “ } \{(x, y)\} \in \text{Integ}$
 ⟨*proof*⟩

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*]:
 $(!!x \ y. \ z = \text{Abs-Integ}(\text{intrel} \text{ “ } \{(x, y)\}) \implies P) \implies P$
 ⟨*proof*⟩

30.3 Arithmetic Operations

lemma *minus*: $-\text{Abs-Integ}(\text{intrel} \text{ “ } \{(x, y)\}) = \text{Abs-Integ}(\text{intrel} \text{ “ } \{(y, x)\})$
 ⟨*proof*⟩

lemma *add*:
 $\text{Abs-Integ}(\text{intrel} \text{ “ } \{(x, y)\}) + \text{Abs-Integ}(\text{intrel} \text{ “ } \{(u, v)\}) =$
 $\text{Abs-Integ}(\text{intrel} \text{ “ } \{(x+u, y+v)\})$
 ⟨*proof*⟩

Congruence property for multiplication

lemma *mult-congruent2*:
 $(\%p1 \ p2. (\%(x, y). (\%(u, v). \text{intrel} \text{ “ } \{(x*u + y*v, x*v + y*u)\}) \ p2) \ p1)$
respects2 intrel
 ⟨*proof*⟩

lemma *mult*:
 $\text{Abs-Integ}((\text{intrel} \text{ “ } \{(x, y)\})) * \text{Abs-Integ}((\text{intrel} \text{ “ } \{(u, v)\})) =$
 $\text{Abs-Integ}(\text{intrel} \text{ “ } \{(x*u + y*v, x*v + y*u)\})$
 ⟨*proof*⟩

The integers form a *comm-ring-1*

instance *int* :: *comm-ring-1*
 ⟨*proof*⟩

lemma *int-def*: $\text{of-nat } m = \text{Abs-Integ}(\text{intrel} \text{ “ } \{(m, 0)\})$
 ⟨*proof*⟩

30.4 The \leq Ordering

lemma *le*:

$(Abs-Integ(intrel^{''}\{(x,y)\}) \leq Abs-Integ(intrel^{''}\{(u,v)\})) = (x+v \leq u+y)$
 $\langle proof \rangle$

lemma *less*:

$(Abs-Integ(intrel^{''}\{(x,y)\}) < Abs-Integ(intrel^{''}\{(u,v)\})) = (x+v < u+y)$
 $\langle proof \rangle$

instance *int* :: *linorder*

$\langle proof \rangle$

instantiation *int* :: *distrib-lattice*

begin

definition

$(inf :: int \Rightarrow int \Rightarrow int) = min$

definition

$(sup :: int \Rightarrow int \Rightarrow int) = max$

instance

$\langle proof \rangle$

end

instance *int* :: *ordered-cancel-ab-semigroup-add*

$\langle proof \rangle$

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on $k_i 0$

lemma *zmult-zless-mono2-lemma*:

$(i :: int) < j ==> 0 < k ==> of-nat k * i < of-nat k * j$
 $\langle proof \rangle$

lemma *zero-le-imp-eq-int*: $(0 :: int) \leq k ==> \exists n. k = of-nat n$

$\langle proof \rangle$

lemma *zero-less-imp-eq-int*: $(0 :: int) < k ==> \exists n > 0. k = of-nat n$

$\langle proof \rangle$

lemma *zmult-zless-mono2*: $[\mid i < j; (0 :: int) < k \mid] ==> k * i < k * j$

$\langle proof \rangle$

The integers form an ordered integral domain

instance *int* :: *linordered-idom*

$\langle proof \rangle$

lemma *zless-imp-add1-zle*: $w < z \implies w + (1::\text{int}) \leq z$
 $\langle \text{proof} \rangle$

lemma *zless-iff-Suc-zadd*:
 $(w :: \text{int}) < z \iff (\exists n. z = w + \text{of-nat} (\text{Suc } n))$
 $\langle \text{proof} \rangle$

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* :: $\text{int} \Rightarrow 'a$ **where**
 $[\text{code del}]: \text{of-int } z = \text{contents } (\bigcup (i, j) \in \text{Rep-Integ } z. \{ \text{of-nat } i - \text{of-nat } j \})$

lemma *of-int*: $\text{of-int } (\text{Abs-Integ } (\text{intrel } “ \{(i,j)\} ”)) = \text{of-nat } i - \text{of-nat } j$
 $\langle \text{proof} \rangle$

lemma *of-int-0* [*simp*]: $\text{of-int } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *of-int-1* [*simp*]: $\text{of-int } 1 = 1$
 $\langle \text{proof} \rangle$

lemma *of-int-add* [*simp*]: $\text{of-int } (w+z) = \text{of-int } w + \text{of-int } z$
 $\langle \text{proof} \rangle$

lemma *of-int-minus* [*simp*]: $\text{of-int } (-z) = - (\text{of-int } z)$
 $\langle \text{proof} \rangle$

lemma *of-int-diff* [*simp*]: $\text{of-int } (w - z) = \text{of-int } w - \text{of-int } z$
 $\langle \text{proof} \rangle$

lemma *of-int-mult* [*simp*]: $\text{of-int } (w*z) = \text{of-int } w * \text{of-int } z$
 $\langle \text{proof} \rangle$

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: $\text{of-int } (\text{of-nat } n) = \text{of-nat } n$
 $\langle \text{proof} \rangle$

lemma *of-int-power*:
 $\text{of-int } (z \wedge n) = \text{of-int } z \wedge n$
 $\langle \text{proof} \rangle$

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*]:
 $of-int\ w = of-int\ z \longleftrightarrow w = z$
 ⟨*proof*⟩

Special cases where either operand is zero

lemma *of-int-eq-0-iff* [*simp*]:
 $of-int\ z = 0 \longleftrightarrow z = 0$
 ⟨*proof*⟩

lemma *of-int-0-eq-iff* [*simp*]:
 $0 = of-int\ z \longleftrightarrow z = 0$
 ⟨*proof*⟩

end

context *linordered-idom*
begin

Every *linordered-idom* has characteristic zero.

subclass *ring-char-0* ⟨*proof*⟩

lemma *of-int-le-iff* [*simp*]:
 $of-int\ w \leq of-int\ z \longleftrightarrow w \leq z$
 ⟨*proof*⟩

lemma *of-int-less-iff* [*simp*]:
 $of-int\ w < of-int\ z \longleftrightarrow w < z$
 ⟨*proof*⟩

lemma *of-int-0-le-iff* [*simp*]:
 $0 \leq of-int\ z \longleftrightarrow 0 \leq z$
 ⟨*proof*⟩

lemma *of-int-le-0-iff* [*simp*]:
 $of-int\ z \leq 0 \longleftrightarrow z \leq 0$
 ⟨*proof*⟩

lemma *of-int-0-less-iff* [*simp*]:
 $0 < of-int\ z \longleftrightarrow 0 < z$
 ⟨*proof*⟩

lemma *of-int-less-0-iff* [simp]:
 $\text{of-int } z < 0 \iff z < 0$
 ⟨proof⟩

end

lemma *of-int-eq-id* [simp]: $\text{of-int} = \text{id}$
 ⟨proof⟩

30.6 Magnitude of an Integer, as a Natural Number: *nat*

definition

$\text{nat} :: \text{int} \Rightarrow \text{nat}$

where

[code del]: $\text{nat } z = \text{contents } (\bigcup (x, y) \in \text{Rep-Integ } z. \{x-y\})$

lemma *nat*: $\text{nat } (\text{Abs-Integ } (\text{intrel}''\{(x,y)\})) = x-y$
 ⟨proof⟩

lemma *nat-int* [simp]: $\text{nat } (\text{of-nat } n) = n$
 ⟨proof⟩

lemma *nat-zero* [simp]: $\text{nat } 0 = 0$
 ⟨proof⟩

lemma *int-nat-eq* [simp]: $\text{of-nat } (\text{nat } z) = (\text{if } 0 \leq z \text{ then } z \text{ else } 0)$
 ⟨proof⟩

corollary *nat-0-le*: $0 \leq z \implies \text{of-nat } (\text{nat } z) = z$
 ⟨proof⟩

lemma *nat-le-0* [simp]: $z \leq 0 \implies \text{nat } z = 0$
 ⟨proof⟩

lemma *nat-le-eq-zle*: $0 < w \mid 0 \leq z \implies (\text{nat } w \leq \text{nat } z) = (w \leq z)$
 ⟨proof⟩

An alternative condition is $(0::'a) \leq w$

corollary *nat-mono-iff*: $0 < z \implies (\text{nat } w < \text{nat } z) = (w < z)$
 ⟨proof⟩

corollary *nat-less-eq-zless*: $0 \leq w \implies (\text{nat } w < \text{nat } z) = (w < z)$
 ⟨proof⟩

lemma *zless-nat-conj* [simp]: $(\text{nat } w < \text{nat } z) = (0 < z \ \& \ w < z)$
 ⟨proof⟩

lemma *nonneg-eq-int*:

fixes $z :: \text{int}$
assumes $0 \leq z$ **and** $\bigwedge m. z = \text{of-nat } m \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *nat-eq-iff*: $(\text{nat } w = m) = (\text{if } 0 \leq w \text{ then } w = \text{of-nat } m \text{ else } m=0)$
 $\langle \text{proof} \rangle$

corollary *nat-eq-iff2*: $(m = \text{nat } w) = (\text{if } 0 \leq w \text{ then } w = \text{of-nat } m \text{ else } m=0)$
 $\langle \text{proof} \rangle$

lemma *nat-less-iff*: $0 \leq w \implies (\text{nat } w < m) = (w < \text{of-nat } m)$
 $\langle \text{proof} \rangle$

lemma *nat-0-iff* [simp]: $\text{nat}(i::\text{int}) = 0 \longleftrightarrow i \leq 0$
 $\langle \text{proof} \rangle$

lemma *int-eq-iff*: $(\text{of-nat } m = z) = (m = \text{nat } z \ \& \ 0 \leq z)$
 $\langle \text{proof} \rangle$

lemma *zero-less-nat-eq* [simp]: $(0 < \text{nat } z) = (0 < z)$
 $\langle \text{proof} \rangle$

lemma *nat-add-distrib*:
 $\llbracket (0::\text{int}) \leq z; \ 0 \leq z' \rrbracket \implies \text{nat } (z+z') = \text{nat } z + \text{nat } z'$
 $\langle \text{proof} \rangle$

lemma *nat-diff-distrib*:
 $\llbracket (0::\text{int}) \leq z'; \ z' \leq z \rrbracket \implies \text{nat } (z-z') = \text{nat } z - \text{nat } z'$
 $\langle \text{proof} \rangle$

lemma *nat-zminus-int* [simp]: $\text{nat } (- (\text{of-nat } n)) = 0$
 $\langle \text{proof} \rangle$

lemma *zless-nat-eq-int-zless*: $(m < \text{nat } z) = (\text{of-nat } m < z)$
 $\langle \text{proof} \rangle$

context *ring-1*
begin

lemma *of-nat-nat*: $0 \leq z \implies \text{of-nat } (\text{nat } z) = \text{of-int } z$
 $\langle \text{proof} \rangle$

end

For termination proofs:

lemma *measure-function-int* [measure-function]: $\text{is-measure } (\text{nat } o \text{ abs}) \langle \text{proof} \rangle$

30.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-(\text{of-nat } (\text{Suc } n)) < (0 :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *negative-zless [iff]*: $-(\text{of-nat } (\text{Suc } n)) < (\text{of-nat } m :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *negative-zle-0*: $-\text{of-nat } n \leq (0 :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *negative-zle [iff]*: $-\text{of-nat } n \leq (\text{of-nat } m :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *not-zle-0-negative [simp]*: $\sim (0 \leq -(\text{of-nat } (\text{Suc } n)) :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *int-zle-neg*: $((\text{of-nat } n :: \text{int}) \leq -\text{of-nat } m) = (n = 0 \ \& \ m = 0)$
 $\langle \text{proof} \rangle$

lemma *not-int-zless-negative [simp]*: $\sim ((\text{of-nat } n :: \text{int}) < -\text{of-nat } m)$
 $\langle \text{proof} \rangle$

lemma *negative-eq-positive [simp]*: $((-\text{of-nat } n :: \text{int}) = \text{of-nat } m) = (n = 0 \ \& \ m = 0)$
 $\langle \text{proof} \rangle$

lemma *zle-iff-zadd*: $(w :: \text{int}) \leq z \iff (\exists n. z = w + \text{of-nat } n)$
 $\langle \text{proof} \rangle$

lemma *zadd-int-left*: $\text{of-nat } m + (\text{of-nat } n + z) = \text{of-nat } (m + n) + (z :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *int-Suc0-eq-1*: $\text{of-nat } (\text{Suc } 0) = (1 :: \text{int})$
 $\langle \text{proof} \rangle$

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]*:
 $P(\text{abs}(a :: 'a :: \text{linordered-idom})) = ((0 \leq a \implies P \ a) \ \& \ (a < 0 \implies P(-a)))$
 $\langle \text{proof} \rangle$

lemma *negD*: $(x :: \text{int}) < 0 \implies \exists n. x = -(\text{of-nat } (\text{Suc } n))$
 $\langle \text{proof} \rangle$

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 :: \text{int}) = \text{of-nat } n ==> P; \ !n. z = - (\text{of-nat } (\text{Suc } n)) ==> P \]]$$

$$==> P$$

$$\langle \text{proof} \rangle$$

theorem *int-of-nat-induct* [*induct type: int, case-names nonneg neg*]:

$$[[!!\ n. P (\text{of-nat } n :: \text{int}); \ !n. P (- (\text{of-nat } (\text{Suc } n))) \]]$$

$$==> P\ z$$

$$\langle \text{proof} \rangle$$

Contributed by Brian Huffman

theorem *int-diff-cases*:

obtains (*diff*) *m n where* (*z::int*) = *of-nat m - of-nat n*

$$\langle \text{proof} \rangle$$

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 mod 2*) is 0 or 1, even if *m* is negative; For instance, $-5 \text{ div } 2 = -3$ and $-5 \text{ mod } 2 = 1$; 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*

$$[\text{code del}]: \text{Pls} = 0$$

definition

Min :: *int where*

$$[\text{code del}]: \text{Min} = -\ 1$$

definition

Bit0 :: *int \Rightarrow int where*

$$[\text{code del}]: \text{Bit0 } k = k + k$$

definition

Bit1 :: *int \Rightarrow int where*

[code del]: $\text{Bit1 } k = 1 + k + k$

class *number* = — for numeric types: nat, int, real, ...
fixes *number-of* :: $\text{int} \Rightarrow 'a$

$\langle ML \rangle$

syntax

$\text{-Numeral} :: \text{num-const} \Rightarrow 'a \quad (-)$

$\langle ML \rangle$

abbreviation

$\text{Numeral0} \equiv \text{number-of } \text{Pls}$

abbreviation

$\text{Numeral1} \equiv \text{number-of } (\text{Bit1 } \text{Pls})$

lemma *Let-number-of [simp]*: $\text{Let } (\text{number-of } v) f = f (\text{number-of } v)$
 — Unfold all *lets* involving constants
 $\langle \text{proof} \rangle$

definition

$\text{succ} :: \text{int} \Rightarrow \text{int}$ **where**
 [code del]: $\text{succ } k = k + 1$

definition

$\text{pred} :: \text{int} \Rightarrow \text{int}$ **where**
 [code del]: $\text{pred } k = k - 1$

lemmas

$\text{max-number-of [simp]} = \text{max-def}$
 $[\text{of number-of } u \text{ number-of } v, \text{ standard}]$

and

$\text{min-number-of [simp]} = \text{min-def}$
 $[\text{of number-of } u \text{ number-of } v, \text{ standard}]$
 — unfolding *minx* and *max* on numerals

lemmas *numeral-simps* =

$\text{succ-def pred-def Pls-def Min-def Bit0-def Bit1-def}$

Removal of leading zeroes

lemma *Bit0-Pls [simp, code-post]*:
 $\text{Bit0 } \text{Pls} = \text{Pls}$
 $\langle \text{proof} \rangle$

lemma *Bit1-Min [simp, code-post]*:
 $\text{Bit1 } \text{Min} = \text{Min}$
 $\langle \text{proof} \rangle$

lemmas *normalize-bin-simps* =
Bit0-Pls Bit1-Min

30.9.2 Successor and predecessor functions

Successor

lemma *succ-Pls*:
succ Pls = Bit1 Pls
<proof>

lemma *succ-Min*:
succ Min = Pls
<proof>

lemma *succ-Bit0*:
succ (Bit0 k) = Bit1 k
<proof>

lemma *succ-Bit1*:
succ (Bit1 k) = Bit0 (succ k)
<proof>

lemmas *succ-bin-simps [simp]* =
succ-Pls succ-Min succ-Bit0 succ-Bit1

Predecessor

lemma *pred-Pls*:
pred Pls = Min
<proof>

lemma *pred-Min*:
pred Min = Bit0 Min
<proof>

lemma *pred-Bit0*:
pred (Bit0 k) = Bit1 (pred k)
<proof>

lemma *pred-Bit1*:
pred (Bit1 k) = Bit0 k
<proof>

lemmas *pred-bin-simps [simp]* =
pred-Pls pred-Min pred-Bit0 pred-Bit1

30.9.3 Binary arithmetic

Addition

lemma *add-Pls*:

$$Pls + k = k$$

<proof>

lemma *add-Min*:

$$Min + k = pred\ k$$

<proof>

lemma *add-Bit0-Bit0*:

$$(Bit0\ k) + (Bit0\ l) = Bit0\ (k + l)$$

<proof>

lemma *add-Bit0-Bit1*:

$$(Bit0\ k) + (Bit1\ l) = Bit1\ (k + l)$$

<proof>

lemma *add-Bit1-Bit0*:

$$(Bit1\ k) + (Bit0\ l) = Bit1\ (k + l)$$

<proof>

lemma *add-Bit1-Bit1*:

$$(Bit1\ k) + (Bit1\ l) = Bit0\ (k + succ\ l)$$

<proof>

lemma *add-Pls-right*:

$$k + Pls = k$$

<proof>

lemma *add-Min-right*:

$$k + Min = pred\ k$$

<proof>

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*:

$$- Pls = Pls$$

<proof>

lemma *minus-Min*:

$$- Min = Bit1\ Pls$$

<proof>

lemma *minus-Bit0*:

$$- (Bit0\ k) = Bit0\ (-\ k)$$

<proof>

lemma *minus-Bit1*:

$$- (Bit1\ k) = Bit1\ (pred\ (-\ k))$$

$\langle proof \rangle$

lemmas *minus-bin-simps* [simp] =

minus-Pls minus-Min minus-Bit0 minus-Bit1

Subtraction

lemma *diff-bin-simps* [simp]:

$$k - Pls = k$$

$$k - Min = succ\ k$$

$$Pls - (Bit0\ l) = Bit0\ (Pls - l)$$

$$Pls - (Bit1\ l) = Bit1\ (Min - l)$$

$$Min - (Bit0\ l) = Bit1\ (Min - l)$$

$$Min - (Bit1\ l) = Bit0\ (Min - l)$$

$$(Bit0\ k) - (Bit0\ l) = Bit0\ (k - l)$$

$$(Bit0\ k) - (Bit1\ l) = Bit1\ (pred\ k - l)$$

$$(Bit1\ k) - (Bit0\ l) = Bit1\ (k - l)$$

$$(Bit1\ k) - (Bit1\ l) = Bit0\ (k - l)$$

$\langle proof \rangle$

Multiplication

lemma *mult-Pls*:

$$Pls * w = Pls$$

$\langle proof \rangle$

lemma *mult-Min*:

$$Min * k = -\ k$$

$\langle proof \rangle$

lemma *mult-Bit0*:

$$(Bit0\ k) * l = Bit0\ (k * l)$$

$\langle proof \rangle$

lemma *mult-Bit1*:

$$(Bit1\ k) * l = (Bit0\ (k * l)) + l$$

$\langle proof \rangle$

lemmas *mult-bin-simps* [simp] =

mult-Pls mult-Min mult-Bit0 mult-Bit1

30.9.4 Binary comparisons

Preliminaries

lemma *even-less-0-iff*:

$$a + a < 0 \iff a < (0 :: 'a :: linordered-idom)$$

$\langle proof \rangle$

lemma *le-imp-0-less*:

assumes *le*: $0 \leq z$

shows $(0::int) < 1 + z$

<proof>

lemma *odd-less-0-iff*:

$(1 + z + z < 0) = (z < (0::int))$

<proof>

lemma *bin-less-0-simps*:

$Pls < 0 \longleftrightarrow False$

$Min < 0 \longleftrightarrow True$

$Bit0\ w < 0 \longleftrightarrow w < 0$

$Bit1\ w < 0 \longleftrightarrow w < 0$

<proof>

lemma *less-bin-lemma*: $k < l \longleftrightarrow k - l < (0::int)$

<proof>

lemma *le-iff-pred-less*: $k \leq l \longleftrightarrow pred\ k < l$

<proof>

lemma *succ-pred*: $succ\ (pred\ x) = x$

<proof>

Less-than

lemma *less-bin-simps [simp]*:

$Pls < Pls \longleftrightarrow False$

$Pls < Min \longleftrightarrow False$

$Pls < Bit0\ k \longleftrightarrow Pls < k$

$Pls < Bit1\ k \longleftrightarrow Pls \leq k$

$Min < Pls \longleftrightarrow True$

$Min < Min \longleftrightarrow False$

$Min < Bit0\ k \longleftrightarrow Min < k$

$Min < Bit1\ k \longleftrightarrow Min < k$

$Bit0\ k < Pls \longleftrightarrow k < Pls$

$Bit0\ k < Min \longleftrightarrow k \leq Min$

$Bit1\ k < Pls \longleftrightarrow k < Pls$

$Bit1\ k < Min \longleftrightarrow k < Min$

$Bit0\ k < Bit0\ l \longleftrightarrow k < l$

$Bit0\ k < Bit1\ l \longleftrightarrow k \leq l$

$Bit1\ k < Bit0\ l \longleftrightarrow k < l$

$Bit1\ k < Bit1\ l \longleftrightarrow k < l$

<proof>

Less-than-or-equal

lemma *le-bin-simps [simp]*:

$Pls \leq Pls \longleftrightarrow True$

$Pls \leq Min \longleftrightarrow False$

```

Pls ≤ Bit0 k ↔ Pls ≤ k
Pls ≤ Bit1 k ↔ Pls ≤ k
Min ≤ Pls ↔ True
Min ≤ Min ↔ True
Min ≤ Bit0 k ↔ Min < k
Min ≤ Bit1 k ↔ Min ≤ k
Bit0 k ≤ Pls ↔ k ≤ Pls
Bit0 k ≤ Min ↔ k ≤ Min
Bit1 k ≤ Pls ↔ k < 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
⟨proof⟩

```

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
⟨proof⟩

```

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⟩

```


end

lemma *number-of-is-id*:
 $\text{number-of } (k::\text{int}) = k$
 $\langle \text{proof} \rangle$

lemma *number-of-succ*:
 $\text{number-of } (\text{succ } k) = (1 + \text{number-of } k :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-pred*:
 $\text{number-of } (\text{pred } w) = (- 1 + \text{number-of } w :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-minus*:
 $\text{number-of } (\text{uminus } w) = (- (\text{number-of } w) :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-add*:
 $\text{number-of } (v + w) = (\text{number-of } v + \text{number-of } w :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-diff*:
 $\text{number-of } (v - w) = (\text{number-of } v - \text{number-of } w :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-mult*:
 $\text{number-of } (v * w) = (\text{number-of } v * \text{number-of } w :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

lemma *double-number-of-Bit0*:
 $(1 + 1) * \text{number-of } w = (\text{number-of } (\text{Bit0 } w) :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

Converting numerals 0 and 1 to their abstract versions.

lemma *numeral-0-eq-0* [*simp*, *code-post*]:
 $\text{Numeral0} = (0 :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *numeral-1-eq-1* [*simp*, *code-post*]:
 $\text{Numeral1} = (1 :: 'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simplrule until later: it is *number-of-Min* re-oriented!

lemma *numeral-m1-eq-minus-1*:
 $(-1 :: 'a :: \text{number-ring}) = -\ 1$
 $\langle \text{proof} \rangle$

lemma *mult-minus1 [simp]*:
 $-1 * z = -(z :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *mult-minus1-right [simp]*:
 $z * -1 = -(z :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *minus-number-of-mult [simp]*:
 $-(\text{number-of } w) * z = \text{number-of } (\text{uminus } w) * (z :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

Subtraction

lemma *diff-number-of-eq*:
 $\text{number-of } v - \text{number-of } w =$
 $(\text{number-of } (v + \text{uminus } w) :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-Pls*:
 $\text{number-of } \text{Pls} = (0 :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-Min*:
 $\text{number-of } \text{Min} = (-\ 1 :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *number-of-Bit0*:
 $\text{number-of } (\text{Bit0 } w) = (0 :: 'a :: \text{number-ring}) + (\text{number-of } w) + (\text{number-of } w)$
 $\langle \text{proof} \rangle$

lemma *number-of-Bit1*:
 $\text{number-of } (\text{Bit1 } w) = (1 :: 'a :: \text{number-ring}) + (\text{number-of } w) + (\text{number-of } w)$
 $\langle \text{proof} \rangle$

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*: $\text{iszero } 0$
 $\langle \text{proof} \rangle$

lemma *iszero-Numeral0*: $\text{iszero } (\text{Numeral0} :: 'a :: \text{number-ring})$

$\langle \text{proof} \rangle$

lemma *not-iszero-1*: $\neg \text{iszero } 1$

$\langle \text{proof} \rangle$

lemma *not-iszero-Numeral1*: $\neg \text{iszero } (\text{Numeral1} :: 'a::\text{number-ring})$

$\langle \text{proof} \rangle$

lemma *eq-number-of-eq* [simp]:

$((\text{number-of } x :: 'a::\text{number-ring}) = \text{number-of } y) =$
 $\text{iszero } (\text{number-of } (x + \text{uminus } y) :: 'a)$

$\langle \text{proof} \rangle$

lemma *iszero-number-of-Pls*:

$\text{iszero } ((\text{number-of } \text{Pls}) :: 'a::\text{number-ring})$

$\langle \text{proof} \rangle$

lemma *nonzero-number-of-Min*:

$\sim \text{iszero } ((\text{number-of } \text{Min}) :: 'a::\text{number-ring})$

$\langle \text{proof} \rangle$

30.10.2 Comparisons, for Ordered Rings

lemmas *double-eq-0-iff* = *double-zero*

lemma *odd-nonzero*:

$1 + z + z \neq (0 :: \text{int})$

$\langle \text{proof} \rangle$

lemma *iszero-number-of-Bit0*:

$\text{iszero } (\text{number-of } (\text{Bit0 } w) :: 'a) =$
 $\text{iszero } (\text{number-of } w :: 'a :: \{\text{ring-char-0}, \text{number-ring}\})$

$\langle \text{proof} \rangle$

lemma *iszero-number-of-Bit1*:

$\sim \text{iszero } (\text{number-of } (\text{Bit1 } w) :: 'a :: \{\text{ring-char-0}, \text{number-ring}\})$

$\langle \text{proof} \rangle$

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*:

$(a + a < 0) = (a < (0 :: 'a :: \text{linordered-idom}))$

$\langle \text{proof} \rangle$

lemma *odd-less-0*:

$(1 + z + z < 0) = (z < (0::int))$
 $\langle proof \rangle$

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*:
 $abs(number-of\ x::'a::\{\text{linordered-idom}, \text{number-ring}\}) =$
 $(if\ number-of\ x < (0::'a)\ then\ -number-of\ x\ else\ number-of\ x)$
 $\langle proof \rangle$

Re-orientation of the equation $nnn=x$

lemma *number-of-reorient*:
 $(number-of\ w = x) = (x = number-of\ w)$
 $\langle proof \rangle$

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]:
 $(number-of\ x::'a::\{\text{linordered-idom}, \text{number-ring}\}) < number-of\ y \longleftrightarrow x < y$
 $\langle proof \rangle$

lemma *le-number-of* [simp]:
 $(number-of\ x::'a::\{\text{linordered-idom}, \text{number-ring}\}) \leq number-of\ y \longleftrightarrow x \leq y$
 $\langle proof \rangle$

lemma *eq-number-of* [simp]:

$(\text{number-of } x :: 'a :: \{\text{ring-char-0}, \text{number-ring}\}) = \text{number-of } y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

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]*:
 $\text{number-of } v + (\text{number-of } w + z) =$
 $(\text{number-of } (v + w) + z :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *mult-number-of-left [simp]*:
 $\text{number-of } v * (\text{number-of } w * z) =$
 $(\text{number-of } (v * w) * z :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *add-number-of-diff1*:
 $\text{number-of } v + (\text{number-of } w - c) =$
 $\text{number-of } (v + w) - (c :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *add-number-of-diff2 [simp]*:
 $\text{number-of } v + (c - \text{number-of } w) =$
 $\text{number-of } (v + \text{uminus } w) + (c :: 'a :: \text{number-ring})$
 $\langle \text{proof} \rangle$

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*)
 $\text{Ints } (\mathbb{Z})$

lemma *Ints-of-int [simp]*: $\text{of-int } z \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-of-nat [simp]*: $\text{of-nat } n \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-0 [simp]*: $0 \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *Ints-1* [*simp*]: $1 \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-add* [*simp*]: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a + b \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-minus* [*simp*]: $a \in \mathbb{Z} \implies -a \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a - b \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \implies b \in \mathbb{Z} \implies a * b \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-power* [*simp*]: $a \in \mathbb{Z} \implies a ^ n \in \mathbb{Z}$

$\langle \text{proof} \rangle$

lemma *Ints-cases* [*cases set: Ints*]:

assumes $q \in \mathbb{Z}$

obtains $(\text{of-int } z) \text{ where } q = \text{of-int } z$

$\langle \text{proof} \rangle$

lemma *Ints-induct* [*case-names of-int, induct set: Ints*]:

$q \in \mathbb{Z} \implies (\bigwedge z. P (\text{of-int } z)) \implies P q$

$\langle \text{proof} \rangle$

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 $(a + a = 0) = (a = (0::'a::\text{ring-char-0}))$

$\langle \text{proof} \rangle$

lemma *Ints-odd-nonzero*:

assumes *in-Ints*: $a \in \text{Ints}$

shows $1 + a + a \neq (0::'a::\text{ring-char-0})$

$\langle \text{proof} \rangle$

lemma *Ints-number-of* [*simp*]:

$(\text{number-of } w :: 'a::\text{number-ring}) \in \text{Ints}$

$\langle \text{proof} \rangle$

lemma *Nats-number-of* [*simp*]:

$\text{Int.Pls} \leq w \implies (\text{number-of } w :: 'a::\text{number-ring}) \in \text{Nats}$

$\langle \text{proof} \rangle$

lemma *Ints-odd-less-0*:
 assumes *in-Ints*: $a \in \text{Ints}$
 shows $(1 + a + a < 0) = (a < (0::'a::\text{linordered-idom}))$
 $\langle \text{proof} \rangle$

30.12 *setsum and setprod*

lemma *of-nat-setsum*: $\text{of-nat} (\text{setsum } f \ A) = (\sum x \in A. \ \text{of-nat}(f \ x))$
 $\langle \text{proof} \rangle$

lemma *of-int-setsum*: $\text{of-int} (\text{setsum } f \ A) = (\sum x \in A. \ \text{of-int}(f \ x))$
 $\langle \text{proof} \rangle$

lemma *of-nat-setprod*: $\text{of-nat} (\text{setprod } f \ A) = (\prod x \in A. \ \text{of-nat}(f \ x))$
 $\langle \text{proof} \rangle$

lemma *of-int-setprod*: $\text{of-int} (\text{setprod } f \ A) = (\prod x \in A. \ \text{of-int}(f \ x))$
 $\langle \text{proof} \rangle$

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*: $\text{Numeral0} + a = (a::'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *add-numeral-0-right*: $a + \text{Numeral0} = (a::'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *mult-numeral-1*: $\text{Numeral1} * a = (a::'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *mult-numeral-1-right*: $a * \text{Numeral1} = (a::'a::\text{number-ring})$
 $\langle \text{proof} \rangle$

lemma *divide-numeral-1*: $a / \text{Numeral1} = (a::'a::\{\text{number-ring}, \text{field}\})$
 $\langle \text{proof} \rangle$

lemma *inverse-numeral-1*:
 $\text{inverse } \text{Numeral1} = (\text{Numeral1}::'a::\{\text{number-ring}, \text{field}\})$
 $\langle \text{proof} \rangle$

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*: $[[f\ x\ y = g\ x\ y; x = x'; y = y']] ==> f\ x'\ y' = g\ x'\ y'$
 $\langle proof \rangle$

lemmas *add-number-of-eq* = *number-of-add* [*symmetric*]

Allow 1 on either or both sides

lemma *one-add-one-is-two*: $1 + 1 = (2::'a::number-ring)$
 $\langle proof \rangle$

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

⟨ML⟩

30.16 Lemmas About Small Numerals

lemma *of-int-m1* [simp]: *of-int -1* = *(-1 :: 'a :: number-ring)*

⟨proof⟩

lemma *abs-minus-one* [simp]: *abs (-1)* = *(1 :: 'a :: {linordered-idom, number-ring})*

⟨proof⟩

lemma *abs-power-minus-one* [simp]:

abs(-1 ^ n) = *(1 :: 'a :: {linordered-idom, number-ring})*

⟨proof⟩

lemma *of-int-number-of-eq* [simp]:

of-int (number-of v) = *(number-of v :: 'a :: number-ring)*

⟨proof⟩

Lemmas for specialist use, NOT as default simprules

lemma *mult-2*: *2 * z* = *(z + z :: 'a :: number-ring)*

⟨proof⟩

lemma *mult-2-right*: *z * 2* = *(z + z :: 'a :: number-ring)*

⟨proof⟩

30.17 More Inequality Reasoning

lemma *zless-add1-eq*: *(w < z + (1 :: int))* = *(w < z | w = z)*

⟨proof⟩

lemma *add1-zle-eq*: *(w + (1 :: int) ≤ z)* = *(w < z)*

⟨proof⟩

lemma *zle-diff1-eq* [*simp*]: $(w \leq z - (1::int)) = (w < z)$
 $\langle proof \rangle$

lemma *zle-add1-eq-le* [*simp*]: $(w < z + (1::int)) = (w \leq z)$
 $\langle proof \rangle$

lemma *int-one-le-iff-zero-less*: $((1::int) \leq z) = (0 < z)$
 $\langle proof \rangle$

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*: $\text{nat } 0 = 0$
 $\langle proof \rangle$

lemma *nat-1*: $\text{nat } 1 = \text{Suc } 0$
 $\langle proof \rangle$

lemma *nat-2*: $\text{nat } 2 = \text{Suc } (\text{Suc } 0)$
 $\langle proof \rangle$

lemma *one-less-nat-eq* [*simp*]: $(\text{Suc } 0 < \text{nat } z) = (1 < z)$
 $\langle proof \rangle$

This simplifies expressions of the form *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*]:

$P(\text{nat}(i::int)) = ((\forall n. i = \text{of-nat } n \longrightarrow P n) \ \& \ (i < 0 \longrightarrow P 0))$

(**is** ?*P* = (?*L* & ?*R*))

$\langle proof \rangle$

context *ring-1*

begin

lemma *of-int-of-nat* [*nitpick-simp*]:

$\text{of-int } k = (\text{if } k < 0 \text{ then } - \text{of-nat } (\text{nat } (- k)) \text{ else } \text{of-nat } (\text{nat } k))$

$\langle proof \rangle$

end

lemma *nat-mult-distrib*:

fixes $z\ z' :: \text{int}$
assumes $0 \leq z$
shows $\text{nat } (z * z') = \text{nat } z * \text{nat } z'$
 $\langle \text{proof} \rangle$

lemma *nat-mult-distrib-neg*: $z \leq (0 :: \text{int}) \implies \text{nat}(z * z') = \text{nat}(-z) * \text{nat}(-z')$
 $\langle \text{proof} \rangle$

lemma *nat-abs-mult-distrib*: $\text{nat } (\text{abs } (w * z)) = \text{nat } (\text{abs } w) * \text{nat } (\text{abs } z)$
 $\langle \text{proof} \rangle$

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*: $\text{wf } (\text{int-ge-less-than } d)$
 $\langle \text{proof} \rangle$

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition

int-ge-less-than2 $:: \text{int} \Rightarrow (\text{int} * \text{int}) \text{ set}$

where

int-ge-less-than2 $d = \{(z', z). d \leq z \ \& \ z' < z\}$

theorem *wf-int-ge-less-than2*: $\text{wf } (\text{int-ge-less-than2 } d)$
 $\langle \text{proof} \rangle$

abbreviation

int $:: \text{nat} \Rightarrow \text{int}$

where

int $\equiv \text{of-nat}$

theorem *int-ge-induct* [*case-names* *base step*, *induct set*: *int*]:
fixes $i :: \text{int}$
assumes *ge*: $k \leq i$ **and**
base: $P\ k$ **and**
step: $\bigwedge i. k \leq i \implies P\ i \implies P\ (i + 1)$
shows $P\ i$
 $\langle \text{proof} \rangle$

theorem *int-gr-induct* [*case-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 $P\ i$
 $\langle proof \rangle$

theorem *int-le-induct* [*consumes 1, case-names base step*]:
assumes $le: i \leq (k::int)$ **and**
 $base: P(k)$ **and**
 $step: \bigwedge i. \llbracket i \leq k; P\ i \rrbracket \implies P(i - 1)$
shows $P\ i$
 $\langle proof \rangle$

theorem *int-less-induct* [*consumes 1, case-names base step*]:
assumes $less: (i::int) < k$ **and**
 $base: P(k - 1)$ **and**
 $step: \bigwedge i. \llbracket i < k; P\ i \rrbracket \implies P(i - 1)$
shows $P\ i$
 $\langle proof \rangle$

theorem *int-induct* [*case-names base step1 step2*]:
fixes $k :: int$
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 $P\ i$
 $\langle proof \rangle$

30.20 Intermediate value theorems

lemma *int-val-lemma*:
 $(\forall i < n::nat. abs(f(i+1) - f\ i) \leq 1) \dashv\dashv$
 $f\ 0 \leq k \dashv\dashv k \leq f\ n \dashv\dashv (\exists i \leq n. f\ i = (k::int))$
 $\langle proof \rangle$

lemmas *nat0-intermed-int-val* = *int-val-lemma* [*rule-format (no-asm)*]

lemma *nat-intermed-int-val*:
 $\llbracket \forall i. m \leq i \ \& \ i < n \dashv\dashv abs(f(i + 1::nat) - f\ i) \leq 1; m < n;$
 $f\ m \leq k; k \leq f\ n \rrbracket \implies ?\ i. m \leq i \ \& \ i \leq n \ \& \ f\ i = (k::int)$
 $\langle proof \rangle$

30.21 Products and 1, by T. M. Rasmussen

lemma *zabs-less-one-iff* [*simp*]: $(|z| < 1) = (z = (0::int))$
 $\langle proof \rangle$

lemma *abs-zmult-eq-1*:
assumes $mn: |m * n| = 1$
shows $|m| = (1::int)$

$\langle proof \rangle$

lemma *pos-zmult-eq-1-iff-lemma*: $(m * n = 1) ==> m = (1::int) \mid m = -1$
 $\langle proof \rangle$

lemma *pos-zmult-eq-1-iff*:
assumes $0 < (m::int)$ **shows** $(m * n = 1) = (m = 1 \ \& \ n = 1)$
 $\langle proof \rangle$

lemma *zmult-eq-1-iff*: $(m*n = (1::int)) = ((m = 1 \ \& \ n = 1) \mid (m = -1 \ \& \ n = -1))$
 $\langle proof \rangle$

lemma *infinite-UNIV-int*: $\neg \text{finite } (UNIV::int \text{ set})$
 $\langle proof \rangle$

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 $(1 < - b) = (b < -1)$
<proof>

lemma *le-minus-iff-1 [simp,no-atp]:*
fixes *b::'b::{linordered-idom,number-ring}*
shows $(1 \leq - b) = (b \leq -1)$
<proof>

lemma *equation-minus-iff-1 [simp,no-atp]:*
fixes *b::'b::number-ring*
shows $(1 = - b) = (b = -1)$
<proof>

lemma *minus-less-iff-1 [simp,no-atp]:*
fixes *a::'a::{linordered-idom,number-ring}*
shows $(- a < 1) = (-1 < a)$
<proof>

lemma *minus-le-iff-1 [simp,no-atp]:*
fixes *a::'a::{linordered-idom,number-ring}*
shows $(- a \leq 1) = (-1 \leq a)$
<proof>

lemma *minus-equation-iff-1 [simp,no-atp]:*

fixes $a::'b::\text{number-ring}$
shows $(- a = 1) = (a = -1)$
 $\langle \text{proof} \rangle$

Cancellation of constant factors in comparisons ($<$ and \leq)

lemmas $\text{mult-less-cancel-left-number-of} \ [\text{simp}, \text{no-atp}] =$
 $\text{mult-less-cancel-left} \ [\text{of number-of } v, \text{standard}]$

lemmas $\text{mult-less-cancel-right-number-of} \ [\text{simp}, \text{no-atp}] =$
 $\text{mult-less-cancel-right} \ [\text{of - number-of } v, \text{standard}]$

lemmas $\text{mult-le-cancel-left-number-of} \ [\text{simp}, \text{no-atp}] =$
 $\text{mult-le-cancel-left} \ [\text{of number-of } v, \text{standard}]$

lemmas $\text{mult-le-cancel-right-number-of} \ [\text{simp}, \text{no-atp}] =$
 $\text{mult-le-cancel-right} \ [\text{of - number-of } v, \text{standard}]$

Multiplying out constant divisors in comparisons ($<$, \leq and $=$)

lemmas $\text{le-divide-eq-number-of1} \ [\text{simp}] = \text{le-divide-eq} \ [\text{of - - number-of } w, \text{standard}]$

lemmas $\text{divide-le-eq-number-of1} \ [\text{simp}] = \text{divide-le-eq} \ [\text{of - number-of } w, \text{standard}]$

lemmas $\text{less-divide-eq-number-of1} \ [\text{simp}] = \text{less-divide-eq} \ [\text{of - - number-of } w, \text{standard}]$

lemmas $\text{divide-less-eq-number-of1} \ [\text{simp}] = \text{divide-less-eq} \ [\text{of - number-of } w, \text{standard}]$

lemmas $\text{eq-divide-eq-number-of1} \ [\text{simp}] = \text{eq-divide-eq} \ [\text{of - - number-of } w, \text{standard}]$

lemmas $\text{divide-eq-eq-number-of1} \ [\text{simp}] = \text{divide-eq-eq} \ [\text{of - number-of } w, \text{standard}]$

30.22.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas $\text{le-divide-eq-number-of} = \text{le-divide-eq} \ [\text{of number-of } w, \text{standard}]$

lemmas $\text{divide-le-eq-number-of} = \text{divide-le-eq} \ [\text{of - - number-of } w, \text{standard}]$

lemmas $\text{less-divide-eq-number-of} = \text{less-divide-eq} \ [\text{of number-of } w, \text{standard}]$

lemmas $\text{divide-less-eq-number-of} = \text{divide-less-eq} \ [\text{of - - number-of } w, \text{standard}]$

lemmas $\text{eq-divide-eq-number-of} = \text{eq-divide-eq} \ [\text{of number-of } w, \text{standard}]$

lemmas $\text{divide-eq-eq-number-of} = \text{divide-eq-eq} \ [\text{of - - number-of } w, \text{standard}]$

Not good as automatic simprules because they cause case splits.

lemmas $\text{divide-const-simps} =$

$\text{le-divide-eq-number-of divide-le-eq-number-of less-divide-eq-number-of}$
 $\text{divide-less-eq-number-of eq-divide-eq-number-of divide-eq-eq-number-of}$
 $\text{le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1}$

Division By -1

lemma $\text{divide-minus1} \ [\text{simp}]$:

$x/-1 = -(x::'a::\{\text{field-inverse-zero, number-ring}\})$

<proof>

lemma *minus1-divide* [simp]:

$$-1 / (x::'a::\{\text{field-inverse-zero, number-ring}\}) = - (1/x)$$

<proof>

lemma *half-gt-zero-iff*:

$$(0 < r/2) = (0 < (r::'a::\{\text{linordered-field-inverse-zero, number-ring}\}))$$

<proof>

lemmas *half-gt-zero* [simp] = *half-gt-zero-iff* [THEN *iffD2*, standard]

lemma *divide-Numeral1*:

$$(x::'a::\{\text{field, number-ring}\}) / \text{Numeral1} = x$$

<proof>

lemma *divide-Numeral0*:

$$(x::'a::\{\text{field-inverse-zero, number-ring}\}) / \text{Numeral0} = 0$$

<proof>

30.23 The divides relation

lemma *zdvd-antisym-nonneg*:

$$0 \leq m \implies 0 \leq n \implies m \text{ dvd } n \implies n \text{ dvd } m \implies m = (n::\text{int})$$

<proof>

lemma *zdvd-antisym-abs*: **assumes** $(a::\text{int}) \text{ dvd } b$ **and** $b \text{ dvd } a$

shows $|a| = |b|$

<proof>

lemma *zdvd-zdiffD*: $k \text{ dvd } m - n \implies k \text{ dvd } n \implies k \text{ dvd } (m::\text{int})$

<proof>

lemma *zdvd-reduce*: $(k \text{ dvd } n + k * m) = (k \text{ dvd } (n::\text{int}))$

<proof>

lemma *dvd-imp-le-int*:

fixes $d \ i :: \text{int}$

assumes $i \neq 0$ **and** $d \text{ dvd } i$

shows $|d| \leq |i|$

<proof>

lemma *zdvd-not-zless*:

fixes $m \ n :: \text{int}$

assumes $0 < m$ **and** $m < n$

shows $\neg n \text{ dvd } m$

<proof>

lemma *zdvd-mult-cancel*: **assumes** $d:k * m \text{ dvd } k * n$ **and** $kz:k \neq (0::\text{int})$

shows $m \text{ dvd } n$
 $\langle \text{proof} \rangle$

theorem zdvd-int : $(x \text{ dvd } y) = (\text{int } x \text{ dvd int } y)$
 $\langle \text{proof} \rangle$

lemma zdvd1-eq[simp] : $(x::\text{int}) \text{ dvd } 1 = (|x| = 1)$
 $\langle \text{proof} \rangle$

lemma zdvd-mult-cancel1 :
assumes $mp:m \neq (0::\text{int})$ **shows** $(m * n \text{ dvd } m) = (|n| = 1)$
 $\langle \text{proof} \rangle$

lemma int-dvd-iff : $(\text{int } m \text{ dvd } z) = (m \text{ dvd nat } (\text{abs } z))$
 $\langle \text{proof} \rangle$

lemma dvd-int-iff : $(z \text{ dvd int } m) = (\text{nat } (\text{abs } z) \text{ dvd } m)$
 $\langle \text{proof} \rangle$

lemma nat-dvd-iff : $(\text{nat } z \text{ dvd } m) = (\text{if } 0 \leq z \text{ then } (z \text{ dvd int } m) \text{ else } m = 0)$
 $\langle \text{proof} \rangle$

lemma eq-nat-nat-iff :
 $0 \leq z \implies 0 \leq z' \implies \text{nat } z = \text{nat } z' \longleftrightarrow z = z'$
 $\langle \text{proof} \rangle$

lemma nat-power-eq :
 $0 \leq z \implies \text{nat } (z ^ n) = \text{nat } z ^ n$
 $\langle \text{proof} \rangle$

lemma zdvd-imp-le : $[| z \text{ dvd } n; 0 < n |] \implies z \leq (n::\text{int})$
 $\langle \text{proof} \rangle$

lemma zdvd-period :
fixes $a \text{ } d :: \text{int}$
assumes $a \text{ dvd } d$
shows $a \text{ dvd } (x + t) \longleftrightarrow a \text{ dvd } ((x + c * d) + t)$
 $\langle \text{proof} \rangle$

30.24 Configuration of the code generator

code-datatype $\text{Pls Min Bit0 Bit1 number-of} :: \text{int} \Rightarrow \text{int}$

lemmas $\text{pred-succ-numeral-code}$ $[code] =$
 $\text{pred-bin-simps succ-bin-simps}$

lemmas plus-numeral-code $[code] =$
 add-bin-simps
 $\text{arith-extra-simps}(1) [\text{where } 'a = \text{int}]$

```

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 <proof>

end

lemma eq-number-of-int-code [code]:
  eq-class.eq (number-of k :: int) (number-of l)  $\longleftrightarrow$  eq-class.eq k l
  <proof>

lemma eq-int-code [code]:
  eq-class.eq Int.Pls Int.Pls  $\longleftrightarrow$  True
  eq-class.eq Int.Pls Int.Min  $\longleftrightarrow$  False
  eq-class.eq Int.Pls (Int.Bit0 k2)  $\longleftrightarrow$  eq-class.eq Int.Pls k2
  eq-class.eq Int.Pls (Int.Bit1 k2)  $\longleftrightarrow$  False
  eq-class.eq Int.Min Int.Pls  $\longleftrightarrow$  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.Pls  $\longleftrightarrow$  eq-class.eq k1 Int.Pls
  eq-class.eq (Int.Bit1 k1) Int.Pls  $\longleftrightarrow$  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
  <proof>

lemma eq-int-refl [code nbe]:
  eq-class.eq (k::int) k  $\longleftrightarrow$  True
  <proof>

lemma less-eq-number-of-int-code [code]:
  (number-of k :: int)  $\leq$  number-of l  $\longleftrightarrow$  k  $\leq$  l
  <proof>

```

lemma *less-eq-int-code* [code]:

$Int.Pls \leq Int.Pls \longleftrightarrow True$
 $Int.Pls \leq Int.Min \longleftrightarrow False$
 $Int.Pls \leq Int.Bit0\ k \longleftrightarrow Int.Pls \leq k$
 $Int.Pls \leq Int.Bit1\ k \longleftrightarrow Int.Pls \leq k$
 $Int.Min \leq Int.Pls \longleftrightarrow True$
 $Int.Min \leq Int.Min \longleftrightarrow True$
 $Int.Min \leq Int.Bit0\ k \longleftrightarrow Int.Min < k$
 $Int.Min \leq Int.Bit1\ k \longleftrightarrow Int.Min \leq k$
 $Int.Bit0\ k \leq Int.Pls \longleftrightarrow k \leq Int.Pls$
 $Int.Bit1\ k \leq Int.Pls \longleftrightarrow k < Int.Pls$
 $Int.Bit0\ k \leq Int.Min \longleftrightarrow k \leq Int.Min$
 $Int.Bit1\ k \leq Int.Min \longleftrightarrow k \leq Int.Min$
 $Int.Bit0\ k1 \leq Int.Bit0\ k2 \longleftrightarrow k1 \leq k2$
 $Int.Bit0\ k1 \leq Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$
 $Int.Bit1\ k1 \leq Int.Bit0\ k2 \longleftrightarrow k1 < k2$
 $Int.Bit1\ k1 \leq Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$
 ⟨proof⟩

lemma *less-number-of-int-code* [code]:

$(number-of\ k :: int) < number-of\ l \longleftrightarrow k < l$
 ⟨proof⟩

lemma *less-int-code* [code]:

$Int.Pls < Int.Pls \longleftrightarrow False$
 $Int.Pls < Int.Min \longleftrightarrow False$
 $Int.Pls < Int.Bit0\ k \longleftrightarrow Int.Pls < k$
 $Int.Pls < Int.Bit1\ k \longleftrightarrow Int.Pls \leq k$
 $Int.Min < Int.Pls \longleftrightarrow True$
 $Int.Min < Int.Min \longleftrightarrow False$
 $Int.Min < Int.Bit0\ k \longleftrightarrow Int.Min < k$
 $Int.Min < Int.Bit1\ k \longleftrightarrow Int.Min < k$
 $Int.Bit0\ k < Int.Pls \longleftrightarrow k < Int.Pls$
 $Int.Bit1\ k < Int.Pls \longleftrightarrow k < Int.Pls$
 $Int.Bit0\ k < Int.Min \longleftrightarrow k \leq Int.Min$
 $Int.Bit1\ k < Int.Min \longleftrightarrow k < Int.Min$
 $Int.Bit0\ k1 < Int.Bit0\ k2 \longleftrightarrow k1 < k2$
 $Int.Bit0\ k1 < Int.Bit1\ k2 \longleftrightarrow k1 \leq k2$
 $Int.Bit1\ k1 < Int.Bit0\ k2 \longleftrightarrow k1 < k2$
 $Int.Bit1\ k1 < Int.Bit1\ k2 \longleftrightarrow k1 < k2$
 ⟨proof⟩

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

<proof>

lemma [*code*]: *nat i = nat-aux i 0*
<proof>

hide-const (**open**) *nat-aux*

lemma *zero-is-num-zero* [*code*, *code-unfold-post*]:
(0::int) = Numeral0
<proof>

lemma *one-is-num-one* [*code*, *code-unfold-post*]:
(1::int) = Numeral1
<proof>

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 = HOLogic.mk-number HOLogic.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;
>>

<ML>

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:
   $x \wedge (y + z) = ((x \wedge y) * (x \wedge z))::int$ 
  <proof>

```

```

lemma zero-less-zpower-abs-iff:
   $(0 < \text{abs } x \wedge n) \longleftrightarrow (x \neq (0::int) \mid n = 0)$ 
  <proof>

```

```

lemma zero-le-zpower-abs:  $(0::int) \leq \text{abs } x \wedge n$ 
  <proof>

```

```

lemma zpower-zpower:
   $(x \wedge y) \wedge z = (x \wedge (y * z))::int$ 
  <proof>

```

```

lemma int-power:
   $\text{int } (m \wedge n) = \text{int } m \wedge n$ 
  <proof>

```

```

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 ⟨proof⟩

end

lemma [code-post]:
  nat (number-of v) = number-of v
  ⟨proof⟩

```

31.2 Special case: squares and cubes

```

lemma numeral-2-eq-2: 2 = Suc (Suc 0)
  ⟨proof⟩

lemma numeral-3-eq-3: 3 = Suc (Suc (Suc 0))
  ⟨proof⟩

context power
begin

abbreviation (xsymbols)
  power2 :: 'a ⇒ 'a ((2) [1000] 999) where
  x2 ≡ x ^ 2

notation (latex output)
  power2 ((2) [1000] 999)

notation (HTML output)
  power2 ((2) [1000] 999)

end

context monoid-mult
begin

lemma power2-eq-square: a2 = a * a
  ⟨proof⟩

lemma power3-eq-cube: a ^ 3 = a * a * a
  ⟨proof⟩

lemma power-even-eq:
  a ^ (2*n) = (a ^ n) ^ 2
  ⟨proof⟩

lemma power-odd-eq:
  a ^ Suc (2*n) = a * (a ^ n) ^ 2
  ⟨proof⟩

```

end

context *semiring-1*
begin

lemma *zero-power2* [*simp*]: $0^2 = 0$
 $\langle proof \rangle$

lemma *one-power2* [*simp*]: $1^2 = 1$
 $\langle proof \rangle$

end

context *ring-1*
begin

lemma *power2-minus* [*simp*]:
 $(- a)^2 = a^2$
 $\langle proof \rangle$

We cannot prove general results about the numeral $-1::'b$, so we have to use $-(1::'a)$ instead.

lemma *power-minus1-even* [*simp*]:
 $(- 1) ^ (2*n) = 1$
 $\langle proof \rangle$

lemma *power-minus1-odd*:
 $(- 1) ^ Suc (2*n) = - 1$
 $\langle proof \rangle$

lemma *power-minus-even* [*simp*]:
 $(-a) ^ (2*n) = a ^ (2*n)$
 $\langle proof \rangle$

end

context *ring-1-no-zero-divisors*
begin

lemma *zero-eq-power2* [*simp*]:
 $a^2 = 0 \longleftrightarrow a = 0$
 $\langle proof \rangle$

lemma *power2-eq-1-iff*:
 $a^2 = 1 \longleftrightarrow a = 1 \vee a = - 1$
 $\langle proof \rangle$

end

context *linordered-ring*

begin

lemma *sum-squares-ge-zero*:

$$0 \leq x * x + y * y$$

<proof>

lemma *not-sum-squares-lt-zero*:

$$\neg x * x + y * y < 0$$

<proof>

end

context *linordered-ring-strict*

begin

lemma *sum-squares-eq-zero-iff*:

$$x * x + y * y = 0 \longleftrightarrow x = 0 \wedge y = 0$$

<proof>

lemma *sum-squares-le-zero-iff*:

$$x * x + y * y \leq 0 \longleftrightarrow x = 0 \wedge y = 0$$

<proof>

lemma *sum-squares-gt-zero-iff*:

$$0 < x * x + y * y \longleftrightarrow x \neq 0 \vee y \neq 0$$

<proof>

end

context *linordered-semidom*

begin

lemma *power2-le-imp-le*:

$$x^2 \leq y^2 \Longrightarrow 0 \leq y \Longrightarrow x \leq y$$

<proof>

lemma *power2-less-imp-less*:

$$x^2 < y^2 \Longrightarrow 0 \leq y \Longrightarrow x < y$$

<proof>

lemma *power2-eq-imp-eq*:

$$x^2 = y^2 \Longrightarrow 0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x = y$$

<proof>

end

context *linordered-idom*

begin

lemma *zero-le-power2* [*simp*]:
 $0 \leq a^2$
 $\langle \text{proof} \rangle$

lemma *zero-less-power2* [*simp*]:
 $0 < a^2 \iff a \neq 0$
 $\langle \text{proof} \rangle$

lemma *power2-less-0* [*simp*]:
 $\neg a^2 < 0$
 $\langle \text{proof} \rangle$

lemma *abs-power2* [*simp*]:
 $\text{abs } (a^2) = a^2$
 $\langle \text{proof} \rangle$

lemma *power2-abs* [*simp*]:
 $(\text{abs } a)^2 = a^2$
 $\langle \text{proof} \rangle$

lemma *odd-power-less-zero*:
 $a < 0 \implies a \wedge \text{Suc } (2*n) < 0$
 $\langle \text{proof} \rangle$

lemma *odd-0-le-power-imp-0-le*:
 $0 \leq a \wedge \text{Suc } (2*n) \implies 0 \leq a$
 $\langle \text{proof} \rangle$

lemma *zero-le-even-power'* [*simp*]:
 $0 \leq a \wedge (2*n)$
 $\langle \text{proof} \rangle$

lemma *sum-power2-ge-zero*:
 $0 \leq x^2 + y^2$
 $\langle \text{proof} \rangle$

lemma *not-sum-power2-lt-zero*:
 $\neg x^2 + y^2 < 0$
 $\langle \text{proof} \rangle$

lemma *sum-power2-eq-zero-iff*:
 $x^2 + y^2 = 0 \iff x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *sum-power2-le-zero-iff*:
 $x^2 + y^2 \leq 0 \iff x = 0 \wedge y = 0$
 $\langle \text{proof} \rangle$

lemma *sum-power2-gt-zero-iff*:
 $0 < x^2 + y^2 \longleftrightarrow x \neq 0 \vee y \neq 0$
 $\langle \text{proof} \rangle$

end

lemma *power2-sum*:
fixes $x\ y :: 'a::\text{number-ring}$
shows $(x + y)^2 = x^2 + y^2 + 2 * x * y$
 $\langle \text{proof} \rangle$

lemma *power2-diff*:
fixes $x\ y :: 'a::\text{number-ring}$
shows $(x - y)^2 = x^2 + y^2 - 2 * x * y$
 $\langle \text{proof} \rangle$

31.3 Predicate for negative binary numbers

definition *neg* :: $\text{int} \Rightarrow \text{bool}$ **where**
 $\text{neg } Z \longleftrightarrow Z < 0$

lemma *not-neg-int* [simp]: $\sim \text{neg } (\text{of-nat } n)$
 $\langle \text{proof} \rangle$

lemma *neg-zminus-int* [simp]: $\text{neg } (- (\text{of-nat } (\text{Suc } n)))$
 $\langle \text{proof} \rangle$

lemmas *neg-eq-less-0* = *neg-def*

lemma *not-neg-eq-ge-0*: $(\sim \text{neg } x) = (0 \leq x)$
 $\langle \text{proof} \rangle$

To simplify inequalities when Numeral1 can get simplified to 1

lemma *not-neg-0*: $\sim \text{neg } 0$
 $\langle \text{proof} \rangle$

lemma *not-neg-1*: $\sim \text{neg } 1$
 $\langle \text{proof} \rangle$

lemma *neg-nat*: $\text{neg } z ==> \text{nat } z = 0$
 $\langle \text{proof} \rangle$

lemma *not-neg-nat*: $\sim \text{neg } z ==> \text{of-nat } (\text{nat } z) = z$
 $\langle \text{proof} \rangle$

If *Numeral0* is rewritten to 0 then this rule can't be applied: *Numeral0* IS
number-of Pls

lemma *not-neg-number-of-Pls*: $\sim \text{neg } (\text{number-of Int.Pls})$

$\langle \text{proof} \rangle$

lemma *neg-number-of-Min*: $\text{neg } (\text{number-of } \text{Int.Min})$
 $\langle \text{proof} \rangle$

lemma *neg-number-of-Bit0*:
 $\text{neg } (\text{number-of } (\text{Int.Bit0 } w)) = \text{neg } (\text{number-of } w)$
 $\langle \text{proof} \rangle$

lemma *neg-number-of-Bit1*:
 $\text{neg } (\text{number-of } (\text{Int.Bit1 } w)) = \text{neg } (\text{number-of } w)$
 $\langle \text{proof} \rangle$

lemmas *neg-simps* [simp] =
 $\text{not-neg-0 not-neg-1}$
 $\text{not-neg-number-of-Pls neg-number-of-Min}$
 $\text{neg-number-of-Bit0 neg-number-of-Bit1}$

31.4 Function *nat*: Coercion from Type *int* to *nat*

declare *nat-1* [simp]

lemma *nat-number-of* [simp]: $\text{nat } (\text{number-of } w) = \text{number-of } w$
 $\langle \text{proof} \rangle$

lemma *nat-numeral-0-eq-0* [simp, code-post]: $\text{Numeral0} = (0::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *nat-numeral-1-eq-1* [simp]: $\text{Numeral1} = (1::\text{nat})$
 $\langle \text{proof} \rangle$

lemma *Numeral1-eq1-nat*:
 $(1::\text{nat}) = \text{Numeral1}$
 $\langle \text{proof} \rangle$

lemma *numeral-1-eq-Suc-0* [code-post]: $\text{Numeral1} = \text{Suc } 0$
 $\langle \text{proof} \rangle$

31.5 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [simp]:
 $\text{int } (\text{number-of } v) =$
 $(\text{if } \text{neg } (\text{number-of } v :: \text{int}) \text{ then } 0$
 $\text{else } (\text{number-of } v :: \text{int}))$
 $\langle \text{proof} \rangle$

31.5.1 Successor

lemma *Suc-nat-eq-nat-zadd1*: $(0::\text{int}) \leq z \implies \text{Suc } (\text{nat } z) = \text{nat } (1 + z)$
 $\langle \text{proof} \rangle$

lemma *Suc-nat-number-of-add*:

$Suc\ (number-of\ v + n) =$
 $(if\ neg\ (number-of\ v :: int)\ then\ 1+n\ else\ number-of\ (Int.succ\ v) + n)$
 $\langle proof \rangle$

lemma *Suc-nat-number-of [simp]*:

$Suc\ (number-of\ v) =$
 $(if\ neg\ (number-of\ v :: int)\ then\ 1\ else\ number-of\ (Int.succ\ v))$
 $\langle proof \rangle$

31.5.2 Addition

lemma *add-nat-number-of [simp]*:

$(number-of\ v :: nat) + number-of\ v' =$
 $(if\ v < Int.Pls\ then\ number-of\ v'$
 $\quad else\ if\ v' < Int.Pls\ then\ number-of\ v$
 $\quad else\ number-of\ (v + v'))$
 $\langle proof \rangle$

lemma *nat-number-of-add-1 [simp]*:

$number-of\ v + (1::nat) =$
 $(if\ v < Int.Pls\ then\ 1\ else\ number-of\ (Int.succ\ v))$
 $\langle proof \rangle$

lemma *nat-1-add-number-of [simp]*:

$(1::nat) + number-of\ v =$
 $(if\ v < Int.Pls\ then\ 1\ else\ number-of\ (Int.succ\ v))$
 $\langle proof \rangle$

lemma *nat-1-add-1 [simp]*: $1 + 1 = (2::nat)$

$\langle proof \rangle$

31.5.3 Subtraction

lemma *diff-nat-eq-if*:

$nat\ z - nat\ z' =$
 $(if\ neg\ z'\ then\ nat\ z$
 $\quad else\ let\ d = z - z'\ in$
 $\quad if\ neg\ d\ then\ 0\ else\ nat\ d)$
 $\langle proof \rangle$

lemma *diff-nat-number-of [simp]*:

$(number-of\ v :: nat) - number-of\ v' =$
 $(if\ v' < Int.Pls\ then\ number-of\ v$
 $\quad else\ let\ d = number-of\ (v + uminus\ v')\ in$
 $\quad if\ neg\ d\ then\ 0\ else\ nat\ d)$
 $\langle proof \rangle$

lemma *nat-number-of-diff-1* [simp]:
 $\text{number-of } v - (1::\text{nat}) =$
 $(\text{if } v \leq \text{Int.Pl}s \text{ then } 0 \text{ else } \text{number-of } (\text{Int.pred } v))$
 ⟨proof⟩

31.5.4 Multiplication

lemma *mult-nat-number-of* [simp]:
 $(\text{number-of } v :: \text{nat}) * \text{number-of } v' =$
 $(\text{if } v < \text{Int.Pl}s \text{ then } 0 \text{ else } \text{number-of } (v * v'))$
 ⟨proof⟩

31.6 Comparisons

31.6.1 Equals (=)

lemma *eq-nat-number-of* [simp]:
 $((\text{number-of } v :: \text{nat}) = \text{number-of } v') =$
 $(\text{if } \text{neg } (\text{number-of } v :: \text{int}) \text{ then } (\text{number-of } v' :: \text{int}) \leq 0$
 $\text{else if } \text{neg } (\text{number-of } v' :: \text{int}) \text{ then } (\text{number-of } v :: \text{int}) = 0$
 $\text{else } v = v')$
 ⟨proof⟩

31.6.2 Less-than (i)

lemma *less-nat-number-of* [simp]:
 $(\text{number-of } v :: \text{nat}) < \text{number-of } v' \longleftrightarrow$
 $(\text{if } v < v' \text{ then } \text{Int.Pl}s < v' \text{ else } \text{False})$
 ⟨proof⟩

31.6.3 Less-than-or-equal

lemma *le-nat-number-of* [simp]:
 $(\text{number-of } v :: \text{nat}) \leq \text{number-of } v' \longleftrightarrow$
 $(\text{if } v \leq v' \text{ then } \text{True} \text{ else } v \leq \text{Int.Pl}s)$
 ⟨proof⟩

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'*: $0 < n \implies n = \text{Suc}(n - 1)$
 ⟨proof⟩

lemmas *expand-Suc* = *Suc-pred'* [of number-of *v*, standard]

31.7.2 Arith

lemma *Suc-eq-plus1*: $\text{Suc } n = n + 1$
 ⟨proof⟩

lemma *Suc-eq-plus1-left*: $\text{Suc } n = 1 + n$
 ⟨proof⟩

lemma *add-eq-if*: $(m :: \text{nat}) + n = (\text{if } m=0 \text{ then } n \text{ else } \text{Suc } ((m - 1) + n))$
 ⟨proof⟩

lemma *mult-eq-if*: $(m :: \text{nat}) * n = (\text{if } m=0 \text{ then } 0 \text{ else } n + ((m - 1) * n))$
 ⟨proof⟩

lemma *power-eq-if*: $(p ^ m :: \text{nat}) = (\text{if } m=0 \text{ then } 1 \text{ else } p * (p ^ (m - 1)))$
 ⟨proof⟩

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]:
 $\text{number-of } v = (0 :: \text{nat}) \iff v \leq \text{Int.Pl}s$
 ⟨proof⟩

lemma *eq-0-number-of* [simp]:
 $(0 :: \text{nat}) = \text{number-of } v \iff v \leq \text{Int.Pl}s$
 ⟨proof⟩

lemma *less-0-number-of* [simp]:
 $(0 :: \text{nat}) < \text{number-of } v \iff \text{Int.Pl}s < v$
 ⟨proof⟩

lemma *neg-imp-number-of-eq-0*: $\text{neg } (\text{number-of } v :: \text{int}) \implies \text{number-of } v = (0 :: \text{nat})$
 <proof>

31.9 Comparisons involving *Suc*

lemma *eq-number-of-Suc* [simp]:
 $(\text{number-of } v = \text{Suc } n) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then False else nat } pv = n)$
 <proof>

lemma *Suc-eq-number-of* [simp]:
 $(\text{Suc } n = \text{number-of } v) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then False else nat } pv = n)$
 <proof>

lemma *less-number-of-Suc* [simp]:
 $(\text{number-of } v < \text{Suc } n) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then True else nat } pv < n)$
 <proof>

lemma *less-Suc-number-of* [simp]:
 $(\text{Suc } n < \text{number-of } v) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then False else } n < \text{nat } pv)$
 <proof>

lemma *le-number-of-Suc* [simp]:
 $(\text{number-of } v \leq \text{Suc } n) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then True else nat } pv \leq n)$
 <proof>

lemma *le-Suc-number-of* [simp]:
 $(\text{Suc } n \leq \text{number-of } v) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then False else } n \leq \text{nat } pv)$
 <proof>

lemma *eq-number-of-Pls-Min*: $(\text{Numeral0} :: \text{int}) \sim = \text{number-of Int.Min}$
 <proof>

31.10 Max and Min Combined with *Suc*

lemma *max-number-of-Suc* [simp]:
 $\text{max } (\text{Suc } n) (\text{number-of } v) =$

(let pv = number-of (Int.pred v) in
 if neg pv then Suc n else Suc(max n (nat pv)))
 <proof>

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))
 <proof>

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)))
 <proof>

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))
 <proof>

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))
 <proof>

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)
 <proof>

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)
 <proof>

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*: *Natural0* = (0::nat)
 ⟨proof⟩

lemma *nat-number-of-Min*: *number-of Int.Min* = (0::nat)
 ⟨proof⟩

lemma *nat-number-of-Bit0*:
number-of (Int.Bit0 w) = (let n::nat = *number-of w* in n + n)
 ⟨proof⟩

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))
 ⟨proof⟩

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-Natural1

lemma *Let-Suc* [simp]: *Let (Suc n) f* == *f (Suc n)*
 ⟨proof⟩

lemma *power-m1-even*: $(-1) ^ (2*n) = (1::'a::\{number-ring\})$
 ⟨proof⟩

lemma *power-m1-odd*: $(-1) ^ \wedge \text{Suc}(2*n) = (-1::'a::\{\text{number-ring}\})$
 ⟨proof⟩

lemma *nat-number-of-add-left*:
 $\text{number-of } v + (\text{number-of } v' + (k::\text{nat})) =$
 $(\text{if } \text{neg } (\text{number-of } v :: \text{int}) \text{ then } \text{number-of } v' + k$
 $\text{else if } \text{neg } (\text{number-of } v' :: \text{int}) \text{ then } \text{number-of } v + k$
 $\text{else } \text{number-of } (v + v') + k)$
 ⟨proof⟩

lemma *nat-number-of-mult-left*:
 $\text{number-of } v * (\text{number-of } v' * (k::\text{nat})) =$
 $(\text{if } v < \text{Int.Pl} \text{ then } 0$
 $\text{else } \text{number-of } (v * v') * k)$
 ⟨proof⟩

31.12 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
 $0 \leq x ==> \text{of-nat } (\text{nat } (2 * x)) = \text{of-nat } (\text{nat } x) + \text{of-nat } (\text{nat } x)$
 ⟨proof⟩

lemma *nat-numeral-m1-eq-0*: $-1 = (0::\text{nat})$
 ⟨proof⟩

lemma *of-nat-number-of-lemma*:
 $\text{of-nat } (\text{number-of } v :: \text{nat}) =$
 $(\text{if } 0 \leq (\text{number-of } v :: \text{int})$
 $\text{then } (\text{number-of } v :: 'a :: \text{number-ring})$
 $\text{else } 0)$
 ⟨proof⟩

lemma *of-nat-number-of-eq [simp]*:
 $\text{of-nat } (\text{number-of } v :: \text{nat}) =$
 $(\text{if } \text{neg } (\text{number-of } v :: \text{int}) \text{ then } 0$
 $\text{else } (\text{number-of } v :: 'a :: \text{number-ring}))$
 ⟨proof⟩

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*: $\text{Numeral0} < n ==> \text{Suc } m - n = m - (n - \text{Numeral1})$
 ⟨proof⟩

Now just instantiating n to $\text{number-of } v$ does the right simplification, but with some redundant inequality tests.

lemma *neg-number-of-pred-iff-0*:
 $\text{neg } (\text{number-of } (\text{Int.pred } v)::\text{int}) = (\text{number-of } v = (0::\text{nat}))$

$\langle \text{proof} \rangle$

No longer required as a simprule because of the *inverse-fold* simproc

lemma *Suc-diff-number-of*:

$\text{Int.Pls} < v ==>$

$\text{Suc } m - (\text{number-of } v) = m - (\text{number-of } (\text{Int.pred } v))$

$\langle \text{proof} \rangle$

lemma *diff-Suc-eq-diff-pred*: $m - \text{Suc } n = (m - 1) - n$

$\langle \text{proof} \rangle$

31.12.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [simp]:

$\text{nat-case } a \text{ f } (\text{number-of } v) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then } a \text{ else f } (\text{nat } pv))$

$\langle \text{proof} \rangle$

lemma *nat-case-add-eq-if* [simp]:

$\text{nat-case } a \text{ f } ((\text{number-of } v) + n) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then nat-case } a \text{ f } n \text{ else f } (\text{nat } pv + n))$

$\langle \text{proof} \rangle$

lemma *nat-rec-number-of* [simp]:

$\text{nat-rec } a \text{ f } (\text{number-of } v) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then } a \text{ else f } (\text{nat } pv) (\text{nat-rec } a \text{ f } (\text{nat } pv)))$

$\langle \text{proof} \rangle$

lemma *nat-rec-add-eq-if* [simp]:

$\text{nat-rec } a \text{ f } (\text{number-of } v + n) =$
 $(\text{let } pv = \text{number-of } (\text{Int.pred } v) \text{ in}$
 $\text{if neg } pv \text{ then nat-rec } a \text{ f } n$
 $\text{else f } (\text{nat } pv + n) (\text{nat-rec } a \text{ f } (\text{nat } pv + n)))$

$\langle \text{proof} \rangle$

31.12.3 Various Other Lemmas

lemma *card-UNIV-bool*[simp]: $\text{card } (\text{UNIV} :: \text{bool set}) = 2$

$\langle \text{proof} \rangle$

Evens and Odds, for Mutilated Chess Board

Lemmas for specialist use, NOT as default simprules

lemma *nat-mult-2*: $2 * z = (z + z :: \text{nat})$

$\langle \text{proof} \rangle$

lemma *nat-mult-2-right*: $z * 2 = (z + z :: \text{nat})$
 ⟨*proof*⟩

Case analysis on $n < (2 :: 'a)$

lemma *less-2-cases*: $(n :: \text{nat}) < 2 \implies n = 0 \mid n = \text{Suc } 0$
 ⟨*proof*⟩

Removal of Small Numerals: 0, 1 and (in additive positions) 2

lemma *add-2-eq-Suc* [*simp*]: $2 + n = \text{Suc } (\text{Suc } n)$
 ⟨*proof*⟩

lemma *add-2-eq-Suc'* [*simp*]: $n + 2 = \text{Suc } (\text{Suc } n)$
 ⟨*proof*⟩

Can be used to eliminate long strings of Sucs, but not by default

lemma *Suc3-eq-add-3*: $\text{Suc } (\text{Suc } (\text{Suc } n)) = 3 + n$
 ⟨*proof*⟩

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 $\bigwedge P y. (\bigwedge x. P x) \implies A y \implies P (f y)$
shows *transfer-morphism* $f A$
 ⟨*proof*⟩

⟨*ML*⟩

32.2 Set up transfer from nat to int

set up transfer direction

lemma *transfer-morphism-nat-int*: *transfer-morphism* *nat* (*op* \leq ($0 :: \text{int}$))
 ⟨*proof*⟩

```

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
  ⟨proof⟩

```

definition

```

  tsub :: int ⇒ int ⇒ int

```

where

```

  tsub x y = (if x >= y then x - y else 0)

```

lemma tsub-eq: $x \geq y \implies \text{tsub } x \ y = x - y$

```

  ⟨proof⟩

```

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)
  ⟨proof⟩

```

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
  ⟨proof⟩

```

lemma transfer-nat-int-relations [transfer key: transfer-morphism-nat-int]:

```

  x >= 0 ⟹ y >= 0 ⟹
    (nat (x::int) = nat y) = (x = y)
  x >= 0 ⟹ y >= 0 ⟹
    (nat (x::int) < nat y) = (x < y)
  x >= 0 ⟹ y >= 0 ⟹
    (nat (x::int) <= nat y) = (x <= y)
  x >= 0 ⟹ y >= 0 ⟹

```

$(\text{nat } (x::\text{int}) \text{ dvd nat } y) = (x \text{ dvd } y)$
 $\langle \text{proof} \rangle$

first-order quantifiers

lemma *all-nat*: $(\forall x. P x) \longleftrightarrow (\forall x \geq 0. P (\text{nat } x))$
 $\langle \text{proof} \rangle$

lemma *ex-nat*: $(\exists x. P x) \longleftrightarrow (\exists x. 0 \leq x \wedge P (\text{nat } x))$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-quantifiers* [*transfer key: transfer-morphism-nat-int*]:
 $(\text{ALL } (x::\text{nat}). P x) = (\text{ALL } (x::\text{int}). x \geq 0 \longrightarrow P (\text{nat } x))$
 $(\text{EX } (x::\text{nat}). P x) = (\text{EX } (x::\text{int}). x \geq 0 \ \& \ P (\text{nat } x))$
 $\langle \text{proof} \rangle$

lemma *all-cong*: $(\bigwedge x. Q x \implies P x = P' x) \implies$
 $(\text{ALL } x. Q x \longrightarrow P x) = (\text{ALL } x. Q x \longrightarrow P' x)$
 $\langle \text{proof} \rangle$

lemma *ex-cong*: $(\bigwedge x. Q x \implies P x = P' x) \implies$
 $(\text{EX } x. Q x \wedge P x) = (\text{EX } x. Q x \wedge P' x)$
 $\langle \text{proof} \rangle$

declare *transfer-morphism-nat-int* [*transfer add*
cong: all-cong ex-cong]

if

lemma *nat-if-cong* [*transfer key: transfer-morphism-nat-int*]:
 $(\text{if } P \text{ then } (\text{nat } x) \text{ else } (\text{nat } y)) = \text{nat } (\text{if } P \text{ then } x \text{ else } y)$
 $\langle \text{proof} \rangle$

operations with sets

definition

nat-set :: *int set* \Rightarrow *bool*

where

nat-set *S* = $(\text{ALL } x:S. x \geq 0)$

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* ‘ $\{x. x \geq 0 \ \& \ P(\text{nat } x)\}$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-set-function-closures*:

nat-set $\{\}$
nat-set *A* \implies *nat-set* *B* \implies *nat-set* (*A Un B*)

$\text{nat-set } A \implies \text{nat-set } B \implies \text{nat-set } (A \text{ Int } B)$
 $\text{nat-set } \{x. x \geq 0 \ \& \ P \ x\}$
 $\text{nat-set } (\text{int} \text{ ' } C)$
 $\text{nat-set } A \implies x : A \implies x \geq 0$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-set-relations:*

$(\text{finite } A) = (\text{finite } (\text{int} \text{ ' } A))$
 $(x : A) = (\text{int } x : \text{int} \text{ ' } A)$
 $(A = B) = (\text{int} \text{ ' } A = \text{int} \text{ ' } B)$
 $(A < B) = (\text{int} \text{ ' } A < \text{int} \text{ ' } B)$
 $(A \leq B) = (\text{int} \text{ ' } A \leq \text{int} \text{ ' } B)$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-set-return-embed:* $\text{nat-set } A \implies$

$(\text{int} \text{ ' } \text{nat} \text{ ' } A = A)$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-set-cong:* $(!!x. x \geq 0 \implies P \ x = P' \ x) \implies$

$\{(x::\text{int}). x \geq 0 \ \& \ P \ x\} = \{x. x \geq 0 \ \& \ P' \ x\}$
 $\langle \text{proof} \rangle$

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
 \rangle

setsum and setprod

lemma *transfer-nat-int-sum-prod:*

$\text{setsum } f \ A = \text{setsum } (\%x. f \ (\text{nat } x)) \ (\text{int} \text{ ' } A)$
 $\text{setprod } f \ A = \text{setprod } (\%x. f \ (\text{nat } x)) \ (\text{int} \text{ ' } A)$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-sum-prod2:*

$\text{setsum } f \ A = \text{nat}(\text{setsum } (\%x. \text{int } (f \ x)) \ A)$
 $\text{setprod } f \ A = \text{nat}(\text{setprod } (\%x. \text{int } (f \ x)) \ A)$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-sum-prod-closure:*

$\text{nat-set } A \implies (!!x. x \geq 0 \implies f \ x \geq (0::\text{int})) \implies \text{setsum } f \ A \geq 0$
 $\text{nat-set } A \implies (!!x. x \geq 0 \implies f \ x \geq (0::\text{int})) \implies \text{setprod } f \ A \geq 0$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-sum-prod-cong*:

$$\begin{aligned} A = B &\implies \text{nat-set } B \implies (!x. x \geq 0 \implies f x = g x) \implies \\ &\quad \text{setsum } f A = \text{setsum } g B \\ A = B &\implies \text{nat-set } B \implies (!x. x \geq 0 \implies f x = g x) \implies \\ &\quad \text{setprod } f A = \text{setprod } g B \\ &\langle \text{proof} \rangle \end{aligned}$$

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* ($\lambda n. \text{True}$)
 $\langle \text{proof} \rangle$

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* \geq 0)

lemma *transfer-int-nat-numerals*:

$$\begin{aligned} 0 &= \text{int } 0 \\ 1 &= \text{int } 1 \\ 2 &= \text{int } 2 \\ 3 &= \text{int } 3 \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *transfer-int-nat-functions*:

$$\begin{aligned} (\text{int } x) + (\text{int } y) &= \text{int } (x + y) \\ (\text{int } x) * (\text{int } y) &= \text{int } (x * y) \\ \text{tsub } (\text{int } x) (\text{int } y) &= \text{int } (x - y) \\ (\text{int } x)^n &= \text{int } (x^n) \\ &\langle \text{proof} \rangle \end{aligned}$$

lemma *transfer-int-nat-function-closures*:

$$\begin{aligned} \text{is-nat } x &\implies \text{is-nat } y \implies \text{is-nat } (x + y) \\ \text{is-nat } x &\implies \text{is-nat } y \implies \text{is-nat } (x * y) \end{aligned}$$

$is\text{-}nat\ x \Longrightarrow is\text{-}nat\ y \Longrightarrow is\text{-}nat\ (tsub\ x\ y)$
 $is\text{-}nat\ x \Longrightarrow is\text{-}nat\ (x^{\wedge}n)$
 $is\text{-}nat\ 0$
 $is\text{-}nat\ 1$
 $is\text{-}nat\ 2$
 $is\text{-}nat\ 3$
 $is\text{-}nat\ (int\ z)$
 $\langle proof \rangle$

lemma *transfer-int-nat-relations:*

$(int\ x = int\ y) = (x = y)$
 $(int\ x < int\ y) = (x < y)$
 $(int\ x \leq int\ y) = (x \leq y)$
 $(int\ x\ dvd\ int\ y) = (x\ dvd\ y)$
 $\langle proof \rangle$

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:*

$(ALL\ (x::int)\ \geq 0.\ P\ x) = (ALL\ (x::nat).\ P\ (int\ x))$
 $(EX\ (x::int)\ \geq 0.\ P\ x) = (EX\ (x::nat).\ P\ (int\ x))$
 $\langle proof \rangle$

declare *transfer-morphism-int-nat* [*transfer add*

return: transfer-int-nat-quantifiers]

if

lemma *int-if-cong:* $(if\ P\ then\ (int\ x)\ else\ (int\ y)) =$

$int\ (if\ P\ then\ x\ else\ y)$
 $\langle proof \rangle$

declare *transfer-morphism-int-nat* [*transfer add return: int-if-cong*]

operations with sets

lemma *transfer-int-nat-set-functions:*

$nat\text{-}set\ A \Longrightarrow card\ A = card\ (nat\ 'A)$
 $\{\} = int\ '(\{\}::nat\ set)$
 $nat\text{-}set\ A \Longrightarrow nat\text{-}set\ B \Longrightarrow A\ Un\ B = int\ ' (nat\ 'A\ Un\ nat\ 'B)$
 $nat\text{-}set\ A \Longrightarrow nat\text{-}set\ B \Longrightarrow A\ Int\ B = int\ ' (nat\ 'A\ Int\ nat\ 'B)$
 $\{x.\ x \geq 0 \ \&\ P\ x\} = int\ ' \{x.\ P(int\ x)\}$

$\langle proof \rangle$

lemma *transfer-int-nat-set-function-closures:*

```

nat-set {}
nat-set A  $\implies$  nat-set B  $\implies$  nat-set (A Un B)
nat-set A  $\implies$  nat-set B  $\implies$  nat-set (A Int B)
nat-set {x. x >= 0 & P x}
nat-set (int ‘ C)
nat-set A  $\implies$  x : A  $\implies$  is-nat x
⟨proof⟩

```

lemma *transfer-int-nat-set-relations:*

```

nat-set A  $\implies$  finite A = finite (nat ‘ A)
is-nat x  $\implies$  nat-set A  $\implies$  (x : A) = (nat x : nat ‘ A)
nat-set A  $\implies$  nat-set B  $\implies$  (A = B) = (nat ‘ A = nat ‘ B)
nat-set A  $\implies$  nat-set B  $\implies$  (A < B) = (nat ‘ A < nat ‘ B)
nat-set A  $\implies$  nat-set B  $\implies$  (A <= B) = (nat ‘ A <= nat ‘ B)
⟨proof⟩

```

lemma *transfer-int-nat-set-return-embed:* nat ‘ int ‘ A = A

⟨proof⟩

lemma *transfer-int-nat-set-cong:* (!x. P x = P' x) \implies

```

{(x::nat). P x} = {x. P' x}
⟨proof⟩

```

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:*

```

nat-set A  $\implies$  setsum f A = setsum (%x. f (int x)) (nat ‘ A)
nat-set A  $\implies$  setprod f A = setprod (%x. f (int x)) (nat ‘ A)
⟨proof⟩

```

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)
⟨proof⟩

```

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 ⇒ 'a ⇒ 'a (infixl div 70)
  and mod :: 'a ⇒ 'a ⇒ '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:  $a \text{ div } b * b + a \text{ mod } b = a$ 
  and div-by-0 [simp]:  $a \text{ div } 0 = 0$ 
  and div-0 [simp]:  $0 \text{ div } a = 0$ 
  and div-mult-self1 [simp]:  $b \neq 0 \implies (a + c * b) \text{ div } b = c + a \text{ div } b$ 
  and div-mult-mult1 [simp]:  $c \neq 0 \implies (c * a) \text{ div } (c * b) = a \text{ div } b$ 
begin
```

op div and *op mod*

```
lemma mod-div-equality2:  $b * (a \text{ div } b) + a \text{ mod } b = a$ 
  <proof>
```

```
lemma mod-div-equality':  $a \text{ mod } b + a \text{ div } b * b = a$ 
  <proof>
```

```
lemma div-mod-equality:  $((a \text{ div } b) * b + a \text{ mod } b) + c = a + c$ 
  <proof>
```

```
lemma div-mod-equality2:  $(b * (a \text{ div } b) + a \text{ mod } b) + c = a + c$ 
  <proof>
```

```
lemma mod-by-0 [simp]:  $a \text{ mod } 0 = a$ 
  <proof>
```

```
lemma mod-0 [simp]:  $0 \text{ mod } a = 0$ 
  <proof>
```

```
lemma div-mult-self2 [simp]:
  assumes  $b \neq 0$ 
  shows  $(a + b * c) \text{ div } b = c + a \text{ div } b$ 
  <proof>
```

lemma *mod-mult-self1* [*simp*]: $(a + c * b) \bmod b = a \bmod b$
 $\langle proof \rangle$

lemma *mod-mult-self2* [*simp*]: $(a + b * c) \bmod b = a \bmod b$
 $\langle proof \rangle$

lemma *div-mult-self1-is-id* [*simp*]: $b \neq 0 \implies b * a \operatorname{div} b = a$
 $\langle proof \rangle$

lemma *div-mult-self2-is-id* [*simp*]: $b \neq 0 \implies a * b \operatorname{div} b = a$
 $\langle proof \rangle$

lemma *mod-mult-self1-is-0* [*simp*]: $b * a \bmod b = 0$
 $\langle proof \rangle$

lemma *mod-mult-self2-is-0* [*simp*]: $a * b \bmod b = 0$
 $\langle proof \rangle$

lemma *div-by-1* [*simp*]: $a \operatorname{div} 1 = a$
 $\langle proof \rangle$

lemma *mod-by-1* [*simp*]: $a \bmod 1 = 0$
 $\langle proof \rangle$

lemma *mod-self* [*simp*]: $a \bmod a = 0$
 $\langle proof \rangle$

lemma *div-self* [*simp*]: $a \neq 0 \implies a \operatorname{div} a = 1$
 $\langle proof \rangle$

lemma *div-add-self1* [*simp*]:
assumes $b \neq 0$
shows $(b + a) \operatorname{div} b = a \operatorname{div} b + 1$
 $\langle proof \rangle$

lemma *div-add-self2* [*simp*]:
assumes $b \neq 0$
shows $(a + b) \operatorname{div} b = a \operatorname{div} b + 1$
 $\langle proof \rangle$

lemma *mod-add-self1* [*simp*]:
 $(b + a) \bmod b = a \bmod b$
 $\langle proof \rangle$

lemma *mod-add-self2* [*simp*]:
 $(a + b) \bmod b = a \bmod b$
 $\langle proof \rangle$

lemma *mod-div-decomp*:

fixes $a\ b$

obtains $q\ r$ **where** $q = a\ \text{div}\ b$ **and** $r = a\ \text{mod}\ b$

and $a = q * b + r$

$\langle \text{proof} \rangle$

lemma *dvd-eq-mod-eq-0* [*code, code-unfold, code-inline del*]: $a\ \text{dvd}\ b \iff b\ \text{mod}\ a = 0$

$\langle \text{proof} \rangle$

lemma *mod-div-trivial* [*simp*]: $a\ \text{mod}\ b\ \text{div}\ b = 0$

$\langle \text{proof} \rangle$

lemma *mod-mod-trivial* [*simp*]: $a\ \text{mod}\ b\ \text{mod}\ b = a\ \text{mod}\ b$

$\langle \text{proof} \rangle$

lemma *dvd-imp-mod-0*: $a\ \text{dvd}\ b \implies b\ \text{mod}\ a = 0$

$\langle \text{proof} \rangle$

lemma *dvd-div-mult-self*: $a\ \text{dvd}\ b \implies (b\ \text{div}\ a) * a = b$

$\langle \text{proof} \rangle$

lemma *dvd-mult-div-cancel*: $a\ \text{dvd}\ b \implies a * (b\ \text{div}\ a) = b$

$\langle \text{proof} \rangle$

lemma *dvd-div-mult*: $a\ \text{dvd}\ b \implies (b\ \text{div}\ a) * c = b * c\ \text{div}\ a$

$\langle \text{proof} \rangle$

lemma *div-dvd-div* [*simp*]:

$a\ \text{dvd}\ b \implies a\ \text{dvd}\ c \implies (b\ \text{div}\ a\ \text{dvd}\ c\ \text{div}\ a) = (b\ \text{dvd}\ c)$

$\langle \text{proof} \rangle$

lemma *dvd-mod-imp-dvd*: $[| k\ \text{dvd}\ m\ \text{mod}\ n; k\ \text{dvd}\ n |] \implies k\ \text{dvd}\ m$

$\langle \text{proof} \rangle$

Addition respects modular equivalence.

lemma *mod-add-left-eq*: $(a + b)\ \text{mod}\ c = (a\ \text{mod}\ c + b)\ \text{mod}\ c$

$\langle \text{proof} \rangle$

lemma *mod-add-right-eq*: $(a + b)\ \text{mod}\ c = (a + b\ \text{mod}\ c)\ \text{mod}\ c$

$\langle \text{proof} \rangle$

lemma *mod-add-eq*: $(a + b)\ \text{mod}\ c = (a\ \text{mod}\ c + b\ \text{mod}\ c)\ \text{mod}\ c$

$\langle \text{proof} \rangle$

lemma *mod-add-cong*:

assumes $a\ \text{mod}\ c = a'\ \text{mod}\ c$

assumes $b\ \text{mod}\ c = b'\ \text{mod}\ c$

shows $(a + b)\ \text{mod}\ c = (a' + b')\ \text{mod}\ c$

$\langle proof \rangle$

lemma *div-add* [*simp*]: $z \text{ dvd } x \implies z \text{ dvd } y$
 $\implies (x + y) \text{ div } z = x \text{ div } z + y \text{ div } z$
 $\langle proof \rangle$

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq*: $(a * b) \text{ mod } c = ((a \text{ mod } c) * b) \text{ mod } c$
 $\langle proof \rangle$

lemma *mod-mult-right-eq*: $(a * b) \text{ mod } c = (a * (b \text{ mod } c)) \text{ mod } c$
 $\langle proof \rangle$

lemma *mod-mult-eq*: $(a * b) \text{ mod } c = ((a \text{ mod } c) * (b \text{ mod } c)) \text{ mod } c$
 $\langle proof \rangle$

lemma *mod-mult-cong*:
assumes $a \text{ mod } c = a' \text{ mod } c$
assumes $b \text{ mod } c = b' \text{ mod } c$
shows $(a * b) \text{ mod } c = (a' * b') \text{ mod } c$
 $\langle proof \rangle$

lemma *mod-mod-cancel*:
assumes $c \text{ dvd } b$
shows $a \text{ mod } b \text{ mod } c = a \text{ mod } c$
 $\langle proof \rangle$

lemma *div-mult-div-if-dvd*:
 $y \text{ dvd } x \implies z \text{ dvd } w \implies (x \text{ div } y) * (w \text{ div } z) = (x * w) \text{ div } (y * z)$
 $\langle proof \rangle$

lemma *div-mult-swap*:
assumes $c \text{ dvd } b$
shows $a * (b \text{ div } c) = (a * b) \text{ div } c$
 $\langle proof \rangle$

lemma *div-mult-mult2* [*simp*]:
 $c \neq 0 \implies (a * c) \text{ div } (b * c) = a \text{ div } b$
 $\langle proof \rangle$

lemma *div-mult-mult1-if* [*simp*]:
 $(c * a) \text{ div } (c * b) = (\text{if } c = 0 \text{ then } 0 \text{ else } a \text{ div } b)$
 $\langle proof \rangle$

lemma *mod-mult-mult1*:
 $(c * a) \text{ mod } (c * b) = c * (a \text{ mod } b)$
 $\langle proof \rangle$

lemma *mod-mult-mult2*:

$$(a * c) \bmod (b * c) = (a \bmod b) * c$$

<proof>

lemma *dvd-mod*: $k \text{ dvd } m \implies k \text{ dvd } n \implies k \text{ dvd } (m \bmod n)$
<proof>

lemma *dvd-mod-iff*: $k \text{ dvd } n \implies k \text{ dvd } (m \bmod n) \longleftrightarrow k \text{ dvd } m$
<proof>

lemma *div-power*:
 $y \text{ dvd } x \implies (x \text{ div } y) ^ n = x ^ n \text{ div } y ^ n$
<proof>

lemma *dvd-div-eq-mult*:
assumes $a \neq 0$ **and** $a \text{ dvd } b$
shows $b \text{ div } a = c \longleftrightarrow b = c * a$
<proof>

lemma *dvd-div-div-eq-mult*:
assumes $a \neq 0$ $c \neq 0$ **and** $a \text{ dvd } b$ $c \text{ dvd } d$
shows $b \text{ div } a = d \text{ div } c \longleftrightarrow b * c = a * d$
<proof>

end

class *ring-div* = *semiring-div* + *comm-ring-1*
begin

subclass *ring-1-no-zero-divisors* *<proof>*

Negation respects modular equivalence.

lemma *mod-minus-eq*: $(- a) \bmod b = (- (a \bmod b)) \bmod b$
<proof>

lemma *mod-minus-cong*:
assumes $a \bmod b = a' \bmod b$
shows $(- a) \bmod b = (- a') \bmod b$
<proof>

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq*: $(a - b) \bmod c = (a \bmod c - b) \bmod c$
<proof>

lemma *mod-diff-right-eq*: $(a - b) \bmod c = (a - b \bmod c) \bmod c$
<proof>

lemma *mod-diff-eq*: $(a - b) \bmod c = (a \bmod c - b \bmod c) \bmod c$
<proof>

lemma *mod-diff-cong*:

assumes $a \bmod c = a' \bmod c$

assumes $b \bmod c = b' \bmod c$

shows $(a - b) \bmod c = (a' - b') \bmod c$

$\langle \text{proof} \rangle$

lemma *dvd-neg-div*: $y \text{ dvd } x \implies -x \text{ div } y = - (x \text{ div } y)$

$\langle \text{proof} \rangle$

lemma *dvd-div-neg*: $y \text{ dvd } x \implies x \text{ div } -y = - (x \text{ div } y)$

$\langle \text{proof} \rangle$

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* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \times \text{nat} \Rightarrow \text{bool}$ **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)*

$\langle \text{proof} \rangle$

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 $qr = qr'$

$\langle \text{proof} \rangle$

We instantiate divisibility on the natural numbers by means of *divmod-nat-rel*:

instantiation *nat* :: *semiring-div*

begin

definition *divmod-nat* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \times \text{nat}$ **where**

$[\text{code del}]: \text{divmod-nat } m \ n = (\text{THE } qr. \text{divmod-nat-rel } m \ n \ qr)$

lemma *divmod-nat-rel-divmod-nat*:

$\text{divmod-nat-rel } m \ n \ (\text{divmod-nat } m \ n)$

$\langle \text{proof} \rangle$

lemma *divmod-nat-eq*:
assumes *divmod-nat-rel* *m n qr*
shows *divmod-nat* *m n = qr*
 $\langle \text{proof} \rangle$

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*:
 $\text{divmod-nat } m \ n = (m \text{ div } n, m \text{ mod } n)$
 $\langle \text{proof} \rangle$

lemma *div-eq*:
assumes *divmod-nat-rel* *m n (q, r)*
shows $m \text{ div } n = q$
 $\langle \text{proof} \rangle$

lemma *mod-eq*:
assumes *divmod-nat-rel* *m n (q, r)*
shows $m \text{ mod } n = r$
 $\langle \text{proof} \rangle$

lemma *divmod-nat-rel*: *divmod-nat-rel* *m n (m div n, m mod n)*
 $\langle \text{proof} \rangle$

lemma *divmod-nat-zero*:
 $\text{divmod-nat } m \ 0 = (0, m)$
 $\langle \text{proof} \rangle$

lemma *divmod-nat-base*:
assumes $m < n$
shows $\text{divmod-nat } m \ n = (0, m)$
 $\langle \text{proof} \rangle$

lemma *divmod-nat-step*:
assumes $0 < n$ **and** $n \leq m$
shows $\text{divmod-nat } m \ n = (\text{Suc } ((m - n) \text{ div } n), (m - n) \text{ mod } n)$
 $\langle \text{proof} \rangle$

The “recursion” equations for *op div* and *op mod*

lemma *div-less [simp]*:
fixes $m \ n :: \text{nat}$
assumes $m < n$
shows $m \text{ div } n = 0$
 $\langle \text{proof} \rangle$

lemma *le-div-geq*:
fixes $m\ n :: \text{nat}$
assumes $0 < n$ **and** $n \leq m$
shows $m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$
 $\langle \text{proof} \rangle$

lemma *mod-less [simp]*:
fixes $m\ n :: \text{nat}$
assumes $m < n$
shows $m \text{ mod } n = m$
 $\langle \text{proof} \rangle$

lemma *le-mod-geq*:
fixes $m\ n :: \text{nat}$
assumes $n \leq m$
shows $m \text{ mod } n = (m - n) \text{ mod } n$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

lemma *divmod-nat-if [code]*: $\text{divmod-nat } m\ n = (\text{if } n = 0 \vee m < n \text{ then } (0, m) \text{ else } \text{let } (q, r) = \text{divmod-nat } (m - n)\ n \text{ in } (\text{Suc } q, r))$
 $\langle \text{proof} \rangle$

Simproc for cancelling *op div* and *op mod*
 $\langle \text{ML} \rangle$

33.3.1 Quotient

lemma *div-geq*: $0 < n \implies \neg m < n \implies m \text{ div } n = \text{Suc } ((m - n) \text{ div } n)$
 $\langle \text{proof} \rangle$

lemma *div-if*: $0 < n \implies m \text{ div } n = (\text{if } m < n \text{ then } 0 \text{ else } \text{Suc } ((m - n) \text{ div } n))$
 $\langle \text{proof} \rangle$

lemma *div-mult-self-is-m [simp]*: $0 < n \implies (m * n) \text{ div } n = (m :: \text{nat})$
 $\langle \text{proof} \rangle$

lemma *div-mult-self1-is-m [simp]*: $0 < n \implies (n * m) \text{ div } n = (m :: \text{nat})$
 $\langle \text{proof} \rangle$

33.3.2 Remainder

lemma *mod-less-divisor [simp]*:
fixes $m\ n :: \text{nat}$
assumes $n > 0$
shows $m \text{ mod } n < (n :: \text{nat})$

$\langle \text{proof} \rangle$

lemma *mod-less-eq-dividend* [*simp*]:

fixes $m\ n :: \text{nat}$

shows $m \bmod n \leq m$

$\langle \text{proof} \rangle$

lemma *mod-geq*: $\neg m < (n :: \text{nat}) \implies m \bmod n = (m - n) \bmod n$

$\langle \text{proof} \rangle$

lemma *mod-if*: $m \bmod (n :: \text{nat}) = (\text{if } m < n \text{ then } m \text{ else } (m - n) \bmod n)$

$\langle \text{proof} \rangle$

lemma *mod-1* [*simp*]: $m \bmod \text{Suc } 0 = 0$

$\langle \text{proof} \rangle$

lemma *mod-mult-distrib*: $(m \bmod n) * (k :: \text{nat}) = (m * k) \bmod (n * k)$

$\langle \text{proof} \rangle$

lemma *mod-mult-distrib2*: $(k :: \text{nat}) * (m \bmod n) = (k * m) \bmod (k * n)$

$\langle \text{proof} \rangle$

lemma *mult-div-cancel*: $(n :: \text{nat}) * (m \text{ div } n) = m - (m \bmod n)$

$\langle \text{proof} \rangle$

lemma *mod-le-divisor* [*simp*]: $0 < n \implies m \bmod n \leq (n :: \text{nat})$

$\langle \text{proof} \rangle$

33.3.3 Quotient and Remainder

lemma *divmod-nat-rel-mult1-eq*:

$\text{divmod-nat-rel } b\ c\ (q, r) \implies c > 0$

$\implies \text{divmod-nat-rel } (a * b)\ c\ (a * q + a * r \text{ div } c, a * r \bmod c)$

$\langle \text{proof} \rangle$

lemma *div-mult1-eq*:

$(a * b) \text{ div } c = a * (b \text{ div } c) + a * (b \bmod c) \text{ div } (c :: \text{nat})$

$\langle \text{proof} \rangle$

lemma *divmod-nat-rel-add1-eq*:

$\text{divmod-nat-rel } a\ c\ (aq, ar) \implies \text{divmod-nat-rel } b\ c\ (bq, br) \implies c > 0$

$\implies \text{divmod-nat-rel } (a + b)\ c\ (aq + bq + (ar + br) \text{ div } c, (ar + br) \bmod c)$

$\langle \text{proof} \rangle$

lemma *div-add1-eq*:

$(a + b) \text{ div } (c :: \text{nat}) = a \text{ div } c + b \text{ div } c + ((a \bmod c + b \bmod c) \text{ div } c)$

$\langle \text{proof} \rangle$

lemma *mod-lemma*: $[(0::nat) < c; r < b] ==> b * (q \text{ mod } c) + r < b * c$
 $\langle proof \rangle$

lemma *divmod-nat-rel-mult2-eq*:
 $divmod\text{-}nat\text{-}rel\ a\ b\ (q, r) ==> 0 < b ==> 0 < c$
 $==> divmod\text{-}nat\text{-}rel\ a\ (b * c)\ (q \text{ div } c, b * (q \text{ mod } c) + r)$
 $\langle proof \rangle$

lemma *div-mult2-eq*: $a \text{ div } (b * c) = (a \text{ div } b) \text{ div } (c::nat)$
 $\langle proof \rangle$

lemma *mod-mult2-eq*: $a \text{ mod } (b * c) = b * (a \text{ div } b \text{ mod } c) + a \text{ mod } (b::nat)$
 $\langle proof \rangle$

33.3.4 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: $m \text{ div } Suc\ 0 = m$
 $\langle proof \rangle$

lemma *div-le-mono* [*rule-format* (*no-asm*)]:
 $\forall m::nat. m \leq n --> (m \text{ div } k) \leq (n \text{ div } k)$
 $\langle proof \rangle$

lemma *div-le-mono2*: $!!m::nat. [(0 < m; m \leq n)] ==> (k \text{ div } n) \leq (k \text{ div } m)$
 $\langle proof \rangle$

lemma *div-le-dividend* [*simp*]: $m \text{ div } n \leq (m::nat)$
 $\langle proof \rangle$

lemma *div-less-dividend* [*rule-format*]:
 $!!n::nat. 1 < n ==> 0 < m --> m \text{ div } n < m$
 $\langle proof \rangle$

declare *div-less-dividend* [*simp*]

A fact for the mutilated chess board

lemma *mod-Suc*: $Suc(m) \text{ mod } n = (if\ Suc(m \text{ mod } n) = n\ then\ 0\ else\ Suc(m \text{ mod } n))$
 $\langle proof \rangle$

lemma *mod-eq-0-iff*: $(m \text{ mod } d = 0) = (\exists q::nat. m = d * q)$
 $\langle proof \rangle$

lemmas *mod-eq-0D* [*dest!*] = *mod-eq-0-iff* [*THEN iffD1*]

lemma *mod-eqD*: $(m \bmod d = r) ==> \exists q::nat. m = r + q*d$
 $\langle proof \rangle$

lemma *split-div*:
 $P(n \text{ div } k :: nat) =$
 $((k = 0 \longrightarrow P\ 0) \wedge (k \neq 0 \longrightarrow (!i. !j < k. n = k*i + j \longrightarrow P\ i)))$
 $(\text{is } ?P = ?Q \text{ is } - = (- \wedge (- \longrightarrow ?R)))$
 $\langle proof \rangle$

lemma *split-div-lemma*:
assumes $0 < n$
shows $n * q \leq m \wedge m < n * \text{Suc } q \longleftrightarrow q = ((m::nat) \text{ div } n) (\text{is } ?lhs \longleftrightarrow ?rhs)$
 $\langle proof \rangle$

theorem *split-div'*:
 $P((m::nat) \text{ div } n) = ((n = 0 \wedge P\ 0) \vee$
 $(\exists q. (n * q \leq m \wedge m < n * (\text{Suc } q)) \wedge P\ q))$
 $\langle proof \rangle$

lemma *split-mod*:
 $P(n \bmod k :: nat) =$
 $((k = 0 \longrightarrow P\ n) \wedge (k \neq 0 \longrightarrow (!i. !j < k. n = k*i + j \longrightarrow P\ j)))$
 $(\text{is } ?P = ?Q \text{ is } - = (- \wedge (- \longrightarrow ?R)))$
 $\langle proof \rangle$

theorem *mod-div-equality'*: $(m::nat) \bmod n = m - (m \text{ div } n) * n$
 $\langle proof \rangle$

lemma *div-mod-equality'*:
fixes $m\ n :: nat$
shows $m \text{ div } n * n = m - m \bmod n$
 $\langle proof \rangle$

33.3.5 An “induction” law for modulus arithmetic.

lemma *mod-induct-0*:
assumes *step*: $\forall i < p. P\ i \longrightarrow P\ ((\text{Suc } i) \bmod p)$
and base: $P\ i$ **and** $i: i < p$
shows $P\ 0$
 $\langle proof \rangle$

lemma *mod-induct*:
assumes *step*: $\forall i < p. P\ i \longrightarrow P\ ((\text{Suc } i) \bmod p)$
and base: $P\ i$ **and** $i: i < p$ **and** $j: j < p$
shows $P\ j$
 $\langle proof \rangle$

lemma *div2-Suc-Suc* [simp]: $\text{Suc} (\text{Suc } m) \text{ div } 2 = \text{Suc} (m \text{ div } 2)$
 <proof>

lemma *add-self-div-2* [simp]: $(m + m) \text{ div } 2 = (m :: \text{nat})$
 <proof>

lemma *mod2-Suc-Suc* [simp]: $\text{Suc}(\text{Suc}(m)) \text{ mod } 2 = m \text{ mod } 2$
 <proof>

lemma *mod2-gr-0* [simp]: $0 < (m :: \text{nat}) \text{ mod } 2 \longleftrightarrow m \text{ mod } 2 = 1$
 <proof>

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]: $m \text{ div} (\text{Suc} (\text{Suc} (\text{Suc } n))) = m \text{ div } (3+n)$
 <proof>

lemma *mod-Suc-eq-mod-add3* [simp]: $m \text{ mod} (\text{Suc} (\text{Suc} (\text{Suc } n))) = m \text{ mod} (3+n)$
 <proof>

lemma *Suc-div-eq-add3-div*: $(\text{Suc} (\text{Suc} (\text{Suc } m))) \text{ div } n = (3+m) \text{ div } n$
 <proof>

lemma *Suc-mod-eq-add3-mod*: $(\text{Suc} (\text{Suc} (\text{Suc } m))) \text{ mod } n = (3+m) \text{ mod } n$
 <proof>

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*: $1 < k \implies \text{Suc} (k * m) \text{ mod } k = 1$
 <proof>

declare *Suc-times-mod-eq* [of number-of w, standard, simp]

lemma [simp]: $n \text{ div } k \leq (\text{Suc } n) \text{ div } k$
 <proof>

lemma *Suc-n-div-2-gt-zero* [simp]: $(0 :: \text{nat}) < n \implies 0 < (n + 1) \text{ div } 2$
 <proof>

lemma *div-2-gt-zero* [simp]: **assumes** $A: (1 :: \text{nat}) < n$ **shows** $0 < n \text{ div } 2$
 <proof>

lemma *mod-mult-self3* [simp]: $(k*n + m) \bmod n = m \bmod (n::nat)$
 ⟨proof⟩

lemma *mod-mult-self4* [simp]: $Suc (k*n + m) \bmod n = Suc m \bmod n$
 ⟨proof⟩

lemma *mod-Suc-eq-Suc-mod*: $Suc m \bmod n = Suc (m \bmod n) \bmod n$
 ⟨proof⟩

33.4 Division on *int*

definition *divmod-int-rel* :: *int* \Rightarrow *int* \Rightarrow *int* \times *int* \Rightarrow *bool* **where**
 — definition of quotient and remainder

[code]: *divmod-int-rel* *a b* = $(\lambda(q, r). a = b * q + r \wedge$
 $(if\ 0 < b\ then\ 0 \leq r \wedge r < b\ else\ b < r \wedge r \leq 0))$

definition *adjust* :: *int* \Rightarrow *int* \times *int* \Rightarrow *int* \times *int* **where**
 — for the division algorithm

[code]: *adjust* *b* = $(\lambda(q, r). if\ 0 \leq r - b\ then\ (2 * q + 1, r - b)$
 $else\ (2 * q, r))$

algorithm for the case $a \geq 0, b > 0$

function *posDivAlg* :: *int* \Rightarrow *int* \Rightarrow *int* \times *int* **where**

posDivAlg *a b* = $(if\ a < b \vee b \leq 0\ then\ (0, a)$
 $else\ adjust\ b\ (posDivAlg\ a\ (2 * b)))$

⟨proof⟩

termination ⟨proof⟩

algorithm for the case $a < 0, b > 0$

function *negDivAlg* :: *int* \Rightarrow *int* \Rightarrow *int* \times *int* **where**

negDivAlg *a b* = $(if\ 0 \leq a + b \vee b \leq 0\ then\ (-1, a + b)$
 $else\ adjust\ b\ (negDivAlg\ a\ (2 * b)))$

⟨proof⟩

termination ⟨proof⟩

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 *<proof>*

end

lemma *divmod-int-mod-div*:
divmod-int p q = (*p div q*, *p mod q*)
<proof>

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<le>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<le>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*:

$$\begin{aligned} & [[b*q' + r' \leq b*q + r; \ 0 \leq r'; \ r' < b; \ r < b]] \\ & \implies q' \leq (q::int) \end{aligned}$$

<proof>

lemma *unique-quotient-lemma-neg*:

$$\begin{aligned} & [[b*q' + r' \leq b*q + r; \ r \leq 0; \ b < r; \ b < r']] \\ & \implies q \leq (q'::int) \end{aligned}$$

<proof>

lemma *unique-quotient*:

$$\begin{aligned} & [[\text{divmod-int-rel } a \ b \ (q, r); \ \text{divmod-int-rel } a \ b \ (q', r'); \ b \neq 0]] \\ & \implies q = q' \end{aligned}$$

<proof>

lemma *unique-remainder*:

$$\begin{aligned} & [[\text{divmod-int-rel } a \ b \ (q, r); \ \text{divmod-int-rel } a \ b \ (q', r'); \ b \neq 0]] \\ & \implies r = r' \end{aligned}$$

<proof>

33.4.2 Correctness of *posDivAlg*, the Algorithm for Non-Negative Dividends

And positive divisors

lemma *adjust-eq [simp]*:

$$\begin{aligned} & \text{adjust } b \ (q, r) = \\ & \quad (\text{let } \text{diff} = r - b \text{ in} \\ & \quad \text{if } 0 \leq \text{diff} \text{ then } (2*q + 1, \text{diff}) \\ & \quad \text{else } (2*q, r)) \end{aligned}$$

<proof>

declare *posDivAlg.simps [simp del]*

use with a *simproc* to avoid repeatedly proving the premise

lemma *posDivAlg-eqn*:

$$\begin{aligned} & 0 < b \implies \\ & \text{posDivAlg } a \ b = (\text{if } a < b \text{ then } (0, a) \text{ else } \text{adjust } b \ (\text{posDivAlg } a \ (2*b))) \end{aligned}$$

<proof>

Correctness of *posDivAlg*: it computes quotients correctly

theorem *posDivAlg-correct*:

$$\begin{aligned} & \text{assumes } 0 \leq a \text{ and } 0 < b \\ & \text{shows } \text{divmod-int-rel } a \ b \ (\text{posDivAlg } a \ b) \end{aligned}$$

<proof>

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*:

$0 < b \implies$
 $\text{negDivAlg } a \ b =$
 $(\text{if } 0 \leq a+b \text{ then } (-1, a+b) \text{ else adjust } b (\text{negDivAlg } a (2*b)))$
 $\langle \text{proof} \rangle$

lemma *negDivAlg-correct*:

assumes $a < 0$ **and** $b > 0$
shows *divmod-int-rel* $a \ b (\text{negDivAlg } a \ b)$
 $\langle \text{proof} \rangle$

33.4.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *divmod-int-rel-0*: $b \neq 0 \implies \text{divmod-int-rel } 0 \ b (0, 0)$
 $\langle \text{proof} \rangle$

lemma *posDivAlg-0* [*simp*]: $\text{posDivAlg } 0 \ b = (0, 0)$
 $\langle \text{proof} \rangle$

lemma *negDivAlg-minus1* [*simp*]: $\text{negDivAlg } -1 \ b = (-1, b - 1)$
 $\langle \text{proof} \rangle$

lemma *negateSnd-eq* [*simp*]: $\text{negateSnd}(q, r) = (q, -r)$
 $\langle \text{proof} \rangle$

lemma *divmod-int-rel-neg*: $\text{divmod-int-rel } (-a) \ (-b) \ qr \implies \text{divmod-int-rel } a \ b (\text{negateSnd } qr)$
 $\langle \text{proof} \rangle$

lemma *divmod-int-correct*: $b \neq 0 \implies \text{divmod-int-rel } a \ b (\text{divmod-int } a \ b)$
 $\langle \text{proof} \rangle$

Arbitrary definitions for division by zero. Useful to simplify certain equations.

lemma *DIVISION-BY-ZERO* [*simp*]: $a \text{ div } (0::\text{int}) = 0 \ \& \ a \text{ mod } (0::\text{int}) = a$
 $\langle \text{proof} \rangle$

Basic laws about division and remainder

lemma *zmod-zdiv-equality*: $(a::\text{int}) = b * (a \text{ div } b) + (a \text{ mod } b)$

$\langle proof \rangle$

lemma *zdiv-zmod-equality*: $(b * (a \text{ div } b) + (a \text{ mod } b)) + k = (a::int)+k$
 $\langle proof \rangle$

lemma *zdiv-zmod-equality2*: $((a \text{ div } b) * b + (a \text{ mod } b)) + k = (a::int)+k$
 $\langle proof \rangle$

Tool setup

$\langle ML \rangle$

lemma *pos-mod-conj* : $(0::int) < b ==> 0 \leq a \text{ mod } b \ \& \ a \text{ mod } b < b$
 $\langle proof \rangle$

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* : $b < (0::int) ==> a \text{ mod } b \leq 0 \ \& \ b < a \text{ mod } b$
 $\langle proof \rangle$

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*: $b \neq 0 ==> \text{divmod-int-rel } a \ b \ (a \text{ div } b, a \text{ mod } b)$
 $\langle proof \rangle$

lemma *divmod-int-rel-div*: $[\text{divmod-int-rel } a \ b \ (q, r); \ b \neq 0] ==> a \text{ div } b = q$
 $\langle proof \rangle$

lemma *divmod-int-rel-mod*: $[\text{divmod-int-rel } a \ b \ (q, r); \ b \neq 0] ==> a \text{ mod } b = r$
 $\langle proof \rangle$

lemma *div-pos-pos-trivial*: $[(0::int) \leq a; \ a < b] ==> a \text{ div } b = 0$
 $\langle proof \rangle$

lemma *div-neg-neg-trivial*: $[a \leq (0::int); \ b < a] ==> a \text{ div } b = 0$
 $\langle proof \rangle$

lemma *div-pos-neg-trivial*: $[(0::int) < a; \ a+b \leq 0] ==> a \text{ div } b = -1$
 $\langle proof \rangle$

lemma *mod-pos-pos-trivial*: $[(0::int) \leq a; \ a < b] ==> a \text{ mod } b = a$
 $\langle proof \rangle$

lemma *mod-neg-neg-trivial*: $[[a \leq (0::int); b < a]] ==> a \bmod b = a$
 $\langle proof \rangle$

lemma *mod-pos-neg-trivial*: $[[(0::int) < a; a+b \leq 0]] ==> a \bmod b = a+b$
 $\langle proof \rangle$

There is no *mod-neg-pos-trivial*.

lemma *zdiv-zminus-zminus* [simp]: $(-a) \operatorname{div} (-b) = a \operatorname{div} (b::int)$
 $\langle proof \rangle$

lemma *zmod-zminus-zminus* [simp]: $(-a) \bmod (-b) = - (a \bmod (b::int))$
 $\langle proof \rangle$

33.4.6 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
 $\operatorname{divmod-int-rel} a b (q, r)$
 $==> \operatorname{divmod-int-rel} (-a) b (\text{if } r=0 \text{ then } -q \text{ else } -q - 1,$
 $\text{if } r=0 \text{ then } 0 \text{ else } b-r)$
 $\langle proof \rangle$

lemma *zdiv-zminus1-eq-if*:
 $b \neq (0::int)$
 $==> (-a) \operatorname{div} b =$
 $(\text{if } a \bmod b = 0 \text{ then } - (a \operatorname{div} b) \text{ else } - (a \operatorname{div} b) - 1)$
 $\langle proof \rangle$

lemma *zmod-zminus1-eq-if*:
 $(-a::int) \bmod b = (\text{if } a \bmod b = 0 \text{ then } 0 \text{ else } b - (a \bmod b))$
 $\langle proof \rangle$

lemma *zmod-zminus1-not-zero*:
fixes $k l :: int$
shows $- k \bmod l \neq 0 \implies k \bmod l \neq 0$
 $\langle proof \rangle$

lemma *zdiv-zminus2*: $a \operatorname{div} (-b) = (-a::int) \operatorname{div} b$
 $\langle proof \rangle$

lemma *zmod-zminus2*: $a \bmod (-b) = - ((-a::int) \bmod b)$
 $\langle proof \rangle$

lemma *zdiv-zminus2-eq-if*:
 $b \neq (0::int)$
 $==> a \operatorname{div} (-b) =$
 $(\text{if } a \bmod b = 0 \text{ then } - (a \operatorname{div} b) \text{ else } - (a \operatorname{div} b) - 1)$
 $\langle proof \rangle$

lemma *zmod-zminus2-eq-if*:

$a \bmod (-b::int) = (if\ a \bmod b = 0\ then\ 0\ else\ (a \bmod b) - b)$
 $\langle proof \rangle$

lemma *zmod-zminus2-not-zero*:

fixes $k\ l :: int$
shows $k \bmod -l \neq 0 \implies k \bmod l \neq 0$
 $\langle proof \rangle$

33.4.7 Division of a Number by Itself

lemma *self-quotient-aux1*: $[(0::int) < a; a = r + a*q; r < a] \implies 1 \leq q$
 $\langle proof \rangle$

lemma *self-quotient-aux2*: $[(0::int) < a; a = r + a*q; 0 \leq r] \implies q \leq 1$
 $\langle proof \rangle$

lemma *self-quotient*: $[divmod\text{-}int\text{-}rel\ a\ a\ (q, r); a \neq (0::int)] \implies q = 1$
 $\langle proof \rangle$

lemma *self-remainder*: $[divmod\text{-}int\text{-}rel\ a\ a\ (q, r); a \neq (0::int)] \implies r = 0$
 $\langle proof \rangle$

lemma *zdiv-self [simp]*: $a \neq 0 \implies a \div a = (1::int)$
 $\langle proof \rangle$

lemma *zmod-self [simp]*: $a \bmod a = (0::int)$
 $\langle proof \rangle$

33.4.8 Computation of Division and Remainder

lemma *zdiv-zero [simp]*: $(0::int) \div b = 0$
 $\langle proof \rangle$

lemma *div-eq-minus1*: $(0::int) < b \implies -1 \div b = -1$
 $\langle proof \rangle$

lemma *zmod-zero [simp]*: $(0::int) \bmod b = 0$
 $\langle proof \rangle$

lemma *zmod-minus1*: $(0::int) < b \implies -1 \bmod b = b - 1$
 $\langle proof \rangle$

a positive, b positive

lemma *div-pos-pos*: $[0 < a; 0 \leq b] \implies a \div b = fst\ (posDivAlg\ a\ b)$
 $\langle proof \rangle$

lemma *mod-pos-pos*: $[0 < a; 0 \leq b] \implies a \bmod b = snd\ (posDivAlg\ a\ b)$

$\langle proof \rangle$

a negative, b positive

lemma *div-neg-pos*: $\llbracket a < 0; \ 0 < b \rrbracket \implies a \text{ div } b = \text{fst } (\text{negDivAlg } a \ b)$
 $\langle proof \rangle$

lemma *mod-neg-pos*: $\llbracket a < 0; \ 0 < b \rrbracket \implies a \text{ mod } b = \text{snd } (\text{negDivAlg } a \ b)$
 $\langle proof \rangle$

a positive, b negative

lemma *div-pos-neg*:
 $\llbracket 0 < a; \ b < 0 \rrbracket \implies a \text{ div } b = \text{fst } (\text{negateSnd } (\text{negDivAlg } (-a) \ (-b)))$
 $\langle proof \rangle$

lemma *mod-pos-neg*:
 $\llbracket 0 < a; \ b < 0 \rrbracket \implies a \text{ mod } b = \text{snd } (\text{negateSnd } (\text{negDivAlg } (-a) \ (-b)))$
 $\langle proof \rangle$

a negative, b negative

lemma *div-neg-neg*:
 $\llbracket a < 0; \ b \leq 0 \rrbracket \implies a \text{ div } b = \text{fst } (\text{negateSnd } (\text{posDivAlg } (-a) \ (-b)))$
 $\langle proof \rangle$

lemma *mod-neg-neg*:
 $\llbracket a < 0; \ b \leq 0 \rrbracket \implies a \text{ mod } b = \text{snd } (\text{negateSnd } (\text{posDivAlg } (-a) \ (-b)))$
 $\langle proof \rangle$

Simplify expresions in which div and mod combine numerical constants

lemma *divmod-int-relI*:
 $\llbracket a == b * q + r; \text{ if } 0 < b \text{ then } 0 \leq r \wedge r < b \text{ else } b < r \wedge r \leq 0 \rrbracket$
 $\implies \text{divmod-int-rel } a \ b \ (q, \ r)$
 $\langle proof \rangle$

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

$\langle ML \rangle$

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]: $a \bmod (-1::int) = 0$
 $\langle proof \rangle$

lemma *zdiv-minus1-right* [simp]: $a \div (-1::int) = -a$
 $\langle proof \rangle$

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*: $[| a \leq a'; 0 < (b::int) |] \implies a \div b \leq a' \div b$
 $\langle proof \rangle$

lemma *zdiv-mono1-neg*: $[| a \leq a'; (b::int) < 0 |] \implies a' \div b \leq a \div b$
 $\langle proof \rangle$

33.4.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
 $[| 0 \leq b'q' + r'; r' < b'; 0 < b' |] \implies 0 \leq (q'::int)$
 $\langle proof \rangle$

lemma *zdiv-mono2-lemma*:

$$\begin{aligned} & [[\ b*q + r = b'*q' + r'; \ 0 \leq b'*q' + r'; \\ & \quad r' < b'; \ 0 \leq r; \ 0 < b'; \ b' \leq b \] \\ & \implies q \leq (q'::int) \\ & \langle proof \rangle \end{aligned}$$

lemma *zdiv-mono2*:

$$[[\ (0::int) \leq a; \ 0 < b'; \ b' \leq b \] \implies a \text{ div } b \leq a \text{ div } b'$$

 $\langle proof \rangle$

lemma *q-neg-lemma*:

$$[[\ b'*q' + r' < 0; \ 0 \leq r'; \ 0 < b' \] \implies q' \leq (0::int)$$

 $\langle proof \rangle$

lemma *zdiv-mono2-neg-lemma*:

$$\begin{aligned} & [[\ 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::int) \\ & \langle proof \rangle \end{aligned}$$

lemma *zdiv-mono2-neg*:

$$[[\ a < (0::int); \ 0 < b'; \ b' \leq b \] \implies a \text{ div } b' \leq a \text{ div } b$$

 $\langle proof \rangle$

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*:

$$\begin{aligned} & [[\ \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) \\ & \langle proof \rangle \end{aligned}$$

lemma *zdiv-zmult1-eq*: $(a*b) \text{ div } c = a*(b \text{ div } c) + a*(b \text{ mod } c) \text{ div } (c::int)$
 $\langle proof \rangle$

lemma *zmod-zmult1-eq*: $(a*b) \text{ mod } c = a*(b \text{ mod } c) \text{ mod } (c::int)$
 $\langle proof \rangle$

lemma *zmod-zdiv-trivial*: $(a \text{ mod } b) \text{ div } b = (0::int)$
 $\langle proof \rangle$

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*:

$$\begin{aligned} & [[\ \text{divmod-int-rel } a \ c \ (aq, ar); \ \text{divmod-int-rel } b \ c \ (bq, br); \ c \neq 0 \] \\ & \implies \text{divmod-int-rel } (a+b) \ c \ (aq + bq + (ar+br) \text{ div } c, (ar+br) \text{ mod } c) \\ & \langle proof \rangle \end{aligned}$$

lemma *zdiv-zadd1-eq*:

$(a+b) \text{ div } (c::int) = a \text{ div } c + b \text{ div } c + ((a \text{ mod } c + b \text{ mod } c) \text{ div } c)$
 $\langle proof \rangle$

instance *int* :: *ring-div*
 $\langle proof \rangle$

lemma *posDivAlg-div-mod*:
 assumes $k \geq 0$
 and $l \geq 0$
 shows $\text{posDivAlg } k \ l = (k \text{ div } l, k \text{ mod } l)$
 $\langle proof \rangle$

lemma *negDivAlg-div-mod*:
 assumes $k < 0$
 and $l > 0$
 shows $\text{negDivAlg } k \ l = (k \text{ div } l, k \text{ mod } l)$
 $\langle proof \rangle$

lemma *zmod-eq-0-iff*: $(m \text{ mod } d = 0) = (EX \ q::int. m = d*q)$
 $\langle proof \rangle$

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*: $[| (0::int) < c; \ b < r; \ r \leq 0 |] ==> b*c < b*(q \text{ mod } c) + r$
 $\langle proof \rangle$

lemma *zmult2-lemma-aux2*:
 $[| (0::int) < c; \ b < r; \ r \leq 0 |] ==> b * (q \text{ mod } c) + r \leq 0$
 $\langle proof \rangle$

lemma *zmult2-lemma-aux3*: $[| (0::int) < c; \ 0 \leq r; \ r < b |] ==> 0 \leq b * (q \text{ mod } c) + r$
 $\langle proof \rangle$

lemma *zmult2-lemma-aux4*: $[| (0::int) < c; \ 0 \leq r; \ r < b |] ==> b * (q \text{ mod } c) + r < b * c$
 $\langle proof \rangle$

lemma *zmult2-lemma*: $[| \text{divmod-int-rel } a \ b \ (q, r); \ b \neq 0; \ 0 < c |]$
 $==> \text{divmod-int-rel } a \ (b * c) \ (q \text{ div } c, b*(q \text{ mod } c) + r)$
 $\langle proof \rangle$

lemma *zdiv-zmult2-eq*: $(0::int) < c ==> a \text{ div } (b*c) = (a \text{ div } b) \text{ div } c$
 $\langle proof \rangle$

lemma *zmod-zmult2-eq*:

$(0::int) < c ==> a \bmod (b*c) = b*(a \operatorname{div} b \bmod c) + a \bmod b$
 $\langle \text{proof} \rangle$

33.4.13 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

lemma *split-pos-lemma*:

$0 < k ==>$
 $P(n \operatorname{div} k :: int)(n \bmod k) = (\forall i j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P \ i \ j)$
 $\langle \text{proof} \rangle$

lemma *split-neg-lemma*:

$k < 0 ==>$
 $P(n \operatorname{div} k :: int)(n \bmod k) = (\forall i j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P \ i \ j)$
 $\langle \text{proof} \rangle$

lemma *split-zdiv*:

$P(n \operatorname{div} k :: int) =$
 $((k = 0 \longrightarrow P \ 0) \ \&$
 $(0 < k \longrightarrow (\forall i j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P \ i)) \ \&$
 $(k < 0 \longrightarrow (\forall i j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P \ i)))$
 $\langle \text{proof} \rangle$

lemma *split-zmod*:

$P(n \bmod k :: int) =$
 $((k = 0 \longrightarrow P \ n) \ \&$
 $(0 < k \longrightarrow (\forall i j. 0 \leq j \ \& \ j < k \ \& \ n = k*i + j \longrightarrow P \ j)) \ \&$
 $(k < 0 \longrightarrow (\forall i j. k < j \ \& \ j \leq 0 \ \& \ n = k*i + j \longrightarrow P \ j)))$
 $\langle \text{proof} \rangle$

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*: $(0::int) \leq a ==> (1 + 2*b) \operatorname{div} (2*a) = b \operatorname{div} a$
 $\langle \text{proof} \rangle$

lemma *neg-zdiv-mult-2*:

assumes *A*: $a \leq (0::int)$ **shows** $(1 + 2*b) \operatorname{div} (2*a) = (b+1) \operatorname{div} a$
 $\langle \text{proof} \rangle$

lemma *zdiv-number-of-Bit0* [*simp*]:

$$\text{number-of } (\text{Int.Bit0 } v) \text{ div number-of } (\text{Int.Bit0 } w) =$$

$$\text{number-of } v \text{ div } (\text{number-of } w :: \text{int})$$
 <proof>

lemma *zdiv-number-of-Bit1* [simp]:

$$\text{number-of } (\text{Int.Bit1 } v) \text{ div number-of } (\text{Int.Bit0 } w) =$$

$$(\text{if } (0 :: \text{int}) \leq \text{number-of } w$$

$$\text{then number-of } v \text{ div } (\text{number-of } w)$$

$$\text{else } (\text{number-of } v + (1 :: \text{int})) \text{ div } (\text{number-of } w))$$
 <proof>

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>

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>

lemma *zmod-number-of-Bit0* [simp]:

$$\text{number-of } (\text{Int.Bit0 } v) \bmod \text{number-of } (\text{Int.Bit0 } w) =$$

$$(2 :: \text{int}) * (\text{number-of } v \bmod \text{number-of } w)$$
 <proof>

lemma *zmod-number-of-Bit1* [simp]:

$$\text{number-of } (\text{Int.Bit1 } v) \bmod \text{number-of } (\text{Int.Bit0 } w) =$$

$$(\text{if } (0 :: \text{int}) \leq \text{number-of } w$$

$$\text{then } 2 * (\text{number-of } v \bmod \text{number-of } w) + 1$$

$$\text{else } 2 * ((\text{number-of } v + (1 :: \text{int})) \bmod \text{number-of } w) - 1)$$
 <proof>

33.4.16 Quotients of Signs

lemma *div-neg-pos-less0*: $[[a < (0 :: \text{int}); \ 0 < b]] ==> a \text{ div } b < 0$
 <proof>

lemma *div-nonneg-neg-le0*: $[[(0 :: \text{int}) \leq a; \ b < 0]] ==> a \text{ div } b \leq 0$
 <proof>

lemma *div-nonpos-pos-le0*: $[[(a :: \text{int}) \leq 0; \ b > 0]] ==> a \text{ div } b \leq 0$
 <proof>

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*: $(0::int) < b \implies (0 \leq a \text{ div } b) = (0 \leq a)$
 $\langle proof \rangle$

lemma *neg-imp-zdiv-nonneg-iff*:
 $b < (0::int) \implies (0 \leq a \text{ div } b) = (a \leq (0::int))$
 $\langle proof \rangle$

lemma *pos-imp-zdiv-neg-iff*: $(0::int) < b \implies (a \text{ div } b < 0) = (a < 0)$
 $\langle proof \rangle$

lemma *neg-imp-zdiv-neg-iff*: $b < (0::int) \implies (a \text{ div } b < 0) = (0 < a)$
 $\langle proof \rangle$

lemma *nonneg1-imp-zdiv-pos-iff*:
 $(0::int) <= a \implies (a \text{ div } b > 0) = (a >= b \ \& \ b > 0)$
 $\langle proof \rangle$

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*: $f \text{ dvd } m \implies f \text{ dvd } (n::int) \implies f \text{ dvd } m \text{ mod } n$
 $\langle proof \rangle$

lemma *zdvd-zmod-imp-zdvd*: $k \text{ dvd } m \text{ mod } n \implies k \text{ dvd } n \implies k \text{ dvd } (m::int)$
 $\langle proof \rangle$

lemma *zmult-div-cancel*: $(n::int) * (m \text{ div } n) = m - (m \text{ mod } n)$
 $\langle proof \rangle$

lemma *zpower-zmod*: $((x::int) \text{ mod } m) ^ y \text{ mod } m = x ^ y \text{ mod } m$
 $\langle proof \rangle$

lemma *zdiv-int*: $\text{int } (a \text{ div } b) = (\text{int } a) \text{ div } (\text{int } b)$
 $\langle proof \rangle$

lemma *zmod-int*: $\text{int } (a \text{ mod } b) = (\text{int } a) \text{ mod } (\text{int } b)$
 $\langle proof \rangle$

lemma *abs-div*: $(y::int) \text{ dvd } x \implies \text{abs } (x \text{ div } y) = \text{abs } x \text{ div } \text{abs } y$
 $\langle proof \rangle$

lemma *zdvd-mult-div-cancel*: $(n::int) \text{ dvd } m \implies n * (m \text{ div } n) = m$
 $\langle proof \rangle$

Suggested by Matthias Daum

lemma *int-power-div-base*:

$\llbracket 0 < m; 0 < k \rrbracket \implies k \wedge m \text{ div } k = (k::\text{int}) \wedge (m - \text{Suc } 0)$
 $\langle \text{proof} \rangle$

by Brian Huffman

lemma *zminus-zmod*: $-(x::\text{int}) \bmod m \bmod m = -x \bmod m$

$\langle \text{proof} \rangle$

lemma *zdiff-zmod-left*: $(x \bmod m - y) \bmod m = (x - y) \bmod (m::\text{int})$

$\langle \text{proof} \rangle$

lemma *zdiff-zmod-right*: $(x - y \bmod m) \bmod m = (x - y) \bmod (m::\text{int})$

$\langle \text{proof} \rangle$

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*: $0 \leq x \implies \text{nat } (x \text{ div } y) = \text{nat } x \text{ div nat } y$

$\langle \text{proof} \rangle$

lemma *nat-mod-distrib*:

$\llbracket 0 \leq x; 0 \leq y \rrbracket \implies \text{nat } (x \bmod y) = \text{nat } x \bmod \text{nat } y$
 $\langle \text{proof} \rangle$

transfer setup

lemma *transfer-nat-int-functions*:

$(x::\text{int}) \geq 0 \implies y \geq 0 \implies (\text{nat } x) \text{ div } (\text{nat } y) = \text{nat } (x \text{ div } y)$
 $(x::\text{int}) \geq 0 \implies y \geq 0 \implies (\text{nat } x) \bmod (\text{nat } y) = \text{nat } (x \bmod y)$
 $\langle \text{proof} \rangle$

lemma *transfer-nat-int-function-closures*:

$(x::\text{int}) \geq 0 \implies y \geq 0 \implies x \text{ div } y \geq 0$
 $(x::\text{int}) \geq 0 \implies y \geq 0 \implies x \bmod y \geq 0$
 $\langle \text{proof} \rangle$

declare *transfer-morphism-nat-int* [transfer add return:

transfer-nat-int-functions
transfer-nat-int-function-closures

]

lemma *transfer-int-nat-functions*:

$(\text{int } x) \text{ div } (\text{int } y) = \text{int } (x \text{ div } y)$

$(\text{int } x) \text{ mod } (\text{int } y) = \text{int } (x \text{ mod } y)$

$\langle \text{proof} \rangle$

lemma *transfer-int-nat-function-closures*:

$\text{is-nat } x \implies \text{is-nat } y \implies \text{is-nat } (x \text{ div } y)$

$\text{is-nat } x \implies \text{is-nat } y \implies \text{is-nat } (x \text{ mod } y)$

$\langle \text{proof} \rangle$

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*: $\llbracket 0 < x; 1 < k \rrbracket \implies x \text{ div } k < (x::\text{int})$

$\langle \text{proof} \rangle$

lemma *zmod-eq-dvd-iff*: $(x::\text{int}) \text{ mod } n = y \text{ mod } n \longleftrightarrow n \text{ dvd } x - y$

$\langle \text{proof} \rangle$

lemma *nat-mod-eq-lemma*: **assumes** xyn : $(x::\text{nat}) \text{ mod } n = y \text{ mod } n$ **and** xy : y

$\leq x$

shows $\exists q. x = y + n * q$

$\langle \text{proof} \rangle$

lemma *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)

$\langle \text{proof} \rangle$

lemma *div-nat-number-of* [simp]:

$(\text{number-of } v :: \text{nat}) \text{ div } \text{number-of } v' =$

$(\text{if neg } (\text{number-of } v :: \text{int}) \text{ then } 0$

$\text{else nat } (\text{number-of } v \text{ div } \text{number-of } v'))$

$\langle \text{proof} \rangle$

lemma *one-div-nat-number-of* [simp]:

$\text{Suc } 0 \text{ div } \text{number-of } v' = \text{nat } (1 \text{ div } \text{number-of } v')$

$\langle \text{proof} \rangle$

lemma *mod-nat-number-of* [simp]:

$(\text{number-of } v :: \text{nat}) \text{ mod } \text{number-of } v' =$

$(\text{if neg } (\text{number-of } v :: \text{int}) \text{ then } 0$

$\text{else if neg } (\text{number-of } v' :: \text{int}) \text{ then } \text{number-of } v$

$\text{else nat } (\text{number-of } v \text{ mod } \text{number-of } v'))$

$\langle \text{proof} \rangle$

lemma *one-mod-nat-number-of* [simp]:

$$\text{Suc } 0 \bmod \text{number-of } v' =$$

$$(\text{if } \text{neg } (\text{number-of } v' :: \text{int}) \text{ then } \text{Suc } 0$$

$$\text{else } \text{nat } (1 \bmod \text{number-of } v'))$$

$$\langle \text{proof} \rangle$$

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'*:

$$(m :: \text{int}) \bmod n = m - (m \text{ div } n) * n$$

$$\langle \text{proof} \rangle$$

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* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{int}$ **where**

$$\text{pdivmod } k \ l = (|k| \text{ div } |l|, |k| \bmod |l|)$$

lemma *pdivmod-posDivAlg* [code]:

$$\text{pdivmod } k \ l = (\text{if } l = 0 \text{ then } (0, |k|) \text{ else } \text{posDivAlg } |k| \ |l|)$$

$$\langle \text{proof} \rangle$$

lemma *divmod-int-pdivmod*: $\text{divmod-int } k \ l = (\text{if } k = 0 \text{ then } (0, 0) \text{ else if } l = 0$

$$\text{then } (0, k) \text{ else}$$

$$\text{apsnd } ((\text{op } *) (\text{sgn } l)) (\text{if } 0 < l \wedge 0 \leq k \vee l < 0 \wedge k < 0$$

$$\text{then } \text{pdivmod } k \ l$$

$$\text{else } (\text{let } (r, s) = \text{pdivmod } k \ l \text{ in}$$

$$\text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, |l| - s)))$$

$$\langle \text{proof} \rangle$$

lemma *divmod-int-code* [code]: $\text{divmod-int } k \ l = (\text{if } k = 0 \text{ then } (0, 0) \text{ else if } l =$

$$0 \text{ then } (0, k) \text{ else}$$

$$\text{apsnd } ((\text{op } *) (\text{sgn } l)) (\text{if } \text{sgn } k = \text{sgn } l$$

$$\text{then } \text{pdivmod } k \ l$$

$$\text{else } (\text{let } (r, s) = \text{pdivmod } k \ l \text{ in}$$

$$\text{if } s = 0 \text{ then } (-r, 0) \text{ else } (-r - 1, |l| - s)))$$

$$\langle \text{proof} \rangle$$

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⟩

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 ⟨proof⟩

```

```

lemma of-nat-nat-of [simp]:
  of-nat (nat-of k) = k
  ⟨proof⟩

```

```

lemma nat-of-of-nat [simp]:
  nat-of (of-nat n) = n
  ⟨proof⟩

```

```

lemma [measure-function]:
  is-measure nat-of ⟨proof⟩

```

```

lemma code-numeral:

```

$(\bigwedge n::\text{code-numeral}. \text{PROP } P \ n) \equiv (\bigwedge n::\text{nat}. \text{PROP } P \ (\text{of-nat } n))$
 $\langle \text{proof} \rangle$

lemma *code-numeral-case*:
 assumes $\bigwedge n. k = \text{of-nat } n \implies P$
 shows P
 $\langle \text{proof} \rangle$

lemma *code-numeral-induct-raw*:
 assumes $\bigwedge n. P \ (\text{of-nat } n)$
 shows $P \ k$
 $\langle \text{proof} \rangle$

lemma *nat-of-inject* [simp]:
 $\text{nat-of } k = \text{nat-of } l \iff k = l$
 $\langle \text{proof} \rangle$

lemma *of-nat-inject* [simp]:
 $\text{of-nat } n = \text{of-nat } m \iff n = m$
 $\langle \text{proof} \rangle$

instantiation *code-numeral* :: *zero*
begin

definition [simp, code del]:
 $0 = \text{of-nat } 0$

instance $\langle \text{proof} \rangle$

end

definition [simp]:
 $\text{Suc-code-numeral } k = \text{of-nat } (\text{Suc } (\text{nat-of } k))$

rep-datatype $0 :: \text{code-numeral } \text{Suc-code-numeral}$
 $\langle \text{proof} \rangle$

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]:
 $k \neq \text{of-nat } 0 \implies \text{nat-of } k - \text{Suc } 0 < \text{nat-of } k$
 $\langle \text{proof} \rangle$

lemma [simp, code]:
 $\text{code-numeral-size} = \text{nat-of}$
 $\langle \text{proof} \rangle$

lemma [simp, code]:

size = *nat-of*
 $\langle \text{proof} \rangle$

lemmas [*code del*] = *code-numeral.recs code-numeral.cases*

lemma [*code*]:
 $\text{eq-class.eq } k \ l \longleftrightarrow \text{eq-class.eq } (\text{nat-of } k) \ (\text{nat-of } l)$
 $\langle \text{proof} \rangle$

lemma [*code nbe*]:
 $\text{eq-class.eq } (k::\text{code-numeral}) \ k \longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

34.2 Indices as datatype of ints

instantiation *code-numeral* :: *number*
begin

definition
 $\text{number-of} = \text{of-nat} \circ \text{nat}$

instance $\langle \text{proof} \rangle$

end

lemma *nat-of-number* [*simp*]:
 $\text{nat-of } (\text{number-of } k) = \text{number-of } k$
 $\langle \text{proof} \rangle$

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*]:
 $(1::\text{code-numeral}) = \text{of-nat } 1$

definition [*simp*, *code del*]:
 $n + m = \text{of-nat } (\text{nat-of } n + \text{nat-of } m)$

definition [*simp*, *code del*]:
 $n - m = \text{of-nat } (\text{nat-of } n - \text{nat-of } m)$

definition [*simp*, *code del*]:
 $n * m = \text{of-nat } (\text{nat-of } n * \text{nat-of } m)$

definition [*simp*, *code del*]:
 $n \text{ div } m = \text{of-nat } (\text{nat-of } n \text{ div } \text{nat-of } m)$

definition [*simp*, *code del*]:
 $n \bmod m = \text{of-nat } (\text{nat-of } n \bmod \text{nat-of } m)$

definition [*simp*, *code del*]:
 $n \leq m \iff \text{nat-of } n \leq \text{nat-of } m$

definition [*simp*, *code del*]:
 $n < m \iff \text{nat-of } n < \text{nat-of } m$

instance $\langle \text{proof} \rangle$

end

lemma *zero-code-numeral-code* [*code*, *code-unfold*]:
 $(0::\text{code-numeral}) = \text{Numeral0}$
 $\langle \text{proof} \rangle$

lemma [*code-post*]: $\text{Numeral0} = (0::\text{code-numeral})$
 $\langle \text{proof} \rangle$

lemma *one-code-numeral-code* [*code*, *code-unfold*]:
 $(1::\text{code-numeral}) = \text{Numeral1}$
 $\langle \text{proof} \rangle$

lemma [*code-post*]: $\text{Numeral1} = (1::\text{code-numeral})$
 $\langle \text{proof} \rangle$

lemma *plus-code-numeral-code* [*code nbe*]:
 $\text{of-nat } n + \text{of-nat } m = \text{of-nat } (n + m)$
 $\langle \text{proof} \rangle$

definition *subtract-code-numeral* :: $\text{code-numeral} \Rightarrow \text{code-numeral} \Rightarrow \text{code-numeral}$
where
 [*simp*, *code del*]: $\text{subtract-code-numeral} = \text{op } -$

lemma *subtract-code-numeral-code* [*code nbe*]:
 $\text{subtract-code-numeral } (\text{of-nat } n) (\text{of-nat } m) = \text{of-nat } (n - m)$
 $\langle \text{proof} \rangle$

lemma *minus-code-numeral-code* [*code*]:
 $n - m = \text{subtract-code-numeral } n m$
 $\langle \text{proof} \rangle$

lemma *times-code-numeral-code* [*code nbe*]:
 $\text{of-nat } n * \text{of-nat } m = \text{of-nat } (n * m)$
 $\langle \text{proof} \rangle$

lemma *less-eq-code-numeral-code* [*code nbe*]:
 $\text{of-nat } n \leq \text{of-nat } m \iff n \leq m$
 $\langle \text{proof} \rangle$

lemma *less-code-numeral-code* [code nbe]:
 $of\text{-}nat\ n < of\text{-}nat\ m \longleftrightarrow n < m$
 ⟨proof⟩

lemma *code-numeral-zero-minus-one*:
 $(0::code\text{-}numeral) - 1 = 0$
 ⟨proof⟩

lemma *Suc-code-numeral-minus-one*:
 $Suc\text{-}code\text{-}numeral\ n - 1 = n$
 ⟨proof⟩

lemma *of-nat-code* [code]:
 $of\text{-}nat = Nat.of\text{-}nat$
 ⟨proof⟩

lemma *code-numeral-not-eq-zero*: $i \neq of\text{-}nat\ 0 \longleftrightarrow i \geq 1$
 ⟨proof⟩

definition *nat-of-aux* :: $code\text{-}numeral \Rightarrow nat \Rightarrow nat$ **where**
 $nat\text{-}of\text{-}aux\ i\ n = nat\text{-}of\ i + n$

lemma *nat-of-aux-code* [code]:
 $nat\text{-}of\text{-}aux\ i\ n = (if\ i = 0\ then\ n\ else\ nat\text{-}of\text{-}aux\ (i - 1)\ (Suc\ n))$
 ⟨proof⟩

lemma *nat-of-code* [code]:
 $nat\text{-}of\ i = nat\text{-}of\text{-}aux\ i\ 0$
 ⟨proof⟩

definition *div-mod-code-numeral* :: $code\text{-}numeral \Rightarrow code\text{-}numeral \Rightarrow code\text{-}numeral$
 $\times code\text{-}numeral$ **where**
 [code del]: $div\text{-}mod\text{-}code\text{-}numeral\ n\ m = (n\ div\ m, n\ mod\ m)$

lemma [code]:
 $div\text{-}mod\text{-}code\text{-}numeral\ n\ m = (if\ m = 0\ then\ (0, n)\ else\ (n\ div\ m, n\ mod\ m))$
 ⟨proof⟩

lemma [code]:
 $n\ div\ m = fst\ (div\text{-}mod\text{-}code\text{-}numeral\ n\ m)$
 ⟨proof⟩

lemma [code]:
 $n\ mod\ m = snd\ (div\text{-}mod\text{-}code\text{-}numeral\ n\ m)$
 ⟨proof⟩

definition *int-of* :: $code\text{-}numeral \Rightarrow int$ **where**
 $int\text{-}of = Nat.of\text{-}nat\ o\ nat\text{-}of$

lemma *int-of-code* [code]:
int-of $k = (\text{if } k = 0 \text{ then } 0$
 $\text{else } (\text{if } k \bmod 2 = 0 \text{ then } 2 * \text{int-of } (k \text{ div } 2) \text{ else } 2 * \text{int-of } (k \text{ div } 2) + 1))$
 ⟨proof⟩

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 $f\ n\ m = \text{Predicate.Seq } (\%u. \text{if } n > m \text{ then } \text{Predicate.Empty} \text{ else } \text{Predicate.Insert } (f\ n) (\text{iterate-upto } f\ (n + 1)\ m))$
 ⟨proof⟩

termination ⟨proof⟩

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 *-*)

⟨ML⟩

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*: $i*u + (j*u + k) = (i+j)*u + (k::nat)$
 <proof>

For *cancel-numerals*

lemma *nat-diff-add-eq1*:
 $j <= (i::nat) ==> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)$
 <proof>

lemma *nat-diff-add-eq2*:
 $i <= (j::nat) ==> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))$
 <proof>

lemma *nat-eq-add-iff1*:
 $j <= (i::nat) ==> (i*u + m = j*u + n) = ((i-j)*u + m = n)$
 <proof>

lemma *nat-eq-add-iff2*:
 $i <= (j::nat) ==> (i*u + m = j*u + n) = (m = (j-i)*u + n)$
 <proof>

lemma *nat-less-add-iff1*:
 $j <= (i::nat) ==> (i*u + m < j*u + n) = ((i-j)*u + m < n)$
 <proof>

lemma *nat-less-add-iff2*:
 $i <= (j::nat) ==> (i*u + m < j*u + n) = (m < (j-i)*u + n)$
 <proof>

lemma *nat-le-add-iff1*:
 $j <= (i::nat) ==> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)$
 <proof>

lemma *nat-le-add-iff2*:
 $i <= (j::nat) ==> (i*u + m <= j*u + n) = (m <= (j-i)*u + n)$
 <proof>

For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k ==> (k*m <= k*n) = (m <= n)$
 <proof>

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m < n)$
 <proof>

lemma *nat-mult-eq-cancel1*: $(0::nat) < k \implies (k*m = k*n) = (m=n)$
 $\langle proof \rangle$

lemma *nat-mult-div-cancel1*: $(0::nat) < k \implies (k*m) \text{ div } (k*n) = (m \text{ div } n)$
 $\langle proof \rangle$

lemma *nat-mult-dvd-cancel-disj*[*simp*]:
 $(k*m) \text{ dvd } (k*n) = (k=0 \mid m \text{ dvd } (n::nat))$
 $\langle proof \rangle$

lemma *nat-mult-dvd-cancel1*: $0 < k \implies (k*m) \text{ dvd } (k*n::nat) = (m \text{ dvd } n)$
 $\langle proof \rangle$

For *cancel-factor*

lemma *nat-mult-le-cancel-disj*: $(k*m \leq k*n) = ((0::nat) < k \longrightarrow m \leq n)$
 $\langle proof \rangle$

lemma *nat-mult-less-cancel-disj*: $(k*m < k*n) = ((0::nat) < k \ \& \ m < n)$
 $\langle proof \rangle$

lemma *nat-mult-eq-cancel-disj*: $(k*m = k*n) = (k = (0::nat) \mid m=n)$
 $\langle proof \rangle$

lemma *nat-mult-div-cancel-disj*[*simp*]:
 $(k*m) \text{ div } (k*n) = (\text{if } k = (0::nat) \text{ then } 0 \text{ else } m \text{ div } n)$
 $\langle proof \rangle$

$\langle ML \rangle$

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*:

$$a \neq b \wedge c \neq d \longleftrightarrow a * c + b * d \neq a * d + b * c$$

<proof>

lemma *add-scale-eq-noteq*:

$$r \neq 0 \implies a = b \wedge c \neq d \implies a + r * c \neq b + r * d$$

<proof>

lemma *add-0-iff*:

$$b = b + a \longleftrightarrow a = 0$$

<proof>

end

sublocale *idom* < *comm-semiring-1-cancel-crossproduct*
<proof>

instance *nat* :: *comm-semiring-1-cancel-crossproduct*
<proof>

Semiring normalization proper

<ML>

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 *<proof>*

lemma *normalizing-semiring-rules*:

$$\begin{aligned} (a * m) + (b * m) &= (a + b) * m \\ (a * m) + m &= (a + 1) * m \\ m + (a * m) &= (a + 1) * m \\ m + m &= (1 + 1) * m \\ 0 + a &= a \\ a + 0 &= a \\ a * b &= b * a \\ (a + b) * c &= (a * c) + (b * c) \\ 0 * a &= 0 \\ a * 0 &= 0 \\ 1 * a &= a \\ a * 1 &= a \\ (lx * ly) * (rx * ry) &= (lx * rx) * (ly * ry) \\ (lx * ly) * (rx * ry) &= lx * (ly * (rx * ry)) \\ (lx * ly) * (rx * ry) &= rx * ((lx * ly) * ry) \\ (lx * ly) * rx &= (lx * rx) * ly \\ (lx * ly) * rx &= lx * (ly * rx) \\ lx * (rx * ry) &= (lx * rx) * ry \\ lx * (rx * ry) &= rx * (lx * ry) \end{aligned}$$

$$\begin{aligned}
(a + b) + (c + d) &= (a + c) + (b + d) \\
(a + b) + c &= a + (b + c) \\
a + (c + d) &= c + (a + d) \\
(a + b) + c &= (a + c) + b \\
a + c &= c + a \\
a + (c + d) &= (a + c) + d \\
(x \wedge p) * (x \wedge q) &= x \wedge (p + q) \\
x * (x \wedge q) &= x \wedge (\text{Suc } q) \\
(x \wedge q) * x &= x \wedge (\text{Suc } q) \\
x * x &= x \wedge 2 \\
(x * y) \wedge q &= (x \wedge q) * (y \wedge q) \\
(x \wedge p) \wedge q &= x \wedge (p * q) \\
x \wedge 0 &= 1 \\
x \wedge 1 &= x \\
x * (y + z) &= (x * y) + (x * z) \\
x \wedge (\text{Suc } q) &= x * (x \wedge q) \\
x \wedge (2 * n) &= (x \wedge n) * (x \wedge n) \\
x \wedge (\text{Suc } (2 * n)) &= x * ((x \wedge n) * (x \wedge n)) \\
\langle \text{proof} \rangle
\end{aligned}$$

lemmas *normalizing-comm-semiring-1-axioms* =
comm-semiring-1-axioms [normalizer
semiring ops: *normalizing-semiring-ops*
semiring rules: *normalizing-semiring-rules*]

$\langle ML \rangle$

end

context *comm-ring-1*
begin

lemma *normalizing-ring-ops*: **shows** *TERM* $(x - y)$ **and** *TERM* $(- x)$ $\langle \text{proof} \rangle$

lemma *normalizing-ring-rules*:

$$\begin{aligned}
- x &= (- 1) * x \\
x - y &= x + (- y) \\
\langle \text{proof} \rangle
\end{aligned}$$

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*]

$\langle ML \rangle$

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]

⟨*ML*⟩

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]

⟨*ML*⟩

end

context *field*
begin

lemma *normalizing-field-ops:*
shows *TERM* (*x* / *y*) **and** *TERM* (*inverse x*) ⟨*proof*⟩

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]

⟨ML⟩

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:
  
$$(P \ \& \ (Q \mid R)) = ((P \ \& \ Q) \mid (P \ \& \ R)) \ ((Q \mid R) \ \& \ P) = ((Q \ \& \ P) \mid (R \ \& \ P))$$

  
$$(P \ \wedge \ Q) = (Q \ \wedge \ P) \ (P \ \vee \ Q) = (Q \ \vee \ P)$$

  ⟨proof⟩

```

```

lemmas weak-dnf-simps = dnf bool-simps

```

```

lemma nnf-simps:
  
$$(\neg(P \ \wedge \ Q)) = (\neg P \ \vee \ \neg Q) \ (\neg(P \ \vee \ Q)) = (\neg P \ \wedge \ \neg Q) \ (P \longrightarrow Q) = (\neg P \ \vee \ Q)$$

  
$$(P = Q) = ((P \ \wedge \ Q) \vee (\neg P \ \wedge \ \neg Q)) \ (\neg \neg(P)) = P$$

  ⟨proof⟩

```

```

lemma PFalse:
  
$$P \equiv \text{False} \implies \neg P$$

  
$$\neg P \implies (P \equiv \text{False})$$

  ⟨proof⟩

```

$\langle ML \rangle$

```

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{..<}-})) where
    {..} == {x. x < u}

definition
  atMost      :: 'a => 'a set ((1{..}-})) where
    {..u} == {x. x ≤ u}

```

definition

greaterThan :: 'a => 'a set ((1{-<..})) **where**
 {l<..} == {x. l<x}

definition

atLeast :: 'a => 'a set ((1{-..})) **where**
 {l..} == {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 ((3∪ -≤.. / -) [0, 0, 10] 10)
 -UNION-less :: 'a => 'a => 'b set => 'b set ((3∪ -<.. / -) [0, 0, 10] 10)
 -INTER-le :: 'a => 'a => 'b set => 'b set ((3∩ -≤.. / -) [0, 0, 10] 10)
 -INTER-less :: 'a => 'a => 'b set => 'b set ((3∩ -<.. / -) [0, 0, 10] 10)

syntax (*latex output*)

-UNION-le :: 'a ⇒ 'a => 'b set => 'b set ((3∪ (00- ≤ -) / -) [0, 0, 10] 10)
 -UNION-less :: 'a ⇒ 'a => 'b set => 'b set ((3∪ (00- < -) / -) [0, 0, 10] 10)
 -INTER-le :: 'a ⇒ 'a => 'b set => 'b set ((3∩ (00- ≤ -) / -) [0, 0, 10] 10)

10)
 $-INTER-less :: 'a \Rightarrow 'a \Rightarrow 'b \text{ set} \Rightarrow 'b \text{ set} \quad ((\exists \cap (00- < -)/ -) [0, 0, 10])$
 10)

translations

$UN\ i \leq n. A == UN\ i:\{..n\}. A$
 $UN\ i < n. A == UN\ i:\{..< n\}. A$
 $INT\ i \leq n. A == INT\ i:\{..n\}. A$
 $INT\ i < n. A == INT\ i:\{..< n\}. A$

38.1 Various equivalences

lemma (in ord) lessThan-iff [iff]: $(i: lessThan\ k) = (i < k)$
 $\langle proof \rangle$

lemma Compl-lessThan [simp]:
 $!!k:: 'a::linorder. \neg lessThan\ k = atLeast\ k$
 $\langle proof \rangle$

lemma single-Diff-lessThan [simp]: $!!k:: 'a::order. \{k\} - lessThan\ k = \{k\}$
 $\langle proof \rangle$

lemma (in ord) greaterThan-iff [iff]: $(i: greaterThan\ k) = (k < i)$
 $\langle proof \rangle$

lemma Compl-greaterThan [simp]:
 $!!k:: 'a::linorder. \neg greaterThan\ k = atMost\ k$
 $\langle proof \rangle$

lemma Compl-atMost [simp]: $!!k:: 'a::linorder. \neg atMost\ k = greaterThan\ k$
 $\langle proof \rangle$

lemma (in ord) atLeast-iff [iff]: $(i: atLeast\ k) = (k \leq i)$
 $\langle proof \rangle$

lemma Compl-atLeast [simp]:
 $!!k:: 'a::linorder. \neg atLeast\ k = lessThan\ k$
 $\langle proof \rangle$

lemma (in ord) atMost-iff [iff]: $(i: atMost\ k) = (i \leq k)$
 $\langle proof \rangle$

lemma atMost-Int-atLeast: $!!n:: 'a::order. atMost\ n\ Int\ atLeast\ n = \{n\}$
 $\langle proof \rangle$

38.2 Logical Equivalences for Set Inclusion and Equality

lemma atLeast-subset-iff [iff]:
 $(atLeast\ x \subseteq atLeast\ y) = (y \leq (x::'a::order))$
 $\langle proof \rangle$

lemma *atLeast-eq-iff* [iff]:
 $(\text{atLeast } x = \text{atLeast } y) = (x = (y::'a::\text{linorder}))$
 $\langle \text{proof} \rangle$

lemma *greaterThan-subset-iff* [iff]:
 $(\text{greaterThan } x \subseteq \text{greaterThan } y) = (y \leq (x::'a::\text{linorder}))$
 $\langle \text{proof} \rangle$

lemma *greaterThan-eq-iff* [iff]:
 $(\text{greaterThan } x = \text{greaterThan } y) = (x = (y::'a::\text{linorder}))$
 $\langle \text{proof} \rangle$

lemma *atMost-subset-iff* [iff]: $(\text{atMost } x \subseteq \text{atMost } y) = (x \leq (y::'a::\text{order}))$
 $\langle \text{proof} \rangle$

lemma *atMost-eq-iff* [iff]: $(\text{atMost } x = \text{atMost } y) = (x = (y::'a::\text{linorder}))$
 $\langle \text{proof} \rangle$

lemma *lessThan-subset-iff* [iff]:
 $(\text{lessThan } x \subseteq \text{lessThan } y) = (x \leq (y::'a::\text{linorder}))$
 $\langle \text{proof} \rangle$

lemma *lessThan-eq-iff* [iff]:
 $(\text{lessThan } x = \text{lessThan } y) = (x = (y::'a::\text{linorder}))$
 $\langle \text{proof} \rangle$

38.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [simp,no-atp]:
 $(i : \{l <..<u\}) = (l < i \ \& \ i < u)$
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-iff* [simp,no-atp]:
 $(i : \{l..<u\}) = (l \leq i \ \& \ i < u)$
 $\langle \text{proof} \rangle$

lemma *greaterThanAtMost-iff* [simp,no-atp]:
 $(i : \{l <..u\}) = (l < i \ \& \ i \leq u)$
 $\langle \text{proof} \rangle$

lemma *atLeastAtMost-iff* [simp,no-atp]:
 $(i : \{l..u\}) = (l \leq i \ \& \ i \leq u)$
 $\langle \text{proof} \rangle$

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 Emptyness, singletons, subset

context *order*

begin

lemma *atLeastatMost-empty[simp]*:

$$b < a \implies \{a..b\} = \{\}$$

<proof>

lemma *atLeastatMost-empty-iff[simp]*:

$$\{a..b\} = \{\} \longleftrightarrow (\sim a \leq b)$$

<proof>

lemma *atLeastatMost-empty-iff2[simp]*:

$$\{\} = \{a..b\} \longleftrightarrow (\sim a \leq b)$$

<proof>

lemma *atLeastLessThan-empty[simp]*:

$$b \leq a \implies \{a..<b\} = \{\}$$

<proof>

lemma *atLeastLessThan-empty-iff[simp]*:

$$\{a..<b\} = \{\} \longleftrightarrow (\sim a < b)$$

<proof>

lemma *atLeastLessThan-empty-iff2[simp]*:

$$\{\} = \{a..<b\} \longleftrightarrow (\sim a < b)$$

<proof>

lemma *greaterThanAtMost-empty[simp]*: $l \leq k \implies \{k<..l\} = \{\}$

<proof>

lemma *greaterThanAtMost-empty-iff[simp]*: $\{k<..l\} = \{\} \longleftrightarrow \sim k < l$

<proof>

lemma *greaterThanAtMost-empty-iff2[simp]*: $\{\} = \{k<..l\} \longleftrightarrow \sim k < l$

<proof>

lemma *greaterThanLessThan-empty[simp]*: $l \leq k \implies \{k<..$

<proof>

lemma *atLeastAtMost-singleton [simp]*: $\{a..a\} = \{a\}$

<proof>

lemma *atLeastAtMost-singleton'*: $a = b \implies \{a .. b\} = \{a\}$ *<proof>*

lemma *atLeastatMost-subset-iff[simp]*:

$\{a..b\} \leq \{c..d\} \longleftrightarrow (\sim a \leq b) \mid c \leq a \ \& \ b \leq d$
 $\langle proof \rangle$

lemma *atLeastAtMost-psubset-iff*:

$\{a..b\} < \{c..d\} \longleftrightarrow$
 $((\sim a \leq b) \mid c \leq a \ \& \ b \leq d \ \& \ (c < a \mid b < d)) \ \& \ c \leq d$
 $\langle proof \rangle$

lemma *atLeastAtMost-singleton-iff[simp]*:

$\{a .. b\} = \{c\} \longleftrightarrow a = b \wedge b = c$
 $\langle proof \rangle$

end

lemma (*in linorder*) *atLeastLessThan-subset-iff*:

$\{a..<b\} \leq \{c..<d\} \implies b \leq a \mid c \leq a \ \& \ b \leq d$
 $\langle proof \rangle$

38.3.2 Intersection

context *linorder*

begin

lemma *Int-atLeastAtMost[simp]*: $\{a..b\} \text{ Int } \{c..d\} = \{\max a \ c .. \min b \ d\}$
 $\langle proof \rangle$

lemma *Int-atLeastAtMostR1[simp]*: $\{..b\} \text{ Int } \{c..d\} = \{c .. \min b \ d\}$
 $\langle proof \rangle$

lemma *Int-atLeastAtMostR2[simp]*: $\{a.. \} \text{ Int } \{c..d\} = \{\max a \ c .. d\}$
 $\langle proof \rangle$

lemma *Int-atLeastAtMostL1[simp]*: $\{a..b\} \text{ Int } \{..d\} = \{a .. \min b \ d\}$
 $\langle proof \rangle$

lemma *Int-atLeastAtMostL2[simp]*: $\{a..b\} \text{ Int } \{c.. \} = \{\max a \ c .. b\}$
 $\langle proof \rangle$

lemma *Int-atLeastLessThan[simp]*: $\{a..<b\} \text{ Int } \{c..<d\} = \{\max a \ c ..< \min b \ d\}$
 $\langle proof \rangle$

lemma *Int-greaterThanAtMost[simp]*: $\{a<..b\} \text{ Int } \{c<..d\} = \{\max a \ c <.. \min b \ d\}$
 $\langle proof \rangle$

lemma *Int-greaterThanLessThan[simp]*: $\{a<..<b\} \text{ Int } \{c<..<d\} = \{\max a \ c <..< \min b \ d\}$
 $\langle proof \rangle$

end

38.4 Intervals of natural numbers

38.4.1 The Constant *lessThan*

lemma *lessThan-0* [simp]: *lessThan* (*0::nat*) = {}
 ⟨proof⟩

lemma *lessThan-Suc*: *lessThan* (*Suc k*) = *insert k (lessThan k)*
 ⟨proof⟩

lemma *lessThan-Suc-atMost*: *lessThan* (*Suc k*) = *atMost k*
 ⟨proof⟩

lemma *UN-lessThan-UNIV*: (*UN m::nat. lessThan m*) = *UNIV*
 ⟨proof⟩

38.4.2 The Constant *greaterThan*

lemma *greaterThan-0* [simp]: *greaterThan 0* = *range Suc*
 ⟨proof⟩

lemma *greaterThan-Suc*: *greaterThan* (*Suc k*) = *greaterThan k* − {*Suc k*}
 ⟨proof⟩

lemma *INT-greaterThan-UNIV*: (*INT m::nat. greaterThan m*) = {}
 ⟨proof⟩

38.4.3 The Constant *atLeast*

lemma *atLeast-0* [simp]: *atLeast* (*0::nat*) = *UNIV*
 ⟨proof⟩

lemma *atLeast-Suc*: *atLeast* (*Suc k*) = *atLeast k* − {*k*}
 ⟨proof⟩

lemma *atLeast-Suc-greaterThan*: *atLeast* (*Suc k*) = *greaterThan k*
 ⟨proof⟩

lemma *UN-atLeast-UNIV*: (*UN m::nat. atLeast m*) = *UNIV*
 ⟨proof⟩

38.4.4 The Constant *atMost*

lemma *atMost-0* [simp]: *atMost* (*0::nat*) = {*0*}
 ⟨proof⟩

lemma *atMost-Suc*: *atMost* (*Suc k*) = *insert (Suc k) (atMost k)*
 ⟨proof⟩

lemma *UN-atMost-UNIV*: $(UN\ m::nat.\ atMost\ m) = UNIV$
 $\langle proof \rangle$

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*: $\{0::nat..<n\} = \{..<n\}$
 $\langle proof \rangle$

lemma *atLeast0AtMost*: $\{0..n::nat\} = \{..n\}$
 $\langle proof \rangle$

declare *atLeast0LessThan*[*symmetric, code-unfold*]
atLeast0AtMost[*symmetric, code-unfold*]

lemma *atLeastLessThan0*: $\{m..<0::nat\} = \{\}$
 $\langle proof \rangle$

38.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
 $\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$
 $\langle proof \rangle$

lemma *atLeastLessThan-singleton* [*simp*]: $\{m..<Suc\ m\} = \{m\}$
 $\langle proof \rangle$

lemma *atLeastLessThanSuc-atLeastAtMost*: $\{l..<Suc\ u\} = \{l..u\}$
 $\langle proof \rangle$

lemma *atLeastSucAtMost-greaterThanAtMost*: $\{Suc\ l..u\} = \{l<..u\}$
 $\langle proof \rangle$

lemma *atLeastSucLessThan-greaterThanLessThan*: $\{Suc\ l..<u\} = \{l<..<u\}$
 $\langle proof \rangle$

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \implies \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
 $\langle proof \rangle$

lemma *atLeastLessThan-add-Un*: $i \leq j \implies \{i..<j+k\} = \{i..<j\} \cup \{j..<j+k::nat\}$
 $\langle proof \rangle$

38.4.7 Image**lemma** *image-add-atLeastAtMost*:
$$(\%n::nat. n+k) \text{ ‘ } \{i..j\} = \{i+k..j+k\} \text{ (is } ?A = ?B)$$

<proof>

lemma *image-add-atLeastLessThan*:
$$(\%n::nat. n+k) \text{ ‘ } \{i..<j\} = \{i+k..<j+k\} \text{ (is } ?A = ?B)$$

<proof>

corollary *image-Suc-atLeastAtMost[simp]*:
$$Suc \text{ ‘ } \{i..j\} = \{Suc\ i..Suc\ j\}$$

<proof>

corollary *image-Suc-atLeastLessThan[simp]*:
$$Suc \text{ ‘ } \{i..<j\} = \{Suc\ i..<Suc\ j\}$$

<proof>

lemma *image-add-int-atLeastLessThan*:
$$(\%x. x + (l::int)) \text{ ‘ } \{0..<u-l\} = \{l..<u\}$$

<proof>

context *ordered-ab-group-add***begin****lemma**

fixes $x :: 'a$
shows *image-uminus-greaterThan[simp]*: $uminus \text{ ‘ } \{x<..\} = \{..<-x\}$
and *image-uminus-atLeast[simp]*: $uminus \text{ ‘ } \{x..\} = \{..-x\}$

<proof>

lemma

fixes $x :: 'a$
shows *image-uminus-lessThan[simp]*: $uminus \text{ ‘ } \{..<x\} = \{-x<..\}$
and *image-uminus-atMost[simp]*: $uminus \text{ ‘ } \{..x\} = \{-x..\}$

<proof>

lemma

fixes $x :: 'a$
shows *image-uminus-atLeastAtMost[simp]*: $uminus \text{ ‘ } \{x..y\} = \{-y..-x\}$
and *image-uminus-greaterThanAtMost[simp]*: $uminus \text{ ‘ } \{x<..y\} = \{-y..<-x\}$
and *image-uminus-atLeastLessThan[simp]*: $uminus \text{ ‘ } \{x..<y\} = \{-y<..-x\}$
and *image-uminus-greaterThanLessThan[simp]*: $uminus \text{ ‘ } \{x<..<y\} = \{-y<..<-x\}$

<proof>

end**38.4.8 Finiteness****lemma** *finite-lessThan [iff]*: **fixes** $k :: nat$ **shows** *finite* $\{..<k\}$ *<proof>*

lemma *finite-atMost* [iff]: **fixes** $k :: nat$ **shows** *finite* $\{..k\}$
 $\langle proof \rangle$

lemma *finite-greaterThanLessThan* [iff]:
fixes $l :: nat$ **shows** *finite* $\{l < .. < u\}$
 $\langle proof \rangle$

lemma *finite-atLeastLessThan* [iff]:
fixes $l :: nat$ **shows** *finite* $\{l .. < u\}$
 $\langle proof \rangle$

lemma *finite-greaterThanAtMost* [iff]:
fixes $l :: nat$ **shows** *finite* $\{l < .. u\}$
 $\langle proof \rangle$

lemma *finite-atLeastAtMost* [iff]:
fixes $l :: nat$ **shows** *finite* $\{l .. u\}$
 $\langle proof \rangle$

A bounded set of natural numbers is finite.

lemma *bounded-nat-set-is-finite*:
 $(ALL\ i:N. i < (n::nat)) ==> finite\ N$
 $\langle proof \rangle$

A set of natural numbers is finite iff it is bounded.

lemma *finite-nat-set-iff-bounded*:
 $finite(N::nat\ set) = (EX\ m. ALL\ n:N. n < m) \text{ (is } ?F = ?B)$
 $\langle proof \rangle$

lemma *finite-nat-set-iff-bounded-le*:
 $finite(N::nat\ set) = (EX\ m. ALL\ n:N. n \leq m)$
 $\langle proof \rangle$

lemma *finite-less-ub*:
 $!!f::nat=>nat. (!!n. n \leq f\ n) ==> finite\ \{n. f\ n \leq u\}$
 $\langle proof \rangle$

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
 $A \leq \{k..<k+card\ A\} \implies A = \{k..<k+card\ A\} \text{ (is } PROP\ ?P)$
 $\langle proof \rangle$

38.4.9 Proving Inclusions and Equalities between Unions

lemma *UN-le-eq-UN0*:
 $(\bigcup i \leq n::nat. M\ i) = (\bigcup i \in \{1..n\}. M\ i) \cup M\ 0 \text{ (is } ?A = ?B)$
 $\langle proof \rangle$

lemma *UN-le-add-shift*:

$(\bigcup_{i \leq n :: \text{nat.}} M(i+k)) = (\bigcup_{i \in \{k..n+k\}.} M i) \text{ (is } ?A = ?B)$
 $\langle \text{proof} \rangle$

lemma *UN-UN-finite-eq*: $(\bigcup_{n :: \text{nat.}} \bigcup_{i \in \{0..<n\}.} A i) = (\bigcup_{n.} A n)$
 $\langle \text{proof} \rangle$

lemma *UN-finite-subset*: $(!!n :: \text{nat.} (\bigcup_{i \in \{0..<n\}.} A i) \subseteq C) \implies (\bigcup_{n.} A n) \subseteq C$
 $\langle \text{proof} \rangle$

lemma *UN-finite2-subset*:

$(!!n :: \text{nat.} (\bigcup_{i \in \{0..<n\}.} A i) \subseteq (\bigcup_{i \in \{0..<n+k\}.} B i)) \implies (\bigcup_{n.} A n) \subseteq (\bigcup_{n.} B n)$
 $\langle \text{proof} \rangle$

lemma *UN-finite2-eq*:

$(!!n :: \text{nat.} (\bigcup_{i \in \{0..<n\}.} A i) = (\bigcup_{i \in \{0..<n+k\}.} B i)) \implies (\bigcup_{n.} A n) = (\bigcup_{n.} B n)$
 $\langle \text{proof} \rangle$

38.4.10 Cardinality

lemma *card-lessThan* [simp]: $\text{card } \{..<u\} = u$
 $\langle \text{proof} \rangle$

lemma *card-atMost* [simp]: $\text{card } \{..u\} = \text{Suc } u$
 $\langle \text{proof} \rangle$

lemma *card-atLeastLessThan* [simp]: $\text{card } \{l..<u\} = u - l$
 $\langle \text{proof} \rangle$

lemma *card-atLeastAtMost* [simp]: $\text{card } \{l..u\} = \text{Suc } u - l$
 $\langle \text{proof} \rangle$

lemma *card-greaterThanAtMost* [simp]: $\text{card } \{l<..u\} = u - l$
 $\langle \text{proof} \rangle$

lemma *card-greaterThanLessThan* [simp]: $\text{card } \{l<..<u\} = u - \text{Suc } l$
 $\langle \text{proof} \rangle$

lemma *ex-bij-betw-nat-finite*:

$\text{finite } M \implies \exists h. \text{bij-betw } h \{0..<\text{card } M\} M$
 $\langle \text{proof} \rangle$

lemma *ex-bij-betw-finite-nat*:

$\text{finite } M \implies \exists h. \text{bij-betw } h M \{0..<\text{card } M\}$
 $\langle \text{proof} \rangle$

lemma *finite-same-card-bij*:

$finite\ A \implies finite\ B \implies card\ A = card\ B \implies \exists h. \text{bij-betw } h\ A\ B$
 $\langle proof \rangle$

lemma *ex-bij-betw-nat-finite-1*:

$finite\ M \implies \exists h. \text{bij-betw } h\ \{1 \ ..\ card\ M\}\ M$
 $\langle proof \rangle$

38.5 Intervals of integers

lemma *atLeastLessThanPlusOne-atLeastAtMost-int*: $\{l..<u+1\} = \{l..(u::int)\}$
 $\langle proof \rangle$

lemma *atLeastPlusOneAtMost-greaterThanAtMost-int*: $\{l+1..u\} = \{l<..(u::int)\}$
 $\langle proof \rangle$

lemma *atLeastPlusOneLessThan-greaterThanLessThan-int*:

$\{l+1..<u\} = \{l<..<u::int\}$
 $\langle proof \rangle$

38.5.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: $0 \leq u \implies$

$\{(0::int)..<u\} = int\ ' \ \{..<nat\ u\}$
 $\langle proof \rangle$

lemma *finite-atLeastZeroLessThan-int*: $finite\ \{(0::int)..<u\}$
 $\langle proof \rangle$

lemma *finite-atLeastLessThan-int* [iff]: $finite\ \{l..<u::int\}$
 $\langle proof \rangle$

lemma *finite-atLeastAtMost-int* [iff]: $finite\ \{l..(u::int)\}$
 $\langle proof \rangle$

lemma *finite-greaterThanAtMost-int* [iff]: $finite\ \{l<..(u::int)\}$
 $\langle proof \rangle$

lemma *finite-greaterThanLessThan-int* [iff]: $finite\ \{l<..<u::int\}$
 $\langle proof \rangle$

38.5.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: $card\ \{(0::int)..<u\} = nat\ u$
 $\langle proof \rangle$

lemma *card-atLeastLessThan-int* [simp]: $card\ \{l..<u\} = nat\ (u - l)$
 $\langle proof \rangle$

lemma *card-atLeastAtMost-int* [simp]: $card\ \{l..u\} = nat\ (u - l + 1)$

<proof>

lemma *card-greaterThanAtMost-int* [simp]: $\text{card } \{l <..u\} = \text{nat } (u - l)$
<proof>

lemma *card-greaterThanLessThan-int* [simp]: $\text{card } \{l <..\} = \text{nat } (u - (l + 1))$
<proof>

lemma *finite-M-bounded-by-nat*: $\text{finite } \{k. P\ k \wedge k < (i::\text{nat})\}$
<proof>

lemma *card-less*:
assumes *zero-in-M*: $0 \in M$
shows $\text{card } \{k \in M. k < \text{Suc } i\} \neq 0$
<proof>

lemma *card-less-Suc2*: $0 \notin M \implies \text{card } \{k. \text{Suc } k \in M \wedge k < i\} = \text{card } \{k \in M. k < \text{Suc } i\}$
<proof>

lemma *card-less-Suc*:
assumes *zero-in-M*: $0 \in M$
shows $\text{Suc } (\text{card } \{k. \text{Suc } k \in M \wedge k < i\}) = \text{card } \{k \in M. k < \text{Suc } i\}$
<proof>

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*:
 $\{l::'a::\text{linorder}\} \text{ Un } \{l<..\} = \{l..\}$
 $\{..\} \text{ Un } \{u::'a::\text{linorder}\} = \{..u\}$
 $(l::'a::\text{linorder}) < u \implies \{l\} \text{ Un } \{l<..\} = \{l..\}$
 $(l::'a::\text{linorder}) < u \implies \{l<..\} \text{ Un } \{u\} = \{l<..u\}$
 $(l::'a::\text{linorder}) \leq u \implies \{l\} \text{ Un } \{l<..u\} = \{l..u\}$
 $(l::'a::\text{linorder}) \leq u \implies \{l..\} \text{ Un } \{u\} = \{l..u\}$
<proof>

One- and two-sided intervals

lemma *ivl-disj-un-one*:
 $(l::'a::\text{linorder}) < u \implies \{..l\} \text{ Un } \{l<..\} = \{..\}$
 $(l::'a::\text{linorder}) \leq u \implies \{..\} \text{ Un } \{l..\} = \{..\}$
 $(l::'a::\text{linorder}) \leq u \implies \{..l\} \text{ Un } \{l<..u\} = \{..u\}$
 $(l::'a::\text{linorder}) \leq u \implies \{..\} \text{ Un } \{l..u\} = \{..\}$
 $(l::'a::\text{linorder}) \leq u \implies \{l<..u\} \text{ Un } \{u<..\} = \{l<..\}$

$(l::'a::linorder) < u \implies \{l<..\} \cup \{u..\} = \{l<..\}$
 $(l::'a::linorder) \leq u \implies \{l..u\} \cup \{u<..\} = \{l..\}$
 $(l::'a::linorder) \leq u \implies \{l..<u\} \cup \{u..\} = \{l..\}$
 <proof>

Two- and two-sided intervals

lemma *ivl-disj-un-two*:

$\llbracket (l::'a::linorder) < m; m \leq u \rrbracket \implies \{l<..\} \cup \{m..\} = \{l<..\}$
 $\llbracket (l::'a::linorder) \leq m; m < u \rrbracket \implies \{l<..\} \cup \{m<..\} = \{l<..\}$
 $\llbracket (l::'a::linorder) \leq m; m \leq u \rrbracket \implies \{l<..\} \cup \{m..\} = \{l<..\}$
 $\llbracket (l::'a::linorder) \leq m; m < u \rrbracket \implies \{l..\} \cup \{m<..\} = \{l..\}$
 $\llbracket (l::'a::linorder) < m; m \leq u \rrbracket \implies \{l<..\} \cup \{m..\} = \{l<..\}$
 $\llbracket (l::'a::linorder) \leq m; m \leq u \rrbracket \implies \{l<..\} \cup \{m<..\} = \{l<..\}$
 $\llbracket (l::'a::linorder) \leq m; m < u \rrbracket \implies \{l..\} \cup \{m..\} = \{l..\}$
 $\llbracket (l::'a::linorder) \leq m; m \leq u \rrbracket \implies \{l..\} \cup \{m<..\} = \{l..\}$
 <proof>

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*:

$\{..l::'a::order\} \cap \{l<..\} = \{\}$
 $\{..<l\} \cap \{l..\} = \{\}$
 $\{..l\} \cap \{l<..\} = \{\}$
 $\{..<l\} \cap \{l..\} = \{\}$
 $\{l<..\} \cap \{u<..\} = \{\}$
 $\{l<..\} \cap \{u..\} = \{\}$
 $\{l..\} \cap \{u<..\} = \{\}$
 $\{l..\} \cap \{u..\} = \{\}$
 <proof>

Two- and two-sided intervals

lemma *ivl-disj-int-two*:

$\{l::'a::order<..\} \cap \{m..\} = \{\}$
 $\{l<..\} \cap \{m<..\} = \{\}$
 $\{l..\} \cap \{m..\} = \{\}$
 $\{l..\} \cap \{m<..\} = \{\}$
 $\{l<..\} \cap \{m..\} = \{\}$
 $\{l<..\} \cap \{m<..\} = \{\}$
 $\{l..\} \cap \{m..\} = \{\}$
 $\{l..\} \cap \{m<..\} = \{\}$
 <proof>

lemmas *ivl-disj-int = ivl-disj-int-one ivl-disj-int-two*

38.6.3 Some Differences

lemma *ivl-diff* [*simp*]:

$$i \leq n \implies \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$$

<proof>

38.6.4 Some Subset Conditions

lemma *ivl-subset* [*simp,no-atp*]:

$$(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \ \& \ j \leq (n::'a::linorder))$$

<proof>

38.7 Summation indexed over intervals

syntax

$$\begin{aligned} \text{-from-to-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((SUM \text{ -} = \text{ --} \cdot \text{ -} \text{ -}) [0,0,0,10] \ 10) \\ \text{-from-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((SUM \text{ -} = \text{ --} < \text{ -} \text{ -} \text{ -}) [0,0,0,10] \\ &10) \\ \text{-upt-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((SUM \text{ -} < \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,10] \ 10) \\ \text{-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((SUM \text{ -} < = \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,10] \ 10) \end{aligned}$$

syntax (*xsymbols*)

$$\begin{aligned} \text{-from-to-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} = \text{ --} \cdot \text{ -} \text{ -} \text{ -}) [0,0,0,10] \ 10) \\ \text{-from-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} = \text{ --} < \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,0,10] \\ &10) \\ \text{-upt-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} < \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,10] \ 10) \\ \text{-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} < = \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,10] \ 10) \end{aligned}$$

syntax (*HTML output*)

$$\begin{aligned} \text{-from-to-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} = \text{ --} \cdot \text{ -} \text{ -} \text{ -}) [0,0,0,10] \ 10) \\ \text{-from-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} = \text{ --} < \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,0,10] \\ &10) \\ \text{-upt-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} < \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,10] \ 10) \\ \text{-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \ ((\mathcal{S}\Sigma \text{ -} < = \text{ -} \text{ -} \text{ -} \text{ -}) [0,0,10] \ 10) \end{aligned}$$

syntax (*latex-sum output*)

$$\begin{aligned} \text{-from-to-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \\ &((\mathcal{S}\Sigma \text{ -} = \text{ -} \text{ -}) [0,0,0,10] \ 10) \\ \text{-from-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \\ &((\mathcal{S}\Sigma \text{ -} < \text{ -} \text{ -}) [0,0,0,10] \ 10) \\ \text{-upt-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \\ &((\mathcal{S}\Sigma \text{ -} < \text{ -} \text{ -}) [0,0,10] \ 10) \\ \text{-upto-setsum} &:: \text{idt} \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b \\ &((\mathcal{S}\Sigma \text{ -} \leq \text{ -} \text{ -}) [0,0,10] \ 10) \end{aligned}$$

translations

$$\begin{aligned} \sum_{x=a..b}. t &== \text{CONST setsum } (\%x. t) \{a..b\} \\ \sum_{x=a..<b}. t &== \text{CONST setsum } (\%x. t) \{a..<b\} \\ \sum_{i \leq n}. t &== \text{CONST setsum } (\lambda i. t) \{..n\} \\ \sum_{i < n}. t &== \text{CONST setsum } (\lambda i. t) \{..<n\} \end{aligned}$$

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	L ^A T _E X
$\sum_{x \in \{a..b\}}. e$	$\sum x = a..b. e$	$\sum_x^b =_a e$
$\sum_{x \in \{a..<b\}}. e$	$\sum x = a..<b. e$	$\sum_x^{<b} =_a e$
$\sum_{x \in \{..b\}}. e$	$\sum x \leq b. e$	$\sum_x \leq_b e$
$\sum_{x \in \{..<b\}}. e$	$\sum x < b. e$	$\sum_x <_b e$

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^AT_EX 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*:

$\llbracket a = c; b = d; !!x. \llbracket c \leq x; x < d \rrbracket \implies f x = g x \rrbracket \implies$
 $\text{setsum } f \{a..<b\} = \text{setsum } g \{c..<d\}$
 $\langle \text{proof} \rangle$

lemma *setsum-atMost-Suc[simp]*: $(\sum i \leq \text{Suc } n. f i) = (\sum i \leq n. f i) + f(\text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *setsum-lessThan-Suc[simp]*: $(\sum i < \text{Suc } n. f i) = (\sum i < n. f i) + f n$
 $\langle \text{proof} \rangle$

lemma *setsum-cl-ivl-Suc[simp]*:

$\text{setsum } f \{m.. \text{Suc } n\} = (\text{if } \text{Suc } n < m \text{ then } 0 \text{ else } \text{setsum } f \{m..n\} + f(\text{Suc } n))$
 $\langle \text{proof} \rangle$

lemma *setsum-op-ivl-Suc[simp]*:

$\text{setsum } f \{m..<\text{Suc } n\} = (\text{if } n < m \text{ then } 0 \text{ else } \text{setsum } f \{m..<n\} + f(n))$
 $\langle \text{proof} \rangle$

lemma *setsum-head*:

fixes $n :: \text{nat}$
assumes $mn: m \leq n$
shows $(\sum_{x \in \{m..n\}}. P x) = P m + (\sum_{x \in \{m<..n\}}. P x)$ (**is** ?lhs = ?rhs)
 $\langle \text{proof} \rangle$

lemma *setsum-head-Suc*:

$m \leq n \implies \text{setsum } f \{m..n\} = f \, m + \text{setsum } f \{ \text{Suc } m..n \}$
 <proof>

lemma *setsum-head-upt-Suc*:

$m < n \implies \text{setsum } f \{m..<n\} = f \, m + \text{setsum } f \{ \text{Suc } m..<n \}$
 <proof>

lemma *setsum-ub-add-nat*: **assumes** $(m::\text{nat}) \leq n + 1$

shows $\text{setsum } f \{m..n + p\} = \text{setsum } f \{m..n\} + \text{setsum } f \{n + 1..n + p\}$
 <proof>

lemma *setsum-add-nat-ivl*: $\llbracket m \leq n; n \leq p \rrbracket \implies$

$\text{setsum } f \{m..<n\} + \text{setsum } f \{n..<p\} = \text{setsum } f \{m..<p::\text{nat}\}$
 <proof>

lemma *setsum-diff-nat-ivl*:

fixes $f :: \text{nat} \Rightarrow 'a::\text{ab-group-add}$

shows $\llbracket m \leq n; n \leq p \rrbracket \implies$

$\text{setsum } f \{m..<p\} - \text{setsum } f \{m..<n\} = \text{setsum } f \{n..<p\}$
 <proof>

lemma *setsum-natinterval-diff*:

fixes $f :: \text{nat} \Rightarrow ('a::\text{ab-group-add})$

shows $\text{setsum } (\lambda k. f \, k - f(k + 1)) \{ (m::\text{nat}) .. n \} =$
 $(\text{if } m \leq n \text{ then } f \, m - f(n + 1) \text{ else } 0)$

<proof>

lemmas *setsum-restrict-set'* = *setsum-restrict-set*[*unfolded Int-def*]

lemma *setsum-setsum-restrict*:

$\text{finite } S \implies \text{finite } T \implies \text{setsum } (\lambda x. \text{setsum } (\lambda y. f \, x \, y) \{y. y \in T \wedge R \, x \, y\}) \, S$
 $= \text{setsum } (\lambda y. \text{setsum } (\lambda x. f \, x \, y) \{x. x \in S \wedge R \, x \, y\}) \, T$
 <proof>

lemma *setsum-image-gen*: **assumes** $fS: \text{finite } S$

shows $\text{setsum } g \, S = \text{setsum } (\lambda y. \text{setsum } g \{x. x \in S \wedge f \, x = y\}) \, (f \, 'S)$
 <proof>

lemma *setsum-le-included*:

fixes $f :: 'a \Rightarrow 'b::\text{ordered-comm-monoid-add}$

assumes $\text{finite } s \, \text{finite } t$

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 $\text{setsum } f \, s \leq \text{setsum } g \, t$

<proof>

lemma *setsum-multicount-gen*:

assumes $\text{finite } s \, \text{finite } t \, \forall j \in t. (\text{card } \{i \in s. R \, i \, j\} = k \, j)$

shows $\text{setsum } (\lambda i. (\text{card } \{j \in t. R \, i \, j\})) \, s = \text{setsum } k \, t \, (\text{is } ?l = ?r)$

$\langle \text{proof} \rangle$

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)

$\langle \text{proof} \rangle$

38.8 Shifting bounds

lemma *setsum-shift-bounds-nat-ivl*:

$\text{setsum } f\ \{m+k..<n+k\} = \text{setsum } (\%i. f(i+k))\{m..<n::\text{nat}\}$

$\langle \text{proof} \rangle$

lemma *setsum-shift-bounds-cl-nat-ivl*:

$\text{setsum } f\ \{m+k..n+k\} = \text{setsum } (\%i. f(i+k))\{m..n::\text{nat}\}$

$\langle \text{proof} \rangle$

corollary *setsum-shift-bounds-cl-Suc-ivl*:

$\text{setsum } f\ \{\text{Suc } m..\text{Suc } n\} = \text{setsum } (\%i. f(\text{Suc } i))\{m..n\}$

$\langle \text{proof} \rangle$

corollary *setsum-shift-bounds-Suc-ivl*:

$\text{setsum } f\ \{\text{Suc } m..<\text{Suc } n\} = \text{setsum } (\%i. f(\text{Suc } i))\{m..<n\}$

$\langle \text{proof} \rangle$

lemma *setsum-shift-lb-Suc0-0*:

$f(0::\text{nat}) = (0::\text{nat}) \implies \text{setsum } f\ \{\text{Suc } 0..k\} = \text{setsum } f\ \{0..k\}$

$\langle \text{proof} \rangle$

lemma *setsum-shift-lb-Suc0-0-upt*:

$f(0::\text{nat}) = 0 \implies \text{setsum } f\ \{\text{Suc } 0..<k\} = \text{setsum } f\ \{0..<k\}$

$\langle \text{proof} \rangle$

38.9 The formula for geometric sums

lemma *geometric-sum*:

assumes $x \neq 1$

shows $(\sum i=0..<n. x^i) = (x^n - 1) / (x - 1::'a::\text{field})$

$\langle \text{proof} \rangle$

38.10 The formula for arithmetic sums

lemma *gauss-sum*:

$((1::'a::\text{comm-semiring-1}) + 1) * (\sum i \in \{1..n\}. \text{of-nat } i) =$
 $\text{of-nat } n * ((\text{of-nat } n) + 1)$

$\langle \text{proof} \rangle$

theorem *arith-series-general*:

$((1::'a::\text{comm-semiring-1}) + 1) * (\sum i \in \{..<n\}. a + \text{of-nat } i * d) =$
 $\text{of-nat } n * (a + (a + \text{of-nat } (n-1) * d))$

$\langle proof \rangle$

lemma *arith-series-nat*:

$Suc (Suc\ 0) * (\sum i \in \{..<n\}. a + i * d) = n * (a + (a + (n - 1) * d))$
 $\langle proof \rangle$

lemma *arith-series-int*:

$(2::int) * (\sum i \in \{..<n\}. a + of\text{-}nat\ i * d) =$
 $of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1) * d))$
 $\langle proof \rangle$

lemma *sum-diff-distrib*:

fixes $P::nat \Rightarrow nat$
shows
 $\forall x. Q\ x \leq P\ x \implies$
 $(\sum x < n. P\ x) - (\sum x < n. Q\ x) = (\sum x < n. P\ x - Q\ x)$
 $\langle proof \rangle$

38.11 Products indexed over intervals

syntax

$\text{-from-to-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((PROD\ - = \dots / -)\ [0,0,0,10]\ 10)$
 $\text{-from-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((PROD\ - = \dots < / -)\ [0,0,0,10]\ 10)$
 $\text{-upt-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((PROD\ - < / -)\ [0,0,10]\ 10)$
 $\text{-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((PROD\ - < = / -)\ [0,0,10]\ 10)$

syntax (*xsymbols*)

$\text{-from-to-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - = \dots / -)\ [0,0,0,10]\ 10)$
 $\text{-from-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - = \dots < / -)\ [0,0,0,10]\ 10)$
 $\text{-upt-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - < / -)\ [0,0,10]\ 10)$
 $\text{-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - \leq / -)\ [0,0,10]\ 10)$

syntax (*HTML output*)

$\text{-from-to-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - = \dots / -)\ [0,0,0,10]\ 10)$
 $\text{-from-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - = \dots < / -)\ [0,0,0,10]\ 10)$
 $\text{-upt-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - < / -)\ [0,0,10]\ 10)$
 $\text{-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b\ ((\mathcal{P}\prod\ - \leq / -)\ [0,0,10]\ 10)$

syntax (*latex-prod output*)

$\text{-from-to-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\mathcal{P}\prod\ - = \dots)\ [0,0,0,10]\ 10)$
 $\text{-from-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\mathcal{P}\prod\ - < \dots)\ [0,0,0,10]\ 10)$
 $\text{-upt-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\mathcal{P}\prod\ - < \dots)\ [0,0,10]\ 10)$
 $\text{-upto-setprod} :: idt \Rightarrow 'a \Rightarrow 'b \Rightarrow 'b$
 $((\mathcal{P}\prod\ - \leq \dots)\ [0,0,10]\ 10)$

translations

$$\prod x=a..b. t == \text{CONST setprod } (\%x. t) \{a..b\}$$

$$\prod x=a..<b. t == \text{CONST setprod } (\%x. t) \{a..<b\}$$

$$\prod i \leq n. t == \text{CONST setprod } (\lambda i. t) \{..n\}$$

$$\prod i < n. t == \text{CONST setprod } (\lambda i. t) \{..<n\}$$

38.12 Transfer setup

lemma *transfer-nat-int-set-functions:*

$$\{..n\} = \text{nat} \text{ ‘ } \{0..int\ n\}$$

$$\{m..n\} = \text{nat} \text{ ‘ } \{int\ m..int\ n\}$$

<proof>

lemma *transfer-nat-int-set-function-closures:*

$$x \geq 0 \implies \text{nat-set } \{x..y\}$$

<proof>

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:*

$$is\text{-}nat\ m \implies is\text{-}nat\ n \implies \{m..n\} = int \text{ ‘ } \{nat\ m..nat\ n\}$$

<proof>

lemma *transfer-int-nat-set-function-closures:*

$$is\text{-}nat\ x \implies \text{nat-set } \{x..y\}$$

<proof>

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*:

$$\begin{aligned}
& \llbracket \exists (z :: 'a::\text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x < z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x) \\
& \llbracket \exists (z :: 'a::\text{linorder}). \forall x < z. P\ x = P'\ x; \exists z. \forall x < z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x < z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x) \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x = t) = \text{False} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x \neq t) = \text{True} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x < t) = \text{True} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x \leq t) = \text{True} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x > t) = \text{False} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x < z. (x \geq t) = \text{False} \\
& \exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x < z. (d\ \text{dvd}\ x + s) = (d\ \text{dvd}\ x + s) \\
& \exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x < z. (\neg d\ \text{dvd}\ x + s) = (\neg d\ \text{dvd}\ x + s) \\
& \exists z. \forall x < z. F = F \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *pinf*:

$$\begin{aligned}
& \llbracket \exists (z :: 'a::\text{linorder}). \forall x > z. P\ x = P'\ x; \exists z. \forall x > z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x > z. (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x) \\
& \llbracket \exists (z :: 'a::\text{linorder}). \forall x > z. P\ x = P'\ x; \exists z. \forall x > z. Q\ x = Q'\ x \rrbracket \\
& \implies \exists z. \forall x > z. (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x) \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x = t) = \text{False} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \neq t) = \text{True} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x < t) = \text{False} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \leq t) = \text{False} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x > t) = \text{True} \\
& \exists (z :: 'a::\{\text{linorder}\}). \forall x > z. (x \geq t) = \text{True} \\
& \exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x > z. (d\ \text{dvd}\ x + s) = (d\ \text{dvd}\ x + s) \\
& \exists z. \forall (x :: 'a::\{\text{linorder}, \text{plus}, \text{Rings.dvd}\}). x > z. (\neg d\ \text{dvd}\ x + s) = (\neg d\ \text{dvd}\ x + s) \\
& \exists z. \forall x > z. F = F \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *inf-period*:

$$\begin{aligned}
& \llbracket \forall x\ k. P\ x = P\ (x - k*D); \forall x\ k. Q\ x = Q\ (x - k*D) \rrbracket \\
& \implies \forall x\ k. (P\ x \wedge Q\ x) = (P\ (x - k*D) \wedge Q\ (x - k*D)) \\
& \llbracket \forall x\ k. P\ x = P\ (x - k*D); \forall x\ k. Q\ x = Q\ (x - k*D) \rrbracket \\
& \implies \forall x\ k. (P\ x \vee Q\ x) = (P\ (x - k*D) \vee Q\ (x - k*D)) \\
& (d :: 'a::\{\text{comm-ring}, \text{Rings.dvd}\})\ \text{dvd}\ D \implies \forall x\ k. (d\ \text{dvd}\ x + t) = (d\ \text{dvd}\ (x - k*D) + t) \\
& (d :: 'a::\{\text{comm-ring}, \text{Rings.dvd}\})\ \text{dvd}\ D \implies \forall x\ k. (\neg d\ \text{dvd}\ x + t) = (\neg d\ \text{dvd}\ (x - k*D) + t) \\
& \forall x\ k. F = F \\
& \langle \text{proof} \rangle
\end{aligned}$$

39.2 The A and B sets

lemma *bset*:

$$\llbracket \forall x. (\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D) \rrbracket ;$$

$$\begin{aligned}
& \forall x.(\forall j \in \{1 \dots D\}. \forall b \in B. x \neq b + j) \longrightarrow Q x \longrightarrow Q(x - D) \Longrightarrow \\
& \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::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::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::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::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::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::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::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 \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma aset:

$$\begin{aligned}
& \llbracket \forall x.(\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow P x \longrightarrow P(x + D) ; \\
& \quad \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) ; \\
& \quad \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))
\end{aligned}$$

$d \text{ dvd } D \implies (\forall (x::int). (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow (\neg d \text{ dvd } x + t) \longrightarrow$
 $(\neg d \text{ dvd } (x + D) + t))$
 $\forall x. (\forall j \in \{1 \dots D\}. \forall b \in A. x \neq b - j) \longrightarrow F \longrightarrow F$
 $\langle \text{proof} \rangle$

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*)

$\langle \text{proof} \rangle$

39.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d::int) \implies x - (abs(x-z)+1) * d < z$

$\langle \text{proof} \rangle$

lemma *incr-lemma*: $0 < (d::int) \implies z < x + (abs(x-z)+1) * d$

$\langle \text{proof} \rangle$

lemma *decr-mult-lemma*:

assumes *dpos*: $(0::int) < d$ **and** *minus*: $\forall x. P\ x \longrightarrow P(x - d)$ **and** *kneg*: $0 \leq k$

shows $ALL\ x. P\ x \longrightarrow P(x - k*d)$

$\langle \text{proof} \rangle$

lemma *minusinfinity*:

assumes *dpos*: $0 < d$ **and**

P1eqP1: $ALL\ x\ k. P1\ x = P1(x - k*d)$ **and** *ePeqP1*: $EX\ z::int. ALL\ x. x < z \longrightarrow (P\ x = P1\ x)$

shows $(EX\ x. P1\ x) \longrightarrow (EX\ x. P\ x)$

$\langle \text{proof} \rangle$

lemma *cpmi*:

assumes *dp*: $0 < D$ **and** *p1*: $\exists z. \forall x < z. P\ x = P'\ x$

and *nb*: $\forall x. (\forall j \in \{1..D\}. \forall (b::int) \in B. x \neq b+j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$

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*))

$\langle \text{proof} \rangle$

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 $(\exists x. P' x) \longrightarrow (\exists x. P x)$
 $\langle proof \rangle$

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 $ALL x. P x \longrightarrow P(x + k*d)$
 $\langle proof \rangle$

lemma *cppl*:

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))
 $\langle proof \rangle$

lemma *simp-from-to*: $\{i..j::int\} = (\text{if } j < i \text{ then } \{\} \text{ else insert } i \{i+1..j\})$
 $\langle proof \rangle$

theorem *unity-coeff-ex*: $(\exists (x::'a::\{\text{semiring-0}, \text{Rings.dvd}\}). P(l * x)) \equiv (\exists x. l \text{ dvd } (x + 0) \wedge P x)$
 $\langle proof \rangle$

lemma *zdvd-mono*: **assumes** *not0*: $(k::int) \neq 0$
shows $((m::int) \text{ dvd } t) \equiv (k*m \text{ dvd } k*t)$
 $\langle proof \rangle$

lemma *uminus-dvd-conv*: $(d \text{ dvd } (t::int)) \equiv (-d \text{ dvd } t) \wedge (d \text{ dvd } (t::int)) \equiv (d \text{ dvd } -t)$
 $\langle proof \rangle$

Theorems for transforming predicates on nat to predicates on *int*

lemma *zdiff-int-split*: $P(\text{int}(x - y)) = ((y \leq x \longrightarrow P(\text{int } x - \text{int } y)) \wedge (x < y \longrightarrow P 0))$
 $\langle proof \rangle$

lemma *number-of1*: $(0::int) \leq \text{number-of } n \implies (0::int) \leq \text{number-of } (\text{Int.Bit0 } n) \wedge (0::int) \leq \text{number-of } (\text{Int.Bit1 } n)$
 $\langle proof \rangle$

lemma *number-of2*: $(0::int) \leq \text{Numeral0} \langle proof \rangle$

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*: $(0 \leq x \implies P = P') \implies (0 \leq (x::int) \longrightarrow P) = (0 \leq x \longrightarrow P') \langle proof \rangle$

theorem *conj-le-cong*: $(0 \leq x \implies P = P') \implies (0 \leq (x::int) \wedge P) = (0 \leq x \wedge P')$
 <proof>

<ML>

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*]: $(a::int) \text{ div } 0 = 0$ **and** [*presburger*]: $a \text{ mod } 0 = a$
 <proof>

lemma [*presburger, algebra*]: $m \text{ mod } 2 = (1::nat) \longleftrightarrow \neg 2 \text{ dvd } m$ <proof>
lemma [*presburger, algebra*]: $m \text{ mod } 2 = \text{Suc } 0 \longleftrightarrow \neg 2 \text{ dvd } m$ <proof>
lemma [*presburger, algebra*]: $m \text{ mod } (\text{Suc } (\text{Suc } 0)) = (1::nat) \longleftrightarrow \neg 2 \text{ dvd } m$
 <proof>
lemma [*presburger, algebra*]: $m \text{ mod } (\text{Suc } (\text{Suc } 0)) = \text{Suc } 0 \longleftrightarrow \neg 2 \text{ dvd } m$
 <proof>
lemma [*presburger, algebra*]: $m \text{ mod } 2 = (1::int) \longleftrightarrow \neg 2 \text{ dvd } m$ <proof>

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 $Eps :: ('a \Rightarrow bool) \Rightarrow 'a$ **where**
 $someI: P\ x \Rightarrow P\ (Eps\ P)$

syntax (*epsilon*)

$-Eps :: [pttrn, bool] \Rightarrow 'a \quad ((\exists \epsilon \text{ -./ -}) [0, 10] 10)$

syntax (*HOL*)

$-Eps :: [pttrn, bool] \Rightarrow 'a \quad ((\exists @ \text{ -./ -}) [0, 10] 10)$

syntax

$-Eps :: [pttrn, bool] \Rightarrow 'a \quad ((\exists SOME \text{ -./ -}) [0, 10] 10)$

translations

$SOME\ x.\ P == CONST\ Eps\ (\%x.\ P)$

$\langle ML \rangle$

definition $inv\text{-}into :: 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**
 $inv\text{-}into\ A\ f == \%x.\ SOME\ y.\ y : A \ \&\ f\ y = x$

abbreviation $inv :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$ **where**
 $inv == inv\text{-}into\ UNIV$

40.2 Hilbert’s Epsilon-operator

Easier to apply than $someI$ if the witness comes from an existential formula

lemma $someI\text{-}ex\ [elim?]: \exists x.\ P\ x \Rightarrow P\ (SOME\ x.\ P\ x)$
 $\langle proof \rangle$

Easier to apply than $someI$ because the conclusion has only one occurrence of P .

lemma $someI2: [| P\ a; !!x.\ P\ x \Rightarrow Q\ x |] \Rightarrow Q\ (SOME\ x.\ P\ x)$
 $\langle proof \rangle$

Easier to apply than $someI2$ if the witness comes from an existential formula

lemma $someI2\text{-}ex: [| \exists a.\ P\ a; !!x.\ P\ x \Rightarrow Q\ x |] \Rightarrow Q\ (SOME\ x.\ P\ x)$
 $\langle proof \rangle$

lemma $some\text{-}equality\ [intro]:$

$[| P\ a; !!x.\ P\ x \Rightarrow x=a |] \Rightarrow (SOME\ x.\ P\ x) = a$

$\langle proof \rangle$

lemma $someI\text{-}equality: [| EX!x.\ P\ x; P\ a |] \Rightarrow (SOME\ x.\ P\ x) = a$
 $\langle proof \rangle$

lemma $some\text{-}eq\text{-}ex: P\ (SOME\ x.\ P\ x) = (\exists x.\ P\ x)$
 $\langle proof \rangle$

lemma $some\text{-}eq\text{-}trivial\ [simp]: (SOME\ y.\ y=x) = x$
 $\langle proof \rangle$

lemma *some-sym-eq-trivial* [simp]: $(\text{SOME } y. x=y) = x$
 $\langle \text{proof} \rangle$

40.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: $\forall x. \exists y. Q\ x\ y \implies \exists f. \forall x. Q\ x\ (f\ x)$
 $\langle \text{proof} \rangle$

lemma *bchoice*: $\forall x \in S. \exists y. Q\ x\ y \implies \exists f. \forall x \in S. Q\ x\ (f\ x)$
 $\langle \text{proof} \rangle$

40.4 Function Inverse

lemma *inv-def*: $\text{inv } f = (\%y. \text{SOME } x. f\ x = y)$
 $\langle \text{proof} \rangle$

lemma *inv-into-into*: $x : f^{-1} A \implies \text{inv-into } A\ f\ x : A$
 $\langle \text{proof} \rangle$

lemma *inv-id* [simp]: $\text{inv } \text{id} = \text{id}$
 $\langle \text{proof} \rangle$

lemma *inv-into-f-f* [simp]:
 $[\text{inj-on } f\ A; x : A] \implies \text{inv-into } A\ f\ (f\ x) = x$
 $\langle \text{proof} \rangle$

lemma *inv-f-f*: $\text{inj } f \implies \text{inv } f\ (f\ x) = x$
 $\langle \text{proof} \rangle$

lemma *f-inv-into-f*: $y : f^{-1} A \implies f\ (\text{inv-into } A\ f\ y) = y$
 $\langle \text{proof} \rangle$

lemma *inv-into-f-eq*: $[\text{inj-on } f\ A; x : A; f\ x = y] \implies \text{inv-into } A\ f\ y = x$
 $\langle \text{proof} \rangle$

lemma *inv-f-eq*: $[\text{inj } f; f\ x = y] \implies \text{inv } f\ y = x$
 $\langle \text{proof} \rangle$

lemma *inj-imp-inv-eq*: $[\text{inj } f; \text{ALL } x. f(g\ x) = x] \implies \text{inv } f = g$
 $\langle \text{proof} \rangle$

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 $P\ x$

$\langle \text{proof} \rangle$

lemma *inj-iff*: $(\text{inj } f) = (\text{inv } f \circ f = \text{id})$
 $\langle \text{proof} \rangle$

lemma *inv-o-cancel[simp]*: $\text{inj } f \implies \text{inv } f \circ f = \text{id}$
 $\langle \text{proof} \rangle$

lemma *o-inv-o-cancel[simp]*: $\text{inj } f \implies g \circ \text{inv } f \circ f = g$
 $\langle \text{proof} \rangle$

lemma *inv-into-image-cancel[simp]*:
 $\text{inj-on } f \ A \implies S \leq A \implies \text{inv-into } A \ f \ ' \ f \ ' \ S = S$
 $\langle \text{proof} \rangle$

lemma *inj-imp-surj-inv*: $\text{inj } f \implies \text{surj } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *surj-f-inv-f*: $\text{surj } f \implies f(\text{inv } f \ y) = y$
 $\langle \text{proof} \rangle$

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 $x=y$
 $\langle \text{proof} \rangle$

lemma *inj-on-inv-into*: $B \leq f \ ' \ A \implies \text{inj-on } (\text{inv-into } A \ f) \ B$
 $\langle \text{proof} \rangle$

lemma *bij-betw-inv-into*: $\text{bij-betw } f \ A \ B \implies \text{bij-betw } (\text{inv-into } A \ f) \ B \ A$
 $\langle \text{proof} \rangle$

lemma *surj-imp-inj-inv*: $\text{surj } f \implies \text{inj } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *surj-iff*: $(\text{surj } f) = (f \circ \text{inv } f = \text{id})$
 $\langle \text{proof} \rangle$

lemma *surj-imp-inv-eq*: $[\text{surj } f; \forall x. g(f \ x) = x] \implies \text{inv } f = g$
 $\langle \text{proof} \rangle$

lemma *bij-imp-bij-inv*: $\text{bij } f \implies \text{bij } (\text{inv } f)$
 $\langle \text{proof} \rangle$

lemma *inv-equality*: $[\forall x. g(f \ x) = x; \forall y. f(g \ y) = y] \implies \text{inv } f = g$
 $\langle \text{proof} \rangle$

lemma *inv-inv-eq*: $\text{bij } f \implies \text{inv } (\text{inv } f) = f$
 $\langle \text{proof} \rangle$

lemma *inv-into-comp*:
 $\llbracket \text{inj-on } f \text{ } (g \text{ } ^\circ A); \text{inj-on } g \text{ } A; x : f \text{ } ^\circ g \text{ } ^\circ A \rrbracket \implies$
 $\text{inv-into } A \text{ } (f \text{ } o \text{ } g) \text{ } x = (\text{inv-into } A \text{ } g \text{ } o \text{ } \text{inv-into } (g \text{ } ^\circ A) \text{ } f) \text{ } x$
 $\langle \text{proof} \rangle$

lemma *o-inv-distrib*: $\llbracket \text{bij } f; \text{bij } g \rrbracket \implies \text{inv } (f \text{ } o \text{ } g) = \text{inv } g \text{ } o \text{ } \text{inv } f$
 $\langle \text{proof} \rangle$

lemma *image-surj-f-inv-f*: $\text{surj } f \implies f \text{ } ^\circ (\text{inv } f \text{ } ^\circ A) = A$
 $\langle \text{proof} \rangle$

lemma *image-inv-f-f*: $\text{inj } f \implies (\text{inv } f) \text{ } ^\circ (f \text{ } ^\circ A) = A$
 $\langle \text{proof} \rangle$

lemma *inv-image-comp*: $\text{inj } f \implies \text{inv } f \text{ } ^\circ (f \text{ } ^\circ X) = X$
 $\langle \text{proof} \rangle$

lemma *bij-image-Collect-eq*: $\text{bij } f \implies f \text{ } ^\circ \text{Collect } P = \{y. P \text{ } (\text{inv } f \text{ } y)\}$
 $\langle \text{proof} \rangle$

lemma *bij-vimage-eq-inv-image*: $\text{bij } f \implies f \text{ } -^\circ A = \text{inv } f \text{ } ^\circ A$
 $\langle \text{proof} \rangle$

lemma *finite-fun-UNIVD1*:
assumes *fin*: $\text{finite } (\text{UNIV} :: ('a \Rightarrow 'b) \text{ set})$
and *card*: $\text{card } (\text{UNIV} :: 'b \text{ set}) \neq \text{Suc } 0$
shows $\text{finite } (\text{UNIV} :: 'a \text{ set})$
 $\langle \text{proof} \rangle$

40.5 Other Consequences of Hilbert’s Epsilon

Hilbert’s Epsilon and the *split* Operator

Looping simprule

lemma *split-paired-Eps*: $(\text{SOME } x. P \text{ } x) = (\text{SOME } (a,b). P(a,b))$
 $\langle \text{proof} \rangle$

lemma *Eps-split*: $\text{Eps } (\text{split } P) = (\text{SOME } xy. P \text{ } (\text{fst } xy) \text{ } (\text{snd } xy))$
 $\langle \text{proof} \rangle$

lemma *Eps-split-eq [simp]*: $(@ (x',y'). x = x' \ \& \ y = y') = (x,y)$
 $\langle \text{proof} \rangle$

A relation is wellfounded iff it has no infinite descending chain

lemma *wf-iff-no-infinite-down-chain*:
 $wf\ r = (\sim(\exists f. \forall i. (f(Suc\ i), f\ i) \in r))$
 $\langle proof \rangle$

lemma *wf-no-infinite-down-chainE*:
assumes $wf\ r$ **obtains** k **where** $(f\ (Suc\ k), f\ k) \notin r$
 $\langle proof \rangle$

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P\ x. P\ x \dashrightarrow P\ (Eps\ P)$
 $\langle proof \rangle$

40.6 Least value operator

definition
 $LeastM :: [a \Rightarrow 'b::ord, 'a \Rightarrow bool] \Rightarrow 'a$ **where**
 $LeastM\ m\ P == SOME\ x. P\ x \ \&\ (\forall y. P\ y \dashrightarrow m\ x \leq m\ y)$

syntax
 $-LeastM :: [pttrn, 'a \Rightarrow 'b::ord, bool] \Rightarrow 'a$ (*LEAST* - *WRT* \cdot - $[0, 4, 10]$
 $10)$

translations
 $LEAST\ x\ WRT\ m. P == CONST\ LeastM\ m\ (\%x. P)$

lemma *LeastMI2*:
 $P\ x \implies (!y. P\ y \implies m\ x \leq m\ y)$
 $\implies (!x. P\ x \implies \forall y. P\ y \dashrightarrow m\ x \leq m\ y \implies Q\ x)$
 $\implies Q\ (LeastM\ m\ P)$
 $\langle proof \rangle$

lemma *LeastM-equality*:
 $P\ k \implies (!x. P\ x \implies m\ k \leq m\ x)$
 $\implies m\ (LEAST\ x\ WRT\ m. P\ x) = (m\ k::'a::order)$
 $\langle proof \rangle$

lemma *wf-linord-ex-has-least*:
 $wf\ r \implies \forall x\ y. ((x,y):r^+) = ((y,x):r^*) \implies P\ k$
 $\implies \exists x. P\ x \ \&\ (!y. P\ y \dashrightarrow (m\ x, m\ y):r^*)$
 $\langle proof \rangle$

lemma *ex-has-least-nat*:
 $P\ k \implies \exists x. P\ x \ \&\ (\forall y. P\ y \dashrightarrow m\ x \leq (m\ y::nat))$
 $\langle proof \rangle$

lemma *LeastM-nat-lemma*:
 $P\ k \implies P\ (LeastM\ m\ P) \ \&\ (\forall y. P\ y \dashrightarrow m\ (LeastM\ m\ P) \leq (m\ y::nat))$
 $\langle proof \rangle$

lemmas $LeastM\ natI = LeastM\ nat\ lemma\ [THEN\ conjunct1, standard]$

lemma *LeastM-nat-le*: $P\ x \implies m\ (LeastM\ m\ P) \leq (m\ x::nat)$
 $\langle proof \rangle$

40.7 Greatest value operator

definition

$GreatestM :: [a \Rightarrow b::ord, a \Rightarrow bool] \Rightarrow a$ **where**
 $GreatestM\ m\ P == SOME\ x.\ P\ x \ \&\ (\forall y.\ P\ y \longrightarrow m\ y \leq m\ x)$

definition

$Greatest :: (a::ord \Rightarrow bool) \Rightarrow a$ (**binder** *GREATEST* 10) **where**
 $Greatest == GreatestM\ (\%x.\ x)$

syntax

$-GreatestM :: [pttrn, a \Rightarrow b::ord, bool] \Rightarrow a$
 $(GREATEST - WRT\ -. - [0, 4, 10]\ 10)$

translations

$GREATEST\ x\ WRT\ m.\ P == CONST\ GreatestM\ m\ (\%x.\ P)$

lemma *GreatestMI2*:

$P\ x \implies (!y.\ P\ y \implies m\ y \leq m\ x)$
 $\implies (!x.\ P\ x \implies \forall y.\ P\ y \longrightarrow m\ y \leq m\ x \implies Q\ x)$
 $\implies Q\ (GreatestM\ m\ P)$
 $\langle proof \rangle$

lemma *GreatestM-equality*:

$P\ k \implies (!x.\ P\ x \implies m\ x \leq m\ k)$
 $\implies m\ (GREATEST\ x\ WRT\ m.\ P\ x) = (m\ k::'a::order)$
 $\langle proof \rangle$

lemma *Greatest-equality*:

$P\ (k::'a::order) \implies (!x.\ P\ x \implies x \leq k) \implies (GREATEST\ x.\ P\ x) = k$
 $\langle proof \rangle$

lemma *ex-has-greatest-nat-lemma*:

$P\ k \implies \forall x.\ P\ x \longrightarrow (\exists y.\ P\ y \ \&\ \sim ((m\ y::nat) \leq m\ x))$
 $\implies \exists y.\ P\ y \ \&\ \sim (m\ y < m\ k + n)$
 $\langle proof \rangle$

lemma *ex-has-greatest-nat*:

$P\ k \implies \forall y.\ P\ y \longrightarrow m\ y < b$
 $\implies \exists x.\ P\ x \ \&\ (\forall y.\ P\ y \longrightarrow (m\ y::nat) \leq m\ x)$
 $\langle proof \rangle$

lemma *GreatestM-nat-lemma*:

$P\ k \implies \forall y.\ P\ y \longrightarrow m\ y < b$
 $\implies P\ (GreatestM\ m\ P) \ \&\ (\forall y.\ P\ y \longrightarrow (m\ y::nat) \leq m\ (GreatestM\ m\ P))$

$\langle \text{proof} \rangle$

lemmas *GreatestM-natI* = *GreatestM-nat-lemma* [*THEN conjunct1, standard*]

lemma *GreatestM-nat-le*:

$P\ x ==> \forall y. P\ y \longrightarrow m\ y < b$
 $==> (m\ x::nat) <= m\ (GreatestM\ m\ P)$
 $\langle \text{proof} \rangle$

Specialization to *GREATEST*.

lemma *GreatestI*: $P\ (k::nat) ==> \forall y. P\ y \longrightarrow y < b ==> P\ (GREATEST\ x. P\ x)$
 $\langle \text{proof} \rangle$

lemma *Greatest-le*:

$P\ x ==> \forall y. P\ y \longrightarrow y < b ==> (x::nat) <= (GREATEST\ x. P\ x)$
 $\langle \text{proof} \rangle$

40.8 The Meson proof procedure

40.8.1 Negation Normal Form

de Morgan laws

lemma *meson-not-conjD*: $\sim(P \& Q) ==> \sim P \mid \sim Q$
and *meson-not-disjD*: $\sim(P \mid Q) ==> \sim P \& \sim Q$
and *meson-not-notD*: $\sim\sim P ==> P$
and *meson-not-allD*: $!!P. \sim(\forall x. P(x)) ==> \exists x. \sim P(x)$
and *meson-not-exD*: $!!P. \sim(\exists x. P(x)) ==> \forall x. \sim P(x)$
 $\langle \text{proof} \rangle$

Removal of \longrightarrow and \longleftrightarrow (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P \longrightarrow Q ==> \sim P \mid Q$
and *meson-not-impD*: $\sim(P \longrightarrow Q) ==> P \& \sim Q$
and *meson-iff-to-disjD*: $P = Q ==> (\sim P \mid Q) \& (\sim Q \mid P)$
and *meson-not-iffD*: $\sim(P = Q) ==> (P \mid Q) \& (\sim P \mid \sim Q)$
 — Much more efficient than $P \wedge \neg Q \vee Q \wedge \neg P$ for computing CNF
and *meson-not-refl-disj-D*: $x \sim = x \mid P ==> P$
 $\langle \text{proof} \rangle$

40.8.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: $!!P\ Q. (\exists x. P(x)) \& Q ==> \exists x. P(x) \& Q$
and *meson-conj-exD2*: $!!P\ Q. P \& (\exists x. Q(x)) ==> \exists x. P \& Q(x)$
 $\langle \text{proof} \rangle$

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*: $!!P \ Q. P \mid (\exists x. Q(x)) \implies \exists x. P \mid Q(x)$
 $\langle proof \rangle$

40.8.3 Generating clauses for the Meson Proof Procedure

Disjunctions

lemma *meson-disj-assoc*: $(P \mid Q) \mid R \implies P \mid (Q \mid R)$
and *meson-disj-comm*: $P \mid Q \implies Q \mid P$
and *meson-disj-FalseD1*: $False \mid P \implies P$
and *meson-disj-FalseD2*: $P \mid False \implies P$
 $\langle proof \rangle$

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*: $\sim P \mid Q \implies ((\sim P \implies P) \implies Q)$
 $\langle proof \rangle$

Version for Plaisted’s “Positive refinement” of the Meson procedure

lemma *make-refined-neg-rule*: $\sim P \mid Q \implies (P \implies Q)$
 $\langle proof \rangle$

P should be a literal

lemma *make-pos-rule*: $P \mid Q \implies ((P \implies \sim P) \implies Q)$
 $\langle proof \rangle$

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'*: $[P \mid Q; \sim P] \implies Q$
 $\langle proof \rangle$

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $\sim P \implies ((\sim P \implies P) \implies False)$
 $\langle proof \rangle$

lemma *make-pos-goal*: $P \implies ((P \implies \sim P) \implies False)$
 $\langle proof \rangle$

40.9.1 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma *conj-forward*: $[[P' \& Q'; P' ==> P; Q' ==> Q]] ==> P \& Q$
 $\langle proof \rangle$

lemma *disj-forward*: $[[P' | Q'; P' ==> P; Q' ==> Q]] ==> P | Q$
 $\langle proof \rangle$

lemma *disj-forward2*:
 $[[P' | Q'; P' ==> P; [[Q'; P ==> False]] ==> Q]] ==> P | Q$
 $\langle proof \rangle$

lemma *all-forward*: $[[\forall x. P'(x); !!x. P'(x) ==> P(x)]] ==> \forall x. P(x)$
 $\langle proof \rangle$

lemma *ex-forward*: $[[\exists x. P'(x); !!x. P'(x) ==> P(x)]] ==> \exists x. P(x)$
 $\langle proof \rangle$

40.10 Meson package

$\langle ML \rangle$

40.11 Specification package – Hilbertized version

lemma *exE-some*: $[[Ex P ; c == Eps P]] ==> P c$
 $\langle proof \rangle$

$\langle ML \rangle$

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*

<proof>

lemma *equal-imp-fequal* [no-atp]: *X* = *Y* \Longrightarrow *fequal* *X* *Y*

<proof>

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*

<proof>

lemma *iff-negative*: \neg *P* \vee \neg *Q* \vee *P* = *Q*

<proof>

Theorems for translation to combinators

lemma *abs-S* [no-atp]: $\lambda x. (f\ x) (g\ x) \equiv \text{COMBS } f\ g$

<proof>

lemma *abs-I* [no-atp]: $\lambda x. x \equiv \text{COMBI}$

<proof>

lemma *abs-K* [no-atp]: $\lambda x. y \equiv \text{COMBK } y$

<proof>

lemma *abs-B* [*no-atp*]: $\lambda x. a (g x) \equiv COMBB a g$
 $\langle proof \rangle$

lemma *abs-C* [*no-atp*]: $\lambda x. (f x) b \equiv COMBC f b$
 $\langle proof \rangle$

41.1 Setup of external ATPs

$\langle ML \rangle$

41.2 The MESON prover

$\langle ML \rangle$

41.3 The Metis prover

$\langle ML \rangle$

end

42 Recdef: TFL: recursive function definitions

theory *Recdef*

imports *FunDef Plain*

uses

(*Tools/TFL/casesplit.ML*)

(*Tools/TFL/utls.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 :: $('a * 'a) \text{ set} \Rightarrow (('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow \text{bool}$

for *R* :: $('a * 'a) \text{ set}$

and *F* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$

where

wfrecI: $ALL z. (z, x) : R \longrightarrow wfrec-rel R F z (g z) \implies$

$wfrec-rel R F x (F g x)$

definition

cut :: $('a \Rightarrow 'b) \Rightarrow ('a * 'a) \text{ set} \Rightarrow 'a \Rightarrow 'a \Rightarrow 'b$ **where**

$cut\ f\ r\ x == (\%y. \text{ if } (y,x):r \text{ then } f\ y \text{ else undefined})$

definition

$adm\text{-}wf :: ('a * 'a) \text{ set} ==> (('a ==> 'b) ==> 'a ==> 'b) ==> \text{bool}$ **where**
 $adm\text{-}wf\ R\ F == ALL\ f\ g\ x.$
 $(ALL\ z. (z, x) : R \dashrightarrow f\ z = g\ z) \dashrightarrow F\ f\ x = F\ g\ x$

definition

$wfrec :: ('a * 'a) \text{ set} ==> (('a ==> 'b) ==> 'a ==> 'b) ==> 'a ==> 'b$ **where**
 $[code\ del]: wfrec\ R\ F == \%x. \text{ THE } y. wfrec\text{-}rel\ R\ (\%f\ x. F\ (cut\ f\ R\ x)\ x)\ x\ y$

42.1 Well-Founded Recursion

cut

lemma *cuts-eq*: $(cut\ f\ r\ x = cut\ g\ r\ x) = (ALL\ y. (y,x):r \dashrightarrow f(y)=g(y))$
 $\langle proof \rangle$

lemma *cut-apply*: $(x,a):r ==> (cut\ f\ r\ a)(x) = f(x)$
 $\langle proof \rangle$

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

lemma *wfrec-unique*: $[[\ adm\text{-}wf\ R\ F; wf\ R\]] ==> EX! y. wfrec\text{-}rel\ R\ F\ x\ y$
 $\langle proof \rangle$

lemma *adm-lemma*: $adm\text{-}wf\ R\ (\%f\ x. F\ (cut\ f\ R\ x)\ x)$
 $\langle proof \rangle$

lemma *wfrec*: $wf(r) ==> wfrec\ r\ H\ a = H\ (cut\ (wfrec\ r\ H)\ r\ a)\ a$
 $\langle proof \rangle$

* 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$
 $\langle proof \rangle$

lemma *tfl-wf-induct*: $ALL\ R. wf\ R \dashrightarrow$
 $(ALL\ P. (ALL\ x. (ALL\ y. (y,x):R \dashrightarrow P\ y) \dashrightarrow P\ x) \dashrightarrow (ALL\ x. P\ x))$
 $\langle proof \rangle$

lemma *tfl-cut-apply*: $ALL\ f\ R. (x,a):R \dashrightarrow (cut\ f\ R\ a)(x) = f(x)$
 $\langle proof \rangle$

lemma *tfl-wfrec*:

$ALL\ M\ R\ f. (f == wfrec\ R\ M) \dashrightarrow wf\ R \dashrightarrow (ALL\ x. f\ x = M\ (cut\ f\ R\ x)\ x)$
 $\langle proof \rangle$

lemma *tfl-eq-True*: $(x = \text{True}) \dashrightarrow x$
 $\langle \text{proof} \rangle$

lemma *tfl-rev-eq-mp*: $(x = y) \dashrightarrow y \dashrightarrow x$
 $\langle \text{proof} \rangle$

lemma *tfl-simp-thm*: $(x \dashrightarrow y) \dashrightarrow (x = x') \dashrightarrow (x' \dashrightarrow y)$
 $\langle \text{proof} \rangle$

lemma *tfl-P-imp-P-iff-True*: $P \implies P = \text{True}$
 $\langle \text{proof} \rangle$

lemma *tfl-imp-trans*: $(A \dashrightarrow B) \implies (B \dashrightarrow C) \implies (A \dashrightarrow C)$
 $\langle \text{proof} \rangle$

lemma *tfl-disj-assoc*: $(a \vee b) \vee c == a \vee (b \vee c)$
 $\langle \text{proof} \rangle$

lemma *tfl-disjE*: $P \vee Q \implies P \dashrightarrow R \implies Q \dashrightarrow R \implies R$
 $\langle \text{proof} \rangle$

lemma *tfl-exE*: $\exists x. P x \implies \forall x. P x \dashrightarrow Q \implies Q$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

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 $P R == \{((x', y'), (x, y)) \mid x' = x \ \& \ P x \ \& \ (y', y) : R x\}$

— For *rec-def* declarations where the first n parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro!*]:

$[[P x; (y', y) : R x]] \implies ((x, y'), (x, y)) : \text{same-fst } P R$
 $\langle \text{proof} \rangle$

lemma *wf-same-fst*:

assumes *prem*: $(!!x. P x \implies \text{wf}(R x))$

shows $\text{wf}(\text{same-fst } P R)$

$\langle \text{proof} \rangle$

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 => 'a list  ([(-)])

translations
  [x, xs] == x#[xs]
  [x] == x#[]

```

43.1 Basic list processing functions

```

primrec
  hd :: 'a list => 'a where
  hd (x # xs) = x

primrec
  tl :: 'a list => 'a list where
  tl [] = []
  | tl (x # xs) = xs

primrec

```

last :: 'a list \Rightarrow 'a **where**
last (*x* # *xs*) = (if *xs* = [] then *x* else *last xs*)

primrec

butlast :: 'a list \Rightarrow 'a list **where**
butlast [] = []
 | *butlast* (*x* # *xs*) = (if *xs* = [] then [] else *x* # *butlast xs*)

primrec

set :: 'a list \Rightarrow 'a set **where**
set [] = {}
 | *set* (*x* # *xs*) = insert *x* (*set xs*)

primrec

map :: ('a \Rightarrow 'b) \Rightarrow 'a list \Rightarrow 'b list **where**
map *f* [] = []
 | *map* *f* (*x* # *xs*) = *f* *x* # *map f xs*

primrec

append :: 'a list \Rightarrow 'a list \Rightarrow 'a list (**infixr** @ 65) **where**
append-Nil: [] @ *ys* = *ys*
 | *append-Cons*: (*x* # *xs*) @ *ys* = *x* # *xs* @ *ys*

primrec

rev :: 'a list \Rightarrow 'a list **where**
rev [] = []
 | *rev* (*x* # *xs*) = *rev xs* @ [*x*]

primrec

filter :: ('a \Rightarrow bool) \Rightarrow 'a list \Rightarrow 'a list **where**
filter *P* [] = []
 | *filter* *P* (*x* # *xs*) = (if *P* *x* then *x* # *filter P xs* else *filter P xs*)

syntax

— Special syntax for filter
 -*filter* :: [*pttrn*, 'a list, bool] \Rightarrow 'a list ((1[-<--./ -]))

translations

[*x* < - *xs* . *P*] == CONST *filter* (%*x*. *P*) *xs*

syntax (*xsymbols*)

-*filter* :: [*pttrn*, 'a list, bool] \Rightarrow 'a list ((1[<--./ -]))

syntax (*HTML output*)

-*filter* :: [*pttrn*, 'a list, bool] \Rightarrow 'a list ((1[<--./ -]))

primrec

foldl :: ('b \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a list \Rightarrow 'b **where**
foldl-Nil: *foldl f a* [] = *a*
 | *foldl-Cons*: *foldl f a* (*x* # *xs*) = *foldl f* (*f a x*) *xs*

primrec

$foldr :: ('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b \Rightarrow 'b$ **where**
 $foldr\ f\ []\ a = a$
 $| foldr\ f\ (x \# xs)\ a = f\ x\ (foldr\ f\ xs\ a)$

primrec

$concat :: 'a\ list\ list \Rightarrow 'a\ list$ **where**
 $concat\ [] = []$
 $| concat\ (x \# xs) = x\ @\ concat\ xs$

primrec (in monoid-add)

$listsum :: 'a\ list \Rightarrow 'a$ **where**
 $listsum\ [] = 0$
 $| listsum\ (x \# xs) = x + listsum\ xs$

primrec

$drop :: nat \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $drop\ Nil: drop\ n\ [] = []$
 $| drop\ Cons: drop\ n\ (x \# xs) = (case\ n\ of\ 0 \Rightarrow x \# xs \mid Suc\ m \Rightarrow drop\ m\ xs)$
 — Warning: simpset does not contain this definition, but separate theorems for
 $n = 0$ and $n = Suc\ k$

primrec

$take :: nat \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $take\ Nil: take\ n\ [] = []$
 $| take\ Cons: take\ n\ (x \# xs) = (case\ n\ of\ 0 \Rightarrow [] \mid Suc\ m \Rightarrow x \# take\ m\ xs)$
 — Warning: simpset does not contain this definition, but separate theorems for
 $n = 0$ and $n = Suc\ k$

primrec

$nth :: 'a\ list \Rightarrow nat \Rightarrow 'a$ (**infixl ! 100**) **where**
 $nth\ Cons: (x \# xs) ! n = (case\ n\ of\ 0 \Rightarrow x \mid Suc\ k \Rightarrow xs ! k)$
 — Warning: simpset does not contain this definition, but separate theorems for
 $n = 0$ and $n = Suc\ k$

primrec

$list-update :: 'a\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ list$ **where**
 $list-update\ []\ i\ v = []$
 $| list-update\ (x \# xs)\ i\ v = (case\ i\ of\ 0 \Rightarrow v \# xs \mid Suc\ j \Rightarrow x \# list-update\ xs\ j\ v)$

nonterminals lupdbinds lupdbind**syntax**

$-lupdbind :: ['a, 'a] \Rightarrow lupdbind\ ((2- := / -))$
 $:: lupdbind \Rightarrow lupdbinds\ (-)$
 $-lupdbinds :: [lupdbind, lupdbinds] \Rightarrow lupdbinds\ (-, / -)$
 $-LUpdate :: ['a, lupdbinds] \Rightarrow 'a\ (-/[(-)]\ [900, 0]\ 900)$

translations

$-LUpdate\ xs\ (-lupdbinds\ b\ bs) == -LUpdate\ (-LUpdate\ xs\ b)\ bs$
 $xs[i:=x] == CONST\ list-update\ xs\ i\ x$

primrec

$takeWhile :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $takeWhile\ P\ [] = []$
 $| takeWhile\ P\ (x \# xs) = (if\ P\ x\ then\ x \# takeWhile\ P\ xs\ else\ [])$

primrec

$dropWhile :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $dropWhile\ P\ [] = []$
 $| dropWhile\ P\ (x \# xs) = (if\ P\ x\ then\ dropWhile\ P\ xs\ else\ x \# xs)$

primrec

$zip :: 'a\ list \Rightarrow 'b\ list \Rightarrow ('a \times 'b)\ list$ **where**
 $zip\ xs\ [] = []$
 $| zip-Cons: zip\ xs\ (y \# ys) = (case\ xs\ of\ [] \Rightarrow [] \mid z \# zs \Rightarrow (z, y) \# zip\ zs\ ys)$
 — Warning: simpset does not contain this definition, but separate theorems for
 $xs = []$ and $xs = z \# zs$

primrec

$upt :: nat \Rightarrow nat \Rightarrow nat\ list\ ((1[-..</-])$ **where**
 $upt-0: [i..<0] = []$
 $| upt-Suc: [i..<(Suc\ j)] = (if\ i \leq j\ then\ [i..<j] @ [j]\ else\ [])$

primrec

$distinct :: 'a\ list \Rightarrow bool$ **where**
 $distinct\ [] \longleftrightarrow True$
 $| distinct\ (x \# xs) \longleftrightarrow x \notin set\ xs \wedge distinct\ xs$

primrec

$remdups :: 'a\ list \Rightarrow 'a\ list$ **where**
 $remdups\ [] = []$
 $| remdups\ (x \# xs) = (if\ x \in set\ xs\ then\ remdups\ xs\ else\ x \# remdups\ xs)$

definition

$insert :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $insert\ x\ xs = (if\ x \in set\ xs\ then\ xs\ else\ x \# xs)$

hide-const (open) $insert$

hide-fact (open) $insert-def$

primrec

$remove1 :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $remove1\ x\ [] = []$
 $| remove1\ x\ (y \# xs) = (if\ x = y\ then\ xs\ else\ y \# remove1\ x\ xs)$

primrec

$removeAll :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $removeAll\ x\ [] = []$
 $| removeAll\ x\ (y \# xs) = (if\ x = y\ then\ removeAll\ x\ xs\ else\ y \# removeAll\ x\ xs)$

primrec

$replicate :: nat \Rightarrow 'a \Rightarrow 'a\ list$ **where**
 $replicate\ 0\ x = []$
 $| replicate\ Suc\ n\ x = x \# replicate\ n\ x$

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

abbreviation

$length :: 'a\ list \Rightarrow nat$ **where**
 $length \equiv size$

definition

$rotate1 :: 'a\ list \Rightarrow 'a\ list$ **where**
 $rotate1\ xs = (case\ xs\ of\ [] \Rightarrow [] \mid x \# xs \Rightarrow xs @ [x])$

definition

$rotate :: nat \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $rotate\ n = rotate1 \wedge^{\wedge} n$

definition

$list_all2 :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow bool$ **where**
 $[code\ del]: list_all2\ P\ xs\ ys =$
 $(length\ xs = length\ ys \wedge (\forall (x, y) \in set\ (zip\ xs\ ys). P\ x\ y))$

definition

$sublist :: 'a\ list \Rightarrow nat\ set \Rightarrow 'a\ list$ **where**
 $sublist\ xs\ A = map\ fst\ (filter\ (\lambda p. snd\ p \in A)\ (zip\ xs\ [0..<size\ xs]))$

primrec

$splice :: 'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$ **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* $\Rightarrow bool$ **where**


```

[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

$sorted [] \longleftrightarrow True$ |
 $sorted [x] \longleftrightarrow True$ |
 $sorted (x\#y\#zs) \longleftrightarrow x \leq y \wedge sorted (y\#zs)$

primrec $insort\text{-}key :: ('b \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
 $insort\text{-}key\ f\ x\ [] = [x]$ |
 $insort\text{-}key\ f\ x\ (y\#ys) = (if\ f\ x \leq f\ y\ then\ (x\#y\#ys)\ else\ y\#(insort\text{-}key\ f\ x\ ys))$

definition $sort\text{-}key :: ('b \Rightarrow 'a) \Rightarrow 'b\ list \Rightarrow 'b\ list$ **where**
 $sort\text{-}key\ f\ xs = foldr\ (insort\text{-}key\ f)\ xs\ []$

abbreviation $sort \equiv sort\text{-}key\ (\lambda x. x)$
abbreviation $insort \equiv insort\text{-}key\ (\lambda x. x)$

definition $insort\text{-}insert :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $insort\text{-}insert\ x\ xs = (if\ x \in set\ xs\ then\ xs\ else\ insort\ x\ xs)$

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 $[e\ |\ x \leftarrow xs, \dots]$.

The qualifiers after the dot are

generators $p \leftarrow xs$, where p is a pattern and xs an expression of list type,
or

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 $map\ (\lambda x. e)\ xs$.

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

$\text{-}listcompr :: 'a \Rightarrow lc\text{-}qual \Rightarrow lc\text{-}quals \Rightarrow 'a\ list$ ($[- \ . \ --]$)
 $\text{-}lc\text{-}gen :: 'a \Rightarrow 'a\ list \Rightarrow lc\text{-}qual$ ($[- \ <- \ -]$)
 $\text{-}lc\text{-}test :: bool \Rightarrow lc\text{-}qual$ ($[-]$)

$-lc-end :: lc-quals \ ()$
 $-lc-quals :: lc-qual \Rightarrow lc-quals \Rightarrow lc-quals \ (, \ --)$
 $-lc-abs :: 'a \Rightarrow 'b \ list \Rightarrow 'b \ list$

syntax (*xsymbols*)
 $-lc-gen :: 'a \Rightarrow 'a \ list \Rightarrow lc-qual \ (- \leftarrow -)$
syntax (*HTML output*)
 $-lc-gen :: 'a \Rightarrow 'a \ list \Rightarrow lc-qual \ (- \leftarrow -)$

$\langle ML \rangle$

term $[(x,y,z). \ b]$
term $[(x,y,z). \ x \leftarrow xs]$
term $[e \ x \ y. \ x \leftarrow xs, \ y \leftarrow ys]$
term $[(x,y,z). \ x < a, \ x > b]$
term $[(x,y,z). \ x \leftarrow xs, \ x > b]$
term $[(x,y,z). \ x < a, \ x \leftarrow xs]$
term $[(x,y). \ Cons \ True \ x \leftarrow xs]$
term $[(x,y,z). \ Cons \ x \ [] \leftarrow xs]$
term $[(x,y,z). \ x < a, \ x > b, \ x = d]$
term $[(x,y,z). \ x < a, \ x > b, \ y \leftarrow ys]$
term $[(x,y,z). \ x < a, \ x \leftarrow xs, y > b]$
term $[(x,y,z). \ x < a, \ x \leftarrow xs, \ y \leftarrow ys]$
term $[(x,y,z). \ x \leftarrow xs, \ x > b, \ y < a]$
term $[(x,y,z). \ x \leftarrow xs, \ x > b, \ y \leftarrow ys]$
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*]:
 $xs \neq x \ \# \ xs$
 $\langle proof \rangle$

lemmas *not-Cons-self2* [*simp*] = *not-Cons-self* [*symmetric*]

lemma *neq-Nil-conv*: $(xs \neq []) = (\exists y \ ys. \ xs = y \ \# \ ys)$
 $\langle proof \rangle$

lemma *length-induct*:
 $(\bigwedge xs. \ \forall ys. \ length \ ys < length \ xs \longrightarrow P \ ys \Longrightarrow P \ xs) \Longrightarrow P \ xs$
 $\langle proof \rangle$

lemma *list-nonempty-induct* [*consumes 1, case-names single cons*]:
assumes $xs \neq []$
assumes *single*: $\bigwedge x. \ P \ [x]$

assumes *cons*: $\bigwedge x \text{ xs}. \text{xs} \neq [] \implies P \text{ xs} \implies P (x \# \text{xs})$
shows $P \text{ xs}$
 $\langle \text{proof} \rangle$

43.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: $\text{length} (\text{xs} @ \text{ys}) = \text{length} \text{xs} + \text{length} \text{ys}$
 $\langle \text{proof} \rangle$

lemma *length-map* [*simp*]: $\text{length} (\text{map } f \text{ xs}) = \text{length} \text{xs}$
 $\langle \text{proof} \rangle$

lemma *length-rev* [*simp*]: $\text{length} (\text{rev xs}) = \text{length} \text{xs}$
 $\langle \text{proof} \rangle$

lemma *length-tl* [*simp*]: $\text{length} (\text{tl xs}) = \text{length} \text{xs} - 1$
 $\langle \text{proof} \rangle$

lemma *length-0-conv* [*iff*]: $(\text{length} \text{xs} = 0) = (\text{xs} = [])$
 $\langle \text{proof} \rangle$

lemma *length-greater-0-conv* [*iff*]: $(0 < \text{length} \text{xs}) = (\text{xs} \neq [])$
 $\langle \text{proof} \rangle$

lemma *length-pos-if-in-set*: $x : \text{set xs} \implies \text{length} \text{xs} > 0$
 $\langle \text{proof} \rangle$

lemma *length-Suc-conv*:
 $(\text{length} \text{xs} = \text{Suc } n) = (\exists y \text{ ys}. \text{xs} = y \# \text{ys} \wedge \text{length} \text{ys} = n)$
 $\langle \text{proof} \rangle$

lemma *Suc-length-conv*:
 $(\text{Suc } n = \text{length} \text{xs}) = (\exists y \text{ ys}. \text{xs} = y \# \text{ys} \wedge \text{length} \text{ys} = n)$
 $\langle \text{proof} \rangle$

lemma *impossible-Cons*: $\text{length} \text{xs} \leq \text{length} \text{ys} \implies \text{xs} = x \# \text{ys} = \text{False}$
 $\langle \text{proof} \rangle$

lemma *list-induct2* [*consumes 1, case-names Nil Cons*]:
 $\text{length} \text{xs} = \text{length} \text{ys} \implies P [] [] \implies$
 $(\bigwedge x \text{ xs } y \text{ ys}. \text{length} \text{xs} = \text{length} \text{ys} \implies P \text{xs ys} \implies P (x \# \text{xs}) (y \# \text{ys}))$
 $\implies P \text{xs ys}$
 $\langle \text{proof} \rangle$

lemma *list-induct3* [*consumes 2, case-names Nil Cons*]:
 $\text{length} \text{xs} = \text{length} \text{ys} \implies \text{length} \text{ys} = \text{length} \text{zs} \implies P [] [] [] \implies$
 $(\bigwedge x \text{ xs } y \text{ ys } z \text{ zs}. \text{length} \text{xs} = \text{length} \text{ys} \implies \text{length} \text{ys} = \text{length} \text{zs} \implies P \text{xs ys zs}$
 $\implies P (x \# \text{xs}) (y \# \text{ys}) (z \# \text{zs}))$

$\implies P\ xs\ ys\ zs$
 $\langle proof \rangle$

lemma *list-induct4* [consumes 3, case-names Nil Cons]:
 $length\ xs = length\ ys \implies length\ ys = length\ zs \implies length\ zs = length\ ws \implies$
 $P\ []\ [] \implies (\bigwedge x\ xs\ y\ ys\ z\ zs\ w\ ws. length\ xs = length\ ys \implies$
 $length\ ys = length\ zs \implies length\ zs = length\ ws \implies P\ xs\ ys\ zs\ ws \implies$
 $P\ (x\#xs)\ (y\#ys)\ (z\#zs)\ (w\#ws)) \implies P\ xs\ ys\ zs\ ws$
 $\langle proof \rangle$

lemma *list-induct2'*:
 $\llbracket P\ []\ [];$
 $\bigwedge x\ xs. P\ (x\#xs)\ [];$
 $\bigwedge y\ ys. P\ []\ (y\#ys);$
 $\bigwedge x\ xs\ y\ ys. P\ xs\ ys \implies P\ (x\#xs)\ (y\#ys) \rrbracket$
 $\implies P\ xs\ ys$
 $\langle proof \rangle$

lemma *neq-if-length-neq*: $length\ xs \neq length\ ys \implies (xs = ys) == False$
 $\langle proof \rangle$

$\langle ML \rangle$

43.1.4 @ – append

lemma *append-assoc* [simp]: $(xs\ @\ ys)\ @\ zs = xs\ @\ (ys\ @\ zs)$
 $\langle proof \rangle$

lemma *append-Nil2* [simp]: $xs\ @\ [] = xs$
 $\langle proof \rangle$

lemma *append-is-Nil-conv* [iff]: $(xs\ @\ ys = []) = (xs = [] \wedge ys = [])$
 $\langle proof \rangle$

lemma *Nil-is-append-conv* [iff]: $([] = xs\ @\ ys) = (xs = [] \wedge ys = [])$
 $\langle proof \rangle$

lemma *append-self-conv* [iff]: $(xs\ @\ ys = xs) = (ys = [])$
 $\langle proof \rangle$

lemma *self-append-conv* [iff]: $(xs = xs\ @\ ys) = (ys = [])$
 $\langle proof \rangle$

lemma *append-eq-append-conv* [simp, no-atp]:
 $length\ xs = length\ ys \vee length\ us = length\ vs$
 $\implies (xs@us = ys@vs) = (xs=ys \wedge us=vs)$
 $\langle proof \rangle$

lemma *append-eq-append-conv2*: $(xs\ @\ ys = zs\ @\ ts) =$

$(EX\ us.\ xs = zs @ us \ \&\ us @ ys = ts \mid xs @ us = zs \ \&\ ys = us @ ts)$
 $\langle proof \rangle$

lemma *same-append-eq* [iff, induct-simp]: $(xs @ ys = xs @ zs) = (ys = zs)$
 $\langle proof \rangle$

lemma *append1-eq-conv* [iff]: $(xs @ [x] = ys @ [y]) = (xs = ys \wedge x = y)$
 $\langle proof \rangle$

lemma *append-same-eq* [iff, induct-simp]: $(ys @ xs = zs @ xs) = (ys = zs)$
 $\langle proof \rangle$

lemma *append-self-conv2* [iff]: $(xs @ ys = ys) = (xs = [])$
 $\langle proof \rangle$

lemma *self-append-conv2* [iff]: $(ys = xs @ ys) = (xs = [])$
 $\langle proof \rangle$

lemma *hd-Cons-tl* [simp, no-atp]: $xs \neq [] \implies hd\ xs \# tl\ xs = xs$
 $\langle proof \rangle$

lemma *hd-append*: $hd\ (xs @ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
 $\langle proof \rangle$

lemma *hd-append2* [simp]: $xs \neq [] \implies hd\ (xs @ ys) = hd\ xs$
 $\langle proof \rangle$

lemma *tl-append*: $tl\ (xs @ ys) = (case\ xs\ of\ [] \implies tl\ ys \mid z \# zs \implies zs @ ys)$
 $\langle proof \rangle$

lemma *tl-append2* [simp]: $xs \neq [] \implies tl\ (xs @ ys) = tl\ xs @ ys$
 $\langle proof \rangle$

lemma *Cons-eq-append-conv*: $x \# xs = ys @ zs =$
 $(ys = [] \ \&\ x \# xs = zs \mid (EX\ ys'.\ x \# ys' = ys \ \&\ xs = ys' @ zs))$
 $\langle proof \rangle$

lemma *append-eq-Cons-conv*: $ys @ zs = x \# xs =$
 $(ys = [] \ \&\ zs = x \# xs \mid (EX\ ys'.\ ys = x \# ys' \ \&\ ys' @ zs = xs))$
 $\langle proof \rangle$

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: $xs = ys \implies xs = [] @ ys$
 $\langle proof \rangle$

lemma *Cons-eq-appendI*:
 $[| x \# xs1 = ys; xs = xs1 @ zs |] \implies x \# xs = ys @ zs$
 $\langle proof \rangle$

lemma *append-eq-appendI*:

$[| xs @ xs1 = zs; ys = xs1 @ us |] ==> xs @ ys = zs @ us$
 $\langle proof \rangle$

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.

$\langle ML \rangle$

43.1.5 map

lemma *map-ext*: $(!x. x : set\ xs \rightarrow f\ x = g\ x) ==> map\ f\ xs = map\ g\ xs$
 $\langle proof \rangle$

lemma *map-ident* [simp]: $map\ (\lambda x. x) = (\lambda xs. xs)$
 $\langle proof \rangle$

lemma *map-append* [simp]: $map\ f\ (xs @ ys) = map\ f\ xs @ map\ f\ ys$
 $\langle proof \rangle$

lemma *map-map* [simp]: $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$
 $\langle proof \rangle$

lemma *map-comp-map*[simp]: $((map\ f) \circ (map\ g)) = map\ (f \circ g)$
 $\langle proof \rangle$

lemma *rev-map*: $rev\ (map\ f\ xs) = map\ f\ (rev\ xs)$
 $\langle proof \rangle$

lemma *map-eq-conv*[simp]: $(map\ f\ xs = map\ g\ xs) = (!x : set\ xs. f\ x = g\ x)$
 $\langle proof \rangle$

lemma *map-cong* [fundef-cong, recdef-cong]:
 $xs = ys ==> (!x. x : set\ ys ==> f\ x = g\ x) ==> map\ f\ xs = map\ g\ ys$
 — a congruence rule for *map*
 $\langle proof \rangle$

lemma *map-is-Nil-conv* [iff]: $(map\ f\ xs = []) = (xs = [])$
 $\langle proof \rangle$

lemma *Nil-is-map-conv* [iff]: $([] = map\ f\ xs) = (xs = [])$
 $\langle proof \rangle$

lemma *map-eq-Cons-conv*:
 $(map\ f\ xs = y \# ys) = (\exists z\ zs. xs = z \# zs \wedge f\ z = y \wedge map\ f\ zs = ys)$
 $\langle proof \rangle$

lemma *Cons-eq-map-conv*:

$(x \# xs = \text{map } f \text{ } ys) = (\exists z \text{ } zs. \text{ } ys = z \# zs \wedge x = f \text{ } z \wedge xs = \text{map } f \text{ } zs)$
 $\langle \text{proof} \rangle$

lemmas $\text{map-eq-Cons-}D = \text{map-eq-Cons-conv} \text{ } [THEN \text{ } iffD1]$

lemmas $\text{Cons-eq-map-}D = \text{Cons-eq-map-conv} \text{ } [THEN \text{ } iffD1]$

declare $\text{map-eq-Cons-}D \text{ } [dest!] \text{ } \text{Cons-eq-map-}D \text{ } [dest!]$

lemma ex-map-conv :

$(EX \text{ } xs. \text{ } ys = \text{map } f \text{ } xs) = (ALL \text{ } y : \text{set } ys. \text{ } EX \text{ } x. \text{ } y = f \text{ } x)$
 $\langle \text{proof} \rangle$

lemma $\text{map-eq-imp-length-eq}$:

assumes $\text{map } f \text{ } xs = \text{map } g \text{ } ys$

shows $\text{length } xs = \text{length } ys$

$\langle \text{proof} \rangle$

lemma map-inj-on :

$[[\text{map } f \text{ } xs = \text{map } f \text{ } ys; \text{inj-on } f \text{ } (\text{set } xs \text{ } Un \text{ } \text{set } ys) \text{ }]]$
 $\implies xs = ys$

$\langle \text{proof} \rangle$

lemma inj-on-map-eq-map :

$\text{inj-on } f \text{ } (\text{set } xs \text{ } Un \text{ } \text{set } ys) \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$
 $\langle \text{proof} \rangle$

lemma map-injective :

$\text{map } f \text{ } xs = \text{map } f \text{ } ys \implies \text{inj } f \implies xs = ys$
 $\langle \text{proof} \rangle$

lemma $\text{inj-map-eq-map[simp]}$: $\text{inj } f \implies (\text{map } f \text{ } xs = \text{map } f \text{ } ys) = (xs = ys)$
 $\langle \text{proof} \rangle$

lemma inj-mapI : $\text{inj } f \implies \text{inj } (\text{map } f)$

$\langle \text{proof} \rangle$

lemma inj-mapD : $\text{inj } (\text{map } f) \implies \text{inj } f$

$\langle \text{proof} \rangle$

lemma inj-map[iff] : $\text{inj } (\text{map } f) = \text{inj } f$

$\langle \text{proof} \rangle$

lemma inj-on-mapI : $\text{inj-on } f \text{ } (\bigcup (\text{set } 'A)) \implies \text{inj-on } (\text{map } f) \text{ } A$

$\langle \text{proof} \rangle$

lemma map-idI : $(\bigwedge x. x \in \text{set } xs \implies f \text{ } x = x) \implies \text{map } f \text{ } xs = xs$

$\langle \text{proof} \rangle$

lemma $\text{map-fun-upd [simp]}$: $y \notin \text{set } xs \implies \text{map } (f(y:=v)) \text{ } xs = \text{map } f \text{ } xs$

$\langle \text{proof} \rangle$

lemma *map-fst-zip*[simp]:

$\text{length } xs = \text{length } ys \implies \text{map fst } (\text{zip } xs \ ys) = xs$
 $\langle \text{proof} \rangle$

lemma *map-snd-zip*[simp]:

$\text{length } xs = \text{length } ys \implies \text{map snd } (\text{zip } xs \ ys) = ys$
 $\langle \text{proof} \rangle$

43.1.6 *rev*

lemma *rev-append* [simp]: $\text{rev } (xs \ @ \ ys) = \text{rev } ys \ @ \ \text{rev } xs$
 $\langle \text{proof} \rangle$

lemma *rev-rev-ident* [simp]: $\text{rev } (\text{rev } xs) = xs$
 $\langle \text{proof} \rangle$

lemma *rev-swap*: $(\text{rev } xs = ys) = (xs = \text{rev } ys)$
 $\langle \text{proof} \rangle$

lemma *rev-is-Nil-conv* [iff]: $(\text{rev } xs = []) = (xs = [])$
 $\langle \text{proof} \rangle$

lemma *Nil-is-rev-conv* [iff]: $([] = \text{rev } xs) = (xs = [])$
 $\langle \text{proof} \rangle$

lemma *rev-singleton-conv* [simp]: $(\text{rev } xs = [x]) = (xs = [x])$
 $\langle \text{proof} \rangle$

lemma *singleton-rev-conv* [simp]: $([x] = \text{rev } xs) = (xs = [x])$
 $\langle \text{proof} \rangle$

lemma *rev-is-rev-conv* [iff]: $(\text{rev } xs = \text{rev } ys) = (xs = ys)$
 $\langle \text{proof} \rangle$

lemma *inj-on-rev*[iff]: *inj-on* *rev* *A*
 $\langle \text{proof} \rangle$

lemma *rev-induct* [case-names *Nil snoc*]:

$[\![\ P \]\!]; \ \forall x \ xs. \ P \ xs \implies P \ (xs \ @ \ [x]) \] \implies P \ xs$
 $\langle \text{proof} \rangle$

lemma *rev-exhaust* [case-names *Nil snoc*]:

$(xs = [] \implies P) \implies (\forall ys \ y. \ xs = ys \ @ \ [y] \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemmas *rev-cases* = *rev-exhaust*

lemma *rev-eq-Cons-iff*[iff]: $(\text{rev } xs = y \# ys) = (xs = \text{rev } ys \ @ \ [y])$

$\langle proof \rangle$

43.1.7 set

lemma *finite-set* [iff]: *finite* (set *xs*)
 $\langle proof \rangle$

lemma *set-append* [simp]: *set* (*xs* @ *ys*) = (*set xs* \cup *set ys*)
 $\langle proof \rangle$

lemma *hd-in-set* [simp]: *xs* $\neq [] \implies \text{hd } xs : \text{set } xs$
 $\langle proof \rangle$

lemma *set-subset-Cons*: *set xs* \subseteq *set* (*x* # *xs*)
 $\langle proof \rangle$

lemma *set-ConsD*: *y* \in *set* (*x* # *xs*) $\implies y = x \vee y \in \text{set } xs$
 $\langle proof \rangle$

lemma *set-empty* [iff]: (*set xs* = {}) = (*xs* = [])
 $\langle proof \rangle$

lemma *set-empty2* [iff]: ({ } = *set xs*) = (*xs* = [])
 $\langle proof \rangle$

lemma *set-rev* [simp]: *set* (*rev xs*) = *set xs*
 $\langle proof \rangle$

lemma *set-map* [simp]: *set* (*map f xs*) = *f*’(*set xs*)
 $\langle proof \rangle$

lemma *set-filter* [simp]: *set* (*filter P xs*) = {*x*. *x* : *set xs* \wedge *P x*}
 $\langle proof \rangle$

lemma *set-upt* [simp]: *set*[*i*..*j*] = {*i*..*j*}
 $\langle proof \rangle$

lemma *split-list*: *x* : *set xs* $\implies \exists ys\ zs. xs = ys @ x \# zs$
 $\langle proof \rangle$

lemma *in-set-conv-decomp*: *x* \in *set xs* $\longleftrightarrow (\exists ys\ zs. xs = ys @ x \# zs)$
 $\langle proof \rangle$

lemma *split-list-first*: *x* : *set xs* $\implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys$
 $\langle proof \rangle$

lemma *in-set-conv-decomp-first*:
(*x* : *set xs*) = ($\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } ys$)

$\langle \text{proof} \rangle$

lemma *split-list-last*: $x : \text{set } xs \implies \exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs$
 $\langle \text{proof} \rangle$

lemma *in-set-conv-decomp-last*:
 $(x : \text{set } xs) = (\exists ys\ zs. xs = ys @ x \# zs \wedge x \notin \text{set } zs)$
 $\langle \text{proof} \rangle$

lemma *split-list-prop*: $\exists x \in \text{set } xs. P\ x \implies \exists ys\ x\ zs. xs = ys @ x \# zs \ \& \ P\ x$
 $\langle \text{proof} \rangle$

lemma *split-list-propE*:
assumes $\exists x \in \text{set } xs. P\ x$
obtains $ys\ x\ zs$ **where** $xs = ys @ x \# zs$ **and** $P\ x$
 $\langle \text{proof} \rangle$

lemma *split-list-first-prop*:
 $\exists x \in \text{set } xs. P\ x \implies$
 $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall y \in \text{set } ys. \neg P\ y)$
 $\langle \text{proof} \rangle$

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** $\forall y \in \text{set } ys. \neg P\ y$
 $\langle \text{proof} \rangle$

lemma *split-list-first-prop-iff*:
 $(\exists x \in \text{set } xs. P\ x) \longleftrightarrow$
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall y \in \text{set } ys. \neg P\ y))$
 $\langle \text{proof} \rangle$

lemma *split-list-last-prop*:
 $\exists x \in \text{set } xs. P\ x \implies$
 $\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in \text{set } zs. \neg P\ z)$
 $\langle \text{proof} \rangle$

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** $\forall z \in \text{set } zs. \neg P\ z$
 $\langle \text{proof} \rangle$

lemma *split-list-last-prop-iff*:
 $(\exists x \in \text{set } xs. P\ x) \longleftrightarrow$
 $(\exists ys\ x\ zs. xs = ys @ x \# zs \wedge P\ x \wedge (\forall z \in \text{set } zs. \neg P\ z))$
 $\langle \text{proof} \rangle$

lemma *finite-list*: $\text{finite } A \implies \exists X\ xs. \text{set } xs = A$
 $\langle \text{proof} \rangle$

lemma *card-length*: $\text{card } (\text{set } xs) \leq \text{length } xs$

<proof>

lemma *set-minus-filter-out*:

$\text{set } xs - \{y\} = \text{set } (\text{filter } (\lambda x. \neg (x = y)) \text{ } xs)$

<proof>

43.1.8 *filter*

lemma *filter-append* [simp]: $\text{filter } P \text{ } (xs @ ys) = \text{filter } P \text{ } xs @ \text{filter } P \text{ } ys$

<proof>

lemma *rev-filter*: $\text{rev } (\text{filter } P \text{ } xs) = \text{filter } P \text{ } (\text{rev } xs)$

<proof>

lemma *filter-filter* [simp]: $\text{filter } P \text{ } (\text{filter } Q \text{ } xs) = \text{filter } (\lambda x. Q \text{ } x \wedge P \text{ } x) \text{ } xs$

<proof>

lemma *length-filter-le* [simp]: $\text{length } (\text{filter } P \text{ } xs) \leq \text{length } xs$

<proof>

lemma *sum-length-filter-compl*:

$\text{length } (\text{filter } P \text{ } xs) + \text{length } (\text{filter } (\lambda x. \neg P \text{ } x) \text{ } xs) = \text{length } xs$

<proof>

lemma *filter-True* [simp]: $\forall x \in \text{set } xs. P \text{ } x \implies \text{filter } P \text{ } xs = xs$

<proof>

lemma *filter-False* [simp]: $\forall x \in \text{set } xs. \neg P \text{ } x \implies \text{filter } P \text{ } xs = []$

<proof>

lemma *filter-empty-conv*: $(\text{filter } P \text{ } xs = []) = (\forall x \in \text{set } xs. \neg P \text{ } x)$

<proof>

lemma *filter-id-conv*: $(\text{filter } P \text{ } xs = xs) = (\forall x \in \text{set } xs. P \text{ } x)$

<proof>

lemma *filter-map*:

$\text{filter } P \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{filter } (P \circ f) \text{ } xs)$

<proof>

lemma *length-filter-map* [simp]:

$\text{length } (\text{filter } P \text{ } (\text{map } f \text{ } xs)) = \text{length } (\text{filter } (P \circ f) \text{ } xs)$

<proof>

lemma *filter-is-subset* [simp]: $\text{set } (\text{filter } P \text{ } xs) \leq \text{set } xs$

<proof>

lemma *length-filter-less*:

$\llbracket x : \text{set } xs; \sim P x \rrbracket \implies \text{length}(\text{filter } P xs) < \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *length-filter-conv-card*:

$\text{length}(\text{filter } p xs) = \text{card}\{i. i < \text{length } xs \ \& \ p(xs!i)\}$
 $\langle \text{proof} \rangle$

lemma *Cons-eq-filterD*:

$x \# xs = \text{filter } P ys \implies$
 $\exists us \ vs. \ ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs$
 $(\text{is } - \implies \exists us \ vs. \ ?P \ ys \ us \ vs)$
 $\langle \text{proof} \rangle$

lemma *filter-eq-ConsD*:

$\text{filter } P ys = x \# xs \implies$
 $\exists us \ vs. \ ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs$
 $\langle \text{proof} \rangle$

lemma *filter-eq-Cons-iff*:

$(\text{filter } P ys = x \# xs) =$
 $(\exists us \ vs. \ ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs)$
 $\langle \text{proof} \rangle$

lemma *Cons-eq-filter-iff*:

$(x \# xs = \text{filter } P ys) =$
 $(\exists us \ vs. \ ys = us @ x \# vs \wedge (\forall u \in \text{set } us. \neg P u) \wedge P x \wedge xs = \text{filter } P vs)$
 $\langle \text{proof} \rangle$

lemma *filter-cong[fundef-cong, recdef-cong]*:

$xs = ys \implies (\bigwedge x. x \in \text{set } ys \implies P x = Q x) \implies \text{filter } P xs = \text{filter } Q ys$
 $\langle \text{proof} \rangle$

43.1.9 List partitioning

primrec *partition* :: $('a \Rightarrow \text{bool}) \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \times 'a \text{ list}$ **where**

$\text{partition } P [] = ([], [])$
 $| \text{partition } P (x \# xs) =$
 $(\text{let } (yes, no) = \text{partition } P xs$
 $\text{in if } P x \text{ then } (x \# yes, no) \text{ else } (yes, x \# no))$

lemma *partition-filter1*:

$\text{fst } (\text{partition } P xs) = \text{filter } P xs$
 $\langle \text{proof} \rangle$

lemma *partition-filter2*:

$\text{snd } (\text{partition } P xs) = \text{filter } (\text{Not } o P) xs$
 $\langle \text{proof} \rangle$

lemma *partition-P*:

assumes *partition* $P\ xs = (yes, no)$

shows $(\forall p \in set\ yes. P\ p) \wedge (\forall p \in set\ no. \neg P\ p)$

$\langle proof \rangle$

lemma *partition-set*:

assumes *partition* $P\ xs = (yes, no)$

shows $set\ yes \cup set\ no = set\ xs$

$\langle proof \rangle$

lemma *partition-filter-conv*[*simp*]:

partition $f\ xs = (filter\ f\ xs, filter\ (Not\ o\ f)\ xs)$

$\langle proof \rangle$

declare *partition.simps*[*simp del*]

43.1.10 *concat*

lemma *concat-append* [*simp*]: $concat\ (xs\ @\ ys) = concat\ xs\ @\ concat\ ys$

$\langle proof \rangle$

lemma *concat-eq-Nil-conv* [*simp*]: $(concat\ xss = []) = (\forall xs \in set\ xss. xs = [])$

$\langle proof \rangle$

lemma *Nil-eq-concat-conv* [*simp*]: $([] = concat\ xss) = (\forall xs \in set\ xss. xs = [])$

$\langle proof \rangle$

lemma *set-concat* [*simp*]: $set\ (concat\ xs) = (UN\ x:set\ xs. set\ x)$

$\langle proof \rangle$

lemma *concat-map-singleton*[*simp*]: $concat(map\ (\%x. [f\ x])\ xs) = map\ f\ xs$

$\langle proof \rangle$

lemma *map-concat*: $map\ f\ (concat\ xs) = concat\ (map\ (map\ f)\ xs)$

$\langle proof \rangle$

lemma *filter-concat*: $filter\ p\ (concat\ xs) = concat\ (map\ (filter\ p)\ xs)$

$\langle proof \rangle$

lemma *rev-concat*: $rev\ (concat\ xs) = concat\ (map\ rev\ (rev\ xs))$

$\langle proof \rangle$

43.1.11 *nth*

lemma *nth-Cons-0* [*simp*, *code*]: $(x\ \# \ xs)!0 = x$

$\langle proof \rangle$

lemma *nth-Cons-Suc* [*simp*, *code*]: $(x\ \# \ xs)!(Suc\ n) = xs!n$

$\langle proof \rangle$

declare *nth.simps* [*simp del*]

lemma *nth-append*:

$(xs @ ys)!n = (\text{if } n < \text{length } xs \text{ then } xs!n \text{ else } ys!(n - \text{length } xs))$
 $\langle \text{proof} \rangle$

lemma *nth-append-length* [*simp*]: $(xs @ x \# ys) ! \text{length } xs = x$
 $\langle \text{proof} \rangle$

lemma *nth-append-length-plus* [*simp*]: $(xs @ ys) ! (\text{length } xs + n) = ys ! n$
 $\langle \text{proof} \rangle$

lemma *nth-map* [*simp*]: $n < \text{length } xs \implies (\text{map } f \text{ } xs)!n = f(xs!n)$
 $\langle \text{proof} \rangle$

lemma *hd-conv-nth*: $xs \neq [] \implies \text{hd } xs = xs!0$
 $\langle \text{proof} \rangle$

lemma *list-eq-iff-nth-eq*:

$(xs = ys) = (\text{length } xs = \text{length } ys \wedge (\text{ALL } i < \text{length } xs. xs!i = ys!i))$
 $\langle \text{proof} \rangle$

lemma *set-conv-nth*: $\text{set } xs = \{xs!i \mid i. i < \text{length } xs\}$
 $\langle \text{proof} \rangle$

lemma *in-set-conv-nth*: $(x \in \text{set } xs) = (\exists i < \text{length } xs. xs!i = x)$
 $\langle \text{proof} \rangle$

lemma *list-ball-nth*: $[| n < \text{length } xs; !x : \text{set } xs. P \ x|] \implies P(xs!n)$
 $\langle \text{proof} \rangle$

lemma *nth-mem* [*simp*]: $n < \text{length } xs \implies xs!n : \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *all-nth-imp-all-set*:

$[| !i < \text{length } xs. P(xs!i); x : \text{set } xs|] \implies P \ x$
 $\langle \text{proof} \rangle$

lemma *all-set-conv-all-nth*:

$(\forall x \in \text{set } xs. P \ x) = (\forall i. i < \text{length } xs \longrightarrow P \ (xs ! i))$
 $\langle \text{proof} \rangle$

lemma *rev-nth*:

$n < \text{size } xs \implies \text{rev } xs ! n = xs ! (\text{length } xs - \text{Suc } n)$
 $\langle \text{proof} \rangle$

lemma *Skolem-list-nth*:

$(\text{ALL } i < k. \text{EX } x. P \ i \ x) = (\text{EX } xs. \text{size } xs = k \ \& \ (\text{ALL } i < k. P \ i \ (xs!i)))$

(**is** - = (EX xs . ?P k xs))
 ⟨proof⟩

43.1.12 list-update

lemma *length-list-update* [simp]: $\text{length}(xs[i:=x]) = \text{length } xs$
 ⟨proof⟩

lemma *nth-list-update*:
 $i < \text{length } xs \implies (xs[i:=x])!j = (\text{if } i = j \text{ then } x \text{ else } xs!j)$
 ⟨proof⟩

lemma *nth-list-update-eq* [simp]: $i < \text{length } xs \implies (xs[i:=x])!i = x$
 ⟨proof⟩

lemma *nth-list-update-neq* [simp]: $i \neq j \implies xs[i:=x]!j = xs!j$
 ⟨proof⟩

lemma *list-update-id* [simp]: $xs[i := xs!i] = xs$
 ⟨proof⟩

lemma *list-update-beyond* [simp]: $\text{length } xs \leq i \implies xs[i:=x] = xs$
 ⟨proof⟩

lemma *list-update-nonempty* [simp]: $xs[k:=x] = [] \longleftrightarrow xs = []$
 ⟨proof⟩

lemma *list-update-same-conv*:
 $i < \text{length } xs \implies (xs[i := x] = xs) = (xs!i = x)$
 ⟨proof⟩

lemma *list-update-append1*:
 $i < \text{size } xs \implies (xs @ ys)[i:=x] = xs[i:=x] @ ys$
 ⟨proof⟩

lemma *list-update-append*:
 $(xs @ ys)[n:=x] =$
 $(\text{if } n < \text{length } xs \text{ then } xs[n:=x] @ ys \text{ else } xs @ (ys[n - \text{length } xs := x]))$
 ⟨proof⟩

lemma *list-update-length* [simp]:
 $(xs @ x \# ys)[\text{length } xs := y] = (xs @ y \# ys)$
 ⟨proof⟩

lemma *map-update*: $\text{map } f (xs[k := y]) = (\text{map } f xs)[k := f y]$
 ⟨proof⟩

lemma *rev-update*:
 $k < \text{length } xs \implies \text{rev } (xs[k := y]) = (\text{rev } xs)[\text{length } xs - k - 1 := y]$

$\langle proof \rangle$

lemma *update-zip*:

$(zip\ xs\ ys)[i:=xy] = zip\ (xs[i:=fst\ xy])\ (ys[i:=snd\ xy])$
 $\langle proof \rangle$

lemma *set-update-subset-insert*: $set(xs[i:=x]) \leq insert\ x\ (set\ xs)$

$\langle proof \rangle$

lemma *set-update-subsetI*: $[\mid set\ xs \leq A; x:A \mid] ==> set(xs[i := x]) \leq A$

$\langle proof \rangle$

lemma *set-update-memI*: $n < length\ xs \implies x \in set\ (xs[n := x])$

$\langle proof \rangle$

lemma *list-update-overwrite[simp]*:

$xs\ [i := x, i := y] = xs\ [i := y]$
 $\langle proof \rangle$

lemma *list-update-swap*:

$i \neq i' \implies xs\ [i := x, i' := x'] = xs\ [i' := x', i := x]$
 $\langle proof \rangle$

lemma *list-update-code [code]*:

$[] [i := y] = []$
 $(x \# xs)[0 := y] = y \# xs$
 $(x \# xs)[Suc\ i := y] = x \# xs[i := y]$
 $\langle proof \rangle$

43.1.13 *last and butlast*

lemma *last-snoc [simp]*: $last\ (xs @ [x]) = x$

$\langle proof \rangle$

lemma *butlast-snoc [simp]*: $butlast\ (xs @ [x]) = xs$

$\langle proof \rangle$

lemma *last-ConsL*: $xs = [] \implies last(x \# xs) = x$

$\langle proof \rangle$

lemma *last-ConsR*: $xs \neq [] \implies last(x \# xs) = last\ xs$

$\langle proof \rangle$

lemma *last-append*: $last(xs @ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$

$\langle proof \rangle$

lemma *last-appendL[simp]*: $ys = [] \implies last(xs @ ys) = last\ xs$

$\langle proof \rangle$

lemma *last-appendR[simp]*: $ys \neq [] \implies \text{last}(xs @ ys) = \text{last } ys$
 $\langle \text{proof} \rangle$

lemma *hd-rev*: $xs \neq [] \implies \text{hd}(\text{rev } xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *last-rev*: $xs \neq [] \implies \text{last}(\text{rev } xs) = \text{hd } xs$
 $\langle \text{proof} \rangle$

lemma *last-in-set[simp]*: $as \neq [] \implies \text{last } as \in \text{set } as$
 $\langle \text{proof} \rangle$

lemma *length-butlast [simp]*: $\text{length } (\text{butlast } xs) = \text{length } xs - 1$
 $\langle \text{proof} \rangle$

lemma *butlast-append*:
 $\text{butlast } (xs @ ys) = (\text{if } ys = [] \text{ then } \text{butlast } xs \text{ else } xs @ \text{butlast } ys)$
 $\langle \text{proof} \rangle$

lemma *append-butlast-last-id [simp]*:
 $xs \neq [] \implies \text{butlast } xs @ [\text{last } xs] = xs$
 $\langle \text{proof} \rangle$

lemma *in-set-butlastD*: $x : \text{set } (\text{butlast } xs) \implies x : \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *in-set-butlast-appendI*:
 $x : \text{set } (\text{butlast } xs) \mid x : \text{set } (\text{butlast } ys) \implies x : \text{set } (\text{butlast } (xs @ ys))$
 $\langle \text{proof} \rangle$

lemma *last-drop[simp]*: $n < \text{length } xs \implies \text{last } (\text{drop } n \text{ } xs) = \text{last } xs$
 $\langle \text{proof} \rangle$

lemma *last-conv-nth*: $xs \neq [] \implies \text{last } xs = xs!(\text{length } xs - 1)$
 $\langle \text{proof} \rangle$

lemma *butlast-conv-take*: $\text{butlast } xs = \text{take } (\text{length } xs - 1) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *last-list-update*:
 $xs \neq [] \implies \text{last}(xs[k:=x]) = (\text{if } k = \text{size } xs - 1 \text{ then } x \text{ else } \text{last } xs)$
 $\langle \text{proof} \rangle$

lemma *butlast-list-update*:
 $\text{butlast}(xs[k:=x]) =$
 $(\text{if } k = \text{size } xs - 1 \text{ then } \text{butlast } xs \text{ else } (\text{butlast } xs)[k:=x])$
 $\langle \text{proof} \rangle$

lemma *last-map*:

$xs \neq [] \implies \text{last } (\text{map } f \text{ } xs) = f \text{ } (\text{last } xs)$
 $\langle \text{proof} \rangle$

lemma *map-butlast*:

$\text{map } f \text{ } (\text{butlast } xs) = \text{butlast } (\text{map } f \text{ } xs)$
 $\langle \text{proof} \rangle$

43.1.14 *take and drop*

lemma *take-0* [simp]: $\text{take } 0 \text{ } xs = []$
 $\langle \text{proof} \rangle$

lemma *drop-0* [simp]: $\text{drop } 0 \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *take-Suc-Cons* [simp]: $\text{take } (\text{Suc } n) \text{ } (x \# xs) = x \# \text{take } n \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *drop-Suc-Cons* [simp]: $\text{drop } (\text{Suc } n) \text{ } (x \# xs) = \text{drop } n \text{ } xs$
 $\langle \text{proof} \rangle$

declare *take-Cons* [simp del] **and** *drop-Cons* [simp del]

lemma *take-1-Cons* [simp]: $\text{take } 1 \text{ } (x \# xs) = [x]$
 $\langle \text{proof} \rangle$

lemma *drop-1-Cons* [simp]: $\text{drop } 1 \text{ } (x \# xs) = xs$
 $\langle \text{proof} \rangle$

lemma *take-Suc*: $xs \sim [] \implies \text{take } (\text{Suc } n) \text{ } xs = \text{hd } xs \# \text{take } n \text{ } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *drop-Suc*: $\text{drop } (\text{Suc } n) \text{ } xs = \text{drop } n \text{ } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *take-tl*: $\text{take } n \text{ } (\text{tl } xs) = \text{tl } (\text{take } (\text{Suc } n) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *drop-tl*: $\text{drop } n \text{ } (\text{tl } xs) = \text{tl } (\text{drop } n \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *tl-take*: $\text{tl } (\text{take } n \text{ } xs) = \text{take } (n - 1) \text{ } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *tl-drop*: $\text{tl } (\text{drop } n \text{ } xs) = \text{drop } n \text{ } (\text{tl } xs)$
 $\langle \text{proof} \rangle$

lemma *nth-via-drop*: $\text{drop } n \text{ } xs = y \# ys \implies xs!n = y$
 $\langle \text{proof} \rangle$

lemma *take-Suc-conv-app-nth*:

$i < \text{length } xs \implies \text{take } (\text{Suc } i) \text{ } xs = \text{take } i \text{ } xs @ [xs!i]$
 $\langle \text{proof} \rangle$

lemma *drop-Suc-conv-tl*:

$i < \text{length } xs \implies (xs!i) \# (\text{drop } (\text{Suc } i) \text{ } xs) = \text{drop } i \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *length-take [simp]*: $\text{length } (\text{take } n \text{ } xs) = \min (\text{length } xs) \ n$
 $\langle \text{proof} \rangle$

lemma *length-drop [simp]*: $\text{length } (\text{drop } n \text{ } xs) = (\text{length } xs - n)$
 $\langle \text{proof} \rangle$

lemma *take-all [simp]*: $\text{length } xs \leq n \implies \text{take } n \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *drop-all [simp]*: $\text{length } xs \leq n \implies \text{drop } n \text{ } xs = []$
 $\langle \text{proof} \rangle$

lemma *take-append [simp]*:

$\text{take } n \text{ } (xs @ ys) = (\text{take } n \text{ } xs @ \text{take } (n - \text{length } xs) \text{ } ys)$
 $\langle \text{proof} \rangle$

lemma *drop-append [simp]*:

$\text{drop } n \text{ } (xs @ ys) = \text{drop } n \text{ } xs @ \text{drop } (n - \text{length } xs) \text{ } ys$
 $\langle \text{proof} \rangle$

lemma *take-take [simp]*: $\text{take } n \text{ } (\text{take } m \text{ } xs) = \text{take } (\min n \ m) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *drop-drop [simp]*: $\text{drop } n \text{ } (\text{drop } m \text{ } xs) = \text{drop } (n + m) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *take-drop*: $\text{take } n \text{ } (\text{drop } m \text{ } xs) = \text{drop } m \text{ } (\text{take } (n + m) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *drop-take*: $\text{drop } n \text{ } (\text{take } m \text{ } xs) = \text{take } (m - n) \text{ } (\text{drop } n \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *append-take-drop-id [simp]*: $\text{take } n \text{ } xs @ \text{drop } n \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *take-eq-Nil [simp]*: $(\text{take } n \text{ } xs = []) = (n = 0 \vee xs = [])$
 $\langle \text{proof} \rangle$

lemma *drop-eq-Nil [simp]*: $(\text{drop } n \text{ } xs = []) = (\text{length } xs \leq n)$
 $\langle \text{proof} \rangle$

lemma *take-map*: $\text{take } n \ (\text{map } f \ xs) = \text{map } f \ (\text{take } n \ xs)$
 $\langle \text{proof} \rangle$

lemma *drop-map*: $\text{drop } n \ (\text{map } f \ xs) = \text{map } f \ (\text{drop } n \ xs)$
 $\langle \text{proof} \rangle$

lemma *rev-take*: $\text{rev} \ (\text{take } i \ xs) = \text{drop} \ (\text{length } xs - i) \ (\text{rev } xs)$
 $\langle \text{proof} \rangle$

lemma *rev-drop*: $\text{rev} \ (\text{drop } i \ xs) = \text{take} \ (\text{length } xs - i) \ (\text{rev } xs)$
 $\langle \text{proof} \rangle$

lemma *nth-take* [simp]: $i < n \implies (\text{take } n \ xs)!i = xs!i$
 $\langle \text{proof} \rangle$

lemma *nth-drop* [simp]:
 $n + i \leq \text{length } xs \implies (\text{drop } n \ xs)!i = xs!(n + i)$
 $\langle \text{proof} \rangle$

lemma *butlast-take*:
 $n \leq \text{length } xs \implies \text{butlast} \ (\text{take } n \ xs) = \text{take} \ (n - 1) \ xs$
 $\langle \text{proof} \rangle$

lemma *butlast-drop*: $\text{butlast} \ (\text{drop } n \ xs) = \text{drop } n \ (\text{butlast } xs)$
 $\langle \text{proof} \rangle$

lemma *take-butlast*: $n < \text{length } xs \implies \text{take } n \ (\text{butlast } xs) = \text{take } n \ xs$
 $\langle \text{proof} \rangle$

lemma *drop-butlast*: $\text{drop } n \ (\text{butlast } xs) = \text{butlast} \ (\text{drop } n \ xs)$
 $\langle \text{proof} \rangle$

lemma *hd-drop-conv-nth*: $\llbracket xs \neq []; n < \text{length } xs \rrbracket \implies \text{hd}(\text{drop } n \ xs) = xs!n$
 $\langle \text{proof} \rangle$

lemma *set-take-subset-set-take*:
 $m \leq n \implies \text{set}(\text{take } m \ xs) \subseteq \text{set}(\text{take } n \ xs)$
 $\langle \text{proof} \rangle$

lemma *set-take-subset*: $\text{set}(\text{take } n \ xs) \subseteq \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *set-drop-subset*: $\text{set}(\text{drop } n \ xs) \subseteq \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *set-drop-subset-set-drop*:
 $m > n \implies \text{set}(\text{drop } m \ xs) \subseteq \text{set}(\text{drop } n \ xs)$
 $\langle \text{proof} \rangle$

lemma *in-set-takeD*: $x : \text{set}(\text{take } n \text{ } xs) \implies x : \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *in-set-dropD*: $x : \text{set}(\text{drop } n \text{ } xs) \implies x : \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *append-eq-conv-conj*:
 $(xs @ ys = zs) = (xs = \text{take } (\text{length } xs) \text{ } zs \wedge ys = \text{drop } (\text{length } xs) \text{ } zs)$
 $\langle \text{proof} \rangle$

lemma *take-add*:
 $i+j \leq \text{length}(xs) \implies \text{take } (i+j) \text{ } xs = \text{take } i \text{ } xs @ \text{take } j \text{ } (\text{drop } i \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *append-eq-append-conv-if*:
 $(xs_1 @ xs_2 = ys_1 @ ys_2) =$
 $(\text{if } \text{size } xs_1 \leq \text{size } ys_1$
 $\text{ then } xs_1 = \text{take } (\text{size } xs_1) \text{ } ys_1 \wedge xs_2 = \text{drop } (\text{size } xs_1) \text{ } ys_1 @ ys_2$
 $\text{ else } \text{take } (\text{size } ys_1) \text{ } xs_1 = ys_1 \wedge \text{drop } (\text{size } ys_1) \text{ } xs_1 @ xs_2 = ys_2)$
 $\langle \text{proof} \rangle$

lemma *take-hd-drop*:
 $n < \text{length } xs \implies \text{take } n \text{ } xs @ [\text{hd } (\text{drop } n \text{ } xs)] = \text{take } (\text{Suc } n) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *id-take-nth-drop*:
 $i < \text{length } xs \implies xs = \text{take } i \text{ } xs @ xs!i \# \text{drop } (\text{Suc } i) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *upd-conv-take-nth-drop*:
 $i < \text{length } xs \implies xs[i:=a] = \text{take } i \text{ } xs @ a \# \text{drop } (\text{Suc } i) \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *nth-drop'*:
 $i < \text{length } xs \implies xs ! i \# \text{drop } (\text{Suc } i) \text{ } xs = \text{drop } i \text{ } xs$
 $\langle \text{proof} \rangle$

43.1.15 *takeWhile* and *dropWhile*

lemma *length-takeWhile-le*: $\text{length } (\text{takeWhile } P \text{ } xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-dropWhile-id* [simp]: $\text{takeWhile } P \text{ } xs @ \text{dropWhile } P \text{ } xs = xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-append1* [simp]:
 $[| x:\text{set } xs; \sim P(x)|] \implies \text{takeWhile } P \text{ } (xs @ ys) = \text{takeWhile } P \text{ } xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-append2* [simp]:

$(!!x. x : \text{set } xs \implies P x) \implies \text{takeWhile } P (xs @ ys) = xs @ \text{takeWhile } P ys$
 $\langle \text{proof} \rangle$

lemma *takeWhile-tail*: $\neg P x \implies \text{takeWhile } P (xs @ (x \# l)) = \text{takeWhile } P xs$
 $\langle \text{proof} \rangle$

lemma *takeWhile-nth*: $j < \text{length } (\text{takeWhile } P xs) \implies \text{takeWhile } P xs ! j = xs ! j$
 $\langle \text{proof} \rangle$

lemma *dropWhile-nth*: $j < \text{length } (\text{dropWhile } P xs) \implies \text{dropWhile } P xs ! j = xs ! (j + \text{length } (\text{takeWhile } P xs))$
 $\langle \text{proof} \rangle$

lemma *length-dropWhile-le*: $\text{length } (\text{dropWhile } P xs) \leq \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *dropWhile-append1* [simp]:

$[| x : \text{set } xs; \sim P(x) |] \implies \text{dropWhile } P (xs @ ys) = (\text{dropWhile } P xs) @ ys$
 $\langle \text{proof} \rangle$

lemma *dropWhile-append2* [simp]:

$(!!x. x : \text{set } xs \implies P(x)) \implies \text{dropWhile } P (xs @ ys) = \text{dropWhile } P ys$
 $\langle \text{proof} \rangle$

lemma *set-takeWhileD*: $x : \text{set } (\text{takeWhile } P xs) \implies x : \text{set } xs \wedge P x$
 $\langle \text{proof} \rangle$

lemma *takeWhile-eq-all-conv*[simp]:

$(\text{takeWhile } P xs = xs) = (\forall x \in \text{set } xs. P x)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-eq-Nil-conv*[simp]:

$(\text{dropWhile } P xs = []) = (\forall x \in \text{set } xs. P x)$
 $\langle \text{proof} \rangle$

lemma *dropWhile-eq-Cons-conv*:

$(\text{dropWhile } P xs = y \# ys) = (xs = \text{takeWhile } P xs @ y \# ys \ \& \ \neg P y)$
 $\langle \text{proof} \rangle$

lemma *distinct-takeWhile*[simp]: $\text{distinct } xs \implies \text{distinct } (\text{takeWhile } P xs)$
 $\langle \text{proof} \rangle$

lemma *distinct-dropWhile*[simp]: $\text{distinct } xs \implies \text{distinct } (\text{dropWhile } P xs)$
 $\langle \text{proof} \rangle$

lemma *takeWhile-map*: $\text{takeWhile } P (\text{map } f xs) = \text{map } f (\text{takeWhile } (P \circ f) xs)$

$\langle proof \rangle$

lemma *dropWhile-map*: $dropWhile\ P\ (map\ f\ xs) = map\ f\ (dropWhile\ (P \circ f)\ xs)$
 $\langle proof \rangle$

lemma *takeWhile-eq-take*: $takeWhile\ P\ xs = take\ (length\ (takeWhile\ P\ xs))\ xs$
 $\langle proof \rangle$

lemma *dropWhile-eq-drop*: $dropWhile\ P\ xs = drop\ (length\ (takeWhile\ P\ xs))\ xs$
 $\langle proof \rangle$

lemma *hd-dropWhile*:
 $dropWhile\ P\ xs \neq [] \implies \neg P\ (hd\ (dropWhile\ P\ xs))$
 $\langle proof \rangle$

lemma *takeWhile-eq-filter*:
assumes $\bigwedge x. x \in set\ (dropWhile\ P\ xs) \implies \neg P\ x$
shows $takeWhile\ P\ xs = filter\ P\ xs$
 $\langle proof \rangle$

lemma *takeWhile-eq-take-P-nth*:
 $\llbracket \bigwedge i. \llbracket i < n ; i < length\ xs \rrbracket \implies P\ (xs\ !\ i) ; n < length\ xs \implies \neg P\ (xs\ !\ n) \rrbracket \implies$
 $takeWhile\ P\ xs = take\ n\ xs$
 $\langle proof \rangle$

lemma *nth-length-takeWhile*:
 $length\ (takeWhile\ P\ xs) < length\ xs \implies \neg P\ (xs\ !\ length\ (takeWhile\ P\ xs))$
 $\langle proof \rangle$

lemma *length-takeWhile-less-P-nth*:
assumes $all: \bigwedge i. i < j \implies P\ (xs\ !\ i)$ **and** $j \leq length\ xs$
shows $j \leq length\ (takeWhile\ P\ xs)$
 $\langle proof \rangle$

The following two lemmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: $\llbracket distinct\ xs ; x \in set\ xs \rrbracket \implies$
 $takeWhile\ (\lambda y. y \neq x)\ (rev\ xs) = rev\ (tl\ (dropWhile\ (\lambda y. y \neq x)\ xs))$
 $\langle proof \rangle$

lemma *dropWhile-neq-rev*: $\llbracket distinct\ xs ; x \in set\ xs \rrbracket \implies$
 $dropWhile\ (\lambda y. y \neq x)\ (rev\ xs) = x \# rev\ (takeWhile\ (\lambda y. y \neq x)\ xs)$
 $\langle proof \rangle$

lemma *takeWhile-not-last*:
 $\llbracket xs \neq [] ; distinct\ xs \rrbracket \implies takeWhile\ (\lambda y. y \neq last\ xs)\ xs = butlast\ xs$
 $\langle proof \rangle$

lemma *takeWhile-cong* [*fundef-cong*, *recdef-cong*]:

$$\begin{aligned} & [| l = k; !!x. x : \text{set } l ==> P\ x = Q\ x \ |] \\ & ==> \text{takeWhile } P\ l = \text{takeWhile } Q\ k \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *dropWhile-cong* [*fundef-cong*, *recdef-cong*]:

$$\begin{aligned} & [| l = k; !!x. x : \text{set } l ==> P\ x = Q\ x \ |] \\ & ==> \text{dropWhile } P\ l = \text{dropWhile } Q\ k \\ & \langle \text{proof} \rangle \end{aligned}$$

43.1.16 *zip*

lemma *zip-Nil* [*simp*]: $\text{zip } []\ ys = []$
 $\langle \text{proof} \rangle$

lemma *zip-Cons-Cons* [*simp*]: $\text{zip } (x \# xs)\ (y \# ys) = (x, y) \# \text{zip } xs\ ys$
 $\langle \text{proof} \rangle$

declare *zip-Cons* [*simp del*]

lemma [*code*]:

$$\begin{aligned} & \text{zip } []\ ys = [] \\ & \text{zip } xs\ [] = [] \\ & \text{zip } (x \# xs)\ (y \# ys) = (x, y) \# \text{zip } xs\ ys \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *zip-Cons1*:

$$\text{zip } (x \# xs)\ ys = (\text{case } ys \text{ of } [] \Rightarrow [] \mid y \# ys \Rightarrow (x, y) \# \text{zip } xs\ ys)$$

 $\langle \text{proof} \rangle$

lemma *length-zip* [*simp*]:

$$\text{length } (\text{zip } xs\ ys) = \min (\text{length } xs)\ (\text{length } ys)$$

 $\langle \text{proof} \rangle$

lemma *zip-obtain-same-length*:

$$\begin{aligned} & \text{assumes } \bigwedge zs\ ws\ n. \text{length } zs = \text{length } ws \implies n = \min (\text{length } xs)\ (\text{length } ys) \\ & \implies zs = \text{take } n\ xs \implies ws = \text{take } n\ ys \implies P\ (\text{zip } zs\ ws) \\ & \text{shows } P\ (\text{zip } xs\ ys) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *zip-append1*:

$$\begin{aligned} & \text{zip } (xs\ @\ ys)\ zs = \\ & \text{zip } xs\ (\text{take } (\text{length } xs)\ zs) @ \text{zip } ys\ (\text{drop } (\text{length } xs)\ zs) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *zip-append2*:

$$\begin{aligned} & \text{zip } xs\ (ys\ @\ zs) = \\ & \text{zip } (\text{take } (\text{length } ys)\ xs)\ ys @ \text{zip } (\text{drop } (\text{length } ys)\ xs)\ zs \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *zip-append* [simp]:

$[[\text{length } xs = \text{length } us; \text{length } ys = \text{length } vs]] ==>$
 $\text{zip } (xs@ys) (us@vs) = \text{zip } xs\ us\ @\ \text{zip } ys\ vs$
 $\langle \text{proof} \rangle$

lemma *zip-rev*:

$\text{length } xs = \text{length } ys ==> \text{zip } (\text{rev } xs) (\text{rev } ys) = \text{rev } (\text{zip } xs\ ys)$
 $\langle \text{proof} \rangle$

lemma *zip-map-map*:

$\text{zip } (\text{map } f\ xs) (\text{map } g\ ys) = \text{map } (\lambda (x, y). (f\ x, g\ y)) (\text{zip } xs\ ys)$
 $\langle \text{proof} \rangle$

lemma *zip-map1*:

$\text{zip } (\text{map } f\ xs)\ ys = \text{map } (\lambda(x, y). (f\ x, y)) (\text{zip } xs\ ys)$
 $\langle \text{proof} \rangle$

lemma *zip-map2*:

$\text{zip } xs\ (\text{map } f\ ys) = \text{map } (\lambda(x, y). (x, f\ y)) (\text{zip } xs\ ys)$
 $\langle \text{proof} \rangle$

lemma *map-zip-map*:

$\text{map } f\ (\text{zip } (\text{map } g\ xs)\ ys) = \text{map } (\%(x,y). f(g\ x, y)) (\text{zip } xs\ ys)$
 $\langle \text{proof} \rangle$

lemma *map-zip-map2*:

$\text{map } f\ (\text{zip } xs\ (\text{map } g\ ys)) = \text{map } (\%(x,y). f(x, g\ y)) (\text{zip } xs\ ys)$
 $\langle \text{proof} \rangle$

Courtesy of Andreas Lochbihler:

lemma *zip-same-conv-map*: $\text{zip } xs\ xs = \text{map } (\lambda x. (x, x))\ xs$
 $\langle \text{proof} \rangle$

lemma *nth-zip* [simp]:

$[[i < \text{length } xs; i < \text{length } ys]] ==> (\text{zip } xs\ ys)!i = (xs!i, ys!i)$
 $\langle \text{proof} \rangle$

lemma *set-zip*:

$\text{set } (\text{zip } xs\ ys) = \{(xs!i, ys!i) \mid i. i < \min (\text{length } xs) (\text{length } ys)\}$
 $\langle \text{proof} \rangle$

lemma *zip-same*: $((a,b) \in \text{set } (\text{zip } xs\ xs)) = (a \in \text{set } xs \wedge a = b)$

$\langle \text{proof} \rangle$

lemma *zip-update*:

$\text{zip } (xs[i:=x])\ (ys[i:=y]) = (\text{zip } xs\ ys)[i:=(x,y)]$
 $\langle \text{proof} \rangle$

lemma *zip-replicate* [simp]:

$zip\ (replicate\ i\ x)\ (replicate\ j\ y) = replicate\ (\min\ i\ j)\ (x,y)$
 $\langle proof \rangle$

lemma *take-zip*:
 $take\ n\ (zip\ xs\ ys) = zip\ (take\ n\ xs)\ (take\ n\ ys)$
 $\langle proof \rangle$

lemma *drop-zip*:
 $drop\ n\ (zip\ xs\ ys) = zip\ (drop\ n\ xs)\ (drop\ n\ ys)$
 $\langle proof \rangle$

lemma *zip-takeWhile-fst*: $zip\ (takeWhile\ P\ xs)\ ys = takeWhile\ (P \circ fst)\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *zip-takeWhile-snd*: $zip\ xs\ (takeWhile\ P\ ys) = takeWhile\ (P \circ snd)\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *set-zip-leftD*:
 $(x,y) \in set\ (zip\ xs\ ys) \implies x \in set\ xs$
 $\langle proof \rangle$

lemma *set-zip-rightD*:
 $(x,y) \in set\ (zip\ xs\ ys) \implies y \in set\ ys$
 $\langle proof \rangle$

lemma *in-set-zipE*:
 $(x,y) : set\ (zip\ xs\ ys) \implies ([\ x : set\ xs;\ y : set\ ys\] \implies R) \implies R$
 $\langle proof \rangle$

lemma *zip-map-fst-snd*:
 $zip\ (map\ fst\ zs)\ (map\ snd\ zs) = zs$
 $\langle proof \rangle$

lemma *zip-eq-conv*:
 $length\ xs = length\ ys \implies zip\ xs\ ys = zs \iff map\ fst\ zs = xs \wedge map\ snd\ zs = ys$
 $\langle proof \rangle$

lemma *distinct-zipI1*:
 $distinct\ xs \implies distinct\ (zip\ xs\ ys)$
 $\langle proof \rangle$

lemma *distinct-zipI2*:
 $distinct\ xs \implies distinct\ (zip\ xs\ ys)$
 $\langle proof \rangle$

43.1.17 *list-all2***lemma** *list-all2-lengthD* [intro?]:

$$\text{list-all2 } P \text{ } xs \text{ } ys ==> \text{length } xs = \text{length } ys$$

*<proof>***lemma** *list-all2-Nil* [iff, code]: *list-all2* *P* [] *ys* = (*ys* = [])*<proof>***lemma** *list-all2-Nil2* [iff, code]: *list-all2* *P* *xs* [] = (*xs* = [])*<proof>***lemma** *list-all2-Cons* [iff, code]:

$$\text{list-all2 } P \text{ } (x \# xs) \text{ } (y \# ys) = (P \text{ } x \text{ } y \wedge \text{list-all2 } P \text{ } xs \text{ } ys)$$

*<proof>***lemma** *list-all2-Cons1*:

$$\text{list-all2 } P \text{ } (x \# xs) \text{ } ys = (\exists z \text{ } zs. \text{ } ys = z \# zs \wedge P \text{ } x \text{ } z \wedge \text{list-all2 } P \text{ } xs \text{ } zs)$$

*<proof>***lemma** *list-all2-Cons2*:

$$\text{list-all2 } P \text{ } xs \text{ } (y \# ys) = (\exists z \text{ } zs. \text{ } xs = z \# zs \wedge P \text{ } z \text{ } y \wedge \text{list-all2 } P \text{ } zs \text{ } ys)$$

*<proof>***lemma** *list-all2-rev* [iff]:

$$\text{list-all2 } P \text{ } (\text{rev } xs) \text{ } (\text{rev } ys) = \text{list-all2 } P \text{ } xs \text{ } ys$$

*<proof>***lemma** *list-all2-rev1*:

$$\text{list-all2 } P \text{ } (\text{rev } xs) \text{ } ys = \text{list-all2 } P \text{ } xs \text{ } (\text{rev } ys)$$

*<proof>***lemma** *list-all2-append1*:

$$\text{list-all2 } P \text{ } (xs @ ys) \text{ } zs =$$

$$(EX \text{ } us \text{ } vs. \text{ } zs = us @ vs \wedge \text{length } us = \text{length } xs \wedge \text{length } vs = \text{length } ys \wedge$$

$$\text{list-all2 } P \text{ } xs \text{ } us \wedge \text{list-all2 } P \text{ } ys \text{ } vs)$$

*<proof>***lemma** *list-all2-append2*:

$$\text{list-all2 } P \text{ } xs \text{ } (ys @ zs) =$$

$$(EX \text{ } us \text{ } vs. \text{ } xs = us @ vs \wedge \text{length } us = \text{length } ys \wedge \text{length } vs = \text{length } zs \wedge$$

$$\text{list-all2 } P \text{ } us \text{ } ys \wedge \text{list-all2 } P \text{ } vs \text{ } zs)$$

*<proof>***lemma** *list-all2-append*:

$$\text{length } xs = \text{length } ys \implies$$

$$\text{list-all2 } P \text{ } (xs @ us) \text{ } (ys @ vs) = (\text{list-all2 } P \text{ } xs \text{ } ys \wedge \text{list-all2 } P \text{ } us \text{ } vs)$$

*<proof>***lemma** *list-all2-appendI* [intro?, trans]:

$\llbracket \text{list-all2 } P \ a \ b; \text{list-all2 } P \ c \ d \rrbracket \Longrightarrow \text{list-all2 } P \ (a @ c) \ (b @ d)$
 $\langle \text{proof} \rangle$

lemma *list-all2-conv-all-nth*:

$\text{list-all2 } P \ xs \ ys =$
 $(\text{length } xs = \text{length } ys \wedge (\forall i < \text{length } xs. P \ (xs!i) \ (ys!i)))$
 $\langle \text{proof} \rangle$

lemma *list-all2-trans*:

assumes $tr: !!a \ b \ c. P1 \ a \ b \Longrightarrow P2 \ b \ c \Longrightarrow P3 \ a \ c$
shows $!!bs \ cs. \text{list-all2 } P1 \ as \ bs \Longrightarrow \text{list-all2 } P2 \ bs \ cs \Longrightarrow \text{list-all2 } P3 \ as \ cs$
 $(\text{is } !!bs \ cs. \text{PROP } ?Q \ as \ bs \ cs)$
 $\langle \text{proof} \rangle$

lemma *list-all2-all-nthI* $[intro?]$:

$\text{length } a = \text{length } b \Longrightarrow (\bigwedge n. n < \text{length } a \Longrightarrow P \ (a!n) \ (b!n)) \Longrightarrow \text{list-all2 } P \ a \ b$
 $\langle \text{proof} \rangle$

lemma *list-all2I*:

$\forall x \in \text{set } (\text{zip } a \ b). \text{split } P \ x \Longrightarrow \text{length } a = \text{length } b \Longrightarrow \text{list-all2 } P \ a \ b$
 $\langle \text{proof} \rangle$

lemma *list-all2-nthD*:

$\llbracket \text{list-all2 } P \ xs \ ys; \ p < \text{size } xs \rrbracket \Longrightarrow P \ (xs!p) \ (ys!p)$
 $\langle \text{proof} \rangle$

lemma *list-all2-nthD2*:

$\llbracket \text{list-all2 } P \ xs \ ys; \ p < \text{size } ys \rrbracket \Longrightarrow P \ (xs!p) \ (ys!p)$
 $\langle \text{proof} \rangle$

lemma *list-all2-map1*:

$\text{list-all2 } P \ (\text{map } f \ as) \ bs = \text{list-all2 } (\lambda x \ y. P \ (f \ x) \ y) \ as \ bs$
 $\langle \text{proof} \rangle$

lemma *list-all2-map2*:

$\text{list-all2 } P \ as \ (\text{map } f \ bs) = \text{list-all2 } (\lambda x \ y. P \ x \ (f \ y)) \ as \ bs$
 $\langle \text{proof} \rangle$

lemma *list-all2-refl* $[intro?]$:

$(\bigwedge x. P \ x \ x) \Longrightarrow \text{list-all2 } P \ xs \ xs$
 $\langle \text{proof} \rangle$

lemma *list-all2-update-cong*:

$\llbracket i < \text{size } xs; \text{list-all2 } P \ xs \ ys; \ P \ x \ y \rrbracket \Longrightarrow \text{list-all2 } P \ (xs[i:=x]) \ (ys[i:=y])$
 $\langle \text{proof} \rangle$

lemma *list-all2-update-cong2*:

$\llbracket \text{list-all2 } P \ xs \ ys; \ P \ x \ y; \ i < \text{length } ys \rrbracket \Longrightarrow \text{list-all2 } P \ (xs[i:=x]) \ (ys[i:=y])$
 $\langle \text{proof} \rangle$

lemma *list-all2-takeI* [*simp,intro?*]:

$list-all2\ P\ xs\ ys \implies list-all2\ P\ (take\ n\ xs)\ (take\ n\ ys)$
 $\langle proof \rangle$

lemma *list-all2-dropI* [*simp,intro?*]:

$list-all2\ P\ as\ bs \implies list-all2\ P\ (drop\ n\ as)\ (drop\ n\ bs)$
 $\langle proof \rangle$

lemma *list-all2-mono* [*intro?*]:

$list-all2\ P\ xs\ ys \implies (\bigwedge xs\ ys. P\ xs\ ys \implies Q\ xs\ ys) \implies list-all2\ Q\ xs\ ys$
 $\langle proof \rangle$

lemma *list-all2-eq*:

$xs = ys \iff list-all2\ (op =)\ xs\ ys$
 $\langle proof \rangle$

43.1.18 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:

$foldl\ f\ a\ (xs\ @\ ys) = foldl\ f\ (foldl\ f\ a\ xs)\ ys$
 $\langle proof \rangle$

lemma *foldr-append*[*simp*]: $foldr\ f\ (xs\ @\ ys)\ a = foldr\ f\ xs\ (foldr\ f\ ys\ a)$

$\langle proof \rangle$

lemma *foldr-map*: $foldr\ g\ (map\ f\ xs)\ a = foldr\ (g\ o\ f)\ xs\ a$

$\langle proof \rangle$

For efficient code generation: avoid intermediate list.

lemma *foldl-map*[*code-unfold*]:

$foldl\ g\ a\ (map\ f\ xs) = foldl\ (\%a\ x. g\ a\ (f\ x))\ a\ xs$
 $\langle proof \rangle$

lemma *foldl-apply*:

assumes $\bigwedge x. x \in set\ xs \implies f\ x \circ h = h \circ g\ x$
shows $foldl\ (\lambda s\ x. f\ x\ s)\ (h\ s)\ xs = h\ (foldl\ (\lambda s\ x. g\ x\ s)\ s\ xs)$
 $\langle proof \rangle$

lemma *foldl-cong* [*fundef-cong, recdef-cong*]:

$[| a = b; l = k; !!a\ x. x : set\ l ==> f\ a\ x = g\ a\ x |]$
 $==> foldl\ f\ a\ l = foldl\ g\ b\ k$
 $\langle proof \rangle$

lemma *foldr-cong* [*fundef-cong, recdef-cong*]:

$[| a = b; l = k; !!a\ x. x : set\ l ==> f\ x\ a = g\ x\ a |]$
 $==> foldr\ f\ l\ a = foldr\ g\ k\ b$
 $\langle proof \rangle$

lemma *foldl-fun-comm*:

assumes $\bigwedge x y s. f (f s x) y = f (f s y) x$
shows $f (foldl f s xs) x = foldl f (f s x) xs$
 $\langle proof \rangle$

lemma (**in** *semigroup-add*) *foldl-assoc*:

shows $foldl op + (x+y) zs = x + (foldl op + y zs)$
 $\langle proof \rangle$

lemma (**in** *monoid-add*) *foldl-absorb0*:

shows $x + (foldl op + 0 zs) = foldl op + x zs$
 $\langle proof \rangle$

lemma *foldl-rev*:

assumes $\bigwedge x y s. f (f s x) y = f (f s y) x$
shows $foldl f s (rev xs) = foldl f s xs$
 $\langle proof \rangle$

The “First Duality Theorem” in Bird & Wadler:

lemma *foldl-foldr1-lemma*:

$foldl op + a xs = a + foldr op + xs (0::'a::monoid-add)$
 $\langle proof \rangle$

corollary *foldl-foldr1*:

$foldl op + 0 xs = foldr op + xs (0::'a::monoid-add)$
 $\langle proof \rangle$

The “Third Duality Theorem” in Bird & Wadler:

lemma *foldr-foldl*: $foldr f xs a = foldl (\%x y. f y x) a (rev xs)$
 $\langle proof \rangle$

lemma *foldl-foldr*: $foldl f a xs = foldr (\%x y. f y x) (rev xs) a$
 $\langle proof \rangle$

lemma (**in** *ab-semigroup-add*) *foldr-conv-foldl*: $foldr op + xs a = foldl op + a xs$
 $\langle proof \rangle$

Note: $n \leq foldl (op +) n ns$ looks simpler, but is more difficult to use because it requires an additional transitivity step.

lemma *start-le-sum*: $(m::nat) \leq n \implies m \leq foldl (op +) n ns$
 $\langle proof \rangle$

lemma *elem-le-sum*: $(n::nat) : set ns \implies n \leq foldl (op +) 0 ns$
 $\langle proof \rangle$

lemma *sum-eq-0-conv* [iff]:

$(foldl (op +) (m::nat) ns = 0) = (m = 0 \wedge (\forall n \in set ns. n = 0))$
 $\langle proof \rangle$

lemma *foldr-invariant*:

$\llbracket Q\ x ; \forall\ x \in \text{set}\ xs.\ P\ x ; \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ x\ y) \rrbracket \Longrightarrow Q\ (\text{foldr}\ f\ xs\ x)$
 $\langle \text{proof} \rangle$

lemma *foldl-invariant*:

$\llbracket Q\ x ; \forall\ x \in \text{set}\ xs.\ P\ x ; \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ y\ x) \rrbracket \Longrightarrow Q\ (\text{foldl}\ f\ x\ xs)$
 $\langle \text{proof} \rangle$

lemma *foldl-weak-invariant*:

assumes $P\ s$
and $\bigwedge s\ x.\ x \in \text{set}\ xs \Longrightarrow P\ s \Longrightarrow P\ (f\ s\ x)$
shows $P\ (\text{foldl}\ f\ s\ xs)$
 $\langle \text{proof} \rangle$

foldl and *concat*

lemma *foldl-conv-concat*:

$\text{foldl}\ (op\ @)\ xs\ xss = xs\ @\ \text{concat}\ xss$
 $\langle \text{proof} \rangle$

lemma *concat-conv-foldl*: $\text{concat}\ xss = \text{foldl}\ (op\ @)\ []\ xss$

$\langle \text{proof} \rangle$

fold and *foldl*

lemma (**in** *fun-left-comm*) *fold-set-remdups*:

$\text{fold}\ f\ y\ (\text{set}\ xs) = \text{foldl}\ (\lambda y\ x.\ f\ x\ y)\ y\ (\text{remdups}\ xs)$
 $\langle \text{proof} \rangle$

lemma (**in** *fun-left-comm-idem*) *fold-set*:

$\text{fold}\ f\ y\ (\text{set}\ xs) = \text{foldl}\ (\lambda y\ x.\ f\ x\ y)\ y\ xs$
 $\langle \text{proof} \rangle$

lemma (**in** *ab-semigroup-idem-mult*) *fold1-set*:

assumes $xs \neq []$
shows $\text{fold1}\ \text{times}\ (\text{set}\ xs) = \text{foldl}\ \text{times}\ (\text{hd}\ xs)\ (\text{tl}\ xs)$
 $\langle \text{proof} \rangle$

lemma (**in** *lattice*) *Inf-fin-set-fold* [code-unfold]:

$\text{Inf-fin}\ (\text{set}\ (x\ \# \ xs)) = \text{foldl}\ \text{inf}\ x\ xs$
 $\langle \text{proof} \rangle$

lemma (**in** *lattice*) *Sup-fin-set-fold* [code-unfold]:

$\text{Sup-fin}\ (\text{set}\ (x\ \# \ xs)) = \text{foldl}\ \text{sup}\ x\ xs$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *Min-fin-set-fold* [code-unfold]:

$\text{Min}\ (\text{set}\ (x\ \# \ xs)) = \text{foldl}\ \text{min}\ x\ xs$
 $\langle \text{proof} \rangle$

lemma (**in** *linorder*) *Max-fin-set-fold* [code-unfold]:

$Max (set (x \# xs)) = foldl\ max\ x\ xs$
 $\langle proof \rangle$

lemma (in complete-lattice) *Inf-set-fold* [code-unfold]:
 $Inf (set\ xs) = foldl\ inf\ top\ xs$
 $\langle proof \rangle$

lemma (in complete-lattice) *Sup-set-fold* [code-unfold]:
 $Sup (set\ xs) = foldl\ sup\ bot\ xs$
 $\langle proof \rangle$

lemma (in complete-lattice) *INFI-set-fold*:
 $INFI (set\ xs)\ f = foldl\ (\lambda y\ x.\ inf\ (f\ x)\ y)\ top\ xs$
 $\langle proof \rangle$

lemma (in complete-lattice) *SUPR-set-fold*:
 $SUPR (set\ xs)\ f = foldl\ (\lambda y\ x.\ sup\ (f\ x)\ y)\ bot\ xs$
 $\langle proof \rangle$

43.1.19 List summation: *listsum* and \sum

lemma *listsum-append* [simp]: $listsum\ (xs\ @\ ys) = listsum\ xs + listsum\ ys$
 $\langle proof \rangle$

lemma *listsum-rev* [simp]:
fixes $xs :: 'a::comm-monoid-add\ list$
shows $listsum\ (rev\ xs) = listsum\ xs$
 $\langle proof \rangle$

lemma *listsum-map-remove1*:
fixes $f :: 'a \Rightarrow ('b::comm-monoid-add)$
shows $x : set\ xs \Longrightarrow listsum(map\ f\ xs) = f\ x + listsum(map\ f\ (remove1\ x\ xs))$
 $\langle proof \rangle$

lemma *list-size-conv-listsum*:
 $list-size\ f\ xs = listsum\ (map\ f\ xs) + size\ xs$
 $\langle proof \rangle$

lemma *listsum-foldr*: $listsum\ xs = foldr\ (op\ +)\ xs\ 0$
 $\langle proof \rangle$

lemma *length-concat*: $length\ (concat\ xss) = listsum\ (map\ length\ xss)$
 $\langle proof \rangle$

lemma *listsum-map-filter*:
fixes $f :: 'a \Rightarrow 'b::comm-monoid-add$
assumes $\bigwedge x. \llbracket x \in set\ xs ; \neg P\ x \rrbracket \Longrightarrow f\ x = 0$
shows $listsum\ (map\ f\ (filter\ P\ xs)) = listsum\ (map\ f\ xs)$
 $\langle proof \rangle$

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

lemma *listsum[code-unfold]*: $\text{listsum } xs = \text{foldl } (op \ +) \ 0 \ xs$
 $\langle \text{proof} \rangle$

lemma *distinct-listsum-conv-Setsum*:
 $\text{distinct } xs \implies \text{listsum } xs = \text{Setsum}(\text{set } xs)$
 $\langle \text{proof} \rangle$

lemma *listsum-eq-0-nat-iff-nat[simp]*:
 $\text{listsum } ns = (0::nat) \iff (\forall \ n \in \text{set } ns. \ n = 0)$
 $\langle \text{proof} \rangle$

lemma *elem-le-listsum-nat*: $k < \text{size } ns \implies ns!k \leq \text{listsum}(ns::nat \text{ list})$
 $\langle \text{proof} \rangle$

lemma *listsum-update-nat*: $k < \text{size } ns \implies$
 $\text{listsum } (ns[k := (n::nat)]) = \text{listsum } ns + n - ns!k$
 $\langle \text{proof} \rangle$

Some syntactic sugar for summing a function over a list:

syntax
 $\text{-listsum} :: \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \quad ((\mathcal{S}UM \text{ } \leftarrow \text{ } \text{ } \text{ }) [0, 51, 10] \ 10)$
syntax (*xsymbols*)
 $\text{-listsum} :: \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \quad ((\mathcal{S}\sum \text{ } \leftarrow \text{ } \text{ } \text{ }) [0, 51, 10] \ 10)$
syntax (*HTML output*)
 $\text{-listsum} :: \text{pttrn} \Rightarrow 'a \text{ list} \Rightarrow 'b \Rightarrow 'b \quad ((\mathcal{S}\sum \text{ } \leftarrow \text{ } \text{ } \text{ }) [0, 51, 10] \ 10)$

translations — Beware of argument permutation!
 $SUM \ x \leftarrow xs. \ b == CONST \ \text{listsum} \ (CONST \ \text{map} \ (\%x. \ b) \ xs)$
 $\sum \ x \leftarrow xs. \ b == CONST \ \text{listsum} \ (CONST \ \text{map} \ (\%x. \ b) \ xs)$

lemma *listsum-triv*: $(\sum \ x \leftarrow xs. \ r) = \text{of-nat } (\text{length } xs) * r$
 $\langle \text{proof} \rangle$

lemma *listsum-0 [simp]*: $(\sum \ x \leftarrow xs. \ 0) = 0$
 $\langle \text{proof} \rangle$

For non-Abelian groups *xs* needs to be reversed on one side:

lemma *uminus-listsum-map*:
fixes $f :: 'a \Rightarrow 'b::\text{ab-group-add}$
shows $-\ \text{listsum} \ (\text{map } f \ xs) = (\text{listsum} \ (\text{map} \ (\text{uminus } o \ f) \ xs))$
 $\langle \text{proof} \rangle$

lemma *listsum-addf*:
fixes $f \ g :: 'a \Rightarrow 'b::\text{comm-monoid-add}$
shows $(\sum \ x \leftarrow xs. \ f \ x + g \ x) = \text{listsum} \ (\text{map } f \ xs) + \text{listsum} \ (\text{map } g \ xs)$
 $\langle \text{proof} \rangle$

lemma *listsum-subtractf*:

fixes $f\ g :: 'a \Rightarrow 'b::ab\text{-group-add}$
shows $(\sum x \leftarrow xs. f\ x - g\ x) = listsum\ (map\ f\ xs) - listsum\ (map\ g\ xs)$
 $\langle proof \rangle$

lemma *listsum-const-mult*:
fixes $f :: 'a \Rightarrow 'b::semiring-0$
shows $(\sum x \leftarrow xs. c * f\ x) = c * (\sum x \leftarrow xs. f\ x)$
 $\langle proof \rangle$

lemma *listsum-mult-const*:
fixes $f :: 'a \Rightarrow 'b::semiring-0$
shows $(\sum x \leftarrow xs. f\ x * c) = (\sum x \leftarrow xs. f\ x) * c$
 $\langle proof \rangle$

lemma *listsum-abs*:
fixes $xs :: 'a::ordered\text{-ab-group-add-abs}\ list$
shows $|listsum\ xs| \leq listsum\ (map\ abs\ xs)$
 $\langle proof \rangle$

lemma *listsum-mono*:
fixes $f\ g :: 'a \Rightarrow 'b::\{comm\text{-monoid-add}, ordered\text{-ab-semigroup-add}\}$
shows $(\bigwedge x. x \in set\ xs \implies f\ x \leq g\ x) \implies (\sum x \leftarrow xs. f\ x) \leq (\sum x \leftarrow xs. g\ x)$
 $\langle proof \rangle$

43.1.20 *upt*

lemma *upt-rec[code]*: $[i..<j] = (if\ i < j\ then\ i \# [Suc\ i..<j]\ else\ [])$
— *simp* does not terminate!
 $\langle proof \rangle$

lemmas *upt-rec-number-of[simp]* = *upt-rec[of number-of m number-of n, standard]*

lemma *upt-conv-Nil [simp]*: $j \leq i \implies [i..<j] = []$
 $\langle proof \rangle$

lemma *upt-eq-Nil-conv[simp]*: $([i..<j] = []) = (j = 0 \vee j \leq i)$
 $\langle proof \rangle$

lemma *upt-eq-Cons-conv*:
 $([i..<j] = x \# xs) = (i < j \ \&\ i = x \ \&\ [i+1..<j] = xs)$
 $\langle proof \rangle$

lemma *upt-Suc-append*: $i \leq j \implies [i..<(Suc\ j)] = [i..<j] @ [j]$
— Only needed if *upt-Suc* is deleted from the simpset.
 $\langle proof \rangle$

lemma *upt-conv-Cons*: $i < j \implies [i..<j] = i \# [Suc\ i..<j]$
 $\langle proof \rangle$

lemma *upt-add-eq-append*: $i \leq j \implies [i..<j+k] = [i..<j] @ [j..<j+k]$
 — LOOPS as a simprule, since $j \leq j$.
 $\langle proof \rangle$

lemma *length-upt [simp]*: $length [i..<j] = j - i$
 $\langle proof \rangle$

lemma *nth-upt [simp]*: $i + k < j \implies [i..<j] ! k = i + k$
 $\langle proof \rangle$

lemma *hd-upt [simp]*: $i < j \implies hd [i..<j] = i$
 $\langle proof \rangle$

lemma *last-upt [simp]*: $i < j \implies last [i..<j] = j - 1$
 $\langle proof \rangle$

lemma *take-upt [simp]*: $i+m \leq n \implies take\ m\ [i..<n] = [i..<i+m]$
 $\langle proof \rangle$

lemma *drop-upt [simp]*: $drop\ m\ [i..<j] = [i+m..<j]$
 $\langle proof \rangle$

lemma *map-Suc-upt*: $map\ Suc\ [m..<n] = [Suc\ m..<Suc\ n]$
 $\langle proof \rangle$

lemma *nth-map-upt*: $i < n-m \implies (map\ f\ [m..<n]) ! i = f(m+i)$
 $\langle proof \rangle$

lemma *nth-take-lemma*:
 $k \leq length\ xs \implies k \leq length\ ys \implies$
 $(!i. i < k \longrightarrow xs!i = ys!i) \implies take\ k\ xs = take\ k\ ys$
 $\langle proof \rangle$

lemma *nth-equalityI*:
 $[| length\ xs = length\ ys; ALL\ i < length\ xs. xs!i = ys!i |] \implies xs = ys$
 $\langle proof \rangle$

lemma *map-nth*:
 $map\ (\lambda i. xs ! i)\ [0..<length\ xs] = xs$
 $\langle proof \rangle$

lemma *list-all2-antisym*:
 $[| (\bigwedge x\ y. [P\ x\ y; Q\ y\ x] \implies x = y); list-all2\ P\ xs\ ys; list-all2\ Q\ ys\ xs |]$
 $\implies xs = ys$
 $\langle proof \rangle$

lemma *take-equalityI*: $(\forall i. take\ i\ xs = take\ i\ ys) \implies xs = ys$

— The famous take-lemma.
 $\langle \text{proof} \rangle$

lemma *take-Cons'*:

$\text{take } n \ (x \# xs) = (\text{if } n = 0 \text{ then } [] \text{ else } x \# \text{take } (n - 1) \ xs)$
 $\langle \text{proof} \rangle$

lemma *drop-Cons'*:

$\text{drop } n \ (x \# xs) = (\text{if } n = 0 \text{ then } x \# xs \text{ else } \text{drop } (n - 1) \ xs)$
 $\langle \text{proof} \rangle$

lemma *nth-Cons'*: $(x \# xs)!n = (\text{if } n = 0 \text{ then } x \text{ else } xs!(n - 1))$
 $\langle \text{proof} \rangle$

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* :: *int* \Rightarrow *int* \Rightarrow *int list* ((1[-./-])) **where**

upto *i j* = (*if* *i* \leq *j* *then* *i* $\#$ [*i*+1..*j*] *else* [])

$\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

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]: $j < i \implies [i..j] = []$

$\langle \text{proof} \rangle$

lemma *set-upto*[simp]: $\text{set}[i..j] = \{i..j\}$

$\langle \text{proof} \rangle$

43.1.22 distinct and remdups

lemma *distinct-append* [simp]:

distinct (*xs* @ *ys*) = (*distinct xs* \wedge *distinct ys* \wedge *set xs* \cap *set ys* = {})

$\langle \text{proof} \rangle$

lemma *distinct-rev*[simp]: *distinct*(*rev xs*) = *distinct xs*

$\langle \text{proof} \rangle$

lemma *set-remdups* [simp]: $\text{set } (\text{remdups } xs) = \text{set } xs$
 ⟨proof⟩

lemma *distinct-remdups* [iff]: $\text{distinct } (\text{remdups } xs)$
 ⟨proof⟩

lemma *distinct-remdups-id*: $\text{distinct } xs \implies \text{remdups } xs = xs$
 ⟨proof⟩

lemma *remdups-id-iff-distinct* [simp]: $\text{remdups } xs = xs \longleftrightarrow \text{distinct } xs$
 ⟨proof⟩

lemma *finite-distinct-list*: $\text{finite } A \implies \exists x. \text{set } xs = A \ \& \ \text{distinct } xs$
 ⟨proof⟩

lemma *remdups-eq-nil-iff* [simp]: $(\text{remdups } x = []) = (x = [])$
 ⟨proof⟩

lemma *remdups-eq-nil-right-iff* [simp]: $([] = \text{remdups } x) = (x = [])$
 ⟨proof⟩

lemma *length-remdups-leq*[iff]: $\text{length}(\text{remdups } xs) \leq \text{length } xs$
 ⟨proof⟩

lemma *length-remdups-eq*[iff]:
 $(\text{length } (\text{remdups } xs) = \text{length } xs) = (\text{remdups } xs = xs)$
 ⟨proof⟩

lemma *remdups-filter*: $\text{remdups}(\text{filter } P \ xs) = \text{filter } P \ (\text{remdups } xs)$
 ⟨proof⟩

lemma *distinct-map*:
 $\text{distinct}(\text{map } f \ xs) = (\text{distinct } xs \ \& \ \text{inj-on } f \ (\text{set } xs))$
 ⟨proof⟩

lemma *distinct-filter* [simp]: $\text{distinct } xs \implies \text{distinct } (\text{filter } P \ xs)$
 ⟨proof⟩

lemma *distinct-upt*[simp]: $\text{distinct}[i..<j]$
 ⟨proof⟩

lemma *distinct-upto*[simp]: $\text{distinct}[i..j]$
 ⟨proof⟩

lemma *distinct-take*[simp]: $\text{distinct } xs \implies \text{distinct } (\text{take } i \ xs)$
 ⟨proof⟩

lemma *distinct-drop[simp]*: $\text{distinct } xs \implies \text{distinct } (\text{drop } i \text{ } xs)$
 ⟨proof⟩

lemma *distinct-list-update*:
assumes d : $\text{distinct } xs$ **and** a : $a \notin \text{set } xs - \{xs!i\}$
shows $\text{distinct } (xs[i:=a])$
 ⟨proof⟩

lemma *distinct-concat*:
assumes $\text{distinct } xs$
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 $\text{distinct } (\text{concat } xs)$
 ⟨proof⟩

It is best to avoid this indexed version of *distinct*, but sometimes it is useful.

lemma *distinct-conv-nth*:
 $\text{distinct } xs = (\forall i < \text{size } xs. \forall j < \text{size } xs. i \neq j \longrightarrow xs!i \neq xs!j)$
 ⟨proof⟩

lemma *nth-eq-iff-index-eq*:
 $\llbracket \text{distinct } xs ; i < \text{length } xs ; j < \text{length } xs \rrbracket \implies (xs!i = xs!j) = (i = j)$
 ⟨proof⟩

lemma *distinct-card*: $\text{distinct } xs \implies \text{card } (\text{set } xs) = \text{size } xs$
 ⟨proof⟩

lemma *card-distinct*: $\text{card } (\text{set } xs) = \text{size } xs \implies \text{distinct } xs$
 ⟨proof⟩

lemma *not-distinct-decomp*: $\sim \text{distinct } ws \implies \exists x \ y \ z. ws = xs@[y]@ys@[y]@zs$
 ⟨proof⟩

lemma *length-remdups-concat*:
 $\text{length}(\text{remdups}(\text{concat } xss)) = \text{card}(\bigcup xs \in \text{set } xss. \text{set } xs)$
 ⟨proof⟩

lemma *length-remdups-card-conv*: $\text{length}(\text{remdups } xs) = \text{card}(\text{set } xs)$
 ⟨proof⟩

lemma *remdups-remdups*:
 $\text{remdups } (\text{remdups } xs) = \text{remdups } xs$
 ⟨proof⟩

lemma *distinct-butlast*:
assumes $xs \neq []$ **and** $\text{distinct } xs$
shows $\text{distinct } (\text{butlast } xs)$
 ⟨proof⟩

43.1.23 *insert***lemma** *in-set-insert* [simp]:

$$x \in \text{set } xs \implies \text{List.insert } x \ xs = xs$$

⟨proof⟩

lemma *not-in-set-insert* [simp]:

$$x \notin \text{set } xs \implies \text{List.insert } x \ xs = x \# \ xs$$

⟨proof⟩

lemma *insert-Nil* [simp]:

$$\text{List.insert } x \ [] = [x]$$

⟨proof⟩

lemma *set-insert* [simp]:

$$\text{set } (\text{List.insert } x \ xs) = \text{insert } x \ (\text{set } xs)$$

⟨proof⟩

lemma *distinct-insert* [simp]:

$$\text{distinct } xs \implies \text{distinct } (\text{List.insert } x \ xs)$$

⟨proof⟩

lemma *insert-remdups*:

$$\text{List.insert } x \ (\text{remdups } xs) = \text{remdups } (\text{List.insert } x \ xs)$$

⟨proof⟩

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** $P \ xs$

⟨proof⟩

43.1.24 *remove1***lemma** *remove1-append*:

$$\text{remove1 } x \ (xs \ @ \ ys) =$$

$$(\text{if } x \in \text{set } xs \text{ then } \text{remove1 } x \ xs \ @ \ ys \text{ else } xs \ @ \ \text{remove1 } x \ ys)$$

⟨proof⟩

lemma *remove1-commute*: $\text{remove1 } x \ (\text{remove1 } y \ zs) = \text{remove1 } y \ (\text{remove1 } x \ zs)$

⟨proof⟩

lemma *in-set-remove1* [simp]:

$$a \neq b \implies a : \text{set}(\text{remove1 } b \ xs) = (a : \text{set } xs)$$

⟨proof⟩

lemma *set-remove1-subset*: $\text{set}(\text{remove1 } x \ xs) \leq \text{set } xs$

⟨proof⟩

lemma *set-remove1-eq* [simp]: $\text{distinct } xs \implies \text{set}(\text{remove1 } x \ xs) = \text{set } xs - \{x\}$
 ⟨proof⟩

lemma *length-remove1*:
 $\text{length}(\text{remove1 } x \ xs) = (\text{if } x : \text{set } xs \text{ then } \text{length } xs - 1 \text{ else } \text{length } xs)$
 ⟨proof⟩

lemma *remove1-filter-not* [simp]:
 $\neg P \ x \implies \text{remove1 } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$
 ⟨proof⟩

lemma *notin-set-remove1* [simp]: $x \sim : \text{set } xs \implies x \sim : \text{set}(\text{remove1 } y \ xs)$
 ⟨proof⟩

lemma *distinct-remove1* [simp]: $\text{distinct } xs \implies \text{distinct}(\text{remove1 } x \ xs)$
 ⟨proof⟩

lemma *remove1-remdups*:
 $\text{distinct } xs \implies \text{remove1 } x \ (\text{remdups } xs) = \text{remdups } (\text{remove1 } x \ xs)$
 ⟨proof⟩

lemma *remove1-idem*:
 assumes $x \notin \text{set } xs$
 shows $\text{remove1 } x \ xs = xs$
 ⟨proof⟩

43.1.25 *removeAll*

lemma *removeAll-filter-not-eq*:
 $\text{removeAll } x = \text{filter } (\lambda y. x \neq y)$
 ⟨proof⟩

lemma *removeAll-append* [simp]:
 $\text{removeAll } x \ (xs \ @ \ ys) = \text{removeAll } x \ xs \ @ \ \text{removeAll } x \ ys$
 ⟨proof⟩

lemma *set-removeAll* [simp]: $\text{set}(\text{removeAll } x \ xs) = \text{set } xs - \{x\}$
 ⟨proof⟩

lemma *removeAll-id* [simp]: $x \notin \text{set } xs \implies \text{removeAll } x \ xs = xs$
 ⟨proof⟩

lemma *removeAll-filter-not* [simp]:
 $\neg P \ x \implies \text{removeAll } x \ (\text{filter } P \ xs) = \text{filter } P \ xs$
 ⟨proof⟩

lemma *distinct-removeAll*:
 $distinct\ xs \implies distinct\ (removeAll\ x\ xs)$
 $\langle proof \rangle$

lemma *distinct-remove1-removeAll*:
 $distinct\ xs \implies remove1\ x\ xs = removeAll\ x\ xs$
 $\langle proof \rangle$

lemma *map-removeAll-inj-on*: $inj\text{-}on\ f\ (insert\ x\ (set\ xs)) \implies$
 $map\ f\ (removeAll\ x\ xs) = removeAll\ (f\ x)\ (map\ f\ xs)$
 $\langle proof \rangle$

lemma *map-removeAll-inj*: $inj\ f \implies$
 $map\ f\ (removeAll\ x\ xs) = removeAll\ (f\ x)\ (map\ f\ xs)$
 $\langle proof \rangle$

43.1.26 replicate

lemma *length-replicate* [simp]: $length\ (replicate\ n\ x) = n$
 $\langle proof \rangle$

lemma *Ex-list-of-length*: $\exists xs. length\ xs = n$
 $\langle proof \rangle$

lemma *map-replicate* [simp]: $map\ f\ (replicate\ n\ x) = replicate\ n\ (f\ x)$
 $\langle proof \rangle$

lemma *map-replicate-const*:
 $map\ (\lambda x. k)\ lst = replicate\ (length\ lst)\ k$
 $\langle proof \rangle$

lemma *replicate-app-Cons-same*:
 $(replicate\ n\ x) @ (x \# xs) = x \# replicate\ n\ x @ xs$
 $\langle proof \rangle$

lemma *rev-replicate* [simp]: $rev\ (replicate\ n\ x) = replicate\ n\ x$
 $\langle proof \rangle$

lemma *replicate-add*: $replicate\ (n + m)\ x = replicate\ n\ x @ replicate\ m\ x$
 $\langle proof \rangle$

Courtesy of Matthias Daum:

lemma *append-replicate-commute*:
 $replicate\ n\ x @ replicate\ k\ x = replicate\ k\ x @ replicate\ n\ x$
 $\langle proof \rangle$

Courtesy of Andreas Lochbihler:

lemma *filter-replicate*:
 $filter\ P\ (replicate\ n\ x) = (if\ P\ x\ then\ replicate\ n\ x\ else\ [])$

$\langle proof \rangle$

lemma *hd-replicate* [simp]: $n \neq 0 \implies hd\ (replicate\ n\ x) = x$
 $\langle proof \rangle$

lemma *tl-replicate* [simp]: $n \neq 0 \implies tl\ (replicate\ n\ x) = replicate\ (n - 1)\ x$
 $\langle proof \rangle$

lemma *last-replicate* [simp]: $n \neq 0 \implies last\ (replicate\ n\ x) = x$
 $\langle proof \rangle$

lemma *nth-replicate* [simp]: $i < n \implies (replicate\ n\ x)!i = x$
 $\langle proof \rangle$

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate* [simp]: $take\ i\ (replicate\ k\ x) = replicate\ (min\ i\ k)\ x$
 $\langle proof \rangle$

lemma *drop-replicate* [simp]: $drop\ i\ (replicate\ k\ x) = replicate\ (k-i)\ x$
 $\langle proof \rangle$

lemma *set-replicate-Suc*: $set\ (replicate\ (Suc\ n)\ x) = \{x\}$
 $\langle proof \rangle$

lemma *set-replicate* [simp]: $n \neq 0 \implies set\ (replicate\ n\ x) = \{x\}$
 $\langle proof \rangle$

lemma *set-replicate-conv-if*: $set\ (replicate\ n\ x) = (if\ n = 0\ then\ \{\}\ else\ \{x\})$
 $\langle proof \rangle$

lemma *in-set-replicateD*: $x : set\ (replicate\ n\ y) \implies x = y$
 $\langle proof \rangle$

lemma *replicate-append-same*:
 $replicate\ i\ x\ @\ [x] = x\ \# \ replicate\ i\ x$
 $\langle proof \rangle$

lemma *map-replicate-trivial*:
 $map\ (\lambda i. x)\ [0..i] = replicate\ i\ x$
 $\langle proof \rangle$

lemma *concat-replicate-trivial* [simp]:
 $concat\ (replicate\ i\ []) = []$
 $\langle proof \rangle$

lemma *replicate-empty* [simp]: $(replicate\ n\ x = []) \longleftrightarrow n=0$
 $\langle proof \rangle$

lemma *empty-replicate[simp]*: $([] = \text{replicate } n \ x) \longleftrightarrow n=0$
 $\langle \text{proof} \rangle$

lemma *replicate-eq-replicate[simp]*:
 $(\text{replicate } m \ x = \text{replicate } n \ y) \longleftrightarrow (m=n \ \& \ (m \neq 0 \longrightarrow x=y))$
 $\langle \text{proof} \rangle$

43.1.27 *rotate1 and rotate*

lemma *rotate-simps[simp]*: $\text{rotate1 } [] = [] \wedge \text{rotate1 } (x\#xs) = xs @ [x]$
 $\langle \text{proof} \rangle$

lemma *rotate0[simp]*: $\text{rotate } 0 = \text{id}$
 $\langle \text{proof} \rangle$

lemma *rotate-Suc[simp]*: $\text{rotate } (\text{Suc } n) \ xs = \text{rotate1 } (\text{rotate } n \ xs)$
 $\langle \text{proof} \rangle$

lemma *rotate-add*:
 $\text{rotate } (m+n) = \text{rotate } m \ o \ \text{rotate } n$
 $\langle \text{proof} \rangle$

lemma *rotate-rotate*: $\text{rotate } m \ (\text{rotate } n \ xs) = \text{rotate } (m+n) \ xs$
 $\langle \text{proof} \rangle$

lemma *rotate1-rotate-swap*: $\text{rotate1 } (\text{rotate } n \ xs) = \text{rotate } n \ (\text{rotate1 } xs)$
 $\langle \text{proof} \rangle$

lemma *rotate1-length01[simp]*: $\text{length } xs \leq 1 \implies \text{rotate1 } xs = xs$
 $\langle \text{proof} \rangle$

lemma *rotate-length01[simp]*: $\text{length } xs \leq 1 \implies \text{rotate } n \ xs = xs$
 $\langle \text{proof} \rangle$

lemma *rotate1-hd-tl*: $xs \neq [] \implies \text{rotate1 } xs = \text{tl } xs @ [\text{hd } xs]$
 $\langle \text{proof} \rangle$

lemma *rotate-drop-take*:
 $\text{rotate } n \ xs = \text{drop } (n \bmod \text{length } xs) \ xs @ \text{take } (n \bmod \text{length } xs) \ xs$
 $\langle \text{proof} \rangle$

lemma *rotate-conv-mod*: $\text{rotate } n \ xs = \text{rotate } (n \bmod \text{length } xs) \ xs$
 $\langle \text{proof} \rangle$

lemma *rotate-id[simp]*: $n \bmod \text{length } xs = 0 \implies \text{rotate } n \ xs = xs$
 $\langle \text{proof} \rangle$

lemma *length-rotate1[simp]*: $\text{length}(\text{rotate1 } xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *length-rotate[simp]*: $\text{length}(\text{rotate } n \text{ } xs) = \text{length } xs$
 $\langle \text{proof} \rangle$

lemma *distinct1-rotate[simp]*: $\text{distinct}(\text{rotate1 } xs) = \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *distinct-rotate[simp]*: $\text{distinct}(\text{rotate } n \text{ } xs) = \text{distinct } xs$
 $\langle \text{proof} \rangle$

lemma *rotate-map*: $\text{rotate } n \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{rotate } n \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *set-rotate1[simp]*: $\text{set}(\text{rotate1 } xs) = \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *set-rotate[simp]*: $\text{set}(\text{rotate } n \text{ } xs) = \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *rotate1-is-Nil-conv[simp]*: $(\text{rotate1 } xs = []) = (xs = [])$
 $\langle \text{proof} \rangle$

lemma *rotate-is-Nil-conv[simp]*: $(\text{rotate } n \text{ } xs = []) = (xs = [])$
 $\langle \text{proof} \rangle$

lemma *rotate-rev*:
 $\text{rotate } n \text{ } (\text{rev } xs) = \text{rev}(\text{rotate } (\text{length } xs - (n \bmod \text{length } xs)) \text{ } xs)$
 $\langle \text{proof} \rangle$

lemma *hd-rotate-conv-nth*: $xs \neq [] \implies \text{hd}(\text{rotate } n \text{ } xs) = xs!(n \bmod \text{length } xs)$
 $\langle \text{proof} \rangle$

43.1.28 *sublist* — a generalization of *nth* to sets

lemma *sublist-empty [simp]*: $\text{sublist } xs \text{ } \{\} = []$
 $\langle \text{proof} \rangle$

lemma *sublist-nil [simp]*: $\text{sublist } [] \text{ } A = []$
 $\langle \text{proof} \rangle$

lemma *length-sublist*:
 $\text{length}(\text{sublist } xs \text{ } I) = \text{card}\{i. i < \text{length } xs \wedge i : I\}$
 $\langle \text{proof} \rangle$

lemma *sublist-shift-lemma-Suc*:
 $\text{map fst } (\text{filter } (\%p. P(\text{Suc}(\text{snd } p))) \text{ } (\text{zip } xs \text{ } is)) =$
 $\text{map fst } (\text{filter } (\%p. P(\text{snd } p)) \text{ } (\text{zip } xs \text{ } (\text{map } \text{Suc } is)))$
 $\langle \text{proof} \rangle$

lemma *sublist-shift-lemma*:

$$\begin{aligned} \text{map fst } [p < - \text{zip } xs \ [i..<i + \text{length } xs] \ . \ \text{snd } p : A] = \\ \text{map fst } [p < - \text{zip } xs \ [0..<\text{length } xs] \ . \ \text{snd } p + i : A] \end{aligned}$$

<proof>

lemma *sublist-append*:

$$\text{sublist } (l \ @ \ l') \ A = \text{sublist } l \ A \ @ \ \text{sublist } l' \ \{j. \ j + \text{length } l : A\}$$

<proof>

lemma *sublist-Cons*:

$$\text{sublist } (x \ # \ l) \ A = (\text{if } 0:A \ \text{then } [x] \ \text{else } []) \ @ \ \text{sublist } l \ \{j. \ \text{Suc } j : A\}$$

<proof>

lemma *set-sublist*: $\text{set}(\text{sublist } xs \ I) = \{xs[i] \mid i. \ i < \text{size } xs \wedge i \in I\}$

<proof>

lemma *set-sublist-subset*: $\text{set}(\text{sublist } xs \ I) \subseteq \text{set } xs$

<proof>

lemma *notin-set-sublistI[simp]*: $x \notin \text{set } xs \implies x \notin \text{set}(\text{sublist } xs \ I)$

<proof>

lemma *in-set-sublistD*: $x \in \text{set}(\text{sublist } xs \ I) \implies x \in \text{set } xs$

<proof>

lemma *sublist-singleton [simp]*: $\text{sublist } [x] \ A = (\text{if } 0 : A \ \text{then } [x] \ \text{else } [])$

<proof>

lemma *distinct-sublistI[simp]*: $\text{distinct } xs \implies \text{distinct}(\text{sublist } xs \ I)$

<proof>

lemma *sublist-upt-eq-take [simp]*: $\text{sublist } l \ \{..<n\} = \text{take } n \ l$

<proof>

lemma *filter-in-sublist*:

$$\text{distinct } xs \implies \text{filter } (\%x. \ x \in \text{set}(\text{sublist } xs \ s)) \ xs = \text{sublist } xs \ s$$

<proof>

43.1.29 splice

lemma *splice-Nil2 [simp, code]*:

$$\text{splice } xs \ [] = xs$$

<proof>

lemma *splice-Cons-Cons [simp, code]*:

$$\text{splice } (x \ # \ xs) \ (y \ # \ ys) = x \ # \ y \ # \ \text{splice } xs \ ys$$

<proof>

declare *splice.simps*(2) [*simp del*, *code del*]

lemma *length-splice*[*simp*]: $\text{length}(\text{splice } xs \ ys) = \text{length } xs + \text{length } ys$
 ⟨*proof*⟩

43.1.30 Transpose

function *transpose* **where**
transpose [] = [] |
transpose ([] # *xss*) = *transpose xss* |
transpose ((*x* # *xs*) # *xss*) =
 (*x* # [*h*. (*h* # *t*) ← *xss*]) # *transpose* (*xs* # [*t*. (*h* # *t*) ← *xss*])
 ⟨*proof*⟩

lemma *transpose-aux-filter-head*:
 $\text{concat } (\text{map } (\text{list-case } [] \ (\lambda h \ t. \ [h])) \ xss) =$
 $\text{map } (\lambda xs. \ \text{hd } xs) \ [ys \leftarrow xss \ . \ ys \neq []]$
 ⟨*proof*⟩

lemma *transpose-aux-filter-tail*:
 $\text{concat } (\text{map } (\text{list-case } [] \ (\lambda h \ t. \ [t])) \ xss) =$
 $\text{map } (\lambda xs. \ \text{tl } xs) \ [ys \leftarrow xss \ . \ ys \neq []]$
 ⟨*proof*⟩

lemma *transpose-aux-max*:
 $\text{max } (\text{Suc } (\text{length } xs)) \ (\text{foldr } (\lambda xs. \ \text{max } (\text{length } xs)) \ xss \ 0) =$
 $\text{Suc } (\text{max } (\text{length } xs) \ (\text{foldr } (\lambda x. \ \text{max } (\text{length } x - \text{Suc } 0)) \ [ys \leftarrow xss \ . \ ys \neq []] \ 0))$
 (is *max* - ?*foldB* = *Suc* (*max* - ?*foldA*))
 ⟨*proof*⟩

termination *transpose*
 ⟨*proof*⟩

lemma *transpose-empty*: $(\text{transpose } xs = []) \longleftrightarrow (\forall x \in \text{set } xs. \ x = [])$
 ⟨*proof*⟩

lemma *length-transpose*:
fixes *xs* :: 'a list list
shows $\text{length } (\text{transpose } xs) = \text{foldr } (\lambda xs. \ \text{max } (\text{length } xs)) \ xs \ 0$
 ⟨*proof*⟩

lemma *nth-transpose*:
fixes *xs* :: 'a list list
assumes $i < \text{length } (\text{transpose } xs)$
shows $\text{transpose } xs \ ! \ i = \text{map } (\lambda xs. \ xs \ ! \ i) \ [ys \leftarrow xs. \ i < \text{length } ys]$
 ⟨*proof*⟩

lemma *transpose-map-map*:

transpose (*map* (*map* *f*) *xs*) = *map* (*map* *f*) (*transpose xs*)
 ⟨*proof*⟩

43.1.31 (In)finiteness

lemma *finite-maxlen*:

finite (*M*::'a list set) ==> *EX n. ALL s:M. size s < n*
 ⟨*proof*⟩

lemma *finite-lists-length-eq*:

assumes *finite A*

shows *finite {xs. set xs ⊆ A ∧ length xs = n}* (**is** *finite (?S n)*)
 ⟨*proof*⟩

lemma *finite-lists-length-le*:

assumes *finite A* **shows** *finite {xs. set xs ⊆ A ∧ length xs ≤ n}*
 (**is** *finite ?S*)
 ⟨*proof*⟩

lemma *infinite-UNIV-listI*: \sim *finite*(*UNIV*::'a list set)

⟨*proof*⟩

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)*

⟨*proof*⟩

lemma *insort-left-comm*:

insort x (insort y xs) = insort y (insort x xs)

⟨*proof*⟩

lemma *fun-left-comm-insort*:

fun-left-comm insort

⟨*proof*⟩

lemma *sort-key-simps [simp]*:

sort-key f [] = []

sort-key f (x#xs) = insort-key f x (sort-key f xs)

⟨*proof*⟩

lemma *sort-foldl-insort*:

$$\text{sort } xs = \text{foldl } (\lambda ys \ x. \text{insort } x \ ys) [] \ xs$$

<proof>

lemma *length-sort[simp]*: $\text{length } (\text{sort-key } f \ xs) = \text{length } xs$

<proof>

lemma *sorted-Cons*: $\text{sorted } (x \# xs) = (\text{sorted } xs \ \& \ (\text{ALL } y : \text{set } xs. \ x \leq y))$

<proof>

lemma *sorted-append*:

$$\text{sorted } (xs @ ys) = (\text{sorted } xs \ \& \ \text{sorted } ys \ \& \ (\forall x \in \text{set } xs. \ \forall y \in \text{set } ys. \ x \leq y))$$

<proof>

lemma *sorted-nth-mono*:

$$\text{sorted } xs \implies i \leq j \implies j < \text{length } xs \implies xs!i \leq xs!j$$

<proof>

lemma *sorted-rev-nth-mono*:

$$\text{sorted } (\text{rev } xs) \implies i \leq j \implies j < \text{length } xs \implies xs!j \leq xs!i$$

<proof>

lemma *sorted-nth-monoI*:

$$(\bigwedge i \ j. \ [i \leq j \ ; \ j < \text{length } xs] \implies xs!i \leq xs!j) \implies \text{sorted } xs$$

<proof>

lemma *sorted-equals-nth-mono*:

$$\text{sorted } xs = (\forall j < \text{length } xs. \ \forall i \leq j. \ xs!i \leq xs!j)$$

<proof>

lemma *set-insort*: $\text{set}(\text{insort-key } f \ x \ xs) = \text{insert } x \ (\text{set } xs)$

<proof>

lemma *set-sort[simp]*: $\text{set}(\text{sort-key } f \ xs) = \text{set } xs$

<proof>

lemma *distinct-insort*: $\text{distinct } (\text{insort-key } f \ x \ xs) = (x \notin \text{set } xs \ \wedge \ \text{distinct } xs)$

<proof>

lemma *distinct-sort[simp]*: $\text{distinct } (\text{sort-key } f \ xs) = \text{distinct } xs$

<proof>

lemma *sorted-insort-key*: $\text{sorted } (\text{map } f \ (\text{insort-key } f \ x \ xs)) = \text{sorted } (\text{map } f \ xs)$

<proof>

lemma *sorted-insort*: $\text{sorted } (\text{insort } x \ xs) = \text{sorted } xs$

<proof>

theorem *sorted-sort-key[simp]*: $\text{sorted } (\text{map } f \ (\text{sort-key } f \ xs))$

<proof>

theorem *sorted-sort[simp]: sorted (sort xs)*
<proof>

lemma *sorted-butlast:*
assumes $xs \neq []$ **and** *sorted xs*
shows *sorted (butlast xs)*
<proof>

lemma *insort-not-Nil [simp]:*
insort-key f a xs $\neq []$
<proof>

lemma *insort-is-Cons:* $\forall x \in \text{set } xs. f a \leq f x \implies \text{insort-key } f a xs = a \# xs$
<proof>

lemma *sorted-remove1:* *sorted xs* \implies *sorted (remove1 a xs)*
<proof>

lemma *insort-key-remove1:* $\llbracket a \in \text{set } xs; \text{sorted } (\text{map } f xs) ; \text{inj-on } f (\text{set } xs) \rrbracket$
 $\implies \text{insort-key } f a (\text{remove1 } a xs) = xs$
<proof>

lemma *insort-remove1:* $\llbracket a \in \text{set } xs; \text{sorted } xs \rrbracket \implies \text{insort } a (\text{remove1 } a xs) = xs$
<proof>

lemma *sorted-remdups[simp]:*
sorted l \implies *sorted (remdups l)*
<proof>

lemma *sorted-distinct-set-unique:*
assumes *sorted xs distinct xs sorted ys distinct ys set xs = set ys*
shows $xs = ys$
<proof>

lemma *map-sorted-distinct-set-unique:*
assumes *inj-on f (set xs \cup set ys)*
assumes *sorted (map f xs) distinct (map f xs)*
sorted (map f ys) distinct (map f ys)
assumes $\text{set } xs = \text{set } ys$
shows $xs = ys$
<proof>

lemma *finite-sorted-distinct-unique:*
shows $\text{finite } A \implies \text{EX! } xs. \text{set } xs = A \ \& \ \text{sorted } xs \ \& \ \text{distinct } xs$
<proof>

lemma *sorted-take:*

$sorted\ xs \implies sorted\ (take\ n\ xs)$
 $\langle proof \rangle$

lemma *sorted-drop*:
 $sorted\ xs \implies sorted\ (drop\ n\ xs)$
 $\langle proof \rangle$

lemma *sorted-dropWhile*: $sorted\ xs \implies sorted\ (dropWhile\ P\ xs)$
 $\langle proof \rangle$

lemma *sorted-takeWhile*: $sorted\ xs \implies sorted\ (takeWhile\ P\ xs)$
 $\langle proof \rangle$

lemma *sorted-filter*:
 $sorted\ (map\ f\ xs) \implies sorted\ (map\ f\ (filter\ P\ xs))$
 $\langle proof \rangle$

lemma *foldr-max-sorted*:
assumes $sorted\ (rev\ xs)$
shows $foldr\ max\ xs\ y = (if\ xs = []\ then\ y\ else\ max\ (xs\ !\ 0)\ y)$
 $\langle proof \rangle$

lemma *filter-equals-takeWhile-sorted-rev*:
assumes $sorted: sorted\ (rev\ (map\ f\ xs))$
shows $[x \leftarrow xs.\ t < f\ x] = takeWhile\ (\lambda\ x.\ t < f\ x)\ xs$
 $(is\ filter\ ?P\ xs = ?tW)$
 $\langle proof \rangle$

lemma *set-insort-insert*:
 $set\ (insort-insert\ x\ xs) = insert\ x\ (set\ xs)$
 $\langle proof \rangle$

lemma *distinct-insort-insert*:
assumes $distinct\ xs$
shows $distinct\ (insort-insert\ x\ xs)$
 $\langle proof \rangle$

lemma *sorted-insort-insert*:
assumes $sorted\ xs$
shows $sorted\ (insort-insert\ x\ xs)$
 $\langle proof \rangle$

lemma *filter-insort-key-triv*:
 $\neg P\ x \implies filter\ P\ (insort-key\ f\ x\ xs) = filter\ P\ xs$
 $\langle proof \rangle$

lemma *filter-insort-key*:
 $sorted\ (map\ f\ xs) \implies P\ x \implies filter\ P\ (insort-key\ f\ x\ xs) = insort-key\ f\ x\ (filter\ P\ xs)$

$\langle \text{proof} \rangle$

lemma *filter-sort-key*:

filter *P* (*sort-key* *f xs*) = *sort-key* *f* (*filter* *P xs*)

$\langle \text{proof} \rangle$

lemma *sorted-same* [*simp*]:

sorted [*x ← xs. x = f xs*]

$\langle \text{proof} \rangle$

lemma *remove1-insort* [*simp*]:

remove1 *x* (*insort* *x xs*) = *xs*

$\langle \text{proof} \rangle$

end

lemma *sorted-upt*[*simp*]: *sorted*[*i..<j*]

$\langle \text{proof} \rangle$

lemma *sorted-upto*[*simp*]: *sorted*[*i..j*]

$\langle \text{proof} \rangle$

43.2.1 *transpose on sorted lists*

lemma *sorted-transpose*[*simp*]:

shows *sorted* (*rev* (*map* *length* (*transpose* *xs*)))

$\langle \text{proof} \rangle$

lemma *transpose-max-length*:

foldr ($\lambda xs. \text{max } (\text{length } xs) \text{) } (\text{transpose } xs) \ 0 = \text{length } [x \leftarrow xs. x \neq []]$

(**is** ?*L* = ?*R*)

$\langle \text{proof} \rangle$

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))

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

lemma *transpose-column-length*:

fixes *xs* :: 'a list list

assumes *sorted*: *sorted* (*rev* (*map length xs*)) **and** $i < \text{length } xs$
shows $\text{length } (\text{filter } (\lambda ys. i < \text{length } ys) (\text{transpose } xs)) = \text{length } (xs ! i)$
 $\langle \text{proof} \rangle$

lemma *transpose-column*:
fixes $xs :: 'a \text{ list list}$
assumes *sorted*: *sorted* (*rev* (*map length xs*)) **and** $i < \text{length } xs$
shows $\text{map } (\lambda ys. ys ! i) (\text{filter } (\lambda ys. i < \text{length } ys) (\text{transpose } xs))$
 $= xs ! i$ (**is** ?*R* = -)
 $\langle \text{proof} \rangle$

lemma *transpose-transpose*:
fixes $xs :: 'a \text{ list list}$
assumes *sorted*: *sorted* (*rev* (*map length xs*))
shows $\text{transpose } (\text{transpose } xs) = \text{takeWhile } (\lambda x. x \neq []) xs$ (**is** ?*L* = ?*R*)
 $\langle \text{proof} \rangle$

theorem *transpose-rectangle*:
assumes $xs = [] \implies n = 0$
assumes *rect*: $\bigwedge i. i < \text{length } xs \implies \text{length } (xs ! i) = n$
shows $\text{transpose } xs = \text{map } (\lambda i. \text{map } (\lambda j. xs ! j ! i) [0..<\text{length } xs]) [0..<n]$
(is ?*trans* = ?*map*)
 $\langle \text{proof} \rangle$

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* :: $'a \text{ set} \Rightarrow 'a \text{ list}$ **where**
 $\text{sorted-list-of-set} = \text{Finite-Set.fold insort } []$

lemma *sorted-list-of-set-empty* [*simp*]:
 $\text{sorted-list-of-set } \{\} = []$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set-insert* [*simp*]:
assumes *finite A*
shows $\text{sorted-list-of-set } (\text{insert } x A) = \text{insort } x (\text{sorted-list-of-set } (A - \{x\}))$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set* [*simp*]:
 $\text{finite } A \implies \text{set } (\text{sorted-list-of-set } A) = A \wedge \text{sorted } (\text{sorted-list-of-set } A)$
 $\wedge \text{distinct } (\text{sorted-list-of-set } A)$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set-sort-remdups*:
 $\text{sorted-list-of-set } (\text{set } xs) = \text{sort } (\text{remdups } xs)$
 $\langle \text{proof} \rangle$

lemma *sorted-list-of-set-remove*:
assumes *finite A*
shows $\text{sorted-list-of-set } (A - \{x\}) = \text{remove1 } x \ (\text{sorted-list-of-set } A)$
 $\langle \text{proof} \rangle$

end

lemma *sorted-list-of-set-range* [simp]:
 $\text{sorted-list-of-set } \{m..<n\} = [m..<n]$
 $\langle \text{proof} \rangle$

43.2.3 *lists*: the list-forming operator over sets

inductive-set
 $\text{lists} :: 'a \text{ set} \Rightarrow 'a \text{ list set}$
for $A :: 'a \text{ set}$
where
 $\text{Nil } [\text{intro!}]: [] : \text{lists } A$
 $| \text{Cons } [\text{intro!}, \text{no-atp}]: [] \ a : A; l : \text{lists } A \Longrightarrow a \# l : \text{lists } A$

inductive-cases *listsE* [elim!,no-atp]: $x \# l : \text{lists } A$
inductive-cases *listspE* [elim!,no-atp]: $\text{listsp } A \ (x \# l)$

lemma *listsp-mono* [mono]: $A \leq B \Longrightarrow \text{listsp } A \leq \text{listsp } B$
 $\langle \text{proof} \rangle$

lemmas *lists-mono* = *listsp-mono* [to-set pred-subset-eq]

lemma *listsp-infI*:
assumes $l : \text{listsp } A \ l$ **shows** $\text{listsp } B \ l \Longrightarrow \text{listsp } (\text{inf } A \ B) \ l$ $\langle \text{proof} \rangle$

lemmas *lists-IntI* = *listsp-infI* [to-set]

lemma *listsp-inf-eq* [simp]: $\text{listsp } (\text{inf } A \ B) = \text{inf } (\text{listsp } A) \ (\text{listsp } B)$
 $\langle \text{proof} \rangle$

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]:
 $(\text{listsp } A \ (xs @ ys)) = (\text{listsp } A \ xs \wedge \text{listsp } A \ ys)$
 $\langle \text{proof} \rangle$

lemmas *append-in-lists-conv* [iff] = *append-in-listsp-conv* [to-set]

lemma *in-listsp-conv-set*: $(listsp\ A\ xs) = (\forall x \in set\ xs. A\ x)$
 — eliminate *listsp* in favour of *set*
 $\langle proof \rangle$

lemmas *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest!*,*no-atp*]: $listsp\ A\ xs \implies \forall x \in set\ xs. A\ x$
 $\langle proof \rangle$

lemmas *in-listsD* [*dest!*,*no-atp*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro!*,*no-atp*]: $\forall x \in set\ xs. A\ x \implies listsp\ A\ xs$
 $\langle proof \rangle$

lemmas *in-listsI* [*intro!*,*no-atp*] = *in-listspI* [*to-set*]

lemma *lists-UNIV* [*simp*]: $lists\ UNIV = UNIV$
 $\langle proof \rangle$

43.2.4 Inductive definition for membership

inductive *ListMem* :: $'a \Rightarrow 'a\ list \Rightarrow bool$
where

elem: $ListMem\ x\ (x \# xs)$
 | *insert*: $ListMem\ x\ xs \implies ListMem\ x\ (y \# xs)$

lemma *ListMem-iff*: $(ListMem\ x\ xs) = (x \in set\ xs)$
 $\langle proof \rangle$

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*]: $set-Cons\ A\ XS = \{z. \exists x\ xs. z = x \# xs \wedge x \in A \wedge xs \in XS\}$

lemma *set-Cons-sing-Nil* [*simp*]: $set-Cons\ A\ \{\ [] \} = (\%x. [x])'A$
 $\langle proof \rangle$

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* $(A \# As) = set-Cons\ A\ (listset\ As)$

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

$lexn :: ('a \times 'a) \text{ set} \Rightarrow \text{nat} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $lexn \ r \ 0 = \{\}$
 $| \ lexn \ r \ (Suc \ n) = (\text{prod-fun } (\%(x, xs). \ x \# xs) \ (\%(x, xs). \ x \# xs)) \ ' (r \ < *lex* >$
 $lexn \ r \ n)) \ Int$
 $\{(xs, ys). \ length \ xs = Suc \ n \wedge length \ ys = Suc \ n\}$

definition

$lex :: ('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $[code \ del]: \ lex \ r = (\bigcup n. \ lexn \ r \ n)$ — Holds only between lists of the same length

definition

$lenlex :: ('a \times 'a) \text{ set} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ set}$ **where**
 $[code \ del]: \ lenlex \ r = \text{inv-image } (less-than \ < *lex* > \ lex \ r) \ (\lambda xs. \ (length \ xs, \ xs))$
 — Compares lists by their length and then lexicographically

lemma $wf\text{-}lexn$: $wf \ r \ ==> \ wf \ (lexn \ r \ n)$

$\langle proof \rangle$

lemma $lexn\text{-}length$:

$(xs, ys) : lexn \ r \ n \ ==> \ length \ xs = n \wedge length \ ys = n$

$\langle proof \rangle$

lemma $wf\text{-}lex$ $[intro!]$: $wf \ r \ ==> \ wf \ (lex \ r)$

$\langle proof \rangle$

lemma $lexn\text{-}conv$:

$lexn \ r \ n =$
 $\{(xs, ys). \ length \ xs = n \wedge length \ ys = n \wedge$
 $(\exists xys \ x \ y \ xs' \ ys'. \ xs = xys \ @ \ x \# xs' \wedge ys = xys \ @ \ y \# ys' \wedge (x, y):r)\}$

$\langle proof \rangle$

lemma $lex\text{-}conv$:

$lex \ r =$
 $\{(xs, ys). \ length \ xs = length \ ys \wedge$
 $(\exists xys \ x \ y \ xs' \ ys'. \ xs = xys \ @ \ x \# xs' \wedge ys = xys \ @ \ y \# ys' \wedge (x, y):r)\}$

$\langle proof \rangle$

lemma $wf\text{-}lenlex$ $[intro!]$: $wf \ r \ ==> \ wf \ (lenlex \ r)$

$\langle proof \rangle$

lemma $lenlex\text{-}conv$:

$lenlex \ r = \{(xs, ys). \ length \ xs < length \ ys \mid$

$length\ xs = length\ ys \wedge (xs, ys) : lex\ r\}$
 $\langle proof \rangle$

lemma *Nil-notin-lex* [iff]: $([], ys) \notin lex\ r$
 $\langle proof \rangle$

lemma *Nil2-notin-lex* [iff]: $(xs, []) \notin lex\ r$
 $\langle proof \rangle$

lemma *Cons-in-lex* [simp]:
 $((x \# xs, y \# ys) : lex\ r) =$
 $((x, y) : r \wedge length\ xs = length\ ys \mid x = y \wedge (xs, ys) : lex\ r)$
 $\langle proof \rangle$

43.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. ”a” i ”ab” i ”b”. This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

definition

$lexord :: ('a \times 'a)\ set \Rightarrow ('a\ list \times 'a\ list)\ set$ **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-Nil-left*[simp]: $([], y) \in lexord\ r = (\exists\ a\ x. y = a \# x)$
 $\langle proof \rangle$

lemma *lexord-Nil-right*[simp]: $(x, []) \notin lexord\ r$
 $\langle proof \rangle$

lemma *lexord-cons-cons*[simp]:
 $((a \# x, b \# y) \in lexord\ r) = ((a,b) \in r \mid (a = b \ \& \ (x,y) \in lexord\ r))$
 $\langle proof \rangle$

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-append-rightI*: $\exists\ b\ z. y = b \# z \implies (x, x @ y) \in lexord\ r$
 $\langle proof \rangle$

lemma *lexord-append-left-rightI*:
 $(a,b) \in r \implies (u @ a \# x, u @ b \# y) \in lexord\ r$
 $\langle proof \rangle$

lemma *lexord-append-leftI*: $(u,v) \in lexord\ r \implies (x @ u, x @ v) \in lexord\ r$
 $\langle proof \rangle$

lemma *lexord-append-leftD*:
 $\llbracket (x @ u, x @ v) \in lexord\ r; (!\ a. (a,a) \notin r) \rrbracket \implies (u,v) \in lexord\ r$
 $\langle proof \rangle$

lemma *lexord-take-index-conv*:

$((x,y) : \text{lexord } r) =$
 $((\text{length } x < \text{length } y \wedge \text{take } (\text{length } x) \ y = x) \vee$
 $(\exists i. i < \min(\text{length } x)(\text{length } y) \ \& \ \text{take } i \ x = \text{take } i \ y \ \& \ (x!i,y!i) \in r))$
 $\langle \text{proof} \rangle$

lemma *lexord-lex*: $(x,y) \in \text{lex } r = ((x,y) \in \text{lexord } r \wedge \text{length } x = \text{length } y)$
 $\langle \text{proof} \rangle$

lemma *lexord-irreflexive*: $(! x. (x,x) \notin r) \implies (y,y) \notin \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-trans*:

$\llbracket (x, y) \in \text{lexord } r; (y, z) \in \text{lexord } r; \text{trans } r \rrbracket \implies (x, z) \in \text{lexord } r$
 $\langle \text{proof} \rangle$

lemma *lexord-transI*: $\text{trans } r \implies \text{trans } (\text{lexord } r)$
 $\langle \text{proof} \rangle$

lemma *lexord-linear*: $(! a \ b. (a,b) \in r \mid a = b \mid (b,a) \in r) \implies (x,y) : \text{lexord } r \mid x = y \mid (y,x) : \text{lexord } r$
 $\langle \text{proof} \rangle$

43.4 Lexicographic combination of measure functions

These are useful for termination proofs

definition

$\text{measures } fs = \text{inv-image } (\text{lex less-than}) \ (\%a. \text{map } (\%f. f \ a) \ fs)$

lemma *wf-measures[recdef-wf, simp]*: $\text{wf } (\text{measures } fs)$
 $\langle \text{proof} \rangle$

lemma *in-measures[simp]*:

$(x, y) \in \text{measures } [] = \text{False}$
 $(x, y) \in \text{measures } (f \ \# \ fs)$
 $= (f \ x < f \ y \vee (f \ x = f \ y \wedge (x, y) \in \text{measures } fs))$
 $\langle \text{proof} \rangle$

lemma *measures-less*: $f \ x < f \ y \implies (x, y) \in \text{measures } (f \ \# \ fs)$
 $\langle \text{proof} \rangle$

lemma *measures-lesseq*: $f \ x \leq f \ y \implies (x, y) \in \text{measures } fs \implies (x, y) \in \text{measures } (f \ \# \ fs)$
 $\langle \text{proof} \rangle$

43.4.1 Lifting a Relation on List Elements to the Lists

inductive-set

$\text{listrel} :: ('a * 'a) \text{set} \implies ('a \text{ list} * 'a \text{ list}) \text{set}$
for $r :: ('a * 'a) \text{set}$

where

$Nil: ([], []) \in listrel\ r$
 $| Cons: [| (x,y) \in r; (xs,ys) \in listrel\ r |] ==> (x\#xs, y\#ys) \in listrel\ r$

inductive-cases *listrel-Nil1* [*elim!*]: $([],xs) \in listrel\ r$

inductive-cases *listrel-Nil2* [*elim!*]: $(xs,[]) \in listrel\ r$

inductive-cases *listrel-Cons1* [*elim!*]: $(y\#ys,xs) \in listrel\ r$

inductive-cases *listrel-Cons2* [*elim!*]: $(xs,y\#ys) \in listrel\ r$

lemma *listrel-mono*: $r \subseteq s \implies listrel\ r \subseteq listrel\ s$
 $\langle proof \rangle$

lemma *listrel-subset*: $r \subseteq A \times A \implies listrel\ r \subseteq lists\ A \times lists\ A$
 $\langle proof \rangle$

lemma *listrel-refl-on*: $refl-on\ A\ r \implies refl-on\ (lists\ A)\ (listrel\ r)$
 $\langle proof \rangle$

lemma *listrel-sym*: $sym\ r \implies sym\ (listrel\ r)$
 $\langle proof \rangle$

lemma *listrel-trans*: $trans\ r \implies trans\ (listrel\ r)$
 $\langle proof \rangle$

theorem *equiv-listrel*: $equiv\ A\ r \implies equiv\ (lists\ A)\ (listrel\ r)$
 $\langle proof \rangle$

lemma *listrel-Nil* [*simp*]: $listrel\ r \text{ “ } \{[]\} = \{[]\}$
 $\langle proof \rangle$

lemma *listrel-Cons*:
 $listrel\ r \text{ “ } \{x\#xs\} = set-Cons\ (r\text{ “ } \{x\})\ (listrel\ r \text{ “ } \{xs\})$
 $\langle proof \rangle$

43.5 Size function

lemma [*measure-function*]: $is-measure\ f \implies is-measure\ (list-size\ f)$
 $\langle proof \rangle$

lemma [*measure-function*]: $is-measure\ f \implies is-measure\ (option-size\ f)$
 $\langle proof \rangle$

lemma *list-size-estimation*[*termination-simp*]:
 $x \in set\ xs \implies y < f\ x \implies y < list-size\ f\ xs$
 $\langle proof \rangle$

lemma *list-size-estimation'*[*termination-simp*]:
 $x \in set\ xs \implies y \leq f\ x \implies y \leq list-size\ f\ xs$

$\langle proof \rangle$

lemma *list-size-map*[simp]: $list\text{-}size\ f\ (map\ g\ xs) = list\text{-}size\ (f\ o\ g)\ xs$
 $\langle proof \rangle$

lemma *list-size-pointwise*[termination-simp]:
 $(\bigwedge x. x \in set\ xs \implies f\ x < g\ x) \implies list\text{-}size\ f\ xs \leq list\text{-}size\ g\ xs$
 $\langle proof \rangle$

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
 $embed\text{-}list\ (return\text{-}list\ l) = l$
 $\langle proof \rangle$

lemma *transfer-nat-int-list-functions*:

l @ *m* = *return-list (embed-list l @ embed-list m)*

$[]$ = *return-list* $[]$

$\langle proof \rangle$

43.7 Code generator

43.7.1 Setup

$\langle ML \rangle$

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 ((- ::/ -))

⟨ML⟩

```

43.7.2 Generation of efficient code

definition *member* :: 'a *list* ⇒ 'a ⇒ bool **where**
mem-iff [*code-post*]: *member xs x* ⇔ *x* ∈ *set xs*

primrec
null:: 'a *list* ⇒ bool
where
null [] = *True*
| *null* (x#*xs*) = *False*

primrec
list-inter :: 'a *list* ⇒ 'a *list* ⇒ 'a *list*
where

```

list-inter [] bs = []
| list-inter (a#as) bs =
  (if a ∈ set bs then a # list-inter as bs else list-inter as bs)

```

primrec

```
list-all :: ('a ⇒ bool) ⇒ ('a list ⇒ bool)
```

where

```
list-all P [] = True
| list-all P (x#xs) = (P x ∧ list-all P xs)
```

primrec

```
list-ex :: ('a ⇒ bool) ⇒ ('a list ⇒ bool)
```

where

```
list-ex P [] = False
| list-ex P (x#xs) = (P x ∨ list-ex P xs)
```

primrec

```
filtermap :: ('a ⇒ 'b option) ⇒ ('a list ⇒ 'b list)
```

where

```
filtermap f [] = []
| filtermap f (x#xs) =
  (case f x of None ⇒ filtermap f xs
   | Some y ⇒ y # filtermap f xs)
```

primrec

```
map-filter :: ('a ⇒ 'b) ⇒ ('a ⇒ bool) ⇒ ('a list ⇒ 'b list)
```

where

```
map-filter f P [] = []
| map-filter f P (x#xs) =
  (if P x then f x # map-filter f P xs else map-filter f P xs)
```

primrec

```
length-unique :: 'a list ⇒ nat
```

where

```
length-unique [] = 0
| length-unique (x#xs) =
  (if x ∈ set xs then length-unique xs else Suc (length-unique xs))
```

primrec

```
concat-map :: ('a ⇒ 'b list) ⇒ 'a list ⇒ 'b list
```

where

```
concat-map f [] = []
| concat-map f (x#xs) = f x @ 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*]:
 $rev\ xs = foldl\ (\lambda xs\ x.\ x \# xs)\ []\ xs$
 ⟨*proof*⟩

lemmas *in-set-code* [*code-unfold*] = *mem-iff* [*symmetric*]

lemma *member-simps* [*simp*, *code*]:
 $member\ (x \# xs)\ y \longleftrightarrow x = y \vee member\ xs\ y$
 $member\ []\ y \longleftrightarrow False$
 ⟨*proof*⟩

lemma *member-set*:
 $member = set$
 ⟨*proof*⟩

abbreviation (*input*) *mem* :: 'a \Rightarrow 'a list \Rightarrow bool (**infixl** *mem* 55) **where**
 $x\ mem\ xs \equiv member\ xs\ x$ — for backward compatibility

lemma *empty-null*:
 $xs = [] \longleftrightarrow null\ xs$
 ⟨*proof*⟩

lemma [*code-unfold*]:
 $eq-class.eq\ xs\ [] \longleftrightarrow null\ xs$
 ⟨*proof*⟩

lemmas *null-empty* [*code-post*] =
empty-null [*symmetric*]

lemma *list-inter-conv*:
 $set\ (list-inter\ xs\ ys) = set\ xs \cap set\ ys$
 ⟨*proof*⟩

lemma *list-all-iff* [*code-post*]:
 $list-all\ P\ xs \longleftrightarrow (\forall x \in set\ xs.\ P\ x)$
 ⟨*proof*⟩

lemmas *list-ball-code* [*code-unfold*] = *list-all-iff* [*symmetric*]

lemma *list-all-append* [*simp*]:
 $list-all\ P\ (xs\ @\ ys) \longleftrightarrow (list-all\ P\ xs \wedge list-all\ P\ ys)$
 ⟨*proof*⟩

lemma *list-all-rev* [*simp*]:

list-all P (*rev* xs) \longleftrightarrow *list-all* P xs
 $\langle proof \rangle$

lemma *list-all-length*:
list-all P $xs \longleftrightarrow (\forall n < \text{length } xs. P (xs ! n))$
 $\langle proof \rangle$

lemma *list-ex-iff* [*code-post*]:
list-ex P $xs \longleftrightarrow (\exists x \in \text{set } xs. P x)$
 $\langle proof \rangle$

lemmas *list-bex-code* [*code-unfold*] =
list-ex-iff [*symmetric*]

lemma *list-ex-length*:
list-ex P $xs \longleftrightarrow (\exists n < \text{length } xs. P (xs ! n))$
 $\langle proof \rangle$

lemma *filtermap-conv*:
filtermap f $xs = \text{map } (\lambda x. \text{the } (f x)) (\text{filter } (\lambda x. f x \neq \text{None}) xs)$
 $\langle proof \rangle$

lemma *map-filter-conv* [*simp*]:
map-filter f P $xs = \text{map } f (\text{filter } P xs)$
 $\langle proof \rangle$

lemma *length-remdups-length-unique* [*code-unfold*]:
 $\text{length } (\text{remdups } xs) = \text{length-unique } xs$
 $\langle proof \rangle$

lemma *concat-map-code* [*code-unfold*]:
 $\text{concat}(\text{map } f xs) = \text{concat-map } f xs$
 $\langle proof \rangle$

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*]:
 $\{..n\} = \text{set } [0..<\text{Suc } n]$
 $\langle proof \rangle$

lemma *atLeast-upt* [*code-unfold*]:
 $\{..<n\} = \text{set } [0..<n]$
 $\langle proof \rangle$

lemma *greaterThanLessThan-upt* [code-unfold]:

$\{n < .. < m\} = \text{set } [\text{Suc } n .. < m]$
 $\langle \text{proof} \rangle$

lemmas *atLeastLessThan-upt* [code-unfold] = *set-upt* [symmetric]

lemma *greaterThanAtMost-upt* [code-unfold]:

$\{n < .. m\} = \text{set } [\text{Suc } n .. < \text{Suc } m]$
 $\langle \text{proof} \rangle$

lemma *atLeastAtMost-upt* [code-unfold]:

$\{n .. m\} = \text{set } [n .. < \text{Suc } m]$
 $\langle \text{proof} \rangle$

lemma *all-nat-less-eq* [code-unfold]:

$(\forall m < n :: \text{nat}. P \ m) \longleftrightarrow (\forall m \in \{0 .. < n\}. P \ m)$
 $\langle \text{proof} \rangle$

lemma *ex-nat-less-eq* [code-unfold]:

$(\exists m < n :: \text{nat}. P \ m) \longleftrightarrow (\exists m \in \{0 .. < n\}. P \ m)$
 $\langle \text{proof} \rangle$

lemma *all-nat-less* [code-unfold]:

$(\forall m \leq n :: \text{nat}. P \ m) \longleftrightarrow (\forall m \in \{0 .. n\}. P \ m)$
 $\langle \text{proof} \rangle$

lemma *ex-nat-less* [code-unfold]:

$(\exists m \leq n :: \text{nat}. P \ m) \longleftrightarrow (\exists m \in \{0 .. n\}. P \ m)$
 $\langle \text{proof} \rangle$

lemma *setsum-set-distinct-conv-listsum*:

$\text{distinct } xs \implies \text{setsum } f \ (\text{set } xs) = \text{listsum } (\text{map } f \ xs)$
 $\langle \text{proof} \rangle$

lemma *setsum-set-upt-conv-listsum* [code-unfold]:

$\text{setsum } f \ (\text{set } [m .. < n]) = \text{listsum } (\text{map } f \ [m .. < n])$
 $\langle \text{proof} \rangle$

General equivalence between *listsum* and *setsum*

lemma *listsum-setsum-nth*:

$\text{listsum } xs = (\sum \ i = 0 .. < \text{length } xs. xs \ ! \ i)$
 $\langle \text{proof} \rangle$

Code for summation over ints.

lemma *greaterThanLessThan-upto* [code-unfold]:

$\{i < .. < j :: \text{int}\} = \text{set } [i+1 .. j - 1]$
 $\langle \text{proof} \rangle$

lemma *atLeastLessThan-upto* [code-unfold]:

$\{i..<j::int\} = \text{set } [i..j - 1]$

$\langle \text{proof} \rangle$

lemma *greaterThanAtMost-upto* [code-unfold]:

$\{i<..j::int\} = \text{set } [i+1..j]$

$\langle \text{proof} \rangle$

lemmas *atLeastAtMost-upto* [code-unfold] = *set-upto*[symmetric]

lemma *setsum-set-upto-conv-listsum* [code-unfold]:

$\text{setsum } f (\text{set } [i..j::int]) = \text{listsum } (\text{map } f [i..j])$

$\langle \text{proof} \rangle$

Optimized code for $\forall i \in \{a..b::int\}$ and $\forall n: \{a..<b::nat\}$ and similarly for \exists .

function *all-from-to-nat* :: (nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat \Rightarrow 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)

$\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

declare *all-from-to-nat.simps*[simp del]

lemma *all-from-to-nat-iff-ball*:

$\text{all-from-to-nat } P \ i \ j = (\text{ALL } n : \{i..<j\}. P \ n)$

$\langle \text{proof} \rangle$

lemma *list-all-iff-all-from-to-nat*[code-unfold]:

$\text{list-all } P \ [i..<j] = \text{all-from-to-nat } P \ i \ j$

$\langle \text{proof} \rangle$

lemma *list-ex-iff-not-all-from-to-not-nat*[code-unfold]:

$\text{list-ex } P \ [i..<j] = (\sim \text{all-from-to-nat } (\%x. \sim P \ x) \ i \ j)$

$\langle \text{proof} \rangle$

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)

$\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

declare *all-from-to-int.simps*[simp del]

lemma *all-from-to-int-iff-ball*:
 $all\text{-}from\text{-}to\text{-}int\ P\ i\ j = (ALL\ n : \{i .. j\}. P\ n)$
 $\langle proof \rangle$

lemma *list-all-iff-all-from-to-int*[code-unfold]:
 $list\text{-}all\ P\ [i..j] = all\text{-}from\text{-}to\text{-}int\ P\ i\ j$
 $\langle proof \rangle$

lemma *list-ex-iff-not-all-from-to-not-int*[code-unfold]:
 $list\text{-}ex\ P\ [i..j] = (\sim all\text{-}from\text{-}to\text{-}int\ (\%x. \sim P\ x)\ i\ j)$
 $\langle proof \rangle$

end

44 Random: A HOL random engine

theory *Random*
imports *Code-Numeral List*
begin

notation *fcomp* (infixl $o> 60$)
notation *scomp* (infixl $o\rightarrow 60$)

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**
 $inc\text{-}shift\ v\ k = (if\ v = k\ then\ 1\ else\ k + 1)$

definition *minus-shift* :: *code-numeral* \Rightarrow *code-numeral* \Rightarrow *code-numeral* \Rightarrow *code-numeral* **where**
 $minus\text{-}shift\ r\ k\ l = (if\ k < l\ then\ r + k - l\ else\ k - l)$

44.2 Random seeds

types *seed* = *code-numeral* \times *code-numeral*

primrec *next* :: *seed* \Rightarrow *code-numeral* \times *seed* **where**
 $next\ (v, w) = (let$
 $\quad k = v\ div\ 53668;$
 $\quad v' = minus\text{-}shift\ 2147483563\ ((v\ mod\ 53668) * 40014)\ (k * 12211);$
 $\quad l = w\ div\ 52774;$
 $\quad w' = minus\text{-}shift\ 2147483399\ ((w\ mod\ 52774) * 40692)\ (l * 3791);$
 $\quad z = minus\text{-}shift\ 2147483562\ v'\ (w' + 1) + 1$
 $in\ (z, (v', w')))$

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*)
 ($\lambda l. \text{next } o \rightarrow (\lambda v. \text{Pair } (v + l * 2147483561)))$ 1
 $o \rightarrow (\lambda v. \text{Pair } (v \bmod k))$

lemma *range*:

k > 0 \implies *fst* (*range* *k* *s*) < *k*
 <proof>

definition *select* :: '*a* list \Rightarrow *seed* \Rightarrow '*a* \times *seed* **where**

select *xs* = *range* (*Code-Numeral.of-nat* (*length* *xs*))
 $o \rightarrow (\lambda k. \text{Pair } (\text{nth } xs \text{ (Code-Numeral.nat-of } k)))$

lemma *select*:

assumes *xs* $\neq []$
shows *fst* (*select* *xs* *s*) \in *set* *xs*
 <proof>

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*)
 <proof>

lemma *pick-drop-zero*:

pick (*filter* ($\lambda(k, -). k > 0$) *xs*) = *pick* *xs*
 <proof>

lemma *pick-same*:

l < *length* *xs* \implies *Random.pick* (*map* (*Pair* 1) *xs*) (*Code-Numeral.of-nat* *l*) =
nth *xs* *l*
 <proof>

definition *select-weight* :: (*code-numeral* \times '*a*) list \Rightarrow *seed* \Rightarrow '*a* \times *seed* **where**

```

select-weight xs = range (listsum (map fst xs))
o→ (λk. Pair (pick xs k))

```

```

lemma select-weight-member:
  assumes 0 < listsum (map fst xs)
  shows fst (select-weight xs s) ∈ set (map snd xs)
  ⟨proof⟩

```

```

lemma select-weight-cons-zero:
  select-weight ((0, x) # xs) = select-weight xs
  ⟨proof⟩

```

```

lemma select-weight-drop-zero:
  select-weight (filter (λ(k, -). k > 0) xs) = select-weight xs
  ⟨proof⟩

```

```

lemma select-weight-select:
  assumes xs ≠ []
  shows select-weight (map (Pair 1) xs) = select xs
  ⟨proof⟩

```

44.4 ML interface

```

code-reflect Random-Engine
  functions range select select-weight

```

```

⟨ML⟩

```

```

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⟩

instance *nibble* :: *finite*

⟨proof⟩

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⟩

instance *char* :: *finite*

⟨proof⟩

lemma *size-char* [*code*, *simp*]:

$\text{size } (c::\text{char}) = 0$ ⟨proof⟩

lemma *char-size* [*code*, *simp*]:

$\text{char-size } (c::\text{char}) = 0$ ⟨proof⟩

primrec *nibble-pair-of-char* :: *char* \Rightarrow *nibble* \times *nibble* **where**

nibble-pair-of-char (*Char n m*) = (*n*, *m*)

⟨ML⟩

lemma *char-case-nibble-pair* [*code*, *code-unfold*]:

$\text{char-case } f = \text{split } f \circ \text{nibble-pair-of-char}$
⟨proof⟩

lemma *char-rec-nibble-pair* [*code*, *code-unfold*]:

$\text{char-rec } f = \text{split } f \circ \text{nibble-pair-of-char}$
⟨proof⟩

syntax

-*Char* :: *xstr* \Rightarrow *char* (*CHR* -)

45.2 Strings

types *string* = *char list*

syntax

-*String* :: *xstr* => *string* (-)

⟨ML⟩

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*
 ⟨*proof*⟩

lemma [*code*]: *literal-size (s::literal) = 0*
 ⟨*proof*⟩

45.4 Code generator

⟨*ML*⟩

code-reserved *SML string*

code-reserved *OCaml string*

code-reserved *Scala string*

code-type *literal*


```

    (SML string)
    (OCaml string)
    (Haskell String)
    (Scala String)

  ⟨ML⟩

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;
  ⟩⟩

  ⟨ML⟩

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'(-'))))

<ML>

lemma [code]:
  eq-class.eq (Typerep tyco1 tys1) (Typerep tyco2 tys2)  $\longleftrightarrow$  eq-class.eq tyco1 tyco2
     $\wedge$  list-all2 eq-class.eq tys1 tys2
  <proof>

code-type typerep
  (Eval Term.typ)

code-const Typerep
  (Eval Term.Type / (-, -))

code-reserved Eval Term

hide-const (open) typerep Typerep

end

```

47 Code-Evaluation: Term evaluation using the generic code generator

```

theory Code-Evaluation
imports Plain Typerep Code-Numeral
begin

```

47.1 Term representation

47.1.1 Terms and class *term-of*

```

datatype term = dummy-term

```

```

definition Const :: String.literal  $\Rightarrow$  typerep  $\Rightarrow$  term where
  Const - - = dummy-term

```

```

definition App :: term  $\Rightarrow$  term  $\Rightarrow$  term where
  App - - = dummy-term

```

```

code-datatype Const App

```

```

class term-of = typerep +
  fixes term-of :: 'a  $\Rightarrow$  term

```

```

lemma term-of-anything: term-of x  $\equiv$  t

```

$\langle \text{proof} \rangle$

definition $\text{valapp} :: ('a \Rightarrow 'b) \times (\text{unit} \Rightarrow \text{term})$
 $\Rightarrow 'a \times (\text{unit} \Rightarrow \text{term}) \Rightarrow 'b \times (\text{unit} \Rightarrow \text{term})$ **where**
 $\text{valapp } f \ x = (\text{fst } f \ (\text{fst } x), \lambda u. \text{App } (\text{snd } f \ ()) \ (\text{snd } x \ ()))$

lemma valapp-code $[\text{code}, \text{code-unfold}]$:
 $\text{valapp } (f, \text{tf}) \ (x, \text{tx}) = (f \ x, \lambda u. \text{App } (\text{tf } ()) \ (\text{tx } ()))$
 $\langle \text{proof} \rangle$

47.1.2 term-of instances

instantiation $\text{fun} :: (\text{typerep}, \text{typerep}) \text{ term-of}$
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 $\langle \text{proof} \rangle$

end

$\langle \text{ML} \rangle$

47.1.3 Code generator setup

lemmas $[\text{code del}] = \text{term.recs term.cases term.size}$

lemma $[\text{code}, \text{code del}]$: $\text{eq-class.eq } (t1 :: \text{term}) \ t2 \longleftrightarrow \text{eq-class.eq } t1 \ t2 \ \langle \text{proof} \rangle$

lemma $[\text{code}, \text{code del}]$: $(\text{term-of} :: \text{typerep} \Rightarrow \text{term}) = \text{term-of} \ \langle \text{proof} \rangle$

lemma $[\text{code}, \text{code del}]$: $(\text{term-of} :: \text{term} \Rightarrow \text{term}) = \text{term-of} \ \langle \text{proof} \rangle$

lemma $[\text{code}, \text{code del}]$: $(\text{term-of} :: \text{String.literal} \Rightarrow \text{term}) = \text{term-of} \ \langle \text{proof} \rangle$

lemma $[\text{code}, \text{code del}]$:
 $(\text{Code-Evaluation.term-of} :: 'a :: \{\text{type}, \text{term-of}\} \text{ Predicate.pred} \Rightarrow \text{Code-Evaluation.term})$
 $= \text{Code-Evaluation.term-of} \ \langle \text{proof} \rangle$

lemma $[\text{code}, \text{code del}]$:
 $(\text{Code-Evaluation.term-of} :: 'a :: \{\text{type}, \text{term-of}\} \text{ Predicate.seq} \Rightarrow \text{Code-Evaluation.term})$
 $= \text{Code-Evaluation.term-of} \ \langle \text{proof} \rangle$

lemma term-of-char $[\text{unfolded typerep-fun-def typerep-char-def typerep-nibble-def},$
 $\text{code}]$: $\text{Code-Evaluation.term-of } c =$
 $(\text{let } (n, m) = \text{nibble-pair-of-char } c$
 $\text{in } \text{Code-Evaluation.App } (\text{Code-Evaluation.App } (\text{Code-Evaluation.Const } (\text{STR } \text{"String.char.Char"})$
 $(\text{TYPEREP}(\text{nibble} \Rightarrow \text{nibble} \Rightarrow \text{char})))$
 $(\text{Code-Evaluation.term-of } n)) \ (\text{Code-Evaluation.term-of } m))$
 $\langle \text{proof} \rangle$

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 HLogic.mk'-literal*)

code-reserved *Eval HLogic*

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*)

$\langle ML \rangle$

locale *term-syntax*
begin

notation *App* (**infixl** $\langle \cdot \rangle$ 70)
and *valapp* (**infixl** $\{ \cdot \}$ 70)

end

interpretation *term-syntax* $\langle proof \rangle$

no-notation *App* (**infixl** $\langle \cdot \rangle$ 70)
and *valapp* (**infixl** $\{ \cdot \}$ 70)

47.2 Numeric types

definition *term-of-num* :: '*a*::{*semiring-div*} \Rightarrow '*a*::{*semiring-div*} \Rightarrow *term* **where**
term-of-num two = ($\lambda -.$ *dummy-term*)

lemma (**in** *term-syntax*) *term-of-num-code* [*code*]:
term-of-num two *k* = (*if* *k* = 0 *then* *termify Int.Pls*
else (*if* *k* mod *two* = 0

then *termify Int.Bit0* $\langle \cdot \rangle$ *term-of-num two* (*k* div *two*)

else *termify Int.Bit1* $\langle \cdot \rangle$ *term-of-num two* (*k* div *two*)))

$\langle proof \rangle$

lemma (**in** *term-syntax*) *term-of-nat-code* [*code*]:
term-of (*n*::*nat*) = *termify* (*number-of* :: *int* \Rightarrow *nat*) $\langle \cdot \rangle$ *term-of-num* (2::*nat*)

n

$\langle proof \rangle$

```

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)))
  ⟨proof⟩

```

```

lemma (in term-syntax) term-of-code-numeral-code [code]:
  term-of (k::code-numeral) = termify (number-of :: int ⇒ code-numeral) <·>
term-of-num (2::code-numeral) k
  ⟨proof⟩

```

47.3 Obfuscate

⟨*ML*⟩

```

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 => 'a => 'a
where
  [code del]: tracing s x = x

```

⟨*ML*⟩

```

code-const tracing :: String.literal => 'a => 'a
  (Eval Code'-Evaluation.tracing)

```

```

hide-const (open) tracing
code-reserved Eval Code-Evaluation

```

47.5 Evaluation setup

⟨*ML*⟩

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* →
 (λ*k*. *Pair* (if *k* = 0 then *Code-Evaluation.valtermify* False else *Code-Evaluation.valtermify* True))

instance ⟨*proof*⟩

end

instantiation *itself* :: (*typerep*) *random*
begin

definition *random-itself* :: *code-numeral* ⇒ *Random.seed* ⇒ ('a *itself* × (*unit* ⇒ *term*)) × *Random.seed* **where**
random-itself - = *Pair* (*Code-Evaluation.valtermify* TYPE('a))

instance ⟨*proof*⟩

end

instantiation *char* :: *random*
begin

definition
random - = *Random.select* *chars* *o* → (λ*c*. *Pair* (*c*, λ*u*. *Code-Evaluation.term-of* *c*))

instance ⟨*proof*⟩

end

instantiation *String.literal* :: *random*
begin

definition
random - = *Pair* (*STR* "", λ*u*. *Code-Evaluation.term-of* (*STR* ""))

instance $\langle proof \rangle$

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) $\circ \rightarrow$ (λk . *Pair* (
 let *n* = *Code-Numeral.nat-of* *k*
 in (*n*, λ -. *Code-Evaluation.term-of* *n*)))

instance $\langle proof \rangle$

end

instantiation *int* :: *random*
begin

definition
 random *i* = *Random.range* ($2 * i + 1$) $\circ \rightarrow$ (λk . *Pair* (
 let *j* = (*if* *k* \geq *i* *then* *Code-Numeral.int-of* (*k* - *i*) *else* - *Code-Numeral.int-of*
 (*i* - *k*))
 in (*j*, λ -. *Code-Evaluation.term-of* *j*)))

instance $\langle proof \rangle$

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 (*unit* \Rightarrow *term*)) \times *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 (*unit* \Rightarrow *term*)) \times *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* :: ($\{eq, \text{term-of}\}$, *random*) *random*
begin

definition *random-fun* :: *code-numeral* \Rightarrow *Random.seed* \Rightarrow (*'a* \Rightarrow *'b*) \times (*unit* \Rightarrow *term*) \times *Random.seed* **where**
random i = *random-fun-lift* (*random i*)

instance \langle *proof* \rangle

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* \rightarrow *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
 \langle *proof* \rangle

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*
 \langle *proof* \rangle

\langle *ML* \rangle

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 $[code]:$

$iter\ nrandom\ sz\ seed = (if\ nrandom = 0\ then\ bot-class.bot\ else$

$let\ ((x, -),\ seed') = random\ sz\ seed$

$in\ Predicate.Seq\ (\%u.\ Predicate.Insert\ x\ (iter\ (nrandom - 1)\ sz\ seed'))$

$\langle proof \rangle$

types $'a\ randompred = Random.seed \Rightarrow ('a\ Predicate.pred \times Random.seed)$

definition $empty :: 'a\ randompred$

where $empty = Pair\ (bot-class.bot)$

definition $single :: 'a \Rightarrow 'a\ randompred$

where $single\ x = Pair\ (Predicate.single\ x)$

definition $bind :: 'a\ randompred \Rightarrow ('a \Rightarrow 'b\ randompred) \Rightarrow 'b\ randompred$

where

$bind\ R\ f = (\lambda s.\ let$

$(P,\ s') = R\ s;$

$(s1,\ s2) = Random.split-seed\ s'$

$in\ (Predicate.bind\ P\ (\%a.\ fst\ (f\ a\ s1)),\ s2))$

definition $union :: 'a\ randompred \Rightarrow 'a\ randompred \Rightarrow 'a\ randompred$

where

$union\ R1\ R2 = (\lambda s.\ let$

$(P1,\ s') = R1\ s;\ (P2,\ s'') = R2\ s'$

$in\ (semilattice-sup-class.sup\ P1\ P2,\ s''))$

definition $if-randompred :: bool \Rightarrow unit\ randompred$

where

$if-randompred\ b = (if\ b\ then\ single\ ()\ else\ empty)$

definition $iterate-upto :: (code-numeral \Rightarrow 'a) \Rightarrow code-numeral \Rightarrow code-numeral \Rightarrow 'a\ randompred$

where

$iterate-upto\ f\ n\ m = Pair\ (Code-Numeral.iterate-upto\ f\ n\ m)$

definition $not-randompred :: unit\ randompred \Rightarrow unit\ randompred$

where

$not-randompred\ P = (\lambda s.\ let$

$(P',\ s') = P\ s$

$in\ if\ Predicate.eval\ P'\ ()\ then\ (Orderings.bot,\ s')\ else\ (Predicate.single\ (),\ s'))$

definition $Random :: (Random.seed \Rightarrow ('a \times (unit \Rightarrow term)) \times Random.seed) \Rightarrow$

```

'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
  <proof>

lemma yield-Seq [code]:
  yield (Lazy-Sequence f) = f ()
  <proof>

lemma Seq-yield:
  Lazy-Sequence (%u. yield f) = f
  <proof>

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)

```

$\langle \text{proof} \rangle$

lemma *size-code* [code]:

size xq = (case *yield xq* of *None* => 0 | *Some* (*x*, *xq'*) => *size xq'* + 1)

$\langle \text{proof} \rangle$

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*) \wedge
(*eq-class.eq xq yq*) | - => *False*)

$\langle \text{proof} \rangle$

lemma *seq-case* [code]:

lazy-sequence-case f g xq = (case (*yield xq*) of *None* => *f* | *Some* (*x*, *xq'*) => *g x xq'*)

$\langle \text{proof} \rangle$

lemma [code]: *lazy-sequence-rec f g xq* = (case (*yield xq*) of *None* => *f* | *Some* (*x*, *xq'*) => *g x xq'* (*lazy-sequence-rec f g xq'*))

$\langle \text{proof} \rangle$

definition *empty* :: 'a lazy-sequence

where

[code]: *empty* = *Lazy-Sequence* (%u. *None*)

definition *single* :: 'a => 'a lazy-sequence

where

[code]: *single x* = *Lazy-Sequence* (%u. *Some* (*x*, *empty*))

primrec *append* :: 'a lazy-sequence => 'a lazy-sequence => 'a lazy-sequence

where

append Empty yq = *yq*

| *append* (*Insert x xq*) *yq* = *Insert x* (*append xq yq*)

lemma [code]:

append xq yq = *Lazy-Sequence* (%u. case *yield xq* of

None => *yield yq*

| *Some* (*x*, *xq'*) => *Some* (*x*, *append xq' yq*))

$\langle \text{proof} \rangle$

primrec *flat* :: 'a lazy-sequence lazy-sequence => 'a lazy-sequence

where

flat Empty = *Empty*

| *flat* (*Insert xq xqq*) = *append xq* (*flat xqq*)

lemma [code]:

flat xqq = *Lazy-Sequence* (%u. case *yield xqq* of

None => *None*

| *Some* (*xq*, *xqq'*) => *yield* (*append xq* (*flat xqq'*)))

$\langle \text{proof} \rangle$

primrec *map* :: ('a => 'b) => 'a lazy-sequence => 'b lazy-sequence

where

map f Empty = *Empty*
| *map f (Insert x xq)* = *Insert (f x) (map f xq)*

lemma [code]:

map f xq = *Lazy-Sequence (%u. Option.map (%(x, xq'). (f x, map f xq')) (yield xq))*
⟨proof⟩

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))*
⟨proof⟩

termination ⟨proof⟩

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')*))
⟨proof⟩

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*))
⟨proof⟩

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⟩

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 $'a \text{ random-dseq} = \text{code-numeral} \Rightarrow \text{code-numeral} \Rightarrow \text{Random.seed} \Rightarrow ('a \text{ DSequence.dseq} \times \text{Random.seed})$

definition $\text{empty} :: 'a \text{ random-dseq}$

where

$\text{empty} = (\%nrandom \text{ size. Pair } (DSequence.empty))$

definition $\text{single} :: 'a \Rightarrow 'a \text{ random-dseq}$

where

$\text{single } x = (\%nrandom \text{ size. Pair } (DSequence.single \ x))$

definition $\text{bind} :: 'a \text{ random-dseq} \Rightarrow ('a \Rightarrow 'b \text{ random-dseq}) \Rightarrow 'b \text{ random-dseq}$

where

$\text{bind } R \ f = (\lambda nrandom \text{ size } s. \text{ let } (P, s') = R \ nrandom \text{ size } s;$
 $(s1, s2) = \text{Random.split-seed } s'$
 $\text{in } (DSequence.bind \ P \ (\%a. \text{fst } (f \ a \ nrandom \text{ size } s1)), s2))$

definition $\text{union} :: 'a \text{ random-dseq} \Rightarrow 'a \text{ random-dseq} \Rightarrow 'a \text{ random-dseq}$

where

$\text{union } R1 \ R2 = (\lambda nrandom \text{ size } s. \text{ let } (S1, s') = R1 \ nrandom \text{ size } s; (S2, s'') = R2 \ nrandom \text{ size } s'$
 $\text{in } (DSequence.union \ S1 \ S2, s''))$

definition $\text{if-random-dseq} :: \text{bool} \Rightarrow \text{unit random-dseq}$

where

$\text{if-random-dseq } b = (\text{if } b \text{ then single } () \text{ else empty})$

definition $\text{not-random-dseq} :: \text{unit random-dseq} \Rightarrow \text{unit random-dseq}$

where

$\text{not-random-dseq } R = (\lambda nrandom \text{ size } s. \text{ let } (S, s') = R \ nrandom \text{ size } s$
 $\text{in } (DSequence.not-seq \ S, s'))$

fun $\text{Random} :: (\text{code-numeral} \Rightarrow \text{Random.seed} \Rightarrow (('a \times (\text{unit} \Rightarrow \text{term})) \times \text{Random.seed})) \Rightarrow 'a \text{ random-dseq}$

where

$\text{Random } g \ nrandom = (\%size. \text{if } nrandom \leq 0 \text{ then } (Pair \ DSequence.empty)$
 $\text{else } (scomp \ (g \ size) \ (\%r. \ scomp \ (\text{Random } g \ (nrandom - 1) \ size) \ (\%rs. \ Pair \ (DSequence.union \ (DSequence.single \ (\text{fst } r)) \ rs))))))$

definition $\text{map} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ random-dseq} \Rightarrow 'b \text{ random-dseq}$

where

$\text{map } f \ P = \text{bind } P \ (\text{single } o \ f)$

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 pos-dseq = code-numeral => 'a Lazy-Sequence.lazy-sequence*

definition *pos-empty :: 'a pos-dseq*

where

pos-empty = (%i. Lazy-Sequence.empty)

definition *pos-single :: 'a => 'a pos-dseq*

where

pos-single x = (%i. Lazy-Sequence.single x)

definition *pos-bind :: 'a pos-dseq => ('a => 'b pos-dseq) => 'b pos-dseq*

where

pos-bind x f = (%i. if i = 0 then Lazy-Sequence.empty else Lazy-Sequence.bind (x (i - 1)) (%a. f a i))

definition *pos-union :: 'a pos-dseq => 'a pos-dseq => 'a pos-dseq*

where

pos-union xq yq = (%i. Lazy-Sequence.append (xq i) (yq i))

definition *pos-if-seq :: bool => unit pos-dseq*

where

pos-if-seq b = (if b then pos-single () else pos-empty)

definition *pos-iterate-upto :: (code-numeral => 'a) => code-numeral => code-numeral => 'a pos-dseq*

where

pos-iterate-upto $f\ n\ m = (\%i. \text{Lazy-Sequence.iterate-upto } f\ n\ m)$

definition *pos-map* $:: ('a \Rightarrow 'b) \Rightarrow 'a\ \text{pos-dseq} \Rightarrow 'b\ \text{pos-dseq}$
where
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 *neg-empty* $:: 'a\ \text{neg-dseq}$
where
neg-empty $= (\%i. \text{Lazy-Sequence.empty})$

definition *neg-single* $:: 'a \Rightarrow 'a\ \text{neg-dseq}$
where
neg-single $x = (\%i. \text{Lazy-Sequence.hb-single } x)$

definition *neg-bind* $:: 'a\ \text{neg-dseq} \Rightarrow ('a \Rightarrow 'b\ \text{neg-dseq}) \Rightarrow 'b\ \text{neg-dseq}$
where
neg-bind $x\ f = (\%i. \text{if } i = 0 \text{ then } \text{Lazy-Sequence.hit-bound } \text{hb-bind } (x\ (i - 1))\ (\%a. f\ a\ i))$

definition *neg-union* $:: 'a\ \text{neg-dseq} \Rightarrow 'a\ \text{neg-dseq} \Rightarrow 'a\ \text{neg-dseq}$
where
neg-union $x\ y = (\%i. \text{Lazy-Sequence.append } (x\ i)\ (y\ i))$

definition *neg-if-seq* $:: \text{bool} \Rightarrow \text{unit}\ \text{neg-dseq}$
where
neg-if-seq $b = (\text{if } b \text{ then } \text{neg-single } () \text{ else } \text{neg-empty})$

definition *neg-iterate-upto*
where
neg-iterate-upto $f\ n\ m = (\%i. \text{Lazy-Sequence.iterate-upto } (\%i. \text{Some } (f\ i))\ n\ m)$

definition *neg-map* $:: ('a \Rightarrow 'b) \Rightarrow 'a\ \text{neg-dseq} \Rightarrow 'b\ \text{neg-dseq}$
where
neg-map $f\ xq = (\%i. \text{Lazy-Sequence.hb-map } f\ (xq\ i))$

51.3 Negation

definition *pos-not-seq* $:: \text{unit}\ \text{neg-dseq} \Rightarrow \text{unit}\ \text{pos-dseq}$
where
pos-not-seq $xq = (\%i. \text{Lazy-Sequence.hb-not-seq } (xq\ i))$

definition *neg-not-seq* $:: \text{unit}\ \text{pos-dseq} \Rightarrow \text{unit}\ \text{neg-dseq}$
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 => code-numeral => Random.seed => 'a
New-DSequence.pos-dseq
types 'a neg-random-dseq = code-numeral => code-numeral => Random.seed => '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 => 'a pos-random-dseq
where
  pos-single x = (%nrandom size seed. New-DSequence.pos-single x)

definition pos-bind :: 'a pos-random-dseq => ('a => 'b pos-random-dseq) => 'b
pos-random-dseq
where
  pos-bind R f = (%nrandom size seed. New-DSequence.pos-bind (R nrandom size
seed) (%a. f a nrandom size seed))

definition pos-union :: 'a pos-random-dseq => 'a pos-random-dseq => 'a pos-random-dseq
where
  pos-union R1 R2 = (%nrandom size seed. New-DSequence.pos-union (R1 nrandom
size seed) (R2 nrandom size seed))

```

definition *pos-if-random-dseq* :: *bool* => *unit pos-random-dseq*

where

pos-if-random-dseq *b* = (if *b* then *pos-single* () else *pos-empty*)

definition *pos-iterate-upto* :: (*code-numeral* => '*a*) => *code-numeral* => *code-numeral* => '*a pos-random-dseq*

where

pos-iterate-upto *f n m* = (λ *nrandom size seed. New-DSequence.pos-iterate-upto f n m*)

definition *pos-not-random-dseq* :: *unit neg-random-dseq* => *unit pos-random-dseq*

where

pos-not-random-dseq *R* = (λ *nrandom size seed. New-DSequence.pos-not-seq (R nrandom size seed)*)

fun *iter* :: (*code-numeral* * *code-numeral* => ('*a* * (*unit* => *term*)) * *code-numeral* * *code-numeral*) => *code-numeral* => *code-numeral* * *code-numeral* => '*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* => '*b*) => '*a pos-random-dseq* => '*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* => '*a neg-random-dseq*

where

neg-single x = (%*nrandom size seed. New-DSequence.neg-single x*)

definition *neg-bind* :: '*a neg-random-dseq* => ('*a* \Rightarrow '*b neg-random-dseq*) \Rightarrow '*b neg-random-dseq*

where

neg-bind R f = (λ *nrandom size seed. New-DSequence.neg-bind (R nrandom size seed) (%a. f a nrandom size seed)*)

definition *neg-union* :: '*a neg-random-dseq* => '*a neg-random-dseq* => '*a neg-random-dseq*

where

neg-union *R1 R2* = (λn random size seed. *New-DSequence.neg-union* (*R1 nrandom size seed*) (*R2 nrandom size seed*))

definition *neg-if-random-dseq* :: *bool* => *unit neg-random-dseq*

where

neg-if-random-dseq *b* = (*if* *b* *then neg-single ()* *else neg-empty*)

definition *neg-iterate-upto* :: (*code-numeral* => '*a*) => *code-numeral* => *code-numeral* => '*a neg-random-dseq*

where

neg-iterate-upto *f n m* = (λn random size seed. *New-DSequence.neg-iterate-upto* *f n m*)

definition *neg-not-random-dseq* :: *unit pos-random-dseq* => *unit neg-random-dseq*

where

neg-not-random-dseq *R* = (λn random size seed. *New-DSequence.neg-not-seq* (*R nrandom size seed*))

definition *neg-map* :: ('*a* => '*b*) => '*a neg-random-dseq* => '*b 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

$\langle ML \rangle$

end

53 Map: Maps

theory *Map*

imports *List*

begin

types $('a, 'b)$ *map* = $'a \Rightarrow 'b$ *option* (**infixr** $\sim=>$ 0)
translations $(type)$ $'a \sim=> 'b <= (type)$ $'a \Rightarrow 'b$ *option*

type-notation $(xsymbols)$
map (**infixr** \rightarrow 0)

abbreviation

empty :: $'a \sim=> 'b$ **where**
empty == $\%x. None$

definition

map-comp :: $('b \sim=> 'c) \Rightarrow ('a \sim=> 'b) \Rightarrow ('a \sim=> 'c)$ (**infixl** *o'-m* 55)

where

$f \circ\text{-}m \ g = (\lambda k. \text{ case } g \ k \text{ of } None \Rightarrow None \mid Some \ v \Rightarrow f \ v)$

notation $(xsymbols)$

map-comp (**infixl** \circ_m 55)

definition

map-add :: $('a \sim=> 'b) \Rightarrow ('a \sim=> 'b) \Rightarrow ('a \sim=> 'b)$ (**infixl** $++$ 100)

where

$m1 \ ++ \ m2 = (\lambda x. \text{ case } m2 \ x \text{ of } None \Rightarrow m1 \ x \mid Some \ y \Rightarrow Some \ y)$

definition

restrict-map :: $('a \sim=> 'b) \Rightarrow 'a$ *set* $\Rightarrow ('a \sim=> 'b)$ (**infixl** $|'$ 110) **where**
 $m|'A = (\lambda x. \text{ if } x : A \text{ then } m \ x \text{ else } None)$

notation $(latex \text{ output})$

restrict-map $(-\restriction [111,110] \ 110)$

definition

dom :: $('a \sim=> 'b) \Rightarrow 'a$ *set* **where**
 $\text{dom } m = \{a. m \ a \sim= None\}$

definition

$ran :: ('a \rightsquigarrow 'b) \Rightarrow 'b \text{ set } \mathbf{where}$
 $ran\ m = \{b. \text{EX } a. m\ a = \text{Some } b\}$

definition

$map\text{-}le :: ('a \rightsquigarrow 'b) \Rightarrow ('a \rightsquigarrow 'b) \Rightarrow \text{bool } (\mathbf{infix} \subseteq_m 50) \mathbf{where}$
 $(m_1 \subseteq_m m_2) = (\forall a \in \text{dom } m_1. m_1\ a = m_2\ a)$

nonterminals

$maplets\ maplet$

syntax

$\text{-}maplet :: ['a, 'a] \Rightarrow maplet \quad (- \ /| \text{-} > / \ -)$
 $\text{-}maplets :: ['a, 'a] \Rightarrow maplet \quad (- \ / [| \text{-} >] / \ -)$
 $\quad \quad \quad :: maplet \Rightarrow maplets \quad (-)$
 $\text{-}Maplets :: [maplet, maplets] \Rightarrow maplets \quad (-, / \ -)$
 $\text{-}MapUpd :: ['a \rightsquigarrow 'b, maplets] \Rightarrow 'a \rightsquigarrow 'b \quad (-/'(-') [900, 0] 900)$
 $\text{-}Map \quad :: maplets \Rightarrow 'a \rightsquigarrow 'b \quad ((1[]))$

syntax (*xsymbols*)

$\text{-}maplet :: ['a, 'a] \Rightarrow maplet \quad (- \ / \mapsto / \ -)$
 $\text{-}maplets :: ['a, 'a] \Rightarrow maplet \quad (- \ / [\mapsto] / \ -)$

translations

$\text{-}MapUpd\ m\ (\text{-}Maplets\ xy\ ms) == \text{-}MapUpd\ (\text{-}MapUpd\ m\ xy)\ ms$
 $\text{-}MapUpd\ m\ (\text{-}maplet\ x\ y) == m(x := \text{CONST}\ \text{Some}\ y)$
 $\text{-}Map\ ms == \text{-}MapUpd\ (\text{CONST}\ \text{empty})\ ms$
 $\text{-}Map\ (\text{-}Maplets\ ms1\ ms2) <= \text{-}MapUpd\ (\text{-}Map\ ms1)\ ms2$
 $\text{-}Maplets\ ms1\ (\text{-}Maplets\ ms2\ ms3) <= \text{-}Maplets\ (\text{-}Maplets\ ms1\ ms2)\ ms3$

primrec

$map\text{-}of :: ('a \times 'b)\ \text{list} \Rightarrow 'a \rightarrow 'b \mathbf{where}$
 $map\text{-}of\ [] = \text{empty}$
 $| map\text{-}of\ (p\ \#\ ps) = (map\text{-}of\ ps)(fst\ p \mapsto snd\ p)$

definition

$map\text{-}upds :: ('a \rightarrow 'b) \Rightarrow 'a\ \text{list} \Rightarrow 'b\ \text{list} \Rightarrow 'a \rightarrow 'b \mathbf{where}$
 $map\text{-}upds\ m\ xs\ ys = m\ ++\ map\text{-}of\ (\text{rev}\ (\text{zip}\ xs\ ys))$

translations

$\text{-}MapUpd\ m\ (\text{-}maplets\ x\ y) == \text{CONST}\ map\text{-}upds\ m\ x\ y$

lemma *map-of-Cons-code* [*code*]:

$map\text{-}of\ []\ k = \text{None}$
 $map\text{-}of\ ((l, v)\ \#\ ps)\ k = (\text{if } l = k \text{ then } \text{Some } v \text{ else } map\text{-}of\ ps\ k)$
 $\langle \text{proof} \rangle$

53.1 *empty*

lemma *empty-upd-none* [*simp*]: $\text{empty}(x := \text{None}) = \text{empty}$

$\langle proof \rangle$

53.2 map-upd

lemma *map-upd-triv*: $t\ k = \text{Some } x \implies t(k|->x) = t$
 $\langle proof \rangle$

lemma *map-upd-nonempty* [simp]: $t(k|->x) \sim \text{empty}$
 $\langle proof \rangle$

lemma *map-upd-eqD1*:
 assumes $m(a \mapsto x) = n(a \mapsto y)$
 shows $x = y$
 $\langle proof \rangle$

lemma *map-upd-Some-unfold*:
 $((m(a|->b))\ x = \text{Some } y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = \text{Some } y)$
 $\langle proof \rangle$

lemma *image-map-upd* [simp]: $x \notin A \implies m(x \mapsto y) \text{ ‘ } A = m \text{ ‘ } A$
 $\langle proof \rangle$

lemma *finite-range-updI*: $\text{finite } (\text{range } f) \implies \text{finite } (\text{range } (f(a|->b)))$
 $\langle proof \rangle$

53.3 map-of

lemma *map-of-eq-None-iff*:
 $(\text{map-of } xys\ x = \text{None}) = (x \notin \text{fst ‘ } (\text{set } xys))$
 $\langle proof \rangle$

lemma *map-of-is-SomeD*: $\text{map-of } xys\ x = \text{Some } y \implies (x, y) \in \text{set } xys$
 $\langle proof \rangle$

lemma *map-of-eq-Some-iff* [simp]:
 $\text{distinct}(\text{map } \text{fst } xys) \implies (\text{map-of } xys\ x = \text{Some } y) = ((x, y) \in \text{set } xys)$
 $\langle proof \rangle$

lemma *Some-eq-map-of-iff* [simp]:
 $\text{distinct}(\text{map } \text{fst } xys) \implies (\text{Some } y = \text{map-of } xys\ x) = ((x, y) \in \text{set } xys)$
 $\langle proof \rangle$

lemma *map-of-is-SomeI* [simp]: $\llbracket \text{distinct}(\text{map } \text{fst } xys); (x, y) \in \text{set } xys \rrbracket$
 $\implies \text{map-of } xys\ x = \text{Some } y$
 $\langle proof \rangle$

lemma *map-of-zip-is-None* [simp]:
 $\text{length } xs = \text{length } ys \implies (\text{map-of } (\text{zip } xs\ ys)\ x = \text{None}) = (x \notin \text{set } xs)$
 $\langle proof \rangle$

lemma *map-of-zip-is-Some*:

assumes $\text{length } xs = \text{length } ys$

shows $x \in \text{set } xs \iff (\exists y. \text{map-of } (\text{zip } xs \ ys) \ x = \text{Some } y)$

<proof>

lemma *map-of-zip-upd*:

fixes $x :: 'a$ **and** $xs :: 'a \text{ list}$ **and** $ys \ zs :: 'b \text{ list}$

assumes $\text{length } ys = \text{length } xs$

and $\text{length } zs = \text{length } xs$

and $x \notin \text{set } xs$

and $\text{map-of } (\text{zip } xs \ ys)(x \mapsto y) = \text{map-of } (\text{zip } xs \ zs)(x \mapsto z)$

shows $\text{map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$

<proof>

lemma *map-of-zip-inject*:

assumes $\text{length } ys = \text{length } xs$

and $\text{length } zs = \text{length } xs$

and $\text{dist: distinct } xs$

and $\text{map-of: map-of } (\text{zip } xs \ ys) = \text{map-of } (\text{zip } xs \ zs)$

shows $ys = zs$

<proof>

lemma *map-of-zip-map*:

$\text{map-of } (\text{zip } xs \ (\text{map } f \ xs)) = (\lambda x. \text{if } x \in \text{set } xs \text{ then } \text{Some } (f \ x) \text{ else } \text{None})$

<proof>

lemma *finite-range-map-of*: $\text{finite } (\text{range } (\text{map-of } \ xys))$

<proof>

lemma *map-of-SomeD*: $\text{map-of } xs \ k = \text{Some } y \implies (k, y) \in \text{set } xs$

<proof>

lemma *map-of-mapk-SomeI*:

$\text{inj } f \implies \text{map-of } t \ k = \text{Some } x \implies$

$\text{map-of } (\text{map } (\text{split } (\%k. \text{Pair } (f \ k))) \ t) \ (f \ k) = \text{Some } x$

<proof>

lemma *weak-map-of-SomeI*: $(k, x) : \text{set } l \implies \exists x. \text{map-of } l \ k = \text{Some } x$

<proof>

lemma *map-of-filter-in*:

$\text{map-of } xs \ k = \text{Some } z \implies P \ k \ z \implies \text{map-of } (\text{filter } (\text{split } P) \ xs) \ k = \text{Some } z$

<proof>

lemma *map-of-map*:

$\text{map-of } (\text{map } (\lambda(k, v). (k, f \ v)) \ xs) = \text{Option.map } f \circ \text{map-of } xs$

<proof>

lemma *dom-option-map*:

$dom (\lambda k. Option.map (f k) (m k)) = dom m$
 $\langle proof \rangle$

53.4 *Option.map* related

lemma *option-map-o-empty* [simp]: $Option.map f o empty = empty$
 $\langle proof \rangle$

lemma *option-map-o-map-upd* [simp]:
 $Option.map f o m(a|->b) = (Option.map f o m)(a|->f b)$
 $\langle proof \rangle$

53.5 *map-comp* related

lemma *map-comp-empty* [simp]:
 $m \circ_m empty = empty$
 $empty \circ_m m = empty$
 $\langle proof \rangle$

lemma *map-comp-simps* [simp]:
 $m2 k = None \implies (m1 \circ_m m2) k = None$
 $m2 k = Some k' \implies (m1 \circ_m m2) k = m1 k'$
 $\langle proof \rangle$

lemma *map-comp-Some-iff*:
 $((m1 \circ_m m2) k = Some v) = (\exists k'. m2 k = Some k' \wedge m1 k' = Some v)$
 $\langle proof \rangle$

lemma *map-comp-None-iff*:
 $((m1 \circ_m m2) k = None) = (m2 k = None \vee (\exists k'. m2 k = Some k' \wedge m1 k' = None))$
 $\langle proof \rangle$

53.6 ++

lemma *map-add-empty*[simp]: $m ++ empty = m$
 $\langle proof \rangle$

lemma *empty-map-add*[simp]: $empty ++ m = m$
 $\langle proof \rangle$

lemma *map-add-assoc*[simp]: $m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3$
 $\langle proof \rangle$

lemma *map-add-Some-iff*:
 $((m ++ n) k = Some x) = (n k = Some x \mid n k = None \ \& \ m k = Some x)$
 $\langle proof \rangle$

lemma *map-add-SomeD* [dest!]:
 $(m ++ n) k = Some x \implies n k = Some x \vee n k = None \wedge m k = Some x$

$\langle proof \rangle$

lemma *map-add-find-right* [simp]: $!!xx. n\ k = \text{Some } xx \implies (m\ ++\ n)\ k = \text{Some } xx$
 $\langle proof \rangle$

lemma *map-add-None* [iff]: $((m\ ++\ n)\ k = \text{None}) = (n\ k = \text{None} \ \&\ m\ k = \text{None})$
 $\langle proof \rangle$

lemma *map-add-upd*[simp]: $f\ ++\ g(x|->y) = (f\ ++\ g)(x|->y)$
 $\langle proof \rangle$

lemma *map-add-upds*[simp]: $m1\ ++\ (m2(xs[\mapsto]ys)) = (m1\ ++\ m2)(xs[\mapsto]ys)$
 $\langle proof \rangle$

lemma *map-add-upd-left*: $m \notin \text{dom } e2 \implies e1(m \mapsto u1) \ ++\ e2 = (e1 \ ++\ e2)(m \mapsto u1)$
 $\langle proof \rangle$

lemma *map-of-append*[simp]: $\text{map-of } (xs\ @\ ys) = \text{map-of } ys\ ++\ \text{map-of } xs$
 $\langle proof \rangle$

lemma *finite-range-map-of-map-add*:
 $\text{finite } (\text{range } f) \implies \text{finite } (\text{range } (f\ ++\ \text{map-of } l))$
 $\langle proof \rangle$

lemma *inj-on-map-add-dom* [iff]:
 $\text{inj-on } (m\ ++\ m')\ (\text{dom } m') = \text{inj-on } m'\ (\text{dom } m')$
 $\langle proof \rangle$

lemma *map-upds-fold-map-upd*:
 $m(ks[\mapsto]vs) = \text{foldl } (\lambda m\ (k, v). m(k \mapsto v))\ m\ (\text{zip } ks\ vs)$
 $\langle proof \rangle$

lemma *map-add-map-of-foldr*:
 $m\ ++\ \text{map-of } ps = \text{foldr } (\lambda(k, v)\ m. m(k \mapsto v))\ ps\ m$
 $\langle proof \rangle$

53.7 restrict-map

lemma *restrict-map-to-empty* [simp]: $m|'\{\} = \text{empty}$
 $\langle proof \rangle$

lemma *restrict-map-insert*: $f\ |'\ (\text{insert } a\ A) = (f\ |'\ A)(a := f\ a)$
 $\langle proof \rangle$

lemma *restrict-map-empty* [simp]: $\text{empty}|'D = \text{empty}$
 $\langle proof \rangle$

lemma *restrict-in* [simp]: $x \in A \implies (m|'A) x = m x$
 ⟨proof⟩

lemma *restrict-out* [simp]: $x \notin A \implies (m|'A) x = \text{None}$
 ⟨proof⟩

lemma *ran-restrictD*: $y \in \text{ran } (m|'A) \implies \exists x \in A. m x = \text{Some } y$
 ⟨proof⟩

lemma *dom-restrict* [simp]: $\text{dom } (m|'A) = \text{dom } m \cap A$
 ⟨proof⟩

lemma *restrict-upd-same* [simp]: $m(x \mapsto y)|'(-\{x\}) = m|'(-\{x\})$
 ⟨proof⟩

lemma *restrict-restrict* [simp]: $m|'A|'B = m|'(A \cap B)$
 ⟨proof⟩

lemma *restrict-fun-upd* [simp]:
 $m(x := y)|'D = (\text{if } x \in D \text{ then } (m|'(D - \{x\}))(x := y) \text{ else } m|'D)$
 ⟨proof⟩

lemma *fun-upd-None-restrict* [simp]:
 $(m|'D)(x := \text{None}) = (\text{if } x:D \text{ then } m|'(D - \{x\}) \text{ else } m|'D)$
 ⟨proof⟩

lemma *fun-upd-restrict*: $(m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$
 ⟨proof⟩

lemma *fun-upd-restrict-conv* [simp]:
 $x \in D \implies (m|'D)(x := y) = (m|'(D - \{x\}))(x := y)$
 ⟨proof⟩

lemma *map-of-map-restrict*:
 $\text{map-of } (\text{map } (\lambda k. (k, f k)) ks) = (\text{Some } \circ f) |' \text{set } ks$
 ⟨proof⟩

lemma *restrict-complement-singleton-eq*:
 $f |' (-\{x\}) = f(x := \text{None})$
 ⟨proof⟩

53.8 map-upds

lemma *map-upds-Nil1* [simp]: $m(\square \ [|->] bs) = m$
 ⟨proof⟩

lemma *map-upds-Nil2* [simp]: $m(\text{as } [|->] \square) = m$
 ⟨proof⟩

lemma *map-upds-Cons* [simp]: $m(a \# as \llbracket - \> \rrbracket b \# bs) = (m(a \llbracket - \> b \rrbracket))(as \llbracket - \> \rrbracket bs)$
 <proof>

lemma *map-upds-append1* [simp]: $\bigwedge ys\ m.\ size\ xs < size\ ys \implies$
 $m(xs @ [x] \llbracket \mapsto \rrbracket ys) = m(xs \llbracket \mapsto \rrbracket ys)(x \mapsto ys!size\ xs)$
 <proof>

lemma *map-upds-list-update2-drop* [simp]:
 $\llbracket size\ xs \leq i; i < size\ ys \rrbracket$
 $\implies m(xs \llbracket \mapsto \rrbracket ys[i := y]) = m(xs \llbracket \mapsto \rrbracket ys)$
 <proof>

lemma *map-upd-upds-conv-if*:
 $(f(x \llbracket - \> y \rrbracket))(xs \llbracket - \> \rrbracket ys) =$
 $(if\ x : set\ (take\ (length\ ys)\ xs)\ then\ f(xs \llbracket - \> \rrbracket ys)$
 $\quad\quad\quad else\ (f(xs \llbracket - \> \rrbracket ys))(x \llbracket - \> y \rrbracket))$
 <proof>

lemma *map-upds-twist* [simp]:
 $a \sim : set\ as \implies m(a \llbracket - \> b \rrbracket)(as \llbracket - \> \rrbracket bs) = m(as \llbracket - \> \rrbracket bs)(a \llbracket - \> b \rrbracket)$
 <proof>

lemma *map-upds-apply-nontin* [simp]:
 $x \sim : set\ xs \implies (f(xs \llbracket - \> \rrbracket ys))\ x = f\ x$
 <proof>

lemma *fun-upds-append-drop* [simp]:
 $size\ xs = size\ ys \implies m(xs @ zs \llbracket \mapsto \rrbracket ys) = m(xs \llbracket \mapsto \rrbracket ys)$
 <proof>

lemma *fun-upds-append2-drop* [simp]:
 $size\ xs = size\ ys \implies m(xs \llbracket \mapsto \rrbracket ys @ zs) = m(xs \llbracket \mapsto \rrbracket ys)$
 <proof>

lemma *restrict-map-upds* [simp]:
 $\llbracket length\ xs = length\ ys; set\ xs \subseteq D \rrbracket$
 $\implies m(xs \llbracket \mapsto \rrbracket ys) \llbracket D = (m \llbracket (D - set\ xs) \rrbracket)(xs \llbracket \mapsto \rrbracket ys)$
 <proof>

53.9 dom

lemma *dom-eq-empty-conv* [simp]: $dom\ f = \{\} \longleftrightarrow f = empty$
 <proof>

lemma *domI*: $m\ a = Some\ b \implies a : dom\ m$
 <proof>

lemma *domD*: $a : \text{dom } m \implies \exists b. m \ a = \text{Some } b$
 $\langle \text{proof} \rangle$

lemma *domIff* [*iff*, *simp del*]: $(a : \text{dom } m) = (m \ a \sim= \text{None})$
 $\langle \text{proof} \rangle$

lemma *dom-empty* [*simp*]: $\text{dom empty} = \{\}$
 $\langle \text{proof} \rangle$

lemma *dom-fun-upd* [*simp*]:
 $\text{dom}(f(x := y)) = (\text{if } y = \text{None} \text{ then } \text{dom } f - \{x\} \text{ else } \text{insert } x (\text{dom } f))$
 $\langle \text{proof} \rangle$

lemma *dom-if*:
 $\text{dom } (\lambda x. \text{if } P \ x \text{ then } f \ x \text{ else } g \ x) = \text{dom } f \cap \{x. P \ x\} \cup \text{dom } g \cap \{x. \neg P \ x\}$
 $\langle \text{proof} \rangle$

lemma *dom-map-of-conv-image-fst*:
 $\text{dom } (\text{map-of } xys) = \text{fst } \text{'set } xys$
 $\langle \text{proof} \rangle$

lemma *dom-map-of-zip* [*simp*]: $[\mid \text{length } xs = \text{length } ys; \text{distinct } xs \mid] \implies$
 $\text{dom}(\text{map-of}(\text{zip } xs \ ys)) = \text{set } xs$
 $\langle \text{proof} \rangle$

lemma *finite-dom-map-of*: $\text{finite } (\text{dom } (\text{map-of } l))$
 $\langle \text{proof} \rangle$

lemma *dom-map-upds* [*simp*]:
 $\text{dom}(m(xs[->]ys)) = \text{set}(\text{take } (\text{length } ys) \ xs) \cup \text{dom } m$
 $\langle \text{proof} \rangle$

lemma *dom-map-add* [*simp*]: $\text{dom}(m++n) = \text{dom } n \cup \text{dom } m$
 $\langle \text{proof} \rangle$

lemma *dom-override-on* [*simp*]:
 $\text{dom}(\text{override-on } f \ g \ A) =$
 $(\text{dom } f - \{a. a : A - \text{dom } g\}) \cup \{a. a : A \mid \text{Int dom } g\}$
 $\langle \text{proof} \rangle$

lemma *map-add-comm*: $\text{dom } m1 \cap \text{dom } m2 = \{\} \implies m1++m2 = m2++m1$
 $\langle \text{proof} \rangle$

lemma *map-add-dom-app-simps*:
 $[\mid m \in \text{dom } l2 \mid] \implies (l1++l2) \ m = l2 \ m$
 $[\mid m \notin \text{dom } l1 \mid] \implies (l1++l2) \ m = l2 \ m$
 $[\mid m \notin \text{dom } l2 \mid] \implies (l1++l2) \ m = l1 \ m$
 $\langle \text{proof} \rangle$

lemma *dom-const* [simp]:
 $\text{dom } (\lambda x. \text{Some } (f x)) = \text{UNIV}$
 ⟨proof⟩

lemma *finite-map-freshness*:
 $\text{finite } (\text{dom } (f :: 'a \rightarrow 'b)) \implies \neg \text{finite } (\text{UNIV} :: 'a \text{ set}) \implies$
 $\exists x. f x = \text{None}$
 ⟨proof⟩

lemma *dom-minus*:
 $f x = \text{None} \implies \text{dom } f - \text{insert } x A = \text{dom } f - A$
 ⟨proof⟩

lemma *insert-dom*:
 $f x = \text{Some } y \implies \text{insert } x (\text{dom } f) = \text{dom } f$
 ⟨proof⟩

lemma *map-of-map-keys*:
 $\text{set } xs = \text{dom } m \implies \text{map-of } (\text{map } (\lambda k. (k, \text{the } (m k)))) xs = m$
 ⟨proof⟩

53.10 ran

lemma *ranI*: $m a = \text{Some } b \implies b : \text{ran } m$
 ⟨proof⟩

lemma *ran-empty* [simp]: $\text{ran empty} = \{\}$
 ⟨proof⟩

lemma *ran-map-upd* [simp]: $m a = \text{None} \implies \text{ran}(m(a|->b)) = \text{insert } b (\text{ran } m)$
 ⟨proof⟩

lemma *ran-distinct*:
 assumes *dist*: $\text{distinct } (\text{map fst } al)$
 shows $\text{ran } (\text{map-of } al) = \text{snd } ` \text{set } al$
 ⟨proof⟩

53.11 map-le

lemma *map-le-empty* [simp]: $\text{empty} \subseteq_m g$
 ⟨proof⟩

lemma *upd-None-map-le* [simp]: $f(x := \text{None}) \subseteq_m f$
 ⟨proof⟩

lemma *map-le-upd*[simp]: $f \subseteq_m g \implies f(a := b) \subseteq_m g(a := b)$

$\langle proof \rangle$

lemma *map-le-imp-upd-le* [*simp*]: $m1 \subseteq_m m2 \implies m1(x := None) \subseteq_m m2(x \mapsto y)$
 $\langle proof \rangle$

lemma *map-le-upds* [*simp*]:
 $f \subseteq_m g \implies f(as \ [|->] \ bs) \subseteq_m g(as \ [|->] \ bs)$
 $\langle proof \rangle$

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \implies (dom\ f \subseteq dom\ g)$
 $\langle proof \rangle$

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
 $\langle proof \rangle$

lemma *map-le-trans*[*trans*]: $\llbracket m1 \subseteq_m m2; m2 \subseteq_m m3 \rrbracket \implies m1 \subseteq_m m3$
 $\langle proof \rangle$

lemma *map-le-antisym*: $\llbracket f \subseteq_m g; g \subseteq_m f \rrbracket \implies f = g$
 $\langle proof \rangle$

lemma *map-le-map-add* [*simp*]: $f \subseteq_m (g ++ f)$
 $\langle proof \rangle$

lemma *map-le-iff-map-add-commute*: $(f \subseteq_m f ++ g) = (f ++ g = g ++ f)$
 $\langle proof \rangle$

lemma *map-add-le-mapE*: $f ++ g \subseteq_m h \implies g \subseteq_m h$
 $\langle proof \rangle$

lemma *map-add-le-mapI*: $\llbracket f \subseteq_m h; g \subseteq_m h; f \subseteq_m f ++ g \rrbracket \implies f ++ g \subseteq_m h$
 $\langle proof \rangle$

lemma *dom-eq-singleton-conv*: $dom\ f = \{x\} \longleftrightarrow (\exists v. f = [x \mapsto v])$
 $\langle proof \rangle$

53.12 Various

lemma *set-map-of-compr*:
assumes *distinct*: *distinct* (*map fst xs*)
shows *set xs* = $\{(k, v). \text{map-of } xs\ k = \text{Some } v\}$
 $\langle proof \rangle$

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*)
 $\langle proof \rangle$

end

54 Quotient: Definition of Quotient Types

theory *Quotient*

imports *Plain Sledgehammer*

uses

(\sim /src/HOL/Tools/Quotient/quotient-info.ML)

(\sim /src/HOL/Tools/Quotient/quotient-typ.ML)

(\sim /src/HOL/Tools/Quotient/quotient-def.ML)

(\sim /src/HOL/Tools/Quotient/quotient-term.ML)

(\sim /src/HOL/Tools/Quotient/quotient-tacs.ML)

begin

Basic definition for equivalence relations that are represented by predicates.

definition

$\text{equivp } E \equiv \forall x y. E x y = (E x = E y)$

definition

$\text{reflp } E \equiv \forall x. E x x$

definition

$\text{symp } E \equiv \forall x y. E x y \longrightarrow E y x$

definition

$\text{transp } E \equiv \forall x y z. E x y \wedge E y z \longrightarrow E x z$

lemma *equivp-reflp-symp-transp*:

shows $\text{equivp } E = (\text{reflp } E \wedge \text{symp } E \wedge \text{transp } E)$

$\langle \text{proof} \rangle$

lemma *equivp-reflp*:

shows $\text{equivp } E \Longrightarrow E x x$

$\langle \text{proof} \rangle$

lemma *equivp-symp*:

shows $\text{equivp } E \Longrightarrow E x y \Longrightarrow E y x$

$\langle \text{proof} \rangle$

lemma *equivp-transp*:

shows $\text{equivp } E \Longrightarrow E x y \Longrightarrow E y z \Longrightarrow E x z$

$\langle \text{proof} \rangle$

lemma *equivpI*:

assumes $\text{reflp } R \text{ symp } R \text{ transp } R$

shows $\text{equivp } R$

$\langle \text{proof} \rangle$

lemma *identity-equivp*:
shows *equivp* (*op* =)
 ⟨*proof*⟩

Partial equivalences: not yet used anywhere

definition

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*
 ⟨*proof*⟩

Composition of Relations

abbreviation

rel-conj (**infixr** *OOO* 75)

where

$r1\ OOO\ r2 \equiv r1\ OO\ r2\ OO\ r1$

lemma *eq-comp-r*:
shows $((op =)\ OOO\ R) = R$
 ⟨*proof*⟩

54.1 Respects predicate

definition

Respects

where

$Respects\ R\ x \equiv R\ x\ x$

lemma *in-respects*:
shows $(x \in Respects\ R) = R\ x\ x$
 ⟨*proof*⟩

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 $(P\ ===>\ Q)\ x\ y$

$\langle \text{proof} \rangle$

lemma *fun-map-id*:

shows $(\text{id} \dashrightarrow \text{id}) = \text{id}$

$\langle \text{proof} \rangle$

lemma *fun-rel-eq*:

shows $((\text{op} =) \implies (\text{op} =)) = (\text{op} =)$

$\langle \text{proof} \rangle$

lemma *fun-rel-id*:

assumes $a: \bigwedge x y. R1\ x\ y \implies R2\ (f\ x)\ (g\ y)$

shows $(R1 \implies R2)\ f\ g$

$\langle \text{proof} \rangle$

lemma *fun-rel-id-asm*:

assumes $a: \bigwedge x y. R1\ x\ y \implies (A \longrightarrow R2\ (f\ x)\ (g\ y))$

shows $A \longrightarrow (R1 \implies R2)\ f\ g$

$\langle \text{proof} \rangle$

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 $a: \text{Quotient}\ E\ Abs\ Rep$

shows $Abs\ (Rep\ a) = a$

$\langle \text{proof} \rangle$

lemma *Quotient-rep-refl*:

assumes $a: \text{Quotient}\ E\ Abs\ Rep$

shows $E\ (Rep\ a)\ (Rep\ a)$

$\langle \text{proof} \rangle$

lemma *Quotient-rel*:

assumes $a: \text{Quotient}\ E\ Abs\ Rep$

shows $E\ r\ s = (E\ r\ r \wedge E\ s\ s \wedge (Abs\ r = Abs\ s))$

$\langle \text{proof} \rangle$

lemma *Quotient-rel-rep*:

assumes $a: \text{Quotient}\ R\ Abs\ Rep$

shows $R\ (Rep\ a)\ (Rep\ b) = (a = b)$

$\langle \text{proof} \rangle$

lemma *Quotient-rep-abs*:

assumes $a: \text{Quotient}\ R\ Abs\ Rep$

shows $R\ r\ r \implies R\ (Rep\ (Abs\ r))\ r$
 $\langle proof \rangle$

lemma *Quotient-rel-abs*:
assumes a : *Quotient* $E\ Abs\ Rep$
shows $E\ r\ s \implies Abs\ r = Abs\ s$
 $\langle proof \rangle$

lemma *Quotient-symp*:
assumes a : *Quotient* $E\ Abs\ Rep$
shows *symp* E
 $\langle proof \rangle$

lemma *Quotient-transp*:
assumes a : *Quotient* $E\ Abs\ Rep$
shows *transp* E
 $\langle proof \rangle$

lemma *identity-quotient*:
shows *Quotient* $(op =) id\ id$
 $\langle proof \rangle$

lemma *fun-quotient*:
assumes $q1$: *Quotient* $R1\ abs1\ rep1$
and $q2$: *Quotient* $R2\ abs2\ rep2$
shows *Quotient* $(R1 ==> R2)\ (rep1 ----> abs2)\ (abs1 ----> rep2)$
 $\langle proof \rangle$

lemma *abs-o-rep*:
assumes a : *Quotient* $R\ Abs\ Rep$
shows $Abs\ o\ Rep = id$
 $\langle proof \rangle$

lemma *equals-rsp*:
assumes q : *Quotient* $R\ Abs\ Rep$
and a : $R\ xa\ xb\ R\ ya\ yb$
shows $R\ xa\ ya = R\ xb\ yb$
 $\langle proof \rangle$

lemma *lambda-prs*:
assumes $q1$: *Quotient* $R1\ Abs1\ Rep1$
and $q2$: *Quotient* $R2\ Abs2\ Rep2$
shows $(Rep1 ----> Abs2)\ (\lambda x. Rep2\ (f\ (Abs1\ x))) = (\lambda x. f\ x)$
 $\langle proof \rangle$

lemma *lambda-prs1*:
assumes $q1$: *Quotient* $R1\ Abs1\ Rep1$
and $q2$: *Quotient* $R2\ Abs2\ Rep2$
shows $(Rep1 ----> Abs2)\ (\lambda x. (Abs1 ----> Rep2)\ f\ x) = (\lambda x. f\ x)$

$\langle proof \rangle$

lemma *rep-abs-rsp*:

assumes q : *Quotient* R Abs Rep

and a : $R\ x1\ x2$

shows $R\ x1\ (Rep\ (Abs\ x2))$

$\langle proof \rangle$

lemma *rep-abs-rsp-left*:

assumes q : *Quotient* R Abs Rep

and a : $R\ x1\ x2$

shows $R\ (Rep\ (Abs\ x1))\ x2$

$\langle proof \rangle$

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 $f\ g$: $'a \Rightarrow 'c$

assumes q : *Quotient* $R1$ $Abs1$ $Rep1$

and a : $(R1\ ==>\ R2)\ f\ g\ R1\ x\ y$

shows $R2\ (f\ x)\ (g\ y)$

$\langle proof \rangle$

lemma *apply-rsp'*:

assumes a : $(R1\ ==>\ R2)\ f\ g\ R1\ x\ y$

shows $R2\ (f\ x)\ (g\ y)$

$\langle proof \rangle$

54.4 lemmas for regularisation of ball and bex

lemma *ball-reg-equiv*:

fixes P :: $'a \Rightarrow bool$

assumes a : *equivp* R

shows $Ball\ (Respects\ R)\ P = (All\ P)$

$\langle proof \rangle$

lemma *bex-reg-equiv*:

fixes P :: $'a \Rightarrow bool$

assumes a : *equivp* R

shows $Bex\ (Respects\ R)\ P = (Ex\ P)$

$\langle proof \rangle$

lemma *ball-reg-right*:

assumes a : $\bigwedge x. R\ x \implies P\ x \longrightarrow Q\ x$

shows $All\ P \longrightarrow Ball\ R\ Q$

$\langle proof \rangle$

lemma *bex-reg-left*:

assumes $a: \bigwedge x. R\ x \implies Q\ x \longrightarrow P\ x$

shows $Bex\ R\ Q \longrightarrow Ex\ P$

<proof>

lemma *ball-reg-left*:

assumes $a: equivp\ R$

shows $(\bigwedge x. (Q\ x \longrightarrow P\ x)) \implies Ball\ (Respects\ R)\ Q \longrightarrow All\ P$

<proof>

lemma *bex-reg-right*:

assumes $a: equivp\ R$

shows $(\bigwedge x. (Q\ x \longrightarrow P\ x)) \implies Ex\ Q \longrightarrow Bex\ (Respects\ R)\ P$

<proof>

lemma *ball-reg-equiv-range*:

fixes $P::'a \Rightarrow bool$

and $x::'a$

assumes $a: equivp\ R2$

shows $(Ball\ (Respects\ (R1 \implies R2))\ (\lambda f. P\ (f\ x))) = All\ (\lambda f. P\ (f\ x))$

<proof>

lemma *bex-reg-equiv-range*:

assumes $a: equivp\ R2$

shows $(Bex\ (Respects\ (R1 \implies R2))\ (\lambda f. P\ (f\ x))) = Ex\ (\lambda f. P\ (f\ x))$

<proof>

lemma *all-reg*:

assumes $a: !x :: 'a. (P\ x \longrightarrow Q\ x)$

and $b: All\ P$

shows $All\ Q$

<proof>

lemma *ex-reg*:

assumes $a: !x :: 'a. (P\ x \longrightarrow Q\ x)$

and $b: Ex\ P$

shows $Ex\ Q$

<proof>

lemma *ball-reg*:

assumes $a: !x :: 'a. (R\ x \longrightarrow P\ x \longrightarrow Q\ x)$

and $b: Ball\ R\ P$

shows $Ball\ R\ Q$

<proof>

lemma *bex-reg*:

assumes $a: !x :: 'a. (R\ x \longrightarrow P\ x \longrightarrow Q\ x)$

and $b: Bex\ R\ P$

shows $Bex\ R\ Q$
 $\langle proof \rangle$

lemma *ball-all-comm*:
assumes $\bigwedge y. (\forall x \in P. A\ x\ y) \longrightarrow (\forall x. B\ x\ y)$
shows $(\forall x \in P. \forall y. A\ x\ y) \longrightarrow (\forall x. \forall y. B\ x\ y)$
 $\langle proof \rangle$

lemma *bex-ex-comm*:
assumes $(\exists y. \exists x. A\ x\ y) \longrightarrow (\exists y. \exists x \in P. B\ x\ y)$
shows $(\exists x. \exists y. A\ x\ y) \longrightarrow (\exists x \in P. \exists y. B\ x\ y)$
 $\langle proof \rangle$

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 $a: (R1\ ==>\ R2)\ f\ g$
shows $(R1\ ==>\ R2)\ (Babs\ (Respects\ R1)\ f)\ (Babs\ (Respects\ R1)\ g)$
 $\langle proof \rangle$

lemma *babs-prs*:
assumes $q1: Quotient\ R1\ Abs1\ Rep1$
and $q2: Quotient\ R2\ Abs2\ Rep2$
shows $((Rep1\ --->\ Abs2)\ (Babs\ (Respects\ R1)\ ((Abs1\ --->\ Rep2)\ f))) =$
 f
 $\langle proof \rangle$

lemma *babs-simp*:
assumes $q: Quotient\ R1\ Abs\ Rep$
shows $((R1\ ==>\ R2)\ (Babs\ (Respects\ R1)\ f)\ (Babs\ (Respects\ R1)\ g)) =$
 $((R1\ ==>\ R2)\ f\ g)$
 $\langle proof \rangle$

lemma *babs-reg-equiv*:
shows $equivp\ R \Longrightarrow Babs\ (Respects\ R)\ P = P$
 $\langle proof \rangle$

lemma *ball-rsp*:
assumes $a: (R\ ==>\ (op\ =))\ f\ g$

shows $Ball\ (Respects\ R)\ f = Ball\ (Respects\ R)\ g$
 $\langle proof \rangle$

lemma *bex-rsp*:

assumes $a: (R ==> (op =))\ f\ g$
shows $(Bex\ (Respects\ R)\ f = Bex\ (Respects\ R)\ g)$
 $\langle proof \rangle$

lemma *bex1-rsp*:

assumes $a: (R ==> (op =))\ f\ g$
shows $Ex1\ (\lambda x. x \in Respects\ R \wedge f\ x) = Ex1\ (\lambda x. x \in Respects\ R \wedge g\ x)$
 $\langle proof \rangle$

lemma *all-prs*:

assumes $a: Quotient\ R\ absf\ repf$
shows $Ball\ (Respects\ R)\ ((absf\ ----> id)\ f) = All\ f$
 $\langle proof \rangle$

lemma *ex-prs*:

assumes $a: Quotient\ R\ absf\ repf$
shows $Bex\ (Respects\ R)\ ((absf\ ----> id)\ f) = Ex\ f$
 $\langle proof \rangle$

54.6 Bex1-rel quantifier

definition

$Bex1\text{-}rel :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$

where

$Bex1\text{-}rel\ R\ P \longleftrightarrow (\exists x \in Respects\ R. P\ x) \wedge (\forall x \in Respects\ R. \forall y \in Respects\ R. ((P\ x \wedge P\ y) \longrightarrow (R\ x\ y)))$

lemma *bex1-rel-aux*:

$\llbracket \forall xa\ ya. R\ xa\ ya \longrightarrow x\ xa = y\ ya; Bex1\text{-}rel\ R\ x \rrbracket \Longrightarrow Bex1\text{-}rel\ R\ y$
 $\langle proof \rangle$

lemma *bex1-rel-aux2*:

$\llbracket \forall xa\ ya. R\ xa\ ya \longrightarrow x\ xa = y\ ya; Bex1\text{-}rel\ R\ y \rrbracket \Longrightarrow Bex1\text{-}rel\ R\ x$
 $\langle proof \rangle$

lemma *bex1-rel-rsp*:

assumes $a: Quotient\ R\ absf\ repf$
shows $((R ==> op =) ==> op =) (Bex1\text{-}rel\ R)\ (Bex1\text{-}rel\ R)$
 $\langle proof \rangle$

lemma *ex1-prs*:

assumes $a: Quotient\ R\ absf\ repf$
shows $((absf\ ----> id)\ ----> id)\ (Bex1\text{-}rel\ R)\ f = Ex1\ f$

$\langle proof \rangle$

lemma *bex1-bexeq-reg*: $(\exists !x \in Respects\ R. P\ x) \longrightarrow (Bex1-rel\ R\ (\lambda x. P\ x))$
 $\langle proof \rangle$

54.7 Various respects and preserve lemmas

lemma *quot-rel-rsp*:
assumes *a*: *Quotient* *R* *Abs* *Rep*
shows $(R\ ==\!\!=\!\!=\!\!>\ R\ ==\!\!=\!\!=\!\!>\ op\ =)\ R\ R$
 $\langle proof \rangle$

lemma *o-prs*:
assumes *q1*: *Quotient* *R1* *Abs1* *Rep1*
and *q2*: *Quotient* *R2* *Abs2* *Rep2*
and *q3*: *Quotient* *R3* *Abs3* *Rep3*
shows $((Abs2\ \text{----}>\ Rep3)\ \text{----}>\ (Abs1\ \text{----}>\ Rep2)\ \text{----}>\ (Rep1\ \text{----}>\ Abs3))\ op\ \circ\ =\ op\ \circ$
and $(id\ \text{----}>\ (Abs1\ \text{----}>\ id)\ \text{----}>\ Rep1\ \text{----}>\ id)\ op\ \circ\ =\ op\ \circ$
 $\langle proof \rangle$

lemma *o-rsp*:
 $((R2\ ==\!\!=\!\!=\!\!>\ R3)\ ==\!\!=\!\!=\!\!>\ (R1\ ==\!\!=\!\!=\!\!>\ R2)\ ==\!\!=\!\!=\!\!>\ (R1\ ==\!\!=\!\!=\!\!>\ R3))\ op\ \circ\ op\ \circ$
 $(op\ =\ ==\!\!=\!\!=\!\!>\ (R1\ ==\!\!=\!\!=\!\!>\ op\ =)\ ==\!\!=\!\!=\!\!>\ R1\ ==\!\!=\!\!=\!\!>\ op\ =)\ op\ \circ\ op\ \circ$
 $\langle proof \rangle$

lemma *cond-prs*:
assumes *a*: *Quotient* *R* *absf* *repf*
shows *absf* $(if\ a\ then\ repf\ b\ else\ repf\ c) = (if\ a\ then\ b\ else\ c)$
 $\langle proof \rangle$

lemma *if-prs*:
assumes *q*: *Quotient* *R* *Abs* *Rep*
shows $(id\ \text{----}>\ Rep\ \text{----}>\ Rep\ \text{----}>\ Abs)\ If\ =\ If$
 $\langle proof \rangle$

lemma *if-rsp*:
assumes *q*: *Quotient* *R* *Abs* *Rep*
shows $(op\ =\ ==\!\!=\!\!=\!\!>\ R\ ==\!\!=\!\!=\!\!>\ R\ ==\!\!=\!\!=\!\!>\ R)\ If\ If$
 $\langle proof \rangle$

lemma *let-prs*:
assumes *q1*: *Quotient* *R1* *Abs1* *Rep1*
and *q2*: *Quotient* *R2* *Abs2* *Rep2*
shows $(Rep2\ \text{----}>\ (Abs2\ \text{----}>\ Rep1)\ \text{----}>\ Abs1)\ Let\ =\ Let$
 $\langle proof \rangle$

lemma *let-rsp*:
shows $(R1\ ==\!\!=\!\!=\!\!>\ (R1\ ==\!\!=\!\!=\!\!>\ R2)\ ==\!\!=\!\!=\!\!>\ R2)\ Let\ Let$

$\langle proof \rangle$

locale *quot-type* =
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$
and $Abs :: ('a \Rightarrow bool) \Rightarrow 'b$
and $Rep :: 'b \Rightarrow ('a \Rightarrow bool)$
assumes *equivp*: $equivp\ R$
and *rep-prop*: $\bigwedge y. \exists x. Rep\ y = R\ x$
and *rep-inverse*: $\bigwedge x. Abs\ (Rep\ x) = x$
and *abs-inverse*: $\bigwedge x. (Rep\ (Abs\ (R\ x))) = (R\ x)$
and *rep-inject*: $\bigwedge x\ y. (Rep\ x = Rep\ y) = (x = y)$
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 $R\ (Eps\ (R\ x)) = R\ x$
 $\langle proof \rangle$

theorem *homeier-thm10*:
shows $abs\ (rep\ a) = a$
 $\langle proof \rangle$

lemma *homeier-lem7*:
shows $(R\ x = R\ y) = (Abs\ (R\ x) = Abs\ (R\ y))$ (**is** *?LHS = ?RHS*)
 $\langle proof \rangle$

theorem *homeier-thm11*:
shows $R\ r\ r' = (abs\ r = abs\ r')$
 $\langle proof \rangle$

lemma *rep-refl*:
shows $R\ (rep\ a)\ (rep\ a)$
 $\langle proof \rangle$

lemma *rep-abs-rsp*:
shows $R\ f\ (rep\ (abs\ g)) = R\ f\ g$
and $R\ (rep\ (abs\ g))\ f = R\ g\ f$
 $\langle proof \rangle$

```

lemma Quotient:
  shows Quotient R abs rep
  <proof>

```

```

end

```

54.8 ML setup

Auxiliary data for the quotient package

<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.

<ML>

Definitions of the quotient types.

<ML>

Definitions for quotient constants.

<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)
<proof>

```

lemma *QT-imp*: *Quot-True a* \equiv *Quot-True b*
<proof>

Tactics for proving the lifted theorems

<ML>

54.9 Methods / Interface

<ML>

no-notation

rel-conj (**infixr** *OOO 75*) **and**
fun-map (**infixr** *---> 55*) **and**
fun-rel (**infixr** *===> 55*)

end

55 Refute: Refute

theory *Refute*

imports *Hilbert-Choice List*

uses

Tools/refute.ML
Tools/refute-isar.ML

begin

<ML>

```
(* ----- *)
(* 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]

```

$\langle ML \rangle$

```

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)
  <proof>

```

lemma *Ex1-def* [*nitpick-def*, *no-atp*]:
 $Ex1\ P \equiv \exists x. P = \{x\}$
 $\langle proof \rangle$

lemma *split-def* [*nitpick-def*]: $split\ f = (\lambda p. f\ (fst\ p)\ (snd\ p))$
 $\langle proof \rangle$

lemma *rtrancl-def* [*nitpick-def*, *no-atp*]: $r^* \equiv (r^+)^=$
 $\langle proof \rangle$

lemma *rtranclp-def* [*nitpick-def*, *no-atp*]:
 $rtranclp\ r\ a\ b \equiv (a = b \vee tranclp\ r\ a\ b)$
 $\langle proof \rangle$

lemma *tranclp-def* [*nitpick-def*, *no-atp*]:
 $tranclp\ r\ a\ b \equiv trancl\ (split\ r)\ (a, b)$
 $\langle proof \rangle$

definition *refl'* :: $('a \times 'a \Rightarrow bool) \Rightarrow bool$ **where**
 $refl'\ r \equiv \forall x. (x, x) \in r$

definition *wf'* :: $('a \times 'a \Rightarrow bool) \Rightarrow bool$ **where**
 $wf'\ r \equiv acyclic\ r \wedge (finite\ r \vee unknown)$

axiomatization *wf-wfrec* :: $('a \times 'a \Rightarrow bool) \Rightarrow ((a \Rightarrow b) \Rightarrow a \Rightarrow b) \Rightarrow a \Rightarrow b$

definition *wf-wfrec'* :: $('a \times 'a \Rightarrow bool) \Rightarrow ((a \Rightarrow b) \Rightarrow a \Rightarrow b) \Rightarrow a \Rightarrow b$
where
 $[nitpick-simp]: wf-wfrec'\ R\ F\ x = F\ (Recdef.cut\ (wf-wfrec\ R\ F)\ R\ x)\ x$

definition *wfrec'* :: $('a \times 'a \Rightarrow bool) \Rightarrow ((a \Rightarrow b) \Rightarrow a \Rightarrow b) \Rightarrow a \Rightarrow b$ **where**
 $wfrec'\ R\ F\ x \equiv if\ wf\ R\ then\ wf-wfrec'\ R\ F\ x$
 $\quad\quad\quad else\ THE\ y. wfrec-rel\ R\ (\%f\ x. F\ (Recdef.cut\ f\ R\ x)\ x)\ x\ y$

definition *card'* :: $('a \Rightarrow bool) \Rightarrow nat$ **where**
 $card'\ A \equiv if\ finite\ A\ then\ length\ (safe-Eps\ (\lambda xs. set\ xs = A \wedge distinct\ xs))\ else\ 0$

definition *setsum'* :: $('a \Rightarrow 'b::comm-monoid-add) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b$ **where**
 $setsum'\ f\ A \equiv if\ finite\ A\ then\ listsum\ (map\ f\ (safe-Eps\ (\lambda xs. set\ xs = A \wedge distinct\ xs)))\ else\ 0$

inductive *fold-graph'* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow ('a \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$ **where**
 $fold-graph'\ f\ z\ \{\} \ z \mid$
 $\llbracket x \in A; fold-graph'\ f\ z\ (A - \{x\})\ y \rrbracket \Longrightarrow fold-graph'\ f\ z\ A\ (f\ x\ y)$

The following lemmas are not strictly necessary but they help the *special level* optimization.

lemma *The-psimp* [*nitpick-psimp*, *no-atp*]:

$P = \{x\} \implies \text{The } P = x$
 $\langle \text{proof} \rangle$

lemma *Eps-psimp* [*nitpick-psimp*, *no-atp*]:
 $\llbracket P\ x; \neg P\ y; \text{Eps } P = y \rrbracket \implies \text{Eps } P = x$
 $\langle \text{proof} \rangle$

lemma *unit-case-def* [*nitpick-def*, *no-atp*]:
 $\text{unit-case } x\ u \equiv x$
 $\langle \text{proof} \rangle$

declare *unit.cases* [*nitpick-simp del*]

lemma *nat-case-def* [*nitpick-def*, *no-atp*]:
 $\text{nat-case } x\ f\ n \equiv \text{if } n = 0 \text{ then } x \text{ else } f\ (n - 1)$
 $\langle \text{proof} \rangle$

declare *nat.cases* [*nitpick-simp del*]

lemma *list-size-simp* [*nitpick-simp*, *no-atp*]:
 $\text{list-size } f\ xs = (\text{if } xs = [] \text{ then } 0$
 $\quad \text{else } \text{Suc } (f\ (\text{hd } xs) + \text{list-size } f\ (\text{tl } xs)))$
 $\text{size } xs = (\text{if } xs = [] \text{ then } 0 \text{ else } \text{Suc } (\text{size } (\text{tl } xs)))$
 $\langle \text{proof} \rangle$

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

function *nat-gcd* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $[simp\ del]: \text{nat-gcd } x\ y = (\text{if } y = 0 \text{ then } x \text{ else } \text{nat-gcd } y\ (x \bmod y))$
 $\langle \text{proof} \rangle$
termination
 $\langle \text{proof} \rangle$

definition *nat-lcm* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **where**
 $\text{nat-lcm } x\ y = x * y \text{ div } (\text{nat-gcd } x\ y)$

definition *int-gcd* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**
 $\text{int-gcd } x\ y = \text{int } (\text{nat-gcd } (\text{nat } (\text{abs } x))\ (\text{nat } (\text{abs } y)))$

definition *int-lcm* :: $\text{int} \Rightarrow \text{int} \Rightarrow \text{int}$ **where**
 $\text{int-lcm } x\ y = \text{int } (\text{nat-lcm } (\text{nat } (\text{abs } x))\ (\text{nat } (\text{abs } y)))$

definition *Frac* :: $\text{int} \times \text{int} \Rightarrow \text{bool}$ **where**
 $\text{Frac} \equiv \lambda(a, b). b > 0 \wedge \text{int-gcd } a\ b = 1$

axiomatization *Abs-Frac* :: $\text{int} \times \text{int} \Rightarrow 'a$
and *Rep-Frac* :: $'a \Rightarrow \text{int} \times \text{int}$

definition *zero-frac* :: $'a$ **where**

zero-frac \equiv *Abs-Frac* (0, 1)

definition *one-frac* :: 'a where
one-frac \equiv *Abs-Frac* (1, 1)

definition *num* :: 'a \Rightarrow int where
num \equiv *fst* o *Rep-Frac*

definition *denom* :: 'a \Rightarrow int where
denom \equiv *snd* o *Rep-Frac*

function *norm-frac* :: int \Rightarrow int \Rightarrow int \times int where
 $[simp\ del]:$ *norm-frac* a b = (if b < 0 then *norm-frac* (- a) (- b)
 else if a = 0 \vee b = 0 then (0, 1)
 else let c = *int-gcd* a b in (a div c, b div c))

$\langle proof \rangle$

termination $\langle proof \rangle$

definition *frac* :: int \Rightarrow int \Rightarrow 'a where
frac a b \equiv *Abs-Frac* (*norm-frac* a b)

definition *plus-frac* :: 'a \Rightarrow 'a \Rightarrow 'a where
 $[nitpick-simp]:$
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
 $[nitpick-simp]:$
times-frac q r = *frac* (num q * num r) (*denom* q * *denom* r)

definition *uminus-frac* :: 'a \Rightarrow 'a where
uminus-frac q \equiv *Abs-Frac* (- num q, *denom* q)

definition *number-of-frac* :: int \Rightarrow 'a where
number-of-frac n \equiv *Abs-Frac* (n, 1)

definition *inverse-frac* :: 'a \Rightarrow 'a where
inverse-frac q \equiv *frac* (*denom* q) (num q)

definition *less-eq-frac* :: 'a \Rightarrow 'a \Rightarrow bool where
 $[nitpick-simp]:$
less-eq-frac q r \longleftrightarrow num (*plus-frac* q (*uminus-frac* r)) \leq 0

definition *of-frac* :: 'a \Rightarrow 'b::{inverse,ring-1} where
of-frac q \equiv *of-int* (num q) / *of-int* (*denom* q)

$\langle ML \rangle$

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* = (if $0 \leq l$ then *k div l* else $-(k \text{ div } (-l))$)

definition *z3mod* :: *int* \Rightarrow *int* \Rightarrow *int* **where**
z3mod *k l* = (if $0 \leq l$ then *k mod l* else *k mod* $(-l)$)

lemma *div-by-z3div*: *k 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* $(-k) (-l)$)

$\langle proof \rangle$

lemma *mod-by-z3mod*: $k \bmod l =$ (
 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 $-z3mod\ (-k)\ (-l)$
 $\langle proof \rangle$

58.5 Setup

$\langle ML \rangle$

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*]:

$(P \longrightarrow Q) = (Q \vee \neg P)$
 $(\neg P \longrightarrow Q) = (P \vee Q)$

$$(\neg P \longrightarrow Q) = (Q \vee P)$$

<proof>

lemma [z3-rule]:

$$((P = Q) \longrightarrow R) = (R \mid (Q = (\neg P)))$$

<proof>

lemma [z3-rule]:

$$\begin{aligned} ((\neg P) = P) &= False \\ (P = (\neg P)) &= False \\ (P \neq Q) &= (Q = (\neg P)) \\ (P = Q) &= ((\neg P \vee Q) \wedge (P \vee \neg Q)) \\ (P \neq Q) &= ((\neg P \vee \neg Q) \wedge (P \vee Q)) \end{aligned}$$

<proof>

lemma [z3-rule]:

$$\begin{aligned} (if\ P\ then\ P\ else\ \neg P) &= True \\ (if\ \neg P\ then\ \neg P\ else\ P) &= True \\ (if\ P\ then\ True\ else\ False) &= P \\ (if\ P\ then\ False\ else\ True) &= (\neg P) \\ (if\ \neg P\ then\ x\ else\ y) &= (if\ P\ then\ y\ else\ x) \end{aligned}$$

<proof>

lemma [z3-rule]:

$$\begin{aligned} P &= Q \vee P \vee Q \\ P &= Q \vee \neg P \vee \neg Q \\ (\neg P) &= Q \vee \neg P \vee Q \\ (\neg P) &= Q \vee P \vee \neg Q \\ P &= (\neg Q) \vee \neg P \vee Q \\ P &= (\neg Q) \vee P \vee \neg Q \\ P \neq Q &= P \vee \neg Q \\ P \neq Q &= \neg P \vee Q \\ P \neq (\neg Q) &= P \vee Q \\ (\neg P) \neq Q &= P \vee Q \\ P \vee Q \vee P &\neq (\neg Q) \\ P \vee Q \vee (\neg P) &\neq Q \\ P \vee \neg Q \vee P &\neq Q \\ \neg P \vee Q \vee P &\neq Q \end{aligned}$$

<proof>

lemma [z3-rule]:

$$\begin{aligned} 0 + (x::int) &= x \\ x + 0 &= x \\ 0 * x &= 0 \\ 1 * x &= x \\ x + y &= y + x \end{aligned}$$

<proof>

```
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
```

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.