

Miscellaneous HOL Examples

June 21, 2010

Contents

1	An experimental alternative numeral representation.	8
1.1	The <i>num</i> type	8
1.2	Numeral operations	9
1.3	Binary numerals	12
1.4	Class-specific numeral rules	14
1.4.1	Class <i>semiring-numeral</i>	14
1.4.2	Structures with a zero: class <i>semiring-1</i>	15
1.4.3	Equality: class <i>semiring-char-0</i>	16
1.4.4	Comparisons: class <i>linordered-semidom</i>	16
1.4.5	Structures with subtraction: class <i>semiring-1-minus</i>	19
1.4.6	Structures with negation: class <i>ring-1</i>	21
1.4.7	Structures with exponentiation	21
1.4.8	Greetings to <i>nat</i>	22
1.5	Code generator setup for <i>int</i>	22
1.6	Numeral equations as default simplification rules	25
1.7	Simplification Procedures	26
1.7.1	Reorientation of equalities	26
1.7.2	Constant folding for multiplication in semirings	26
1.8	Toy examples	27
2	Foundations of HOL	27
2.1	Pure Logic	28
2.1.1	Basic logical connectives	28
2.1.2	Extensional equality	28
2.1.3	Derived connectives	29
2.2	Classical logic	32
3	Abstract Natural Numbers primitive recursion	34
4	Proof by guessing	37

5	Simple and efficient binary numerals	37
5.1	Binary representation of natural numbers	37
5.2	Direct operations – plain normalization	38
5.3	Indirect operations – ML will produce witnesses	39
5.4	Concrete syntax	41
5.5	Examples	41
6	Examples of recdef definitions	44
7	Examples of function definitions	46
7.1	Very basic	46
7.2	Currying	47
7.3	Nested recursion	47
7.4	More general patterns	48
7.4.1	Overlapping patterns	48
7.4.2	Guards	48
7.5	Mutual Recursion	49
7.6	Definitions in local contexts	50
7.7	Regression tests	50
8	Examples of automatically derived induction rules	53
8.1	Some simple induction principles on <i>nat</i>	54
9	Some of the results in Inductive Invariants for Nested Re- cursion	54
10	Example use if an inductive invariant to solve termination conditions	56
11	Test of Locale Interpretation	59
12	Interpretation of Defined Concepts	59
12.1	Lattices	59
12.1.1	Definitions	59
12.1.2	Total order \leq on <i>int</i>	68
12.1.3	Total order \leq on <i>nat</i>	69
12.1.4	Lattice <i>dvd</i> on <i>nat</i>	71
12.2	Group example with defined operations <i>inv</i> and <i>unit</i>	72
12.2.1	Locale declarations and lemmas	72
12.2.2	Interpretation of Functions	76
13	Monoids and Groups as predicates over record schemes	78

14 Binary arithmetic examples	78
14.1 Regression Testing for Cancellation Simprocs	78
14.2 Arithmetic Method Tests	80
14.3 The Integers	81
14.4 The Natural Numbers	84
14.5 Real Arithmetic	86
14.5.1 Addition	86
14.5.2 Negation	86
14.5.3 Multiplication	86
14.5.4 Inequalities	86
14.5.5 Powers	87
14.5.6 Tests	87
14.6 Complex Arithmetic	93
15 Examples for hexadecimal and binary numerals	93
16 Antiquotations	94
17 Multiple nested quotations and anti-quotations	95
18 Partial equivalence relations	96
18.1 Partial equivalence	96
18.2 Equivalence on function spaces	97
18.3 Total equivalence	98
18.4 Quotient types	98
18.5 Equality on quotients	99
18.6 Picking representing elements	100
19 Summing natural numbers	101
20 Three Divides Theorem	103
20.1 Abstract	103
20.2 Formal proof	103
20.2.1 Miscellaneous summation lemmas	103
20.2.2 Generalised Three Divides	104
20.2.3 Three Divides Natural	105
21 Higher-Order Logic: Intuitionistic predicate calculus problems	107
22 CTL formulae	113
22.1 Basic fixed point properties	114
22.2 The tree induction principle	116
22.3 An application of tree induction	118

23 Arithmetic	119
23.1 Splitting of Operators: <i>max, min, abs, op −, nat, op mod, op</i> <i>div</i>	119
23.2 Meta-Logic	121
23.3 Various Other Examples	121
24 Binary trees	123
25 2-3 Trees	127
26 Merge Sort	131
27 A lemma for Lagrange’s theorem	132
28 Groebner Basis Examples	133
28.1 Basic examples	133
28.2 Lemmas for Lagrange’s theorem	134
28.3 Colinearity is invariant by rotation	135
29 Milner-Tofte: Co-induction in Relational Semantics	135
30 Case study: Unification Algorithm	155
30.1 Basic definitions	156
30.2 Basic lemmas	156
30.3 Specification: Most general unifiers	157
30.4 The unification algorithm	158
30.5 Partial correctness	158
30.6 Properties used in termination proof	160
30.7 Termination proof	165
31 Primitive Recursive Functions	166
31.1 Ackermann’s Function	166
31.2 Primitive Recursive Functions	168
32 The Full Theorem of Tarski	171
32.1 Partial Order	174
32.2 sublattice	178
32.3 lub	178
32.4 glb	180
32.5 fixed points	181
32.6 lemmas for Tarski, lub	181
32.7 Tarski fixpoint theorem 1, first part	182
32.8 interval	183
32.9 Top and Bottom	186
32.10fixed points form a partial order	187

33 Classical Predicate Calculus Problems	190
33.1 Traditional Classical Reasoner	190
33.1.1 Pelletier’s examples	190
33.1.2 Classical Logic: examples with quantifiers	192
33.1.3 Problems requiring quantifier duplication	192
33.1.4 Hard examples with quantifiers	193
33.1.5 Problems (mainly) involving equality or functions	196
33.2 Model Elimination Prover	198
33.2.1 Pelletier’s examples	199
33.2.2 Classical Logic: examples with quantifiers	200
33.2.3 Hard examples with quantifiers	201
34 Set Theory examples: Cantor’s Theorem, Schröder-Bernstein Theorem, etc.	207
34.1 Examples for the <i>blast</i> paper	207
34.2 Cantor’s Theorem: There is no surjection from a set to its powerset	208
34.3 The Schröder-Bernstein Theorem	208
34.4 A simple party theorem	209
35 Meson test cases	211
35.1 Interactive examples	211
36 Examples and regression tests for automated termination proofs	287
36.1 Manually giving termination relations using <i>relation</i> and <i>measure</i>	287
36.2 <i>lexicographic-order</i> : Trivial examples	287
36.3 Examples on natural numbers	287
36.4 Simple examples with other datatypes than nat, e.g. trees and lists	289
36.5 Examples with mutual recursion	289
36.6 Refined analysis: The <i>size-change</i> method	290
37 Coherent Logic Problems	291
37.1 Equivalence of two versions of Pappus’ Axiom	291
37.2 Preservation of the Diamond Property under reflexive closure	292
38 Some examples for Presburger Arithmetic	293
39 Generic reflection and reification	295
40 Examples for generic reflection and reification	296

41 Square roots of primes are irrational	305
41.1 Variations	306
42 Square roots of primes are irrational (script version)	307
42.1 Preliminaries	307
42.2 Main theorem	308
43 Various examples for transfer procedure	308
44 Arithmetic Series for Reals	309
45 Divergence of the Harmonic Series	310
45.1 Abstract	310
45.2 Formal Proof	310
46 Examples for the 'refute' command	316
46.1 Examples and Test Cases	316
46.1.1 Propositional logic	316
46.1.2 Predicate logic	317
46.1.3 Equality	317
46.1.4 First-Order Logic	318
46.1.5 Higher-Order Logic	320
46.1.6 Meta-logic	321
46.1.7 Schematic variables	322
46.1.8 Abstractions	322
46.1.9 Sets	323
46.1.10 undefined	323
46.1.11 The	324
46.1.12 Eps	324
46.1.13 Subtypes (typedef), typedecl	325
46.1.14 Inductive datatypes	325
46.1.15 Records	341
46.1.16 Inductively defined sets	341
46.1.17 Examples involving special functions	342
46.1.18 Type classes and overloading	343
47 Examples for the 'quickcheck' command	345
47.1 Lists	345
47.2 Trees	347
48 Preorders with explicit equivalence relation	348

49 Comparing growth of functions on natural numbers by a preorder relation	349
49.1 Auxiliary	349
49.2 The \lesssim relation	350
49.3 The \approx relation, the equivalence relation induced by \lesssim	351
49.4 The \prec relation, the strict part of \lesssim	352
49.5 Assert that \lesssim is indeed a preorder	353
49.6 Simple examples	353
50 A simple cookbook example how to eliminate choice in programs.	354
51 Some basic facts about discrete summation	356
52 Theory of Integration on real intervals	358
52.1 Gauges	358
52.2 Gauge-fine divisions	358
52.3 Riemann sum	362
52.4 Gauge integrability (definite)	363
53 Positive real numbers	370
53.1 Properties of Ordering	374
53.2 Properties of Addition	375
53.3 Properties of Multiplication	377
53.4 Distribution of Multiplication across Addition	380
53.5 Existence of Inverse, a Positive Real	382
53.6 Gleason's Lemma 9-3.4, page 122	383
53.7 Gleason's Lemma 9-3.6	385
53.8 Existence of Inverse: Part 2	386
53.9 Subtraction for Positive Reals	388
53.10 proving that $S \leq R + D$ — trickier	390
53.11 Completeness of type <i>preal</i>	392
54 Defining the Reals from the Positive Reals	393
54.1 Equivalence relation over positive reals	395
54.2 Addition and Subtraction	396
54.3 Multiplication	397
54.4 Inverse and Division	398
54.5 The Real Numbers form a Field	399
54.6 The \leq Ordering	399
54.7 The Reals Form an Ordered Field	401
54.8 Theorems About the Ordering	403
54.9 Numerals and Arithmetic	403
54.10 Completeness of Positive Reals	403

54.11 The Archimedean Property of the Reals	409
55 Hilbert's choice and classical logic	411
55.1 Proof text	411
55.2 Proof term of text	413
55.3 Proof script	414
55.4 Proof term of script	415
56 Installing an oracle for SVC (Stanford Validity Checker)	416

1 An experimental alternative numeral representation.

```
theory Numeral
imports Main
begin
```

1.1 The *num* type

```
datatype num = One | Dig0 num | Dig1 num
```

Increment function for type *Numeral.num*

```
primrec inc :: num ⇒ num where
  inc One = Dig0 One
| inc (Dig0 x) = Dig1 x
| inc (Dig1 x) = Dig0 (inc x)
```

Converting between type *Numeral.num* and type *nat*

```
primrec nat-of-num :: num ⇒ nat where
  nat-of-num One = Suc 0
| nat-of-num (Dig0 x) = nat-of-num x + nat-of-num x
| nat-of-num (Dig1 x) = Suc (nat-of-num x + nat-of-num x)
```

```
primrec num-of-nat :: nat ⇒ num where
  num-of-nat 0 = One
| num-of-nat (Suc n) = (if 0 < n then inc (num-of-nat n) else One)
```

```
lemma nat-of-num-pos: 0 < nat-of-num x
  by (induct x) simp-all
```

```
lemma nat-of-num-neg-0: nat-of-num x ≠ 0
  by (induct x) simp-all
```

```
lemma nat-of-num-inc: nat-of-num (inc x) = Suc (nat-of-num x)
  by (induct x) simp-all
```

```
lemma num-of-nat-double:
```



```

0 < n ==> num-of-nat (n + n) = Dig0 (num-of-nat n)
by (induct n) simp-all

```

Type *Numeral.num* is isomorphic to the strictly positive natural numbers.

```

lemma nat-of-num-inverse: num-of-nat (nat-of-num x) = x
by (induct x) (simp-all add: num-of-nat-double nat-of-num-pos)

```

```

lemma num-of-nat-inverse: 0 < n ==> nat-of-num (num-of-nat n) = n
by (induct n) (simp-all add: nat-of-num-inc)

```

```

lemma num-eq-iff: x = y <=> nat-of-num x = nat-of-num y
  apply safe
  apply (drule arg-cong [where f=num-of-nat])
  apply (simp add: nat-of-num-inverse)
done

```

```

lemma num-induct [case-names One inc]:
  fixes P :: num => bool
  assumes One: P One
  and inc:  $\bigwedge x. P\ x \implies P\ (inc\ x)$ 
  shows P x
proof -
  obtain n where n: Suc n = nat-of-num x
  by (cases nat-of-num x, simp-all add: nat-of-num-neq-0)
  have P (num-of-nat (Suc n))
  proof (induct n)
    case 0 show ?case using One by simp
  next
    case (Suc n)
    then have P (inc (num-of-nat (Suc n))) by (rule inc)
    then show P (num-of-nat (Suc (Suc n))) by simp
  qed
  with n show P x
  by (simp add: nat-of-num-inverse)
qed

```

From now on, there are two possible models for *Numeral.num*: as positive naturals (rule *num-induct*) and as digit representation (rules *num.induct*, *num.cases*).

It is not entirely clear in which context it is better to use the one or the other, or whether the construction should be reversed.

1.2 Numeral operations

```

ML <<
structure Dig-Simps = Named-Thms
(
  val name = numeral

```

```

    val description = Simplification rules for numerals
  )
  >>

setup Dig-Simps.setup

instantiation num :: {plus,times,ord}
begin

definition plus-num :: num  $\Rightarrow$  num  $\Rightarrow$  num where
  [code del]: m + n = num-of-nat (nat-of-num m + nat-of-num n)

definition times-num :: num  $\Rightarrow$  num  $\Rightarrow$  num where
  [code del]: m * n = num-of-nat (nat-of-num m * nat-of-num n)

definition less-eq-num :: num  $\Rightarrow$  num  $\Rightarrow$  bool where
  [code del]: m  $\leq$  n  $\iff$  nat-of-num m  $\leq$  nat-of-num n

definition less-num :: num  $\Rightarrow$  num  $\Rightarrow$  bool where
  [code del]: m < n  $\iff$  nat-of-num m < nat-of-num n

instance ..

end

lemma nat-of-num-add: nat-of-num (x + y) = nat-of-num x + nat-of-num y
  unfolding plus-num-def
  by (intro num-of-nat-inverse add-pos-pos nat-of-num-pos)

lemma nat-of-num-mult: nat-of-num (x * y) = nat-of-num x * nat-of-num y
  unfolding times-num-def
  by (intro num-of-nat-inverse mult-pos-pos nat-of-num-pos)

lemma Dig-plus [numeral, simp, code]:
  One + One = Dig0 One
  One + Dig0 m = Dig1 m
  One + Dig1 m = Dig0 (m + One)
  Dig0 n + One = Dig1 n
  Dig0 n + Dig0 m = Dig0 (n + m)
  Dig0 n + Dig1 m = Dig1 (n + m)
  Dig1 n + One = Dig0 (n + One)
  Dig1 n + Dig0 m = Dig1 (n + m)
  Dig1 n + Dig1 m = Dig0 (n + m + One)
  by (simp-all add: num-eq-iff nat-of-num-add)

lemma Dig-times [numeral, simp, code]:
  One * One = One
  One * Dig0 n = Dig0 n
  One * Dig1 n = Dig1 n

```

$Dig0\ n * One = Dig0\ n$
 $Dig0\ n * Dig0\ m = Dig0\ (n * Dig0\ m)$
 $Dig0\ n * Dig1\ m = Dig0\ (n * Dig1\ m)$
 $Dig1\ n * One = Dig1\ n$
 $Dig1\ n * Dig0\ m = Dig0\ (n * Dig0\ m + m)$
 $Dig1\ n * Dig1\ m = Dig1\ (n * Dig1\ m + m)$
by (simp-all add: num-eq-iff nat-of-num-add nat-of-num-mult
left-distrib right-distrib)

lemma *Dig-eq*:

$One = One \longleftrightarrow True$
 $One = Dig0\ n \longleftrightarrow False$
 $One = Dig1\ n \longleftrightarrow False$
 $Dig0\ m = One \longleftrightarrow False$
 $Dig1\ m = One \longleftrightarrow False$
 $Dig0\ m = Dig0\ n \longleftrightarrow m = n$
 $Dig0\ m = Dig1\ n \longleftrightarrow False$
 $Dig1\ m = Dig0\ n \longleftrightarrow False$
 $Dig1\ m = Dig1\ n \longleftrightarrow m = n$
by simp-all

lemma *less-eq-num-code* [numeral, simp, code]:

$One \leq n \longleftrightarrow True$
 $Dig0\ m \leq One \longleftrightarrow False$
 $Dig1\ m \leq One \longleftrightarrow False$
 $Dig0\ m \leq Dig0\ n \longleftrightarrow m \leq n$
 $Dig0\ m \leq Dig1\ n \longleftrightarrow m \leq n$
 $Dig1\ m \leq Dig1\ n \longleftrightarrow m \leq n$
 $Dig1\ m \leq Dig0\ n \longleftrightarrow m < n$
using nat-of-num-pos [of n] nat-of-num-pos [of m]
by (auto simp add: less-eq-num-def less-num-def)

lemma *less-num-code* [numeral, simp, code]:

$m < One \longleftrightarrow False$
 $One < One \longleftrightarrow False$
 $One < Dig0\ n \longleftrightarrow True$
 $One < Dig1\ n \longleftrightarrow True$
 $Dig0\ m < Dig0\ n \longleftrightarrow m < n$
 $Dig0\ m < Dig1\ n \longleftrightarrow m \leq n$
 $Dig1\ m < Dig1\ n \longleftrightarrow m < n$
 $Dig1\ m < Dig0\ n \longleftrightarrow m < n$
using nat-of-num-pos [of n] nat-of-num-pos [of m]
by (auto simp add: less-eq-num-def less-num-def)

Rules using *One* and *inc* as constructors

lemma *add-One*: $x + One = inc\ x$

by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma *add-inc*: $x + inc\ y = inc\ (x + y)$

```

    by (simp add: num-eq-iff nat-of-num-add nat-of-num-inc)

lemma mult-One:  $x * One = x$ 
  by (simp add: num-eq-iff nat-of-num-mult)

lemma mult-inc:  $x * inc\ y = x * y + x$ 
  by (simp add: num-eq-iff nat-of-num-mult nat-of-num-add nat-of-num-inc)

A double-and-decrement function

primrec DigM :: num  $\Rightarrow$  num where
  DigM One = One
  | DigM (Dig0 n) = Dig1 (DigM n)
  | DigM (Dig1 n) = Dig1 (Dig0 n)

lemma DigM-plus-one:  $DigM\ n + One = Dig0\ n$ 
  by (induct n) simp-all

lemma add-One-commute:  $One + n = n + One$ 
  by (induct n) simp-all

lemma one-plus-DigM:  $One + DigM\ n = Dig0\ n$ 
  unfolding add-One-commute DigM-plus-one ..

```

Squaring and exponentiation

```

primrec square :: num  $\Rightarrow$  num where
  square One = One
  | square (Dig0 n) = Dig0 (Dig0 (square n))
  | square (Dig1 n) = Dig1 (Dig0 (square n + n))

primrec pow :: num  $\Rightarrow$  num  $\Rightarrow$  num
where
  pow x One = x
  | pow x (Dig0 y) = square (pow x y)
  | pow x (Dig1 y) = x * square (pow x y)

```

1.3 Binary numerals

We embed binary representations into a generic algebraic structure using *of-num*.

```

class semiring-numeral = semiring + monoid-mult
begin

```

```

primrec of-num :: num  $\Rightarrow$  'a where
  of-num-One [numeral]: of-num One = 1
  | of-num (Dig0 n) = of-num n + of-num n
  | of-num (Dig1 n) = of-num n + of-num n + 1

```

```

lemma of-num-inc: of-num (inc x) = of-num x + 1

```

```

    by (induct x) (simp-all add: add-ac)

lemma of-num-add: of-num (m + n) = of-num m + of-num n
  apply (induct n rule: num-induct)
  apply (simp-all add: add-One add-inc of-num-inc add-ac)
  done

lemma of-num-mult: of-num (m * n) = of-num m * of-num n
  apply (induct n rule: num-induct)
  apply (simp add: mult-One)
  apply (simp add: mult-inc of-num-add of-num-inc right-distrib)
  done

declare of-num.simps [simp del]

end

ML stuff and syntax.

ML <<
fun mk-num k =
  if k > 1 then
    let
      val (l, b) = Integer.div-mod k 2;
      val bit = (if b = 0 then @{term Dig0} else @{term Dig1});
    in bit $ (mk-num l) end
  else if k = 1 then @{term One}
  else error (mk-num ^ string-of-int k);

fun dest-num @{term One} = 1
  | dest-num (@{term Dig0} $ n) = 2 * dest-num n
  | dest-num (@{term Dig1} $ n) = 2 * dest-num n + 1
  | dest-num t = raise TERM (dest-num, [t]);

(*FIXME these have to gain proper context via morphisms phi*)

fun mk-numeral T k = Const (@{const-name of-num}, @{typ num} --> T)
  $ mk-num k

fun dest-numeral (Const (@{const-name of-num}, Type (fun, [@{typ num}, T])))
  $ t) =
  (T, dest-num t)
>>

syntax
  -Numerals :: xnum => 'a    (-)

parse-translation <<
let
  fun num-of-int n = if n > 0 then case IntInf.quotRem (n, 2)

```

```

    of (0, 1) => Const (@{const-name One}, dummyT)
    | (n, 0) => Const (@{const-name Dig0}, dummyT) $ num-of-int n
    | (n, 1) => Const (@{const-name Dig1}, dummyT) $ num-of-int n
    else raise Match;
fun numeral-tr [Free (num, -)] =
  let
    val {leading-zeros, value, ...} = Syntax.read-xnum num;
    val - = leading-zeros = 0 andalso value > 0
    orelse error (Bad numeral: ^ num);
  in Const (@{const-name of-num}, @ {typ num} --> dummyT) $ num-of-int
value end
  | numeral-tr ts = raise TERM (numeral-tr, ts);
in [(@{syntax-const -Numerals}, numeral-tr)] end
>>

```

typed-print-translation <<

```

let
  fun dig b n = b + 2 * n;
  fun int-of-num' (Const (@{const-syntax Dig0}, -) $ n) =
    dig 0 (int-of-num' n)
  | int-of-num' (Const (@{const-syntax Dig1}, -) $ n) =
    dig 1 (int-of-num' n)
  | int-of-num' (Const (@{const-syntax One}, -)) = 1;
  fun num-tr' show-sorts T [n] =
    let
      val k = int-of-num' n;
      val t' = Syntax.const @ {syntax-const -Numerals} $ Syntax.free (# ^ string-of-int
k);
    in case T
      of Type (@{type-name fun}, [-, T']) =>
         if not (! show-types) andalso can Term.dest-Type T' then t'
         else Syntax.const Syntax.constrainC $ t' $ Syntax.term-of-typ show-sorts
T'
      | T' => if T' = dummyT then t' else raise Match
    end;
in [(@{const-syntax of-num}, num-tr')] end
>>

```

1.4 Class-specific numeral rules

of-num is a morphism.

1.4.1 Class *semiring-numeral*

context *semiring-numeral*
begin

abbreviation *Num1* \equiv *of-num One*

Alas, there is still the duplication of $1::'a$, though the duplicated $0::'b$ has disappeared. We could get rid of it by replacing the constructor $1::'a$ in *Numeral.num* by two constructors *two* and *three*, resulting in a further blow-up. But it could be worth the effort.

```
lemma of-num-plus-one [numeral]:
  of-num  $n + 1 = \textit{of-num}$  ( $n + \textit{One}$ )
by (simp only: of-num-add of-num-One)
```

```
lemma of-num-one-plus [numeral]:
   $1 + \textit{of-num}$   $n = \textit{of-num}$  ( $\textit{One} + n$ )
by (simp only: of-num-add of-num-One)
```

```
lemma of-num-plus [numeral]:
  of-num  $m + \textit{of-num}$   $n = \textit{of-num}$  ( $m + n$ )
unfolding of-num-add ..
```

```
lemma of-num-times-one [numeral]:
  of-num  $n * 1 = \textit{of-num}$   $n$ 
by simp
```

```
lemma of-num-one-times [numeral]:
   $1 * \textit{of-num}$   $n = \textit{of-num}$   $n$ 
by simp
```

```
lemma of-num-times [numeral]:
  of-num  $m * \textit{of-num}$   $n = \textit{of-num}$  ( $m * n$ )
unfolding of-num-mult ..
```

end

1.4.2 Structures with a zero: class *semiring-1*

```
context semiring-1
begin
```

```
subclass semiring-numeral ..
```

```
lemma of-nat-of-num [numeral]: of-nat (of-num  $n$ ) = of-num  $n$ 
by (induct  $n$ )
  (simp-all add: semiring-numeral-class.of-num.simps of-num.simps add-ac)
```

```
declare of-nat-1 [numeral]
```

```
lemma Dig-plus-zero [numeral]:
   $0 + 1 = 1$ 
   $0 + \textit{of-num}$   $n = \textit{of-num}$   $n$ 
   $1 + 0 = 1$ 
  of-num  $n + 0 = \textit{of-num}$   $n$ 
by simp-all
```

```

lemma Dig-times-zero [numeral]:
  0 * 1 = 0
  0 * of-num n = 0
  1 * 0 = 0
  of-num n * 0 = 0
  by simp-all

end

lemma nat-of-num-of-num: nat-of-num = of-num
proof
  fix n
  have of-num n = nat-of-num n
    by (induct n) (simp-all add: of-num.simps)
  then show nat-of-num n = of-num n by simp
qed

```

1.4.3 Equality: class *semiring-char-0*

```

context semiring-char-0
begin

lemma of-num-eq-iff [numeral]: of-num m = of-num n  $\longleftrightarrow$  m = n
  unfolding of-nat-of-num [symmetric] nat-of-num-of-num [symmetric]
    of-nat-eq-iff num-eq-iff ..

lemma of-num-eq-one-iff [numeral]: of-num n = 1  $\longleftrightarrow$  n = One
  using of-num-eq-iff [of n One] by (simp add: of-num-One)

lemma one-eq-of-num-iff [numeral]: 1 = of-num n  $\longleftrightarrow$  One = n
  using of-num-eq-iff [of One n] by (simp add: of-num-One)

end

```

1.4.4 Comparisons: class *linordered-semidom*

Could be perhaps more general than here.

```

context linordered-semidom
begin

lemma of-num-pos [numeral]: 0 < of-num n
  by (induct n) (simp-all add: of-num.simps add-pos-pos)

lemma of-num-less-eq-iff [numeral]: of-num m ≤ of-num n  $\longleftrightarrow$  m ≤ n
proof –
  have of-nat (of-num m) ≤ of-nat (of-num n)  $\longleftrightarrow$  m ≤ n
    unfolding less-eq-num-def nat-of-num-of-num of-nat-le-iff ..
  then show ?thesis by (simp add: of-nat-of-num)

```


qed

lemma *of-num-less-eq-one-iff* [numeral]: $of\text{-}num\ n \leq 1 \iff n \leq One$
using *of-num-less-eq-iff* [of n One] **by** (simp add: of-num-One)

lemma *one-less-eq-of-num-iff* [numeral]: $1 \leq of\text{-}num\ n$
using *of-num-less-eq-iff* [of One n] **by** (simp add: of-num-One)

lemma *of-num-less-iff* [numeral]: $of\text{-}num\ m < of\text{-}num\ n \iff m < n$
proof –
have $of\text{-}nat\ (of\text{-}num\ m) < of\text{-}nat\ (of\text{-}num\ n) \iff m < n$
unfolding *less-num-def* *nat-of-num-of-num* *of-nat-less-iff* ..
then show ?thesis **by** (simp add: of-nat-of-num)
qed

lemma *of-num-less-one-iff* [numeral]: $\neg of\text{-}num\ n < 1$
using *of-num-less-iff* [of n One] **by** (simp add: of-num-One)

lemma *one-less-of-num-iff* [numeral]: $1 < of\text{-}num\ n \iff One < n$
using *of-num-less-iff* [of One n] **by** (simp add: of-num-One)

lemma *of-num-nonneg* [numeral]: $0 \leq of\text{-}num\ n$
by (induct n) (simp-all add: of-num.simps add-nonneg-nonneg)

lemma *of-num-less-zero-iff* [numeral]: $\neg of\text{-}num\ n < 0$
by (simp add: not-less of-num-nonneg)

lemma *of-num-le-zero-iff* [numeral]: $\neg of\text{-}num\ n \leq 0$
by (simp add: not-le of-num-pos)

end

context *linordered-idom*
begin

lemma *minus-of-num-less-of-num-iff*: $- of\text{-}num\ m < of\text{-}num\ n$
proof –
have $- of\text{-}num\ m < 0$ **by** (simp add: of-num-pos)
also have $0 < of\text{-}num\ n$ **by** (simp add: of-num-pos)
finally show ?thesis .
qed

lemma *minus-of-num-less-one-iff*: $- of\text{-}num\ n < 1$
using *minus-of-num-less-of-num-iff* [of n One] **by** (simp add: of-num-One)

lemma *minus-one-less-of-num-iff*: $- 1 < of\text{-}num\ n$
using *minus-of-num-less-of-num-iff* [of One n] **by** (simp add: of-num-One)

lemma *minus-one-less-one-iff*: $- 1 < 1$

using *minus-of-num-less-of-num-iff* [*of One One*] **by** (*simp add: of-num-One*)

lemma *minus-of-num-le-of-num-iff*: $- \text{of-num } m \leq \text{of-num } n$
by (*simp add: less-imp-le minus-of-num-less-of-num-iff*)

lemma *minus-of-num-le-one-iff*: $- \text{of-num } n \leq 1$
by (*simp add: less-imp-le minus-of-num-less-one-iff*)

lemma *minus-one-le-of-num-iff*: $- 1 \leq \text{of-num } n$
by (*simp add: less-imp-le minus-one-less-of-num-iff*)

lemma *minus-one-le-one-iff*: $- 1 \leq 1$
by (*simp add: less-imp-le minus-one-less-one-iff*)

lemma *of-num-le-minus-of-num-iff*: $\neg \text{of-num } m \leq - \text{of-num } n$
by (*simp add: not-le minus-of-num-less-of-num-iff*)

lemma *one-le-minus-of-num-iff*: $\neg 1 \leq - \text{of-num } n$
by (*simp add: not-le minus-of-num-less-one-iff*)

lemma *of-num-le-minus-one-iff*: $\neg \text{of-num } n \leq - 1$
by (*simp add: not-le minus-one-less-of-num-iff*)

lemma *one-le-minus-one-iff*: $\neg 1 \leq - 1$
by (*simp add: not-le minus-one-less-one-iff*)

lemma *of-num-less-minus-of-num-iff*: $\neg \text{of-num } m < - \text{of-num } n$
by (*simp add: not-less minus-of-num-le-of-num-iff*)

lemma *one-less-minus-of-num-iff*: $\neg 1 < - \text{of-num } n$
by (*simp add: not-less minus-of-num-le-one-iff*)

lemma *of-num-less-minus-one-iff*: $\neg \text{of-num } n < - 1$
by (*simp add: not-less minus-one-le-of-num-iff*)

lemma *one-less-minus-one-iff*: $\neg 1 < - 1$
by (*simp add: not-less minus-one-le-one-iff*)

lemmas *le-signed-numeral-special* [*numeral*] =
minus-of-num-le-of-num-iff
minus-of-num-le-one-iff
minus-one-le-of-num-iff
minus-one-le-one-iff
of-num-le-minus-of-num-iff
one-le-minus-of-num-iff
of-num-le-minus-one-iff
one-le-minus-one-iff

lemmas *less-signed-numeral-special* [*numeral*] =

```

minus-of-num-less-of-num-iff
minus-of-num-less-one-iff
minus-one-less-of-num-iff
minus-one-less-one-iff
of-num-less-minus-of-num-iff
one-less-minus-of-num-iff
of-num-less-minus-one-iff
one-less-minus-one-iff

end

1.4.5 Structures with subtraction: class semiring-1-minus

class semiring-minus = semiring + minus + zero +
  assumes minus-inverts-plus1:  $a + b = c \implies c - b = a$ 
  assumes minus-minus-zero-inverts-plus1:  $a + b = c \implies b - c = 0 - a$ 
begin

lemma minus-inverts-plus2:  $a + b = c \implies c - a = b$ 
  by (simp add: add-ac minus-inverts-plus1 [of b a])

lemma minus-minus-zero-inverts-plus2:  $a + b = c \implies a - c = 0 - b$ 
  by (simp add: add-ac minus-minus-zero-inverts-plus1 [of b a])

end

class semiring-1-minus = semiring-1 + semiring-minus
begin

lemma Dig-of-num-pos:
  assumes  $k + n = m$ 
  shows  $\text{of-num } m - \text{of-num } n = \text{of-num } k$ 
  using assms by (simp add: of-num-plus minus-inverts-plus1)

lemma Dig-of-num-zero:
  shows  $\text{of-num } n - \text{of-num } n = 0$ 
  by (rule minus-inverts-plus1) simp

lemma Dig-of-num-neg:
  assumes  $k + m = n$ 
  shows  $\text{of-num } m - \text{of-num } n = 0 - \text{of-num } k$ 
  by (rule minus-minus-zero-inverts-plus1) (simp add: of-num-plus assms)

lemmas Dig-plus-eval =
  of-num-plus of-num-eq-iff Dig-plus refl [of One, THEN eqTrueI] num.inject

simproc-setup numeral-minus ( $\text{of-num } m - \text{of-num } n$ ) = ⟨⟨
  let
    (*TODO proper implicit use of morphism via pattern antiquotations*)

```

```

    fun cdest-of-num ct = (snd o split-last o snd o Drule.strip-comb) ct;
    fun cdest-minus ct = case (rev o snd o Drule.strip-comb) ct of [n, m] => (m,
n);
    fun attach-num ct = (dest-num (Thm.term-of ct), ct);
    fun cdifference t = (pairself (attach-num o cdest-of-num) o cdest-minus) t;
    val simplify = MetaSimplifier.rewrite false (map mk-meta-eq @ {thms Dig-plus-eval});
    fun cert ck cl cj = @ {thm eqTrueE} OF [@ {thm meta-eq-to-obj-eq} OF [simplify
(Drule.list-comb (@ {cterm op = :: num => -},
    [Drule.list-comb (@ {cterm op + :: num => -}, [ck, cl]), cj])]]];
    in fn phi => fn - => fn ct => case try cdifference ct
    of NONE => (NONE)
    | SOME ((k, ck), (l, cl)) => SOME (let val j = k - l in if j = 0
    then MetaSimplifier.rewrite false [mk-meta-eq (Morphism.thm phi @ {thm
Dig-of-num-zero})] ct
    else mk-meta-eq (let
    val cj = Thm.cterm-of (Thm.theory-of-cterm ct) (mk-num (abs j));
    in
    (if j > 0 then (Morphism.thm phi @ {thm Dig-of-num-pos}) OF [cert cj cl
ck]
    else (Morphism.thm phi @ {thm Dig-of-num-neg}) OF [cert cj ck cl])
    end) end)
    end
  >>

```

lemma *Dig-of-num-minus-zero* [numeral]:

```

  of-num n - 0 = of-num n
  by (simp add: minus-inverts-plus1)

```

lemma *Dig-one-minus-zero* [numeral]:

```

  1 - 0 = 1
  by (simp add: minus-inverts-plus1)

```

lemma *Dig-one-minus-one* [numeral]:

```

  1 - 1 = 0
  by (simp add: minus-inverts-plus1)

```

lemma *Dig-of-num-minus-one* [numeral]:

```

  of-num (Dig0 n) - 1 = of-num (DigM n)
  of-num (Dig1 n) - 1 = of-num (Dig0 n)
  by (auto intro: minus-inverts-plus1 simp add: DigM-plus-one of-num.simps of-num-plus-one)

```

lemma *Dig-one-minus-of-num* [numeral]:

```

  1 - of-num (Dig0 n) = 0 - of-num (DigM n)
  1 - of-num (Dig1 n) = 0 - of-num (Dig0 n)
  by (auto intro: minus-minus-zero-inverts-plus1 simp add: DigM-plus-one of-num.simps
of-num-plus-one)

```

end

1.4.6 Structures with negation: class *ring-1*

context *ring-1*

begin

subclass *semiring-1-minus*

proof **qed** (*simp-all add: algebra-simps*)

lemma *Dig-zero-minus-of-num* [*numeral*]:

$0 - \text{of-num } n = - \text{of-num } n$

by *simp*

lemma *Dig-zero-minus-one* [*numeral*]:

$0 - 1 = - 1$

by *simp*

lemma *Dig-uminus-uminus* [*numeral*]:

$- (- \text{of-num } n) = \text{of-num } n$

by *simp*

lemma *Dig-plus-uminus* [*numeral*]:

$\text{of-num } m + - \text{of-num } n = \text{of-num } m - \text{of-num } n$

$- \text{of-num } m + \text{of-num } n = \text{of-num } n - \text{of-num } m$

$- \text{of-num } m + - \text{of-num } n = - (\text{of-num } m + \text{of-num } n)$

$\text{of-num } m - - \text{of-num } n = \text{of-num } m + \text{of-num } n$

$- \text{of-num } m - \text{of-num } n = - (\text{of-num } m + \text{of-num } n)$

$- \text{of-num } m - - \text{of-num } n = \text{of-num } n - \text{of-num } m$

by (*simp-all add: diff-minus add-commute*)

lemma *Dig-times-uminus* [*numeral*]:

$- \text{of-num } n * \text{of-num } m = - (\text{of-num } n * \text{of-num } m)$

$\text{of-num } n * - \text{of-num } m = - (\text{of-num } n * \text{of-num } m)$

$- \text{of-num } n * - \text{of-num } m = \text{of-num } n * \text{of-num } m$

by *simp-all*

lemma *of-int-of-num* [*numeral*]: $\text{of-int } (\text{of-num } n) = \text{of-num } n$

by (*induct n*)

(*simp-all only: of-num.simps semiring-numeral-class.of-num.simps of-int-add, simp-all*)

declare *of-int-1* [*numeral*]

end

1.4.7 Structures with exponentiation

lemma *of-num-square*: $\text{of-num } (\text{square } x) = \text{of-num } x * \text{of-num } x$

by (*induct x*)

(*simp-all add: of-num.simps of-num-add algebra-simps*)

```

lemma of-num-pow: of-num (pow x y) = of-num x ^ of-num y
by (induct y)
  (simp-all add: of-num.simps of-num-square of-num-mult power-add)

lemma power-of-num [numeral]: of-num x ^ of-num y = of-num (pow x y)
  unfolding of-num-pow ..

lemma power-zero-of-num [numeral]:
  0 ^ of-num n = (0::'a::semiring-1)
  using of-num-pos [where n=n and ?'a=nat]
  by (simp add: power-0-left)

lemma power-minus-Dig0 [numeral]:
  fixes x :: 'a::ring-1
  shows (- x) ^ of-num (Dig0 n) = x ^ of-num (Dig0 n)
  by (induct n rule: num-induct) (simp-all add: of-num.simps of-num-inc)

lemma power-minus-Dig1 [numeral]:
  fixes x :: 'a::ring-1
  shows (- x) ^ of-num (Dig1 n) = - (x ^ of-num (Dig1 n))
  by (induct n rule: num-induct) (simp-all add: of-num.simps of-num-inc)

declare power-one [numeral]

1.4.8 Greetings to nat.

instance nat :: semiring-1-minus proof qed simp-all

lemma Suc-of-num [numeral]: Suc (of-num n) = of-num (n + One)
  unfolding of-num-plus-one [symmetric] by simp

lemma nat-number:
  1 = Suc 0
  of-num One = Suc 0
  of-num (Dig0 n) = Suc (of-num (DigM n))
  of-num (Dig1 n) = Suc (of-num (Dig0 n))
  by (simp-all add: of-num.simps DigM-plus-one Suc-of-num)

declare diff-0-eq-0 [numeral]

```

1.5 Code generator setup for int

```

definition Pls :: num  $\Rightarrow$  int where
  [simp, code-post]: Pls n = of-num n

definition Mns :: num  $\Rightarrow$  int where
  [simp, code-post]: Mns n = - of-num n

code-datatype 0::int Pls Mns

```

lemmas [code-unfold] = Pls-def [symmetric] Mns-def [symmetric]

definition sub :: num \Rightarrow num \Rightarrow int **where**
 [simp, code del]: sub m n = (of-num m - of-num n)

definition dup :: int \Rightarrow int **where**
 [code del]: dup k = 2 * k

lemma Dig-sub [code]:
 sub One One = 0
 sub (Dig0 m) One = of-num (DigM m)
 sub (Dig1 m) One = of-num (Dig0 m)
 sub One (Dig0 n) = - of-num (DigM n)
 sub One (Dig1 n) = - of-num (Dig0 n)
 sub (Dig0 m) (Dig0 n) = dup (sub m n)
 sub (Dig1 m) (Dig1 n) = dup (sub m n)
 sub (Dig1 m) (Dig0 n) = dup (sub m n) + 1
 sub (Dig0 m) (Dig1 n) = dup (sub m n) - 1
apply (simp-all add: dup-def algebra-simps)
apply (simp-all add: of-num-plus one-plus-DigM)[4]
apply (simp-all add: of-num.simps)
done

lemma dup-code [code]:
 dup 0 = 0
 dup (Pls n) = Pls (Dig0 n)
 dup (Mns n) = Mns (Dig0 n)
by (simp-all add: dup-def of-num.simps)

lemma [code, code del]:
 (1 :: int) = 1
 (op + :: int \Rightarrow int \Rightarrow int) = op +
 (uminus :: int \Rightarrow int) = uminus
 (op - :: int \Rightarrow int \Rightarrow int) = op -
 (op * :: int \Rightarrow int \Rightarrow int) = op *
 (eq-class.eq :: int \Rightarrow int \Rightarrow bool) = eq-class.eq
 (op \leq :: int \Rightarrow int \Rightarrow bool) = op \leq
 (op < :: int \Rightarrow int \Rightarrow bool) = op <
by rule+

lemma one-int-code [code]:
 1 = Pls One
by (simp add: of-num-One)

lemma plus-int-code [code]:
 k + 0 = (k::int)
 0 + l = (l::int)
 Pls m + Pls n = Pls (m + n)
 Pls m - Pls n = sub m n

$Mns\ m + Mns\ n = Mns\ (m + n)$
 $Mns\ m - Mns\ n = sub\ n\ m$
by (*simp-all add: of-num-add*)

lemma *uminus-int-code* [code]:
 $uminus\ 0 = (0::int)$
 $uminus\ (Pls\ m) = Mns\ m$
 $uminus\ (Mns\ m) = Pls\ m$
by *simp-all*

lemma *minus-int-code* [code]:
 $k - 0 = (k::int)$
 $0 - l = uminus\ (l::int)$
 $Pls\ m - Pls\ n = sub\ m\ n$
 $Pls\ m - Mns\ n = Pls\ (m + n)$
 $Mns\ m - Pls\ n = Mns\ (m + n)$
 $Mns\ m - Mns\ n = sub\ n\ m$
by (*simp-all add: of-num-add*)

lemma *times-int-code* [code]:
 $k * 0 = (0::int)$
 $0 * l = (0::int)$
 $Pls\ m * Pls\ n = Pls\ (m * n)$
 $Pls\ m * Mns\ n = Mns\ (m * n)$
 $Mns\ m * Pls\ n = Mns\ (m * n)$
 $Mns\ m * Mns\ n = Pls\ (m * n)$
by (*simp-all add: of-num-mult*)

lemma *eq-int-code* [code]:
 $eq_class.eq\ 0\ (0::int) \longleftrightarrow True$
 $eq_class.eq\ 0\ (Pls\ l) \longleftrightarrow False$
 $eq_class.eq\ 0\ (Mns\ l) \longleftrightarrow False$
 $eq_class.eq\ (Pls\ k)\ 0 \longleftrightarrow False$
 $eq_class.eq\ (Pls\ k)\ (Pls\ l) \longleftrightarrow eq_class.eq\ k\ l$
 $eq_class.eq\ (Pls\ k)\ (Mns\ l) \longleftrightarrow False$
 $eq_class.eq\ (Mns\ k)\ 0 \longleftrightarrow False$
 $eq_class.eq\ (Mns\ k)\ (Pls\ l) \longleftrightarrow False$
 $eq_class.eq\ (Mns\ k)\ (Mns\ l) \longleftrightarrow eq_class.eq\ k\ l$
using *of-num-pos* [of *l*, **where** $?a = int$] *of-num-pos* [of *k*, **where** $?a = int$]
by (*simp-all add: of-num-eq-iff eq*)

lemma *less-eq-int-code* [code]:
 $0 \leq (0::int) \longleftrightarrow True$
 $0 \leq Pls\ l \longleftrightarrow True$
 $0 \leq Mns\ l \longleftrightarrow False$
 $Pls\ k \leq 0 \longleftrightarrow False$
 $Pls\ k \leq Pls\ l \longleftrightarrow k \leq l$
 $Pls\ k \leq Mns\ l \longleftrightarrow False$
 $Mns\ k \leq 0 \longleftrightarrow True$

$Mns\ k \leq Pls\ l \longleftrightarrow True$
 $Mns\ k \leq Mns\ l \longleftrightarrow l \leq k$
using *of-num-pos* [*of l, where ?'a = int*] *of-num-pos* [*of k, where ?'a = int*]
by (*simp-all add: of-num-less-eq-iff*)

lemma *less-int-code* [*code*]:
 $0 < (0::int) \longleftrightarrow False$
 $0 < Pls\ l \longleftrightarrow True$
 $0 < Mns\ l \longleftrightarrow False$
 $Pls\ k < 0 \longleftrightarrow False$
 $Pls\ k < Pls\ l \longleftrightarrow k < l$
 $Pls\ k < Mns\ l \longleftrightarrow False$
 $Mns\ k < 0 \longleftrightarrow True$
 $Mns\ k < Pls\ l \longleftrightarrow True$
 $Mns\ k < Mns\ l \longleftrightarrow l < k$
using *of-num-pos* [*of l, where ?'a = int*] *of-num-pos* [*of k, where ?'a = int*]
by (*simp-all add: of-num-less-iff*)

lemma [*code-unfold del*]: $(0::int) \equiv Numeral0$ **by** *simp*
lemma [*code-unfold del*]: $(1::int) \equiv Numeral1$ **by** *simp*
declare *zero-is-num-zero* [*code-unfold del*]
declare *one-is-num-one* [*code-unfold del*]

hide-const (**open**) *sub dup*

1.6 Numeral equations as default simplification rules

declare (**in** *semiring-numeral*) *of-num-One* [*simp*]
declare (**in** *semiring-numeral*) *of-num-plus-one* [*simp*]
declare (**in** *semiring-numeral*) *of-num-one-plus* [*simp*]
declare (**in** *semiring-numeral*) *of-num-plus* [*simp*]
declare (**in** *semiring-numeral*) *of-num-times* [*simp*]

declare (**in** *semiring-1*) *of-nat-of-num* [*simp*]

declare (**in** *semiring-char-0*) *of-num-eq-iff* [*simp*]
declare (**in** *semiring-char-0*) *of-num-eq-one-iff* [*simp*]
declare (**in** *semiring-char-0*) *one-eq-of-num-iff* [*simp*]

declare (**in** *linordered-semidom*) *of-num-pos* [*simp*]
declare (**in** *linordered-semidom*) *of-num-less-eq-iff* [*simp*]
declare (**in** *linordered-semidom*) *of-num-less-eq-one-iff* [*simp*]
declare (**in** *linordered-semidom*) *one-less-eq-of-num-iff* [*simp*]
declare (**in** *linordered-semidom*) *of-num-less-iff* [*simp*]
declare (**in** *linordered-semidom*) *of-num-less-one-iff* [*simp*]
declare (**in** *linordered-semidom*) *one-less-of-num-iff* [*simp*]
declare (**in** *linordered-semidom*) *of-num-nonneg* [*simp*]
declare (**in** *linordered-semidom*) *of-num-less-zero-iff* [*simp*]
declare (**in** *linordered-semidom*) *of-num-le-zero-iff* [*simp*]

```

declare (in linordered-idom) le-signed-numeral-special [simp]
declare (in linordered-idom) less-signed-numeral-special [simp]

declare (in semiring-1-minus) Dig-of-num-minus-one [simp]
declare (in semiring-1-minus) Dig-one-minus-of-num [simp]

declare (in ring-1) Dig-plus-uminus [simp]
declare (in ring-1) of-int-of-num [simp]

declare power-of-num [simp]
declare power-zero-of-num [simp]
declare power-minus-Dig0 [simp]
declare power-minus-Dig1 [simp]

declare Suc-of-num [simp]

```

1.7 Simplification Procedures

1.7.1 Reorientation of equalities

```

setup «
  Reorient-Proc.add
  (fn Const(@{const-name of-num}, -) $ - => true
   | Const(@{const-name uminus}, -) $
     (Const(@{const-name of-num}, -) $ -) => true
   | - => false)
  »

simproc-setup reorient-num (of-num  $n = x$  | - of-num  $m = y$ ) = Reorient-Proc.proc

```

1.7.2 Constant folding for multiplication in semirings

```

context semiring-numeral
begin

lemma mult-of-num-commute:  $x * \text{of-num } n = \text{of-num } n * x$ 
by (induct  $n$ )
  (simp-all only: of-num.simps left-distrib right-distrib mult-1-left mult-1-right)

definition
  commutes-with  $a \ b \longleftrightarrow a * b = b * a$ 

lemma commutes-with-commute: commutes-with  $a \ b \implies a * b = b * a$ 
unfolding commutes-with-def .

lemma commutes-with-left-commute: commutes-with  $a \ b \implies a * (b * c) = b * (a * c)$ 
unfolding commutes-with-def by (simp only: mult-assoc [symmetric])

```

lemma *commutes-with-numeral*: *commutes-with* x (*of-num* n) *commutes-with* (*of-num* n) x

unfolding *commutes-with-def* **by** (*simp-all* *add*: *mult-of-num-commute*)

lemmas *mult-ac-numeral* =
mult-assoc
commutes-with-commute
commutes-with-left-commute
commutes-with-numeral

end

ML $\langle\langle$
structure Semiring-Times-Assoc-Data : *ASSOC-FOLD-DATA* =
struct
 val assoc-ss = *HOL-ss* *addsims* @{*thms mult-ac-numeral*}
 val eq-reflection = *eq-reflection*
 fun is-numeral (*Const*(@{*const-name of-num*}, -) \$ -) = *true*
 | *is-numeral* - = *false*;
end;

structure Semiring-Times-Assoc = *Assoc-Fold* (*Semiring-Times-Assoc-Data*);
 $\rangle\rangle$

simproc-setup *semiring-assoc-fold'* ((*a*::*'a::semiring-numeral*) * *b*) =
 $\langle\langle$ *fn phi* => *fn ss* => *fn ct* =>
 Semiring-Times-Assoc.proc ss (*Thm.term-of ct*) $\rangle\rangle$

1.8 Toy examples

definition *bar* \longleftrightarrow $\#4 * \#2 + \#7 = (\#8 :: \text{nat}) \wedge \#4 * \#2 + \#7 \geq (\#8 :: \text{int}) - \#3$

code-thms *bar*

export-code *bar* **in** *Haskell* **file** -

export-code *bar* **in** *OCaml* **module-name** *Foo* **file** -

ML $\langle\langle$ @{*code bar*} $\rangle\rangle$

end

2 Foundations of HOL

theory *Higher-Order-Logic* **imports** *Pure* **begin**

The following theory development demonstrates Higher-Order Logic itself, represented directly within the Pure framework of Isabelle. The “HOL” logic given here is essentially that of Gordon [1], although we prefer to present basic concepts in a slightly more conventional manner oriented towards plain Natural Deduction.

2.1 Pure Logic

classes *type*
default-sort *type*

typeddecl *o*
arities
 o :: *type*
 fun :: (*type*, *type*) *type*

2.1.1 Basic logical connectives

judgment
Trueprop :: *o* \Rightarrow *prop* (- 5)

axiomatization
 imp :: *o* \Rightarrow *o* \Rightarrow *o* (**infixr** \longrightarrow 25) **and**
 All :: (*'a* \Rightarrow *o*) \Rightarrow *o* (**binder** \forall 10)
where
 impI [*intro*]: (*A* \Longrightarrow *B*) \Longrightarrow *A* \longrightarrow *B* **and**
 impE [*dest*, *trans*]: *A* \longrightarrow *B* \Longrightarrow *A* \Longrightarrow *B* **and**
 allI [*intro*]: ($\bigwedge x. P\ x$) \Longrightarrow $\forall x. P\ x$ **and**
 allE [*dest*]: $\forall x. P\ x \Longrightarrow P\ a$

2.1.2 Extensional equality

axiomatization
 equal :: *'a* \Rightarrow *'a* \Rightarrow *o* (**infixl** = 50)
where
 refl [*intro*]: *x* = *x* **and**
 subst: *x* = *y* $\Longrightarrow P\ x \Longrightarrow P\ y$

axiomatization where
 ext [*intro*]: ($\bigwedge x. f\ x = g\ x$) $\Longrightarrow f = g$ **and**
 iff [*intro*]: (*A* \Longrightarrow *B*) \Longrightarrow (*B* \Longrightarrow *A*) $\Longrightarrow A = B$

theorem *sym* [*sym*]: *x* = *y* $\Longrightarrow y = x$
proof –
 assume *x* = *y*
 then show *y* = *x* **by** (*rule subst*) (*rule refl*)
qed

lemma [*trans*]: *x* = *y* $\Longrightarrow P\ y \Longrightarrow P\ x$
 by (*rule subst*) (*rule sym*)

lemma [*trans*]: *P* *x* $\Longrightarrow x = y \Longrightarrow P\ y$
 by (*rule subst*)

theorem *trans* [*trans*]: *x* = *y* $\Longrightarrow y = z \Longrightarrow x = z$
 by (*rule subst*)

theorem *iff1* [*elim*]: $A = B \implies A \implies B$
by (*rule subst*)

theorem *iff2* [*elim*]: $A = B \implies B \implies A$
by (*rule subst*) (*rule sym*)

2.1.3 Derived connectives

definition

false :: $o \rightarrow \perp$ **where**
 $\perp \equiv \forall A. A$

definition

true :: $o \rightarrow \top$ **where**
 $\top \equiv \perp \longrightarrow \perp$

definition

not :: $o \Rightarrow o \rightarrow \perp$ (\neg - [40] 40) **where**
 $\text{not} \equiv \lambda A. A \longrightarrow \perp$

definition

conj :: $o \Rightarrow o \Rightarrow o$ (**infixr** \wedge 35) **where**
 $\text{conj} \equiv \lambda A B. \forall C. (A \longrightarrow B \longrightarrow C) \longrightarrow C$

definition

disj :: $o \Rightarrow o \Rightarrow o$ (**infixr** \vee 30) **where**
 $\text{disj} \equiv \lambda A B. \forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$

definition

Ex :: $(\lambda a. a \Rightarrow o) \Rightarrow o$ (**binder** \exists 10) **where**
 $\exists x. P x \equiv \forall C. (\forall x. P x \longrightarrow C) \longrightarrow C$

abbreviation

not-equal :: $\lambda a. \lambda b. a \Rightarrow b \Rightarrow o$ (**infixl** \neq 50) **where**
 $x \neq y \equiv \neg (x = y)$

theorem *falseE* [*elim*]: $\perp \implies A$

proof (*unfold false-def*)

assume $\forall A. A$

then show A ..

qed

theorem *trueI* [*intro*]: \top

proof (*unfold true-def*)

show $\perp \longrightarrow \perp$..

qed

theorem *notI* [*intro*]: $(A \implies \perp) \implies \neg A$

```

proof (unfold not-def)
  assume  $A \implies \perp$ 
  then show  $A \longrightarrow \perp$  ..
qed

```

```

theorem notE [elim]:  $\neg A \implies A \implies B$ 
proof (unfold not-def)
  assume  $A \longrightarrow \perp$ 
  also assume  $A$ 
  finally have  $\perp$  ..
  then show  $B$  ..
qed

```

```

lemma notE':  $A \implies \neg A \implies B$ 
  by (rule notE)

```

lemmas *contradiction* = *notE notE'* — proof by contradiction in any order

```

theorem conjI [intro]:  $A \implies B \implies A \wedge B$ 
proof (unfold conj-def)
  assume  $A$  and  $B$ 
  show  $\forall C. (A \longrightarrow B \longrightarrow C) \longrightarrow C$ 
  proof
    fix  $C$  show  $(A \longrightarrow B \longrightarrow C) \longrightarrow C$ 
    proof
      assume  $A \longrightarrow B \longrightarrow C$ 
      also note  $\langle A \rangle$ 
      also note  $\langle B \rangle$ 
      finally show  $C$  .
    qed
  qed
qed

```

```

theorem conjE [elim]:  $A \wedge B \implies (A \implies B \implies C) \implies C$ 
proof (unfold conj-def)
  assume  $c$ :  $\forall C. (A \longrightarrow B \longrightarrow C) \longrightarrow C$ 
  assume  $A \implies B \implies C$ 
  moreover {
    from  $c$  have  $(A \longrightarrow B \longrightarrow A) \longrightarrow A$  ..
    also have  $A \longrightarrow B \longrightarrow A$ 
    proof
      assume  $A$ 
      then show  $B \longrightarrow A$  ..
    qed
    finally have  $A$  .
  } moreover {
    from  $c$  have  $(A \longrightarrow B \longrightarrow B) \longrightarrow B$  ..
    also have  $A \longrightarrow B \longrightarrow B$ 
    proof

```

```

    show  $B \longrightarrow B$  ..
  qed
  finally have  $B$  .
} ultimately show  $C$  .
qed

```

```

theorem disjI1 [intro]:  $A \Longrightarrow A \vee B$ 
proof (unfold disj-def)
  assume  $A$ 
  show  $\forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$ 
  proof
    fix  $C$  show  $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$ 
    proof
      assume  $A \longrightarrow C$ 
      also note  $\langle A \rangle$ 
      finally have  $C$  .
      then show  $(B \longrightarrow C) \longrightarrow C$  ..
    qed
  qed
qed

```

```

theorem disjI2 [intro]:  $B \Longrightarrow A \vee B$ 
proof (unfold disj-def)
  assume  $B$ 
  show  $\forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$ 
  proof
    fix  $C$  show  $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$ 
    proof
      show  $(B \longrightarrow C) \longrightarrow C$ 
      proof
        assume  $B \longrightarrow C$ 
        also note  $\langle B \rangle$ 
        finally show  $C$  .
      qed
    qed
  qed
qed

```

```

theorem disjE [elim]:  $A \vee B \Longrightarrow (A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C$ 
proof (unfold disj-def)
  assume  $c$ :  $\forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$ 
  assume  $r1$ :  $A \Longrightarrow C$  and  $r2$ :  $B \Longrightarrow C$ 
  from  $c$  have  $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$  ..
  also have  $A \longrightarrow C$ 
  proof
    assume  $A$  then show  $C$  by (rule r1)
  qed
  also have  $B \longrightarrow C$ 
  proof

```

```

    assume  $B$  then show  $C$  by (rule  $r2$ )
  qed
  finally show  $C$  .
qed

```

```

theorem  $exI$  [intro]:  $P\ a \implies \exists x. P\ x$ 
proof (unfold  $Ex-def$ )
  assume  $P\ a$ 
  show  $\forall C. (\forall x. P\ x \longrightarrow C) \longrightarrow C$ 
proof
    fix  $C$  show  $(\forall x. P\ x \longrightarrow C) \longrightarrow C$ 
    proof
      assume  $\forall x. P\ x \longrightarrow C$ 
      then have  $P\ a \longrightarrow C$  ..
      also note  $\langle P\ a \rangle$ 
      finally show  $C$  .
    qed
  qed
qed

```

```

theorem  $exE$  [elim]:  $\exists x. P\ x \implies (\bigwedge x. P\ x \implies C) \implies C$ 
proof (unfold  $Ex-def$ )
  assume  $c$ :  $\forall C. (\forall x. P\ x \longrightarrow C) \longrightarrow C$ 
  assume  $r$ :  $\bigwedge x. P\ x \implies C$ 
  from  $c$  have  $(\forall x. P\ x \longrightarrow C) \longrightarrow C$  ..
  also have  $\forall x. P\ x \longrightarrow C$ 
proof
    fix  $x$  show  $P\ x \longrightarrow C$ 
    proof
      assume  $P\ x$ 
      then show  $C$  by (rule  $r$ )
    qed
  qed
  finally show  $C$  .
qed

```

2.2 Classical logic

```

locale  $classical$  =
  assumes  $classical$ :  $(\neg A \implies A) \implies A$ 

```

```

theorem (in  $classical$ )
  Peirce's-Law:  $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$ 
proof
  assume  $a$ :  $(A \longrightarrow B) \longrightarrow A$ 
  show  $A$ 
proof (rule  $classical$ )
    assume  $\neg A$ 
    have  $A \longrightarrow B$ 

```



```

    proof
      assume A
      with  $\langle \neg A \rangle$  show B by (rule contradiction)
    qed
  with a show A ..
qed
qed

theorem (in classical)
  double-negation:  $\neg \neg A \implies A$ 
proof -
  assume  $\neg \neg A$ 
  show A
  proof (rule classical)
    assume  $\neg A$ 
    with  $\langle \neg \neg A \rangle$  show ?thesis by (rule contradiction)
  qed
qed

theorem (in classical)
  tertium-non-datur:  $A \vee \neg A$ 
proof (rule double-negation)
  show  $\neg \neg (A \vee \neg A)$ 
  proof
    assume  $\neg (A \vee \neg A)$ 
    have  $\neg A$ 
    proof
      assume A then have  $A \vee \neg A$  ..
      with  $\langle \neg (A \vee \neg A) \rangle$  show  $\perp$  by (rule contradiction)
    qed
    then have  $A \vee \neg A$  ..
    with  $\langle \neg (A \vee \neg A) \rangle$  show  $\perp$  by (rule contradiction)
  qed
qed

theorem (in classical)
  classical-cases:  $(A \implies C) \implies (\neg A \implies C) \implies C$ 
proof -
  assume r1:  $A \implies C$  and r2:  $\neg A \implies C$ 
  from tertium-non-datur show C
  proof
    assume A
    then show ?thesis by (rule r1)
  next
    assume  $\neg A$ 
    then show ?thesis by (rule r2)
  qed
qed

```

```

lemma (in classical)  $(\neg A \implies A) \implies A$ 
proof –
  assume  $r: \neg A \implies A$ 
  show  $A$ 
  proof (rule classical-cases)
    assume  $A$  then show  $A$  .
  next
    assume  $\neg A$  then show  $A$  by (rule r)
  qed
qed
end

```

3 Abstract Natural Numbers primitive recursion

```

theory Abstract-NAT
imports Main
begin

```

Axiomatic Natural Numbers (Peano) – a monomorphic theory.

```

locale NAT =
  fixes  $zero :: 'n$ 
  and  $succ :: 'n \Rightarrow 'n$ 
  assumes succ-inject [simp]:  $(succ\ m = succ\ n) = (m = n)$ 
  and succ-neq-zero [simp]:  $succ\ m \neq zero$ 
  and induct [case-names zero succ, induct type: 'n]:
     $P\ zero \implies (\bigwedge n. P\ n \implies P\ (succ\ n)) \implies P\ n$ 
begin

```

```

lemma zero-neq-succ [simp]:  $zero \neq succ\ m$ 
  by (rule succ-neq-zero [symmetric])

```

Primitive recursion as a (functional) relation – polymorphic!

```

inductive
   $Rec :: 'a \Rightarrow ('n \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'n \Rightarrow 'a \Rightarrow bool$ 
  for  $e :: 'a$  and  $r :: 'n \Rightarrow 'a \Rightarrow 'a$ 
where
     $Rec\ zero: Rec\ e\ r\ zero\ e$ 
     $| Rec\ succ: Rec\ e\ r\ m\ n \implies Rec\ e\ r\ (succ\ m)\ (r\ m\ n)$ 

```

```

lemma Rec-functional:
  fixes  $x :: 'n$ 
  shows  $\exists! y :: 'a. Rec\ e\ r\ x\ y$ 
proof –
  let  $?R = Rec\ e\ r$ 
  show ?thesis
  proof (induct x)

```

```

case zero
show  $\exists!y. ?R \text{ zero } y$ 
proof
  show  $?R \text{ zero } e \dots$ 
  fix  $y$  assume  $?R \text{ zero } y$ 
  then show  $y = e$  by cases simp-all
qed
next
case (succ m)
from  $\langle \exists!y. ?R \text{ m } y \rangle$ 
obtain  $y$  where  $y: ?R \text{ m } y$ 
  and  $yy': \bigwedge y'. ?R \text{ m } y' \implies y = y'$  by blast
show  $\exists!z. ?R (\text{succ } m) z$ 
proof
  from  $y$  show  $?R (\text{succ } m) (r \text{ m } y) \dots$ 
  fix  $z$  assume  $?R (\text{succ } m) z$ 
  then obtain  $u$  where  $z = r \text{ m } u$  and  $?R \text{ m } u$  by cases simp-all
  with  $yy'$  show  $z = r \text{ m } y$  by (simp only:)
qed
qed
qed

```

The recursion operator – polymorphic!

definition

```

rec :: 'a  $\Rightarrow$  ('n  $\Rightarrow$  'a  $\Rightarrow$  'a)  $\Rightarrow$  'n  $\Rightarrow$  'a where
rec e r x = (THE  $y. \text{Rec } e \text{ r } x \text{ y}$ )

```

lemma *rec-eval*:

```

assumes Rec: Rec e r x y
shows rec e r x =  $y$ 
unfolding rec-def
using Rec-functional and Rec by (rule the1-equality)

```

lemma *rec-zero* [*simp*]: *rec e r zero* = e

proof (*rule rec-eval*)

```

  show Rec e r zero e ..

```

qed

lemma *rec-succ* [*simp*]: *rec e r (succ m)* = $r \text{ m } (\text{rec } e \text{ r } m)$

proof (*rule rec-eval*)

```

  let  $?R = \text{Rec } e \text{ r}$ 
  have  $?R \text{ m } (\text{rec } e \text{ r } m)$ 
    unfolding rec-def using Rec-functional by (rule theI')
  then show  $?R (\text{succ } m) (r \text{ m } (\text{rec } e \text{ r } m)) \dots$ 

```

qed

Example: addition (monomorphic)

definition

```

add :: 'n ⇒ 'n ⇒ 'n where
add m n = rec n (λ- k. succ k) m

lemma add-zero [simp]: add zero n = n
and add-succ [simp]: add (succ m) n = succ (add m n)
unfolding add-def by simp-all

lemma add-assoc: add (add k m) n = add k (add m n)
by (induct k) simp-all

lemma add-zero-right: add m zero = m
by (induct m) simp-all

lemma add-succ-right: add m (succ n) = succ (add m n)
by (induct m) simp-all

lemma add (succ (succ (succ zero))) (succ (succ zero)) =
  succ (succ (succ (succ (succ zero))))
by simp

Example: replication (polymorphic)

definition
  repl :: 'n ⇒ 'a ⇒ 'a list where
  repl n x = rec [] (λ- xs. x # xs) n

lemma repl-zero [simp]: repl zero x = []
and repl-succ [simp]: repl (succ n) x = x # repl n x
unfolding repl-def by simp-all

lemma repl (succ (succ (succ zero))) True = [True, True, True]
by simp

end

```

Just see that our abstract specification makes sense ...

```

interpretation NAT 0 Suc
proof (rule NAT.intro)
  fix m n
  show (Suc m = Suc n) = (m = n) by simp
  show Suc m ≠ 0 by simp
  fix P
  assume zero: P 0
  and succ: ∧n. P n ⇒ P (Suc n)
  show P n
  proof (induct n)
    case 0 show ?case by (rule zero)
  next
    case Suc then show ?case by (rule succ)

```

```

    qed
  qed
end

```

4 Proof by guessing

```

theory Guess
imports Main
begin

lemma True
proof

  have 1:  $\exists x. x = x$  by simp

  from 1 guess ..
  from 1 guess x ..
  from 1 guess x :: 'a ..
  from 1 guess x :: nat ..

  have 2:  $\exists x y. x = x \ \& \ y = y$  by simp
  from 2 guess apply - apply (erule exE conjE) + done
  from 2 guess x apply - apply (erule exE conjE) + done
  from 2 guess x y apply - apply (erule exE conjE) + done
  from 2 guess x :: 'a and y :: 'b apply - apply (erule exE conjE) + done
  from 2 guess x y :: nat apply - apply (erule exE conjE) + done

qed

end

```

5 Simple and efficient binary numerals

```

theory Binary
imports Main
begin

```

5.1 Binary representation of natural numbers

```

definition
  bit :: nat  $\Rightarrow$  bool  $\Rightarrow$  nat where
    bit n b = (if b then 2 * n + 1 else 2 * n)

```

```

lemma bit-simps:
  bit n False = 2 * n

```

$bit\ n\ True = 2 * n + 1$
unfolding *bit-def* **by** *simp-all*

ML $\langle\langle$
 $fun\ dest-bit\ (Const\ (@\{const-name\ False\},\ -)) = 0$
 $\quad | dest-bit\ (Const\ (@\{const-name\ True\},\ -)) = 1$
 $\quad | dest-bit\ t = raise\ TERM\ (dest-bit,\ [t]);$

 $fun\ dest-binary\ (Const\ (@\{const-name\ Groups.zero\},\ Type\ (@\{type-name\ nat\},$
 $-))) = 0$
 $\quad | dest-binary\ (Const\ (@\{const-name\ Groups.one\},\ Type\ (@\{type-name\ nat\},$
 $-))) = 1$
 $\quad | dest-binary\ (Const\ (@\{const-name\ bit\},\ -)\ \$\ bs\ \$\ b) = 2 * dest-binary\ bs +$
 $dest-bit\ b$
 $\quad | dest-binary\ t = raise\ TERM\ (dest-binary,\ [t]);$

 $fun\ mk-bit\ 0 = @\{term\ False\}$
 $\quad | mk-bit\ 1 = @\{term\ True\}$
 $\quad | mk-bit\ - = raise\ TERM\ (mk-bit,\ []);$

 $fun\ mk-binary\ 0 = @\{term\ 0::nat\}$
 $\quad | mk-binary\ 1 = @\{term\ 1::nat\}$
 $\quad | mk-binary\ n =$
 $\quad\quad if\ n < 0\ then\ raise\ TERM\ (mk-binary,\ [])$
 $\quad\quad else$
 $\quad\quad\quad let\ val\ (q,\ r) = Integer.div-mod\ n\ 2$
 $\quad\quad\quad in\ @\{term\ bit\}\ \$\ mk-binary\ q\ \$\ mk-bit\ r\ end;$
 $\rangle\rangle$

5.2 Direct operations – plain normalization

lemma *binary-norm*:
 $bit\ 0\ False = 0$
 $bit\ 0\ True = 1$
unfolding *bit-def* **by** *simp-all*

lemma *binary-add*:
 $n + 0 = n$
 $0 + n = n$
 $1 + 1 = bit\ 1\ False$
 $bit\ n\ False + 1 = bit\ n\ True$
 $bit\ n\ True + 1 = bit\ (n + 1)\ False$
 $1 + bit\ n\ False = bit\ n\ True$
 $1 + bit\ n\ True = bit\ (n + 1)\ False$
 $bit\ m\ False + bit\ n\ False = bit\ (m + n)\ False$
 $bit\ m\ False + bit\ n\ True = bit\ (m + n)\ True$
 $bit\ m\ True + bit\ n\ False = bit\ (m + n)\ True$
 $bit\ m\ True + bit\ n\ True = bit\ ((m + n) + 1)\ False$
by (*simp-all add: bit-simps*)

lemma *binary-mult*:

```

   $n * 0 = 0$ 
   $0 * n = 0$ 
   $n * 1 = n$ 
   $1 * n = n$ 
   $\text{bit } m \text{ True} * n = \text{bit } (m * n) \text{ False} + n$ 
   $\text{bit } m \text{ False} * n = \text{bit } (m * n) \text{ False}$ 
   $n * \text{bit } m \text{ True} = \text{bit } (m * n) \text{ False} + n$ 
   $n * \text{bit } m \text{ False} = \text{bit } (m * n) \text{ False}$ 
by (simp-all add: bit-simps)

```

lemmas *binary-simps* = *binary-norm binary-add binary-mult*

5.3 Indirect operations – ML will produce witnesses

lemma *binary-less-eq*:

```

fixes  $n :: \text{nat}$ 
shows  $n \equiv m + k \implies (m \leq n) \equiv \text{True}$ 
  and  $m \equiv n + k + 1 \implies (m \leq n) \equiv \text{False}$ 
by simp-all

```

lemma *binary-less*:

```

fixes  $n :: \text{nat}$ 
shows  $m \equiv n + k \implies (m < n) \equiv \text{False}$ 
  and  $n \equiv m + k + 1 \implies (m < n) \equiv \text{True}$ 
by simp-all

```

lemma *binary-diff*:

```

fixes  $n :: \text{nat}$ 
shows  $m \equiv n + k \implies m - n \equiv k$ 
  and  $n \equiv m + k \implies m - n \equiv 0$ 
by simp-all

```

lemma *binary-divmod*:

```

fixes  $n :: \text{nat}$ 
assumes  $m \equiv n * k + l$  and  $0 < n$  and  $l < n$ 
shows  $m \text{ div } n \equiv k$ 
  and  $m \text{ mod } n \equiv l$ 

```

proof –

```

from  $\langle m \equiv n * k + l \rangle$  have  $m = l + k * n$  by simp
with  $\langle 0 < n \rangle$  and  $\langle l < n \rangle$  show  $m \text{ div } n \equiv k$  and  $m \text{ mod } n \equiv l$  by simp-all

```

qed

ML \ll

local

```

  infix ==;
  val op == = Logic.mk-equals;
  fun plus  $m \ n = @\{\text{term plus} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}\} \$ m \$ n;$ 

```

```

fun mult m n = @{term times :: nat ⇒ nat ⇒ nat} $ m $ n;

val binary-ss = HOL-basic-ss addsimps @{thms binary-simps};
fun prove ctxt prop =
  Goal.prove ctxt [] [] prop (fn - => ALLGOALS (full-simp-tac binary-ss));

fun binary-proc proc ss ct =
  (case Thm.term-of ct of
   - $ t $ u =>
     (case try (pairstelf ('dest-binary)) (t, u) of
      SOME args => proc (Simplifier.the-context ss) args
      | NONE => NONE)
   | - => NONE);
in

val less-eq-proc = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
  let val k = n - m in
    if k >= 0 then
      SOME (@{thm binary-less-eq(1)} OF [prove ctxt (u == plus t (mk-binary k))])
    else
      SOME (@{thm binary-less-eq(2)} OF
        [prove ctxt (t == plus (plus u (mk-binary (~ k - 1))) (mk-binary 1))])
  end);

val less-proc = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
  let val k = m - n in
    if k >= 0 then
      SOME (@{thm binary-less(1)} OF [prove ctxt (t == plus u (mk-binary k))])
    else
      SOME (@{thm binary-less(2)} OF
        [prove ctxt (u == plus (plus t (mk-binary (~ k - 1))) (mk-binary 1))])
  end);

val diff-proc = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
  let val k = m - n in
    if k >= 0 then
      SOME (@{thm binary-diff(1)} OF [prove ctxt (t == plus u (mk-binary k))])
    else
      SOME (@{thm binary-diff(2)} OF [prove ctxt (u == plus t (mk-binary (~ k)))]))
  end);

fun divmod-proc rule = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
  if n = 0 then NONE
  else
    let val (k, l) = Integer.div-mod m n
    in SOME (rule OF [prove ctxt (t == plus (mult u (mk-binary k)) (mk-binary l))]) end);

```



```

end;
>>

simproc-setup binary-nat-less-eq (m <= (n::nat)) = << K less-eq-proc >>
simproc-setup binary-nat-less (m < (n::nat)) = << K less-proc >>
simproc-setup binary-nat-diff (m - (n::nat)) = << K diff-proc >>
simproc-setup binary-nat-div (m div (n::nat)) = << K (divmod-proc @{thm binary-divmod(1)}) >>
simproc-setup binary-nat-mod (m mod (n::nat)) = << K (divmod-proc @{thm
binary-divmod(2)}) >>

method-setup binary-simp = <<
  Scan.succeed (K (SIMPLE-METHOD'
    (full-simp-tac
      (HOL-basic-ss
        addsimps @{thms binary-simps}
        addsimprocs
          [@{simproc binary-nat-less-eq},
           @{simproc binary-nat-less},
           @{simproc binary-nat-diff},
           @{simproc binary-nat-div},
           @{simproc binary-nat-mod}])))
  >> binary simplification

```

5.4 Concrete syntax

syntax

-Binary :: num-const \Rightarrow 'a (\$-)

parse-translation <<

let

val syntax-consts =
 map-aterms (fn Const (c, T) => Const (Syntax.mark-const c, T) | a => a);

fun binary-tr [Const (num, -)] =

let

val {leading-zeros = z, value = n, ...} = Syntax.read-xnum num;

val - = z = 0 andalso n >= 0 orelse error (Bad binary number: ^ num);

in syntax-consts (mk-binary n) end

| binary-tr ts = raise TERM (binary-tr, ts);

in [(@{syntax-const -Binary}, binary-tr)] end

>>

5.5 Examples

lemma \$6 = 6

by (simp add: bit-simps)

```

lemma bit (bit (bit 0 False) False) True = 1
  by (simp add: bit-simps)

lemma bit (bit (bit 0 False) False) True = bit 0 True
  by (simp add: bit-simps)

lemma $5 + $3 = $8
  by binary-simp

lemma $5 * $3 = $15
  by binary-simp

lemma $5 - $3 = $2
  by binary-simp

lemma $3 - $5 = 0
  by binary-simp

lemma $123456789 - $123 = $123456666
  by binary-simp

lemma $1111111112222222222333333333334444444444 - $998877665544332211
=
  $111111111222222222232334455668900112233
  by binary-simp

lemma (1111111112222222222333333333334444444444::nat) - 998877665544332211
=
  111111111222222222232334455668900112233
  by simp

lemma (1111111112222222222333333333334444444444::int) - 998877665544332211
=
  111111111222222222232334455668900112233
  by simp

lemma $1111111112222222222333333333334444444444 * $998877665544332211
=
  $1109864072938022197293802219729380221972383090160869185684
  by binary-simp

lemma $1111111112222222222333333333334444444444 * $998877665544332211
=
  $5555555555666666666677777777778888888888 =
  $1109864072938022191738246664062713555294605312381980296796
  by binary-simp

lemma $42 < $4 = False
  by binary-simp

```

```

lemma $4 < $42 = True
  by binary-simp

lemma $42 <= $4 = False
  by binary-simp

lemma $4 <= $42 = True
  by binary-simp

lemma $11111111112222222222333333333334444444444 < $998877665544332211
  = False
  by binary-simp

lemma $998877665544332211 < $11111111112222222222333333333334444444444
  = True
  by binary-simp

lemma $11111111112222222222333333333334444444444 <= $998877665544332211
  = False
  by binary-simp

lemma $998877665544332211 <= $11111111112222222222333333333334444444444
  = True
  by binary-simp

lemma $1234 div $23 = $53
  by binary-simp

lemma $1234 mod $23 = $15
  by binary-simp

lemma $11111111112222222222333333333334444444444 div $998877665544332211
  =
    $1112359550673033707875
  by binary-simp

lemma $11111111112222222222333333333334444444444 mod $998877665544332211
  =
    $42245174317582819
  by binary-simp

lemma ((11111111112222222222333333333334444444444)::int) div 998877665544332211
  =
    1112359550673033707875
  by simp — legacy numerals: 30 times slower

lemma ((11111111112222222222333333333334444444444)::int) mod 998877665544332211
  =

```

```

42245174317582819
by simp — legacy numerals: 30 times slower

end

```

6 Examples of recdef definitions

theory *Recdefs* **imports** *Main* **begin**

```

consts fact :: nat => nat
recdef fact less-than
  fact x = (if x = 0 then 1 else x * fact (x - 1))

```

```

consts Fact :: nat => nat
recdef Fact less-than
  Fact 0 = 1
  Fact (Suc x) = Fact x * Suc x

```

```

consts fib :: int => int
recdef fib measure nat
  eqn: fib n = (if n < 1 then 0
                else if n=1 then 1
                else fib(n - 2) + fib(n - 1))

```

```

lemma fib 7 = 13
by simp

```

```

consts map2 :: ('a => 'b => 'c) * 'a list * 'b list => 'c list
recdef map2 measure (λ(f, l1, l2). size l1)
  map2 (f, [], []) = []
  map2 (f, h # t, []) = []
  map2 (f, h1 # t1, h2 # t2) = f h1 h2 # map2 (f, t1, t2)

```

```

consts finiteRchain :: ('a => 'a => bool) * 'a list => bool
recdef finiteRchain measure (λ(R, l). size l)
  finiteRchain(R, []) = True
  finiteRchain(R, [x]) = True
  finiteRchain(R, x # y # rst) = (R x y ∧ finiteRchain (R, y # rst))

```

Not handled automatically: too complicated.

```

consts variant :: nat * nat list => nat
recdef (permissive) variant measure (λ(n,ns). size (filter (λy. n ≤ y) ns))
  variant (x, L) = (if x mem L then variant (Suc x, L) else x)

```

```

consts gcd :: nat * nat => nat
recdef gcd measure (λ(x, y). x + y)
  gcd (0, y) = y

```

```

gcd (Suc x, 0) = Suc x
gcd (Suc x, Suc y) =
  (if y ≤ x then gcd (x - y, Suc y) else gcd (Suc x, y - x))

```

The silly g function: example of nested recursion. Not handled automatically. In fact, g is the zero constant function.

```

consts g :: nat => nat
recdef (permissive) g less-than
  g 0 = 0
  g (Suc x) = g (g x)

```

```

lemma g-terminates: g x < Suc x
apply (induct x rule: g.induct)
apply (auto simp add: g.simps)
done

```

```

lemma g-zero: g x = 0
apply (induct x rule: g.induct)
apply (simp-all add: g.simps g-terminates)
done

```

```

consts Div :: nat * nat => nat * nat
recdef Div measure fst
  Div (0, x) = (0, 0)
  Div (Suc x, y) =
    (let (q, r) = Div (x, y)
     in if y ≤ Suc r then (Suc q, 0) else (q, Suc r))

```

Not handled automatically. Should be the predecessor function, but there is an unnecessary "looping" recursive call in $k\ 1$.

```

consts k :: nat => nat

recdef (permissive) k less-than
  k 0 = 0
  k (Suc n) =
    (let x = k 1
     in if False then k (Suc 1) else n)

```

```

consts part :: ('a => bool) * 'a list * 'a list * 'a list => 'a list * 'a list
recdef part measure (λ(P, l, l1, l2). size l)
  part (P, [], l1, l2) = (l1, l2)
  part (P, h # rst, l1, l2) =
    (if P h then part (P, rst, h # l1, l2)
     else part (P, rst, l1, h # l2))

```

```

consts fqsrt :: ('a => 'a => bool) * 'a list => 'a list
recdef (permissive) fqsrt measure (size o snd)

```

```

fqsrt (ord, []) = []
fqsrt (ord, x # rst) =
  (let (less, more) = part ((λy. ord y x), rst, ([], []))
  in fqsrt (ord, less) @ [x] @ fqsrt (ord, more))

```

Silly example which demonstrates the occasional need for additional congruence rules (here: *map-cong*). If the congruence rule is removed, an unprovable termination condition is generated! Termination not proved automatically. TFL requires $\lambda x. \text{mapf } x$ instead of *mapf*.

```

consts mapf :: nat => nat list
recdef (permissive) mapf measure (λm. m)
  mapf 0 = []
  mapf (Suc n) = concat (map (λx. mapf x) (replicate n n))
  (hints cong: map-cong)

recdef-tc mapf-tc: mapf
  apply (rule allI)
  apply (case-tac n = 0)
  apply simp-all
done

```

Removing the termination condition from the generated thms:

```

lemma mapf (Suc n) = concat (map mapf (replicate n n))
  apply (simp add: mapf.simps mapf-tc)
done

```

```

lemmas mapf-induct = mapf.induct [OF mapf-tc]

```

```

end

```

7 Examples of function definitions

```

theory Fundefs
imports Main
begin

```

7.1 Very basic

```

fun fib :: nat => nat
where
  fib 0 = 1
  | fib (Suc 0) = 1
  | fib (Suc (Suc n)) = fib n + fib (Suc n)

```

partial simp and induction rules:

```

thm fib.psimps

```

thm *fib.pinduct*

There is also a cases rule to distinguish cases along the definition

thm *fib.cases*

total simp and induction rules:

thm *fib.simps*

thm *fib.induct*

7.2 Currying

fun *add*

where

add 0 *y* = *y*
| *add* (*Suc* *x*) *y* = *Suc* (*add* *x* *y*)

thm *add.simps*

thm *add.induct* — Note the curried induction predicate

7.3 Nested recursion

function *nz*

where

nz 0 = 0
| *nz* (*Suc* *x*) = *nz* (*nz* *x*)
by *pat-completeness auto*

lemma *nz-is-zero*: — A lemma we need to prove termination

assumes *trm*: *nz-dom* *x*

shows *nz* *x* = 0

using *trm*

by *induct auto*

termination *nz*

by (*relation less-than*) (*auto simp:nz-is-zero*)

thm *nz.simps*

thm *nz.induct*

Here comes McCarthy's 91-function

function *f91* :: *nat* => *nat*

where

f91 *n* = (*if* 100 < *n* *then* *n* - 10 *else* *f91* (*f91* (*n* + 11)))
by *pat-completeness auto*

lemma *f91-estimate*:

assumes *trm*: *f91-dom* *n*

shows *n* < *f91* *n* + 11

```

termination
proof
  let  $?R = \text{measure } (\%x. 101 - x)$ 
  show  $wf ?R ..$ 

  fix  $n::nat$  assume  $\sim 100 < n$ 
  thus  $(n + 11, n) : ?R$  by simp

  assume inner-trm:  $f91\text{-dom } (n + 11)$ 
  with  $f91\text{-estimate}$  have  $n + 11 < f91 (n + 11) + 11 ..$ 
  with  $(\sim 100 < n)$  show  $(f91 (n + 11), n) : ?R$  by simp
qed

```

```
lemma f91 n = (if 100 < n then n - 10 else 91)
by (induct n rule:f91.induct) auto
```

7.4.1 Overlapping patterns

```

fun gcd2 :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
where
  gcd2 x 0 = x
| gcd2 0 y = y
| gcd2 (Suc x) (Suc y) = (if x < y then gcd2 (Suc x) (y - x)
                           else gcd2 (x - y) (Suc y))

```

7.4.2 Guards

```

function gcd3 :: nat => nat => nat
where
  gcd3 x 0 = x
| gcd3 0 y = y
| x < y ==> gcd3 (Suc x) (Suc y) = gcd3 (Suc x) (y - x)
| ~ x < y ==> gcd3 (Suc x) (Suc y) = gcd3 (x - y) (Suc y)
apply (case-tac x, case-tac a, auto)
apply (case-tac ba, auto)
done

```


termination by *lexicographic-order*

thm *gcd3.simps*
thm *gcd3.induct*

General patterns allow even strange definitions:

```
function ev :: nat  $\Rightarrow$  bool
where
  ev (2 * n) = True
| ev (2 * n + 1) = False
proof — — completeness is more difficult here ...
  fix P :: bool
    and x :: nat
  assume c1:  $\bigwedge n. x = 2 * n \implies P$ 
    and c2:  $\bigwedge n. x = 2 * n + 1 \implies P$ 
  have divmod:  $x = 2 * (x \text{ div } 2) + (x \text{ mod } 2)$  by auto
  show P
proof cases
  assume  $x \text{ mod } 2 = 0$ 
  with divmod have  $x = 2 * (x \text{ div } 2)$  by simp
  with c1 show P .
next
  assume  $x \text{ mod } 2 \neq 0$ 
  hence  $x \text{ mod } 2 = 1$  by simp
  with divmod have  $x = 2 * (x \text{ div } 2) + 1$  by simp
  with c2 show P .
qed
qed presburger+ — solve compatibility with presburger
termination by lexicographic-order
```

thm *ev.simps*
thm *ev.induct*
thm *ev.cases*

7.5 Mutual Recursion

```
fun evn od :: nat  $\Rightarrow$  bool
where
  evn 0 = True
| od 0 = False
| evn (Suc n) = od n
| od (Suc n) = evn n
```

thm *evn.simps*
thm *od.simps*

thm *evn-od.induct*
thm *evn-od.termination*

7.6 Definitions in local contexts

```
locale my-monoid =
fixes opr :: 'a ⇒ 'a ⇒ 'a
  and un :: 'a
assumes assoc: opr (opr x y) z = opr x (opr y z)
  and lunit: opr un x = x
  and runit: opr x un = x
begin

fun foldR :: 'a list ⇒ 'a
where
  foldR [] = un
| foldR (x#xs) = opr x (foldR xs)

fun foldL :: 'a list ⇒ 'a
where
  foldL [] = un
| foldL [x] = x
| foldL (x#y#ys) = foldL (opr x y # ys)

thm foldL.simps

lemma foldR-foldL: foldR xs = foldL xs
by (induct xs rule: foldL.induct) (auto simp:lunit runit assoc)

thm foldR-foldL

end

thm my-monoid.foldL.simps
thm my-monoid.foldR-foldL
```

7.7 Regression tests

The following examples mainly serve as tests for the function package

```
fun listlen :: 'a list ⇒ nat
where
  listlen [] = 0
| listlen (x#xs) = Suc (listlen xs)

fun f :: nat ⇒ nat
where
  zero: f 0 = 0
| succ: f (Suc n) = (if f n = 0 then 0 else f n)
```

```

function  $h :: \text{nat} \Rightarrow \text{nat}$ 
where
   $h\ 0 = 0$ 
|  $h\ (\text{Suc}\ n) = (\text{if}\ h\ n = 0\ \text{then}\ h\ (h\ n)\ \text{else}\ h\ n)$ 
  by pat-completeness auto

```

```

fun  $i :: \text{nat} \Rightarrow \text{nat}$ 
where
   $i\ 0 = 0$ 
|  $i\ (\text{Suc}\ n) = (\text{if}\ n = 0\ \text{then}\ 0\ \text{else}\ i\ n)$ 

```

```

fun  $fa :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ 
where
   $fa\ 0\ y = 0$ 
|  $fa\ (\text{Suc}\ n)\ y = (\text{if}\ fa\ n\ y = 0\ \text{then}\ 0\ \text{else}\ fa\ n\ y)$ 

```

```

fun  $j :: \text{nat} \Rightarrow \text{nat}$ 
where
   $j\ 0 = 0$ 
|  $j\ (\text{Suc}\ n) = (\text{let}\ u = n\ \text{in}\ \text{Suc}\ (j\ u))$ 

```

```

function  $k :: \text{nat} \Rightarrow \text{nat}$ 
where
   $k\ x = (\text{let}\ a = x;\ b = x\ \text{in}\ k\ x)$ 
  by pat-completeness auto

```

```

function  $f2 :: (\text{nat} \times \text{nat}) \Rightarrow (\text{nat} \times \text{nat})$ 
where
   $f2\ p = (\text{let}\ (x,y) = p\ \text{in}\ f2\ (y,x))$ 
  by pat-completeness auto

```

```

fun  $f3 :: 'a\ \text{set} \Rightarrow \text{bool}$ 
where
   $f3\ x = \text{finite}\ x$ 

```

```

datatype 'a tree =
  Leaf 'a
| Branch 'a tree list

fun treemap :: ('a  $\Rightarrow$  'a)  $\Rightarrow$  'a tree  $\Rightarrow$  'a tree
where
  treemap fn (Leaf n) = (Leaf (fn n))
| treemap fn (Branch l) = (Branch (map (treemap fn) l))

fun tinc :: nat tree  $\Rightarrow$  nat tree
where
  tinc (Leaf n) = Leaf (Suc n)
| tinc (Branch l) = Branch (map tinc l)

fun testcase :: 'a tree  $\Rightarrow$  'a list
where
  testcase (Leaf a) = [a]
| testcase (Branch x) =
  (let xs = concat (map testcase x);
   ys = concat (map testcase x) in
   xs @ ys)

record point =
  Xcoord :: int
  Ycoord :: int

function swp :: point  $\Rightarrow$  point
where
  swp ( $\lfloor$  Xcoord = x, Ycoord = y  $\rfloor$ ) = ( $\lfloor$  Xcoord = y, Ycoord = x  $\rfloor$ )
proof -
  fix P x
  assume  $\bigwedge xa y. x = (\lfloor Xcoord = xa, Ycoord = y \rfloor) \Longrightarrow P$ 
  thus P
  by (cases x)
qed auto
termination by rule auto

```

```

fun diag :: bool  $\Rightarrow$  bool  $\Rightarrow$  bool  $\Rightarrow$  nat
where
  diag x True False = 1
| diag False y True = 2
| diag True False z = 3
| diag True True True = 4
| diag False False False = 5

```

```

datatype DT =
  A | B | C | D | E | F | G | H | I | J | K | L | M | N | P
  | Q | R | S | T | U | V

```

```

fun big :: DT  $\Rightarrow$  nat

```

```

where

```

```

  big A = 0
| big B = 0
| big C = 0
| big D = 0
| big E = 0
| big F = 0
| big G = 0
| big H = 0
| big I = 0
| big J = 0
| big K = 0
| big L = 0
| big M = 0
| big N = 0
| big P = 0
| big Q = 0
| big R = 0
| big S = 0
| big T = 0
| big U = 0
| big V = 0

```

```

fun

```

```

  f4 :: nat  $\Rightarrow$  nat  $\Rightarrow$  bool

```

```

where

```

```

  f4 0 0 = True
| f4 - - = False

```

```

end

```

8 Examples of automatically derived induction rules

```

theory Induction-Schema

```

```

imports Main

```

```

begin

```

8.1 Some simple induction principles on nat

lemma *nat-standard-induct*:

$\llbracket P\ 0; \bigwedge n. P\ n \implies P\ (Suc\ n) \rrbracket \implies P\ x$

by *induction-schema* (*pat-completeness*, *lexicographic-order*)

lemma *nat-induct2*:

$\llbracket P\ 0; P\ (Suc\ 0); \bigwedge k. P\ k \implies P\ (Suc\ k) \implies P\ (Suc\ (Suc\ k)) \rrbracket$
 $\implies P\ n$

by *induction-schema* (*pat-completeness*, *lexicographic-order*)

lemma *minus-one-induct*:

$\llbracket \bigwedge n::nat. (n \neq 0 \implies P\ (n - 1)) \implies P\ n \rrbracket \implies P\ x$

by *induction-schema* (*pat-completeness*, *lexicographic-order*)

theorem *diff-induct*:

$(!!x. P\ x\ 0) \implies (!!y. P\ 0\ (Suc\ y)) \implies$
 $(!!x\ y. P\ x\ y \implies P\ (Suc\ x)\ (Suc\ y)) \implies P\ m\ n$

by *induction-schema* (*pat-completeness*, *lexicographic-order*)

lemma *list-induct2'*:

$\llbracket P\ [];$
 $\bigwedge x\ xs. P\ (x\#xs)\ [];$
 $\bigwedge y\ ys. P\ []\ (y\#ys);$
 $\bigwedge x\ xs\ y\ ys. P\ xs\ ys \implies P\ (x\#xs)\ (y\#ys) \rrbracket$
 $\implies P\ xs\ ys$

by *induction-schema* (*pat-completeness*, *lexicographic-order*)

theorem *even-odd-induct*:

assumes $R\ 0$

assumes $Q\ 0$

assumes $\bigwedge n. Q\ n \implies R\ (Suc\ n)$

assumes $\bigwedge n. R\ n \implies Q\ (Suc\ n)$

shows $R\ n\ Q\ n$

using *assms*

by *induction-schema* (*pat-completeness+*, *lexicographic-order*)

end

9 Some of the results in Inductive Invariants for Nested Recursion

theory *InductiveInvariant* **imports** *Main* **begin**

A formalization of some of the results in *Inductive Invariants for Nested Recursion*, by Sava Krstić and John Matthews. Appears in the proceedings of TPHOLs 2003, LNCS vol. 2758, pp. 253-269.

S is an inductive invariant of the functional F with respect to the wellfounded

relation r .

definition

$indinv :: ('a * 'a) \text{ set} \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow \text{bool}$ **where**
 $indinv \ r \ S \ F = (\forall f \ x. (\forall y. (y, x) : r \longrightarrow S \ y \ (f \ y)) \longrightarrow S \ x \ (F \ f \ x))$

S is an inductive invariant of the functional F on set D with respect to the wellfounded relation r .

definition

$indinv\text{-}on :: ('a * 'a) \text{ set} \Rightarrow 'a \text{ set} \Rightarrow ('a \Rightarrow 'b \Rightarrow \text{bool}) \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow \text{bool}$ **where**
 $indinv\text{-}on \ r \ D \ S \ F = (\forall f. \forall x \in D. (\forall y \in D. (y, x) \in r \longrightarrow S \ y \ (f \ y)) \longrightarrow S \ x \ (F \ f \ x))$

The key theorem, corresponding to theorem 1 of the paper. All other results in this theory are proved using instances of this theorem, and theorems derived from this theorem.

theorem *indinv-wfrec*:

assumes wf : $wf \ r$ **and**
 inv : $indinv \ r \ S \ F$
shows $S \ x \ (wfrec \ r \ F \ x)$
using wf
proof (*induct* x)
fix x
assume $IHYP$: $!!y. (y, x) \in r \Longrightarrow S \ y \ (wfrec \ r \ F \ y)$
then have $!!y. (y, x) \in r \Longrightarrow S \ y \ (cut \ (wfrec \ r \ F) \ r \ x \ y)$ **by** (*simp add: tfl-cut-apply*)
with inv **have** $S \ x \ (F \ (cut \ (wfrec \ r \ F) \ r \ x) \ x)$ **by** (*unfold indinv-def, blast*)
thus $S \ x \ (wfrec \ r \ F \ x)$ **using** wf **by** (*simp add: wfrec*)
qed

theorem *indinv-on-wfrec*:

assumes WF : $wf \ r$ **and**
 INV : $indinv\text{-}on \ r \ D \ S \ F$ **and**
 D : $x \in D$
shows $S \ x \ (wfrec \ r \ F \ x)$
apply (*insert INV D indinv-wfrec [OF WF, of % x y. x ∈ D ⟶ S x y]*)
by (*simp add: indinv-on-def indinv-def*)

theorem *ind-fixpoint-on-lemma*:

assumes WF : $wf \ r$ **and**
 INV : $\forall f. \forall x \in D. (\forall y \in D. (y, x) \in r \longrightarrow S \ y \ (wfrec \ r \ F \ y) \ \& \ f \ y = wfrec \ r \ F \ y)$
 $\longrightarrow S \ x \ (wfrec \ r \ F \ x) \ \& \ F \ f \ x = wfrec \ r \ F \ x$ **and**
 D : $x \in D$
shows $F \ (wfrec \ r \ F) \ x = wfrec \ r \ F \ x \ \& \ S \ x \ (wfrec \ r \ F \ x)$
proof (*rule indinv-on-wfrec [OF WF - D, of % a b. F (wfrec r F) a = b & wfrec r F a = b & S a b F, simplified]*)

```

show indinv-on r D (%a b. F (wfrec r F) a = b & wfrec r F a = b & S a b) F
proof (unfold indinv-on-def, clarify)
  fix f x
    assume A1:  $\forall y \in D. (y, x) \in r \dashrightarrow F (wfrec\ r\ F)\ y = f\ y \ \&\ wfrec\ r\ F\ y = f\ y \ \&\ S\ y\ (f\ y)$ 
    assume D':  $x \in D$ 
    from A1 INV [THEN spec, of f, THEN bspec, OF D']
      have S x (wfrec r F x) and
        F f x = wfrec r F x by auto
    moreover
      from A1 have  $\forall y \in D. (y, x) \in r \dashrightarrow S\ y\ (wfrec\ r\ F\ y)$  by auto
    with D' INV [THEN spec, of wfrec r F, simplified]
      have F (wfrec r F) x = wfrec r F x by blast
    ultimately show F (wfrec r F) x = F f x & wfrec r F x = F f x & S x (F f
x) by auto
  qed
qed

theorem ind-fixpoint-lemma:
  assumes WF: wf r and
    INV:  $\forall f\ x. (\forall y. (y, x) \in r \dashrightarrow S\ y\ (wfrec\ r\ F\ y) \ \&\ f\ y = wfrec\ r\ F\ y) \dashrightarrow S\ x\ (wfrec\ r\ F\ x) \ \&\ F\ f\ x = wfrec\ r\ F\ x$ 
  shows F (wfrec r F) x = wfrec r F x & S x (wfrec r F x)
apply (rule ind-fixpoint-on-lemma [OF WF - UNIV-I, simplified])
by (rule INV)

theorem tfl-indinv-wfrec:
  [f == wfrec r F; wf r; indinv r S F ]
  ==> S x (f x)
by (simp add: indinv-wfrec)

theorem tfl-indinv-on-wfrec:
  [f == wfrec r F; wf r; indinv-on r D S F;  $x \in D$  ]
  ==> S x (f x)
by (simp add: indinv-on-wfrec)

end

```

10 Example use if an inductive invariant to solve termination conditions

theory *InductiveInvariant-examples* **imports** *InductiveInvariant* **begin**

A simple example showing how to use an inductive invariant to solve termination conditions generated by *recdef* on nested recursive function definitions.

consts *g* :: *nat* ==> *nat*


```

recdef (permissive) g less-than
  g 0 = 0
  g (Suc n) = g (g n)

```

We can prove the unsolved termination condition for *g* by showing it is an inductive invariant.

```

recdef-tc g-tc[simp]: g
apply (rule allI)
apply (rule-tac x=n in tfl-indinv-wfrec [OF g-def])
apply (auto simp add: indinv-def split: nat.split)
apply (frule-tac x=nat in spec)
apply (drule-tac x=f nat in spec)
by auto

```

This declaration invokes Isabelle’s simplifier to remove any termination conditions before adding *g*’s rules to the simpset.

```

declare g.simps [simplified, simp]

```

This is an example where the termination condition generated by **recdef** is not itself an inductive invariant.

```

consts g' :: nat => nat
recdef (permissive) g' less-than
  g' 0 = 0
  g' (Suc n) = g' n + g' (g' n)

```

```

thm g'.simps

```

The strengthened inductive invariant is as follows (this invariant also works for the first example above):

```

lemma g'-inv: g' n = 0
thm tfl-indinv-wfrec [OF g'-def]
apply (rule-tac x=n in tfl-indinv-wfrec [OF g'-def])
by (auto simp add: indinv-def split: nat.split)

```

```

recdef-tc g'-tc[simp]: g'
by (simp add: g'-inv)

```

Now we can remove the termination condition from the rules for *g'*.

```

thm g'.simps [simplified]

```

Sometimes a recursive definition is partial, that is, it is only meant to be invoked on ”good” inputs. As a contrived example, we will define a new version of *g* that is only well defined for even inputs greater than zero.

```

consts g-even :: nat => nat
recdef (permissive) g-even less-than
  g-even (Suc (Suc 0)) = 3

```

$$g\text{-even } n = g\text{-even } (g\text{-even } (n - 2) - 1)$$

We can prove a conditional version of the unsolved termination condition for *g-even* by proving a stronger inductive invariant.

lemma *g-even-indinv*: $\exists k. n = \text{Suc } (\text{Suc } (2*k)) \implies g\text{-even } n = 3$
apply (*rule-tac* $D = \{n. \exists k. n = \text{Suc } (\text{Suc } (2*k))\}$ **and** $x=n$ **in** *tfl-indinv-on-wfrec* [*OF g-even-def*])
apply (*auto simp add: indinv-on-def split: nat.split*)
by (*case-tac ka, auto*)

Now we can prove that the second recursion equation for *g-even* holds, provided that *n* is an even number greater than two.

theorem *g-even-n*: $\exists k. n = 2*k + 4 \implies g\text{-even } n = g\text{-even } (g\text{-even } (n - 2) - 1)$
apply (*subgoal-tac* $(\exists k. n - 2 = 2*k + 2) \ \& \ (\exists k. n = 2*k + 2)$)
by (*auto simp add: g-even-indinv, arith*)

McCarthy's ninety-one function. This function requires a non-standard measure to prove termination.

consts *ninety-one* :: *nat* => *nat*
recdef (**permissive**) *ninety-one measure* (%*n*. 101 - *n*)
 ninety-one *x* = (*if* 100 < *x*
 then *x* - 10
 else (*ninety-one* (*ninety-one* (*x*+11))))

To discharge the termination condition, we will prove a strengthened inductive invariant: $S \ x \ y == x \dot{+} y + 11$

lemma *ninety-one-inv*: $n < \text{ninety-one } n + 11$
apply (*rule-tac* $x=n$ **in** *tfl-indinv-wfrec* [*OF ninety-one-def*])
apply *force*
apply (*auto simp add: indinv-def*)
apply (*frule-tac* $x=x+11$ **in** *spec*)
apply (*frule-tac* $x=f \ (x + 11)$ **in** *spec*)
by *arith*

Proving the termination condition using the strengthened inductive invariant.

recdef-tc *ninety-one-tc*[*rule-format*]: *ninety-one*
apply *clarify*
by (*cut-tac* $n=x+11$ **in** *ninety-one-inv, arith*)

Now we can remove the termination condition from the simplification rule for *ninety-one*.

theorem *def-ninety-one*:
ninety-one *x* = (*if* 100 < *x*
 then *x* - 10
 else *ninety-one* (*ninety-one* (*x*+11)))

```
by (subst ninety-one.simps,
    simp add: ninety-one-tc)
```

```
end
```

11 Test of Locale Interpretation

```
theory LocaleTest2
imports Main GCD
begin
```

12 Interpretation of Defined Concepts

Naming convention for global objects: prefixes D and d

12.1 Lattices

Much of the lattice proofs are from HOL/Lattice.

12.1.1 Definitions

```
locale dpo =
  fixes le :: ['a, 'a] => bool (infixl  $\sqsubseteq$  50)
  assumes refl [intro, simp]:  $x \sqsubseteq x$ 
    and antisym [intro]:  $[x \sqsubseteq y; y \sqsubseteq x] \implies x = y$ 
    and trans [trans]:  $[x \sqsubseteq y; y \sqsubseteq z] \implies x \sqsubseteq z$ 
```

```
begin
```

```
theorem circular:
   $[x \sqsubseteq y; y \sqsubseteq z; z \sqsubseteq x] \implies x = y \ \& \ y = z$ 
  by (blast intro: trans)
```

```
definition
```

```
less :: ['a, 'a] => bool (infixl  $\sqsubset$  50)
where  $(x \sqsubset y) = (x \sqsubseteq y \ \& \ x \not\sim y)$ 
```

```
theorem abs-test:
```

```
op  $\sqsubset = (\%x \ y. x \sqsubset y)$ 
by simp
```

```
definition
```

```
is-inf :: ['a, 'a, 'a] => bool
where  $is-inf \ x \ y \ i = (i \sqsubseteq x \ \& \ i \sqsubseteq y \ \& \ (\forall z. z \sqsubseteq x \ \& \ z \sqsubseteq y \longrightarrow z \sqsubseteq i))$ 
```

```
definition
```

```

is-sup :: ['a, 'a, 'a] => bool
where is-sup x y s = (x ⊆ s ∧ y ⊆ s ∧ (∀ z. x ⊆ z ∧ y ⊆ z ⟹ s ⊆ z))

end

locale dlat = dpo +
  assumes ex-inf: EX inf. dpo.is-inf le x y inf
  and ex-sup: EX sup. dpo.is-sup le x y sup

begin

definition
  meet :: ['a, 'a] => 'a (infixl ⊓ 70)
  where x ⊓ y = (THE inf. is-inf x y inf)

definition
  join :: ['a, 'a] => 'a (infixl ⊔ 65)
  where x ⊔ y = (THE sup. is-sup x y sup)

lemma is-infI [intro?]: i ⊆ x ⟹ i ⊆ y ⟹
  (⋀ z. z ⊆ x ⟹ z ⊆ y ⟹ z ⊆ i) ⟹ is-inf x y i
  by (unfold is-inf-def) blast

lemma is-inf-lower [elim?]:
  is-inf x y i ⟹ (i ⊆ x ⟹ i ⊆ y ⟹ C) ⟹ C
  by (unfold is-inf-def) blast

lemma is-inf-greatest [elim?]:
  is-inf x y i ⟹ z ⊆ x ⟹ z ⊆ y ⟹ z ⊆ i
  by (unfold is-inf-def) blast

theorem is-inf-uniq: is-inf x y i ⟹ is-inf x y i' ⟹ i = i'
proof -
  assume inf: is-inf x y i
  assume inf': is-inf x y i'
  show ?thesis
  proof (rule antisym)
    from inf' show i ⊆ i'
    proof (rule is-inf-greatest)
      from inf show i ⊆ x ..
      from inf show i ⊆ y ..
    qed
    from inf show i' ⊆ i
    proof (rule is-inf-greatest)
      from inf' show i' ⊆ x ..
      from inf' show i' ⊆ y ..
    qed
  qed
qed
qed

```

theorem *is-inf-related* [*elim?*]: $x \sqsubseteq y \implies \text{is-inf } x \ y \ x$
proof –
 assume $x \sqsubseteq y$
 show *?thesis*
 proof
 show $x \sqsubseteq x$..
 show $x \sqsubseteq y$ **by** *fact*
 fix z **assume** $z \sqsubseteq x$ **and** $z \sqsubseteq y$ **show** $z \sqsubseteq x$ **by** *fact*
 qed
qed

lemma *meet-equality* [*elim?*]: $\text{is-inf } x \ y \ i \implies x \sqcap y = i$
proof (*unfold meet-def*)
 assume $\text{is-inf } x \ y \ i$
 then show (*THE* i . $\text{is-inf } x \ y \ i$) = i
 by (*rule the-equality*) (*rule is-inf-uniq* [*OF* - $\langle \text{is-inf } x \ y \ i \rangle$])
qed

lemma *meetI* [*intro?*]:
 $i \sqsubseteq x \implies i \sqsubseteq y \implies (\bigwedge z. z \sqsubseteq x \implies z \sqsubseteq y \implies z \sqsubseteq i) \implies x \sqcap y = i$
by (*rule meet-equality*, *rule is-infI*) *blast*+

lemma *is-inf-meet* [*intro?*]: $\text{is-inf } x \ y \ (x \sqcap y)$
proof (*unfold meet-def*)
 from *ex-inf* **obtain** i **where** $\text{is-inf } x \ y \ i$..
 then show $\text{is-inf } x \ y \ (THE \ i. \ \text{is-inf } x \ y \ i)$
 by (*rule theI*) (*rule is-inf-uniq* [*OF* - $\langle \text{is-inf } x \ y \ i \rangle$])
qed

lemma *meet-left* [*intro?*]:
 $x \sqcap y \sqsubseteq x$
by (*rule is-inf-lower*) (*rule is-inf-meet*)

lemma *meet-right* [*intro?*]:
 $x \sqcap y \sqsubseteq y$
by (*rule is-inf-lower*) (*rule is-inf-meet*)

lemma *meet-le* [*intro?*]:
 $[| z \sqsubseteq x; z \sqsubseteq y |] \implies z \sqsubseteq x \sqcap y$
by (*rule is-inf-greatest*) (*rule is-inf-meet*)

lemma *is-supI* [*intro?*]: $x \sqsubseteq s \implies y \sqsubseteq s \implies$
 $(\bigwedge z. x \sqsubseteq z \implies y \sqsubseteq z \implies s \sqsubseteq z) \implies \text{is-sup } x \ y \ s$
by (*unfold is-sup-def*) *blast*

lemma *is-sup-least* [*elim?*]:
 $\text{is-sup } x \ y \ s \implies x \sqsubseteq z \implies y \sqsubseteq z \implies s \sqsubseteq z$
by (*unfold is-sup-def*) *blast*

lemma *is-sup-upper* [elim?]:
 $is-sup\ x\ y\ s \implies (x \sqsubseteq s \implies y \sqsubseteq s \implies C) \implies C$
by (*unfold is-sup-def*) *blast*

theorem *is-sup-uniq*: $is-sup\ x\ y\ s \implies is-sup\ x\ y\ s' \implies s = s'$
proof –
assume *sup*: $is-sup\ x\ y\ s$
assume *sup'*: $is-sup\ x\ y\ s'$
show ?thesis
proof (*rule antisym*)
from *sup* **show** $s \sqsubseteq s'$
proof (*rule is-sup-least*)
from *sup'* **show** $x \sqsubseteq s' ..$
from *sup'* **show** $y \sqsubseteq s' ..$
qed
from *sup'* **show** $s' \sqsubseteq s$
proof (*rule is-sup-least*)
from *sup* **show** $x \sqsubseteq s ..$
from *sup* **show** $y \sqsubseteq s ..$
qed
qed
qed

theorem *is-sup-related* [elim?]: $x \sqsubseteq y \implies is-sup\ x\ y\ y$
proof –
assume $x \sqsubseteq y$
show ?thesis
proof
show $x \sqsubseteq y$ **by** *fact*
show $y \sqsubseteq y ..$
fix *z* **assume** $x \sqsubseteq z$ **and** $y \sqsubseteq z$
show $y \sqsubseteq z$ **by** *fact*
qed
qed

lemma *join-equality* [elim?]: $is-sup\ x\ y\ s \implies x \sqcup y = s$
proof (*unfold join-def*)
assume $is-sup\ x\ y\ s$
then show (*THE* *s*. $is-sup\ x\ y\ s$) = *s*
by (*rule the-equality*) (*rule is-sup-uniq* [*OF* - ($is-sup\ x\ y\ s$)])
qed

lemma *joinI* [intro?]: $x \sqsubseteq s \implies y \sqsubseteq s \implies$
 $(\bigwedge z. x \sqsubseteq z \implies y \sqsubseteq z \implies s \sqsubseteq z) \implies x \sqcup y = s$
by (*rule join-equality*, *rule is-supI*) *blast*+

lemma *is-sup-join* [intro?]: $is-sup\ x\ y\ (x \sqcup y)$
proof (*unfold join-def*)

from *ex-sup* **obtain** *s* **where** *is-sup* *x y s* ..
then show *is-sup* *x y* (*THE s. is-sup* *x y s*)
by (*rule theI*) (*rule is-sup-uniq* [*OF* - $\langle is-sup\ x\ y\ s \rangle$])
qed

lemma *join-left* [*intro?*]:
 $x \sqsubseteq x \sqcup y$
by (*rule is-sup-upper*) (*rule is-sup-join*)

lemma *join-right* [*intro?*]:
 $y \sqsubseteq x \sqcup y$
by (*rule is-sup-upper*) (*rule is-sup-join*)

lemma *join-le* [*intro?*]:
 $[x \sqsubseteq z; y \sqsubseteq z] \implies x \sqcup y \sqsubseteq z$
by (*rule is-sup-least*) (*rule is-sup-join*)

theorem *meet-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$

proof (*rule meetI*)
show $x \sqcap (y \sqcap z) \sqsubseteq x \sqcap y$
proof
show $x \sqcap (y \sqcap z) \sqsubseteq x$..
show $x \sqcap (y \sqcap z) \sqsubseteq y$
proof –
have $x \sqcap (y \sqcap z) \sqsubseteq y \sqcap z$..
also have ... $\sqsubseteq y$..
finally show *?thesis* .
qed

qed
show $x \sqcap (y \sqcap z) \sqsubseteq z$
proof –
have $x \sqcap (y \sqcap z) \sqsubseteq y \sqcap z$..
also have ... $\sqsubseteq z$..
finally show *?thesis* .
qed

fix *w* **assume** $w \sqsubseteq x \sqcap y$ **and** $w \sqsubseteq z$
show $w \sqsubseteq x \sqcap (y \sqcap z)$
proof
show $w \sqsubseteq x$
proof –
have $w \sqsubseteq x \sqcap y$ **by** *fact*
also have ... $\sqsubseteq x$..
finally show *?thesis* .
qed

show $w \sqsubseteq y \sqcap z$
proof
show $w \sqsubseteq y$
proof –
have $w \sqsubseteq x \sqcap y$ **by** *fact*

```

    also have ...  $\sqsubseteq$   $y$  ..
    finally show ?thesis .
  qed
  show  $w \sqsubseteq z$  by fact
  qed
  qed
  qed

theorem meet-commute:  $x \sqcap y = y \sqcap x$ 
proof (rule meetI)
  show  $y \sqcap x \sqsubseteq x$  ..
  show  $y \sqcap x \sqsubseteq y$  ..
  fix  $z$  assume  $z \sqsubseteq y$  and  $z \sqsubseteq x$ 
  then show  $z \sqsubseteq y \sqcap x$  ..
qed

theorem meet-join-absorb:  $x \sqcap (x \sqcup y) = x$ 
proof (rule meetI)
  show  $x \sqsubseteq x$  ..
  show  $x \sqsubseteq x \sqcup y$  ..
  fix  $z$  assume  $z \sqsubseteq x$  and  $z \sqsubseteq x \sqcup y$ 
  show  $z \sqsubseteq x$  by fact
qed

theorem join-assoc:  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ 
proof (rule joinI)
  show  $x \sqcup y \sqsubseteq x \sqcup (y \sqcup z)$ 
  proof
    show  $x \sqsubseteq x \sqcup (y \sqcup z)$  ..
    show  $y \sqsubseteq x \sqcup (y \sqcup z)$ 
    proof -
      have  $y \sqsubseteq y \sqcup z$  ..
      also have ...  $\sqsubseteq x \sqcup (y \sqcup z)$  ..
      finally show ?thesis .
    qed
  qed
  show  $z \sqsubseteq x \sqcup (y \sqcup z)$ 
  proof -
    have  $z \sqsubseteq y \sqcup z$  ..
    also have ...  $\sqsubseteq x \sqcup (y \sqcup z)$  ..
    finally show ?thesis .
  qed
  fix  $w$  assume  $x \sqcup y \sqsubseteq w$  and  $z \sqsubseteq w$ 
  show  $x \sqcup (y \sqcup z) \sqsubseteq w$ 
  proof
    show  $x \sqsubseteq w$ 
    proof -
      have  $x \sqsubseteq x \sqcup y$  ..
      also have ...  $\sqsubseteq w$  by fact

```



```

    finally show ?thesis .
  qed
  show  $y \sqcup z \sqsubseteq w$ 
  proof
    show  $y \sqsubseteq w$ 
    proof -
      have  $y \sqsubseteq x \sqcup y$  ..
      also have  $\dots \sqsubseteq w$  by fact
      finally show ?thesis .
    qed
    show  $z \sqsubseteq w$  by fact
  qed
qed
qed

```

```

theorem join-commute:  $x \sqcup y = y \sqcup x$ 
proof (rule joinI)
  show  $x \sqsubseteq y \sqcup x$  ..
  show  $y \sqsubseteq y \sqcup x$  ..
  fix z assume  $y \sqsubseteq z$  and  $x \sqsubseteq z$ 
  then show  $y \sqcup x \sqsubseteq z$  ..
qed

```

```

theorem join-meet-absorb:  $x \sqcup (x \sqcap y) = x$ 
proof (rule joinI)
  show  $x \sqsubseteq x$  ..
  show  $x \sqcap y \sqsubseteq x$  ..
  fix z assume  $x \sqsubseteq z$  and  $x \sqcap y \sqsubseteq z$ 
  show  $x \sqsubseteq z$  by fact
qed

```

```

theorem meet-idem:  $x \sqcap x = x$ 
proof -
  have  $x \sqcap (x \sqcup (x \sqcap x)) = x$  by (rule meet-join-absorb)
  also have  $x \sqcup (x \sqcap x) = x$  by (rule join-meet-absorb)
  finally show ?thesis .
qed

```

```

theorem meet-related [elim?]:  $x \sqsubseteq y \implies x \sqcap y = x$ 
proof (rule meetI)
  assume  $x \sqsubseteq y$ 
  show  $x \sqsubseteq x$  ..
  show  $x \sqsubseteq y$  by fact
  fix z assume  $z \sqsubseteq x$  and  $z \sqsubseteq y$ 
  show  $z \sqsubseteq x$  by fact
qed

```

```

theorem meet-related2 [elim?]:  $y \sqsubseteq x \implies x \sqcap y = y$ 
  by (drule meet-related) (simp add: meet-commute)

```

```

theorem join-related [elim?]:  $x \sqsubseteq y \implies x \sqcup y = y$ 
proof (rule joinI)
  assume  $x \sqsubseteq y$ 
  show  $y \sqsubseteq y$  ..
  show  $x \sqsubseteq y$  by fact
  fix  $z$  assume  $x \sqsubseteq z$  and  $y \sqsubseteq z$ 
  show  $y \sqsubseteq z$  by fact
qed

```

```

theorem join-related2 [elim?]:  $y \sqsubseteq x \implies x \sqcup y = x$ 
  by (drule join-related) (simp add: join-commute)

```

Additional theorems

```

theorem meet-connection:  $(x \sqsubseteq y) = (x \sqcap y = x)$ 
proof
  assume  $x \sqsubseteq y$ 
  then have is-inf  $x \ y \ x$  ..
  then show  $x \sqcap y = x$  ..
next
  have  $x \sqcap y \sqsubseteq y$  ..
  also assume  $x \sqcap y = x$ 
  finally show  $x \sqsubseteq y$  .
qed

```

```

theorem meet-connection2:  $(x \sqsubseteq y) = (y \sqcap x = x)$ 
  using meet-commute meet-connection by simp

```

```

theorem join-connection:  $(x \sqsubseteq y) = (x \sqcup y = y)$ 
proof
  assume  $x \sqsubseteq y$ 
  then have is-sup  $x \ y \ y$  ..
  then show  $x \sqcup y = y$  ..
next
  have  $x \sqsubseteq x \sqcup y$  ..
  also assume  $x \sqcup y = y$ 
  finally show  $x \sqsubseteq y$  .
qed

```

```

theorem join-connection2:  $(x \sqsubseteq y) = (x \sqcup y = y)$ 
  using join-commute join-connection by simp

```

Naming according to Jacobson I, p. 459.

```

lemmas L1 = join-commute meet-commute
lemmas L2 = join-assoc meet-assoc

```

```

lemmas L4 = join-meet-absorb meet-join-absorb

```

```

end

```

locale $dlat = dlat +$
assumes $meet-distr$:
 $dlat.meet\ le\ x\ (dlat.join\ le\ y\ z) =$
 $dlat.join\ le\ (dlat.meet\ le\ x\ y)\ (dlat.meet\ le\ x\ z)$

begin

lemma $join-distr$:
 $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

Jacobson I, p. 462

proof –

have $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$ **by** ($simp\ add: L4$)
also have $\dots = x \sqcup ((x \sqcap z) \sqcup (y \sqcap z))$ **by** ($simp\ add: L2$)
also have $\dots = x \sqcup ((x \sqcup y) \sqcap z)$ **by** ($simp\ add: L1\ meet-distr$)
also have $\dots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$ **by** ($simp\ add: L1\ L4$)
also have $\dots = (x \sqcup y) \sqcap (x \sqcup z)$ **by** ($simp\ add: meet-distr$)
finally show $?thesis$.

qed

end

locale $dlo = dpo +$
assumes $total: x \sqsubseteq y \mid y \sqsubseteq x$

begin

lemma $less-total: x \sqsubset y \mid x = y \mid y \sqsubset x$
using $total$
by ($unfold\ less-def$) $blast$

end

sublocale $dlo < dlat$

proof

fix $x\ y$
from $total$ **have** $is-inf\ x\ y$ ($if\ x \sqsubseteq y\ then\ x\ else\ y$) **by** ($auto\ simp: is-inf-def$)
then show $EX\ inf. is-inf\ x\ y\ inf$ **by** $blast$

next

fix $x\ y$
from $total$ **have** $is-sup\ x\ y$ ($if\ x \sqsubseteq y\ then\ y\ else\ x$) **by** ($auto\ simp: is-sup-def$)
then show $EX\ sup. is-sup\ x\ y\ sup$ **by** $blast$

qed

sublocale $dlo < dlat$

proof

fix $x\ y\ z$

show $x \sqcap (y \sqcup z) = x \sqcap y \sqcup x \sqcap z$ (**is** $?l = ?r$)

Jacobson I, p. 462

```

proof –
  { assume  $c: y \sqsubseteq x \sqsubseteq z$ 
    from  $c$  have  $?l = y \sqcup z$ 
      by (metis c join-connection2 join-related2 meet-connection meet-related2
total)
    also from  $c$  have  $\dots = ?r$  by (metis meet-related2)
    finally have  $?l = ?r$  . }
moreover
  { assume  $c: x \sqsubseteq y \mid x \sqsubseteq z$ 
    from  $c$  have  $?l = x$ 
      by (metis join-connection2 join-related2 meet-connection total trans)
    also from  $c$  have  $\dots = ?r$ 
      by (metis join-commute join-related2 meet-connection meet-related2 total)
    finally have  $?l = ?r$  . }
moreover note total
ultimately show ?thesis by blast
qed
qed

```

12.1.2 Total order \leq on *int*

interpretation *int*: $dpo\ op \leq :: [int, int] \Rightarrow bool$
where (*dpo.less* ($op \leq$) ($x::int$) y) = ($x < y$)

We give interpretation for less, but not *is-inf* and *is-sub*.

```

proof –
  show  $dpo\ op \leq :: [int, int] \Rightarrow bool$ 
    proof qed auto
  then interpret int:  $dpo\ op \leq :: [int, int] \Rightarrow bool$  .

```

Gives interpreted version of *less-def* (without condition).

```

  show (dpo.less ( $op \leq$ ) ( $x::int$ )  $y$ ) = ( $x < y$ )
    by (unfold int.less-def) auto
qed

```

thm *int.circular*

lemma $\llbracket (x::int) \leq y; y \leq z; z \leq x \rrbracket \Longrightarrow x = y \wedge y = z$

apply (*rule int.circular*) **apply** *assumption* **apply** *assumption* **apply** *assumption*
done

thm *int.abs-test*

lemma ($op < :: [int, int] \Rightarrow bool$) = $op <$
apply (*rule int.abs-test*) **done**

interpretation *int*: $dlat\ op \leq :: [int, int] \Rightarrow bool$

where *meet-eq*: $dlat.meet\ (op \leq) (x::int)\ y = \min\ x\ y$
and *join-eq*: $dlat.join\ (op \leq) (x::int)\ y = \max\ x\ y$

```

proof –
  show dlat (op <= :: [int, int] => bool)
    apply unfold-locales
    apply (unfold int.is-inf-def int.is-sup-def)
    apply arith+
    done
  then interpret int: dlat op <= :: [int, int] => bool .

```

Interpretation to ease use of definitions, which are conditional in general but unconditional after interpretation.

```

  show dlat.meet (op <=) (x::int) y = min x y
    apply (unfold int.meet-def)
    apply (rule the-equality)
    apply (unfold int.is-inf-def)
    by auto
  show dlat.join (op <=) (x::int) y = max x y
    apply (unfold int.join-def)
    apply (rule the-equality)
    apply (unfold int.is-sup-def)
    by auto
qed

```

```

interpretation int: dlo op <= :: [int, int] => bool
  proof qed arith

```

Interpreted theorems from the locales, involving defined terms.

```

thm int.less-def

from dpo
thm int.meet-left

from dlat
thm int.meet-distr

from ddlat
thm int.less-total

from dlo

```

12.1.3 Total order <= on nat

```

interpretation nat: dpo op <= :: [nat, nat] => bool
  where dpo.less (op <=) (x::nat) y = (x < y)

```

We give interpretation for less, but not *is-inf* and *is-sub*.

```

proof –
  show dpo (op <= :: [nat, nat] => bool)
    proof qed auto

```

then interpret *nat*: *dpo op* \leq :: [*nat*, *nat*] \Rightarrow *bool* .

Gives interpreted version of *less-def* (without condition).

```

show dpo.less (op  $\leq$ ) (x::nat) y = (x < y)
apply (unfold nat.less-def)
apply auto
done
qed

```

interpretation *nat*: *dlat op* \leq :: [*nat*, *nat*] \Rightarrow *bool*

where *dlat.meet* (*op* \leq) (*x*::*nat*) *y* = *min x y*
and *dlat.join* (*op* \leq) (*x*::*nat*) *y* = *max x y*

proof –

```

show dlat (op  $\leq$  :: [nat, nat]  $\Rightarrow$  bool)
apply unfold-locales
apply (unfold nat.is-inf-def nat.is-sup-def)
apply arith+
done

```

then interpret *nat*: *dlat op* \leq :: [*nat*, *nat*] \Rightarrow *bool* .

Interpretation to ease use of definitions, which are conditional in general but unconditional after interpretation.

```

show dlat.meet (op  $\leq$ ) (x::nat) y = min x y
apply (unfold nat.meet-def)
apply (rule the-equality)
apply (unfold nat.is-inf-def)
by auto
show dlat.join (op  $\leq$ ) (x::nat) y = max x y
apply (unfold nat.join-def)
apply (rule the-equality)
apply (unfold nat.is-sup-def)
by auto
qed

```

interpretation *nat*: *dlo op* \leq :: [*nat*, *nat*] \Rightarrow *bool*

proof **qed** *arith*

Interpreted theorems from the locales, involving defined terms.

thm *nat.less-def*

from *dpo*

thm *nat.meet-left*

from *dlat*

thm *nat.meet-distr*

from *ddlat*

thm *nat.less-total*

from *ldo*

12.1.4 Lattice *dvd* on *nat*

interpretation *nat-dvd*: *dpo op dvd* :: [*nat*, *nat*] => *bool*
where *dpo.less* (*op dvd*) (*x::nat*) *y* = (*x dvd y* & *x ~*= *y*)

We give interpretation for less, but not *is-inf* and *is-sub*.

proof –
show *dpo* (*op dvd* :: [*nat*, *nat*] => *bool*)
proof **qed** (*auto simp: dvd-def*)
then interpret *nat-dvd*: *dpo op dvd* :: [*nat*, *nat*] => *bool* .

Gives interpreted version of *less-def* (without condition).

show *dpo.less* (*op dvd*) (*x::nat*) *y* = (*x dvd y* & *x ~*= *y*)
apply (*unfold nat-dvd.less-def*)
apply *auto*
done
qed

interpretation *nat-dvd*: *dlat op dvd* :: [*nat*, *nat*] => *bool*
where *dlat.meet* (*op dvd*) (*x::nat*) *y* = *gcd x y*
and *dlat.join* (*op dvd*) (*x::nat*) *y* = *lcm x y*

proof –
show *dlat* (*op dvd* :: [*nat*, *nat*] => *bool*)
apply *unfold-locales*
apply (*unfold nat-dvd.is-inf-def nat-dvd.is-sup-def*)
apply (*rule-tac x = gcd x y in exI*)
apply *auto* [1]
apply (*rule-tac x = lcm x y in exI*)
apply (*auto intro: lcm-dvd1-nat lcm-dvd2-nat lcm-least-nat*)
done
then interpret *nat-dvd*: *dlat op dvd* :: [*nat*, *nat*] => *bool* .

Interpretation to ease use of definitions, which are conditional in general but unconditional after interpretation.

show *dlat.meet* (*op dvd*) (*x::nat*) *y* = *gcd x y*
apply (*unfold nat-dvd.meet-def*)
apply (*rule the-equality*)
apply (*unfold nat-dvd.is-inf-def*)
by *auto*
show *dlat.join* (*op dvd*) (*x::nat*) *y* = *lcm x y*
apply (*unfold nat-dvd.join-def*)
apply (*rule the-equality*)
apply (*unfold nat-dvd.is-sup-def*)
by (*auto intro: lcm-dvd1-nat lcm-dvd2-nat lcm-least-nat*)
qed

Interpreted theorems from the locales, involving defined terms.

thm *nat-dvd.less-def*

from *dpo*

```

lemma ((x::nat) dvd y & x ~ = y) = (x dvd y & x ~ = y)
  apply (rule nat-dvd.less-def) done
thm nat-dvd.meet-left

```

from dlat

```

lemma gcd x y dvd (x::nat)
  apply (rule nat-dvd.meet-left) done

```

12.2 Group example with defined operations *inv* and *unit*

12.2.1 Locale declarations and lemmas

```

locale Dsemi =
  fixes prod (infixl ** 65)
  assumes assoc: (x ** y) ** z = x ** (y ** z)

```

```

locale Dmonoid = Dsemi +
  fixes one
  assumes l-one [simp]: one ** x = x
  and r-one [simp]: x ** one = x

```

begin

definition

inv **where** *inv* *x* = (*THE* *y*. *x* ** *y* = *one* & *y* ** *x* = *one*)

definition

unit **where** *unit* *x* = (*EX* *y*. *x* ** *y* = *one* & *y* ** *x* = *one*)

lemma *inv-unique*:

assumes *eq*: *y* ** *x* = *one* & *x* ** *y'* = *one*
shows *y* = *y'*

proof –

from *eq* **have** *y* = *y* ** (*x* ** *y'*) **by** (*simp add: r-one*)
also have ... = (*y* ** *x*) ** *y'* **by** (*simp add: assoc*)
also from *eq* **have** ... = *y'* **by** (*simp add: l-one*)
finally show ?*thesis* .

qed

lemma *unit-one* [*intro*, *simp*]:

unit one
by (*unfold unit-def*) *auto*

lemma *unit-l-inv-ex*:

*unit x ==> ∃ y. y ** x = one*
by (*unfold unit-def*) *auto*

lemma *unit-r-inv-ex*:

*unit x ==> ∃ y. x ** y = one*
by (*unfold unit-def*) *auto*


```

lemma unit-l-inv:
  unit x ==> inv x ** x = one
  apply (simp add: unit-def inv-def) apply (erule exE)
  apply (rule theI2, fast)
  apply (rule inv-unique)
  apply fast+
  done

```

```

lemma unit-r-inv:
  unit x ==> x ** inv x = one
  apply (simp add: unit-def inv-def) apply (erule exE)
  apply (rule theI2, fast)
  apply (rule inv-unique)
  apply fast+
  done

```

```

lemma unit-inv-unit [intro, simp]:
  unit x ==> unit (inv x)
proof –
  assume x: unit x
  show unit (inv x)
    by (auto simp add: unit-def
      intro: unit-l-inv unit-r-inv x)
qed

```

```

lemma unit-l-cancel [simp]:
  unit x ==> (x ** y = x ** z) = (y = z)
proof
  assume eq: x ** y = x ** z
  and G: unit x
  then have (inv x ** x) ** y = (inv x ** x) ** z
    by (simp add: assoc)
  with G show y = z by (simp add: unit-l-inv)
next
  assume eq: y = z
  and G: unit x
  then show x ** y = x ** z by simp
qed

```

```

lemma unit-inv-inv [simp]:
  unit x ==> inv (inv x) = x
proof –
  assume x: unit x
  then have inv x ** inv (inv x) = inv x ** x
    by (simp add: unit-l-inv unit-r-inv)
  with x show ?thesis by simp
qed

```

```

lemma inv-inj-on-unit:

```

```

    inj-on inv {x. unit x}
proof (rule inj-onI, simp)
  fix x y
  assume G: unit x unit y and eq: inv x = inv y
  then have inv (inv x) = inv (inv y) by simp
  with G show x = y by simp
qed

lemma unit-inv-comm:
  assumes inv: x ** y = one
  and G: unit x unit y
  shows y ** x = one
proof –
  from G have x ** y ** x = x ** one by (auto simp add: inv)
  with G show ?thesis by (simp del: r-one add: assoc)
qed

end

locale Dgrp = Dmonoid +
  assumes unit [intro, simp]: Dmonoid.unit (op **) one x

begin

lemma l-inv-ex [simp]:
   $\exists y. y ** x = one$ 
  using unit-l-inv-ex by simp

lemma r-inv-ex [simp]:
   $\exists y. x ** y = one$ 
  using unit-r-inv-ex by simp

lemma l-inv [simp]:
  inv x ** x = one
  using unit-l-inv by simp

lemma l-cancel [simp]:
   $(x ** y = x ** z) = (y = z)$ 
  using unit-l-inv by simp

lemma r-inv [simp]:
  x ** inv x = one
proof –
  have inv x ** (x ** inv x) = inv x ** one
  by (simp add: assoc [symmetric] l-inv)
  then show ?thesis by (simp del: r-one)
qed

```

```

lemma r-cancel [simp]:
  ( $y ** x = z ** x$ ) = ( $y = z$ )
proof
  assume eq:  $y ** x = z ** x$ 
  then have  $y ** (x ** \text{inv } x) = z ** (x ** \text{inv } x)$ 
    by (simp add: assoc [symmetric] del: r-inv)
  then show  $y = z$  by simp
qed simp

lemma inv-one [simp]:
   $\text{inv one} = \text{one}$ 
proof –
  have  $\text{inv one} = \text{one} ** (\text{inv one})$  by (simp del: r-inv)
  moreover have  $\dots = \text{one}$  by simp
  finally show ?thesis .
qed

lemma inv-inv [simp]:
   $\text{inv} (\text{inv } x) = x$ 
  using unit-inv-inv by simp

lemma inv-inj:
  inj-on inv UNIV
  using inv-inj-on-unit by simp

lemma inv-mult-group:
   $\text{inv} (x ** y) = \text{inv } y ** \text{inv } x$ 
proof –
  have  $\text{inv} (x ** y) ** (x ** y) = (\text{inv } y ** \text{inv } x) ** (x ** y)$ 
    by (simp add: assoc l-inv) (simp add: assoc [symmetric])
  then show ?thesis by (simp del: l-inv)
qed

lemma inv-comm:
   $x ** y = \text{one} \implies y ** x = \text{one}$ 
  by (rule unit-inv-comm) auto

lemma inv-equality:
   $y ** x = \text{one} \implies \text{inv } x = y$ 
  apply (simp add: inv-def)
  apply (rule the-equality)
  apply (simp add: inv-comm [of y x])
  apply (rule r-cancel [THEN iffD1], auto)
  done

end

```

locale *Dhom* = *prod: Dgrp prod one* + *sum: Dgrp sum zero*

```

    for prod (infixl ** 65) and one and sum (infixl +++ 60) and zero +
    fixes hom
    assumes hom-mult [simp]: hom (x ** y) = hom x +++ hom y

begin

lemma hom-one [simp]:
  hom one = zero
proof -
  have hom one +++ zero = hom one +++ hom one
    by (simp add: hom-mult [symmetric] del: hom-mult)
  then show ?thesis by (simp del: r-one)
qed

end

```

12.2.2 Interpretation of Functions

```

interpretation Dfun: Dmonoid op o id :: 'a => 'a
  where Dmonoid.unit (op o) id f = bij (f::'a => 'a)

proof -
  show Dmonoid op o (id :: 'a => 'a) proof qed (simp-all add: o-assoc)
  note Dmonoid = this

  show Dmonoid.unit (op o) (id :: 'a => 'a) f = bij f
    apply (unfold Dmonoid.unit-def [OF Dmonoid])
    apply rule apply clarify
  proof -
    fix f g
    assume id1: f o g = id and id2: g o f = id
    show bij f
    proof (rule bijI)
      show inj f
      proof (rule inj-onI)
        fix x y
        assume f x = f y
        then have (g o f) x = (g o f) y by simp
        with id2 show x = y by simp
      qed
    next
      show surj f
      proof (rule surjI)
        fix x
        from id1 have (f o g) x = x by simp
        then show f (g x) = x by simp
      qed
    qed
  next

```

```

    fix f
    assume bij: bij f
    then
    have inv: f o Hilbert-Choice.inv f = id & Hilbert-Choice.inv f o f = id
      by (simp add: bij-def surj-iff inj-iff)
    show EX g. f o g = id & g o f = id by rule (rule inv)
  qed
qed

thm Dmonoid.unit-def Dfun.unit-def

thm Dmonoid.inv-inj-on-unit Dfun.inv-inj-on-unit

lemma unit-id:
  (f :: unit => unit) = id
  by rule simp

interpretation Dfun: Dgrp op o id :: unit => unit
  where Dmonoid.inv (op o) id f = inv (f :: unit => unit)
proof -
  have Dmonoid op o (id :: 'a => 'a) ..
  note Dmonoid = this

  show Dgrp (op o) (id :: unit => unit)
  apply unfold-locales
  apply (unfold Dmonoid.unit-def [OF Dmonoid])
  apply (insert unit-id)
  apply simp
done
  show Dmonoid.inv (op o) id f = inv (f :: unit => unit)
  apply (unfold Dmonoid.inv-def [OF Dmonoid])
  apply (insert unit-id)
  apply simp
  apply (rule the-equality)
  apply rule
  apply rule
  apply simp
done
qed

thm Dfun.unit-l-inv Dfun.l-inv

thm Dfun.inv-equality
thm Dfun.inv-equality

end

```

13 Monoids and Groups as predicates over record schemes

theory *MonoidGroup* **imports** *Main* **begin**

record *'a monoid-sig* =
 times :: *'a* => *'a* => *'a*
 one :: *'a*

record *'a group-sig* = *'a monoid-sig* +
 inv :: *'a* => *'a*

definition

monoid :: (*| times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) => *bool* **where**
monoid *M* = ($\forall x y z.$
 times *M* (*times* *M* *x* *y*) *z* = *times* *M* *x* (*times* *M* *y* *z*) \wedge
 times *M* (*one* *M*) *x* = *x* \wedge *times* *M* *x* (*one* *M*) = *x*)

definition

group :: (*| times* :: *'a* => *'a* => *'a*, *one* :: *'a*, *inv* :: *'a* => *'a*, ... :: *'b* |) => *bool*
where
group *G* = (*monoid* *G* \wedge ($\forall x. \text{times } G (\text{inv } G \ x) \ x = \text{one } G$))

definition

reverse :: (*| times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) =>
 (*| times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) **where**
reverse *M* = *M* (*| times* := $\lambda x y. \text{times } M \ y \ x$ |)

end

14 Binary arithmetic examples

theory *BinEx*
imports *Complex-Main*
begin

14.1 Regression Testing for Cancellation Simprocs

lemma $l + 2 + 2 + 2 + (l + 2) + (oo + 2) = (uu::int)$
apply *simp* **oops**

lemma $2*u = (u::int)$
apply *simp* **oops**

lemma $(i + j + 12 + (k::int)) - 15 = y$
apply *simp* **oops**

lemma $(i + j + 12 + (k::int)) - 5 = y$

apply simp **oops**

lemma $y - b < (b::int)$

apply simp **oops**

lemma $y - (3*b + c) < (b::int) - 2*c$

apply simp **oops**

lemma $(2*x - (u*v) + y) - v*3*u = (w::int)$

apply simp **oops**

lemma $(2*x*u*v + (u*v)*4 + y) - v*u*4 = (w::int)$

apply simp **oops**

lemma $(2*x*u*v + (u*v)*4 + y) - v*u = (w::int)$

apply simp **oops**

lemma $u*v - (x*u*v + (u*v)*4 + y) = (w::int)$

apply simp **oops**

lemma $(i + j + 12 + (k::int)) = u + 15 + y$

apply simp **oops**

lemma $(i + j*2 + 12 + (k::int)) = j + 5 + y$

apply simp **oops**

lemma $2*y + 3*z + 6*w + 2*y + 3*z + 2*u = 2*y' + 3*z' + 6*w' + 2*y' + 3*z' + u + (v::int)$

apply simp **oops**

lemma $a + -(b+c) + b = (d::int)$

apply simp **oops**

lemma $a + -(b+c) - b = (d::int)$

apply simp **oops**

lemma $(i + j + -2 + (k::int)) - (u + 5 + y) = zz$

apply simp **oops**

lemma $(i + j + -3 + (k::int)) < u + 5 + y$

apply simp **oops**

lemma $(i + j + 3 + (k::int)) < u + -6 + y$

apply simp **oops**

lemma $(i + j + -12 + (k::int)) - 15 = y$

apply simp **oops**

lemma $(i + j + 12 + (k::int)) - -15 = y$
apply *simp* **oops**

lemma $(i + j + -12 + (k::int)) - -15 = y$
apply *simp* **oops**

lemma $-(2*i) + 3 + (2*i + 4) = (0::int)$
apply *simp* **oops**

14.2 Arithmetic Method Tests

lemma $!!a::int. [a \leq b; c \leq d; x+y < z] \implies a+c \leq b+d$
by *arith*

lemma $!!a::int. [a < b; c < d] \implies a-d+2 \leq b+(-c)$
by *arith*

lemma $!!a::int. [a < b; c < d] \implies a+c+1 < b+d$
by *arith*

lemma $!!a::int. [a \leq b; b+b \leq c] \implies a+a \leq c$
by *arith*

lemma $!!a::int. [a+b \leq i+j; a \leq b; i \leq j] \implies a+a \leq j+j$
by *arith*

lemma $!!a::int. [a+b < i+j; a < b; i < j] \implies a+a - - -1 < j+j - 3$
by *arith*

lemma $!!a::int. a+b+c \leq i+j+k \ \& \ a \leq b \ \& \ b \leq c \ \& \ i \leq j \ \& \ j \leq k \implies a+a+a \leq k+k+k$
by *arith*

lemma $!!a::int. [a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k; k \leq l] \implies a \leq l$
by *arith*

lemma $!!a::int. [a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k; k \leq l] \implies a+a+a+a \leq l+l+l+l$
by *arith*

lemma $!!a::int. [a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k; k \leq l] \implies a+a+a+a+a \leq l+l+l+l+i$
by *arith*

lemma $!!a::int. [a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k;$

$k \leq l$ $||$
 $\implies a+a+a+a+a+a \leq l+l+l+l+i+l$
by *arith*

lemma $!!a::int. [| a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k;$
 $k \leq l |]$
 $\implies 6*a \leq 5*l+i$
by *arith*

14.3 The Integers

Addition

lemma $(13::int) + 19 = 32$
by *simp*

lemma $(1234::int) + 5678 = 6912$
by *simp*

lemma $(1359::int) + -2468 = -1109$
by *simp*

lemma $(93746::int) + -46375 = 47371$
by *simp*

Negation

lemma $-(65745::int) = -65745$
by *simp*

lemma $-(-54321::int) = 54321$
by *simp*

Multiplication

lemma $(13::int) * 19 = 247$
by *simp*

lemma $(-84::int) * 51 = -4284$
by *simp*

lemma $(255::int) * 255 = 65025$
by *simp*

lemma $(1359::int) * -2468 = -3354012$
by *simp*

lemma $(89::int) * 10 \neq 889$
by *simp*

lemma $(13::int) < 18 - 4$
by *simp*

lemma $(-345::int) < -242 + -100$
by *simp*

lemma $(13557456::int) < 18678654$
by *simp*

lemma $(999999::int) \leq (1000001 + 1) - 2$
by *simp*

lemma $(1234567::int) \leq 1234567$
by *simp*

No integer overflow!

lemma $1234567 * (1234567::int) < 1234567 * 1234567 * 1234567$
by *simp*

Quotient and Remainder

lemma $(10::int) \text{ div } 3 = 3$
by *simp*

lemma $(10::int) \text{ mod } 3 = 1$
by *simp*

A negative divisor

lemma $(10::int) \text{ div } -3 = -4$
by *simp*

lemma $(10::int) \text{ mod } -3 = -2$
by *simp*

A negative dividend¹

lemma $(-10::int) \text{ div } 3 = -4$
by *simp*

lemma $(-10::int) \text{ mod } 3 = 2$
by *simp*

A negative dividend *and* divisor

lemma $(-10::int) \text{ div } -3 = 3$
by *simp*

lemma $(-10::int) \text{ mod } -3 = -1$

¹The definition agrees with mathematical convention and with ML, but not with the hardware of most computers

by *simp*

A few bigger examples

lemma $(8452::int) \bmod 3 = 1$
by *simp*

lemma $(59485::int) \operatorname{div} 434 = 137$
by *simp*

lemma $(1000006::int) \bmod 10 = 6$
by *simp*

Division by shifting

lemma $10000000 \operatorname{div} 2 = (5000000::int)$
by *simp*

lemma $10000001 \bmod 2 = (1::int)$
by *simp*

lemma $10000055 \operatorname{div} 32 = (312501::int)$
by *simp*

lemma $10000055 \bmod 32 = (23::int)$
by *simp*

lemma $100094 \operatorname{div} 144 = (695::int)$
by *simp*

lemma $100094 \bmod 144 = (14::int)$
by *simp*

Powers

lemma $2 ^ 10 = (1024::int)$
by *simp*

lemma $-3 ^ 7 = (-2187::int)$
by *simp*

lemma $13 ^ 7 = (62748517::int)$
by *simp*

lemma $3 ^ 15 = (14348907::int)$
by *simp*

lemma $-5 ^ 11 = (-48828125::int)$
by *simp*

14.4 The Natural Numbers

Successor

lemma $Suc\ 99999 = 100000$
 by (*simp add: Suc-nat-number-of*)
 — not a default rewrite since sometimes we want to have $Suc\ nnn$

Addition

lemma $(13::nat) + 19 = 32$
 by *simp*

lemma $(1234::nat) + 5678 = 6912$
 by *simp*

lemma $(973646::nat) + 6475 = 980121$
 by *simp*

Subtraction

lemma $(32::nat) - 14 = 18$
 by *simp*

lemma $(14::nat) - 15 = 0$
 by *simp*

lemma $(14::nat) - 1576644 = 0$
 by *simp*

lemma $(48273776::nat) - 3873737 = 44400039$
 by *simp*

Multiplication

lemma $(12::nat) * 11 = 132$
 by *simp*

lemma $(647::nat) * 3643 = 2357021$
 by *simp*

Quotient and Remainder

lemma $(10::nat) \text{ div } 3 = 3$
 by *simp*

lemma $(10::nat) \text{ mod } 3 = 1$
 by *simp*

lemma $(10000::nat) \text{ div } 9 = 1111$
 by *simp*

lemma $(10000::nat) \bmod 9 = 1$
by *simp*

lemma $(10000::nat) \operatorname{div} 16 = 625$
by *simp*

lemma $(10000::nat) \bmod 16 = 0$
by *simp*

Powers

lemma $2^{12} = (4096::nat)$
by *simp*

lemma $3^{10} = (59049::nat)$
by *simp*

lemma $12^7 = (35831808::nat)$
by *simp*

lemma $3^{14} = (4782969::nat)$
by *simp*

lemma $5^{11} = (48828125::nat)$
by *simp*

Testing the cancellation of complementary terms

lemma $y + (x + -x) = (0::int) + y$
by *simp*

lemma $y + (-x + (-y + x)) = (0::int)$
by *simp*

lemma $-x + (y + (-y + x)) = (0::int)$
by *simp*

lemma $x + (x + (-x + (-x + (-y + -z)))) = (0::int) - y - z$
by *simp*

lemma $x + x - x - x - y - z = (0::int) - y - z$
by *simp*

lemma $x + y + z - (x + z) = y - (0::int)$
by *simp*

lemma $x + (y + (y + (y + (-x + -x)))) = (0::int) + y - x + y + y$
by *simp*

lemma $x + (y + (y + (y + (-y + -x)))) = y + (0::int) + y$
by *simp*

lemma $x + y - x + z - x - y - z + x < (1::int)$
by *simp*

14.5 Real Arithmetic

14.5.1 Addition

lemma $(1359::real) + -2468 = -1109$
by *simp*

lemma $(93746::real) + -46375 = 47371$
by *simp*

14.5.2 Negation

lemma $-(65745::real) = -65745$
by *simp*

lemma $-(-54321::real) = 54321$
by *simp*

14.5.3 Multiplication

lemma $(-84::real) * 51 = -4284$
by *simp*

lemma $(255::real) * 255 = 65025$
by *simp*

lemma $(1359::real) * -2468 = -3354012$
by *simp*

14.5.4 Inequalities

lemma $(89::real) * 10 \neq 889$
by *simp*

lemma $(13::real) < 18 - 4$
by *simp*

lemma $(-345::real) < -242 + -100$
by *simp*

lemma $(13557456::real) < 18678654$
by *simp*

lemma $(999999::real) \leq (1000001 + 1) - 2$

by *simp*

lemma $(1234567::\text{real}) \leq 1234567$
by *simp*

14.5.5 Powers

lemma $2^{15} = (32768::\text{real})$
by *simp*

lemma $-3^7 = (-2187::\text{real})$
by *simp*

lemma $13^7 = (62748517::\text{real})$
by *simp*

lemma $3^{15} = (14348907::\text{real})$
by *simp*

lemma $-5^{11} = (-48828125::\text{real})$
by *simp*

14.5.6 Tests

lemma $(x + y = x) = (y = (0::\text{real}))$
by *arith*

lemma $(x + y = y) = (x = (0::\text{real}))$
by *arith*

lemma $(x + y = (0::\text{real})) = (x = -y)$
by *arith*

lemma $(x + y = (0::\text{real})) = (y = -x)$
by *arith*

lemma $((x + y) < (x + z)) = (y < (z::\text{real}))$
by *arith*

lemma $((x + z) < (y + z)) = (x < (y::\text{real}))$
by *arith*

lemma $(\neg x < y) = (y \leq (x::\text{real}))$
by *arith*

lemma $\neg (x < y \wedge y < (x::\text{real}))$
by *arith*

lemma $(x::\text{real}) < y ==> \neg y < x$
by *arith*

lemma $((x::real) \neq y) = (x < y \vee y < x)$
by *arith*

lemma $(\neg x \leq y) = (y < (x::real))$
by *arith*

lemma $x \leq y \vee y \leq (x::real)$
by *arith*

lemma $x \leq y \vee y < (x::real)$
by *arith*

lemma $x < y \vee y \leq (x::real)$
by *arith*

lemma $x \leq (x::real)$
by *arith*

lemma $((x::real) \leq y) = (x < y \vee x = y)$
by *arith*

lemma $((x::real) \leq y \wedge y \leq x) = (x = y)$
by *arith*

lemma $\neg(x < y \wedge y \leq (x::real))$
by *arith*

lemma $\neg(x \leq y \wedge y < (x::real))$
by *arith*

lemma $(-x < (0::real)) = (0 < x)$
by *arith*

lemma $((0::real) < -x) = (x < 0)$
by *arith*

lemma $(-x \leq (0::real)) = (0 \leq x)$
by *arith*

lemma $((0::real) \leq -x) = (x \leq 0)$
by *arith*

lemma $(x::real) = y \vee x < y \vee y < x$
by *arith*

lemma $(x::real) = 0 \vee 0 < x \vee 0 < -x$
by *arith*

lemma $(0::real) \leq x \vee 0 \leq -x$
by *arith*

lemma $((x::real) + y \leq x + z) = (y \leq z)$
by *arith*

lemma $((x::real) + z \leq y + z) = (x \leq y)$
by *arith*

lemma $(w::real) < x \wedge y < z ==> w + y < x + z$
by *arith*

lemma $(w::real) \leq x \wedge y \leq z ==> w + y \leq x + z$
by *arith*

lemma $(0::real) \leq x \wedge 0 \leq y ==> 0 \leq x + y$
by *arith*

lemma $(0::real) < x \wedge 0 < y ==> 0 < x + y$
by *arith*

lemma $(-x < y) = (0 < x + (y::real))$
by *arith*

lemma $(x < -y) = (x + y < (0::real))$
by *arith*

lemma $(y < x + -z) = (y + z < (x::real))$
by *arith*

lemma $(x + -y < z) = (x < z + (y::real))$
by *arith*

lemma $x \leq y ==> x < y + (1::real)$
by *arith*

lemma $(x - y) + y = (x::real)$
by *arith*

lemma $y + (x - y) = (x::real)$
by *arith*

lemma $x - x = (0::real)$
by *arith*

lemma $(x - y = 0) = (x = (y::real))$
by *arith*

lemma $((0::real) \leq x + x) = (0 \leq x)$

by *arith*

lemma $(-x \leq x) = ((0::real) \leq x)$
by *arith*

lemma $(x \leq -x) = (x \leq (0::real))$
by *arith*

lemma $(-x = (0::real)) = (x = 0)$
by *arith*

lemma $-(x - y) = y - (x::real)$
by *arith*

lemma $((0::real) < x - y) = (y < x)$
by *arith*

lemma $((0::real) \leq x - y) = (y \leq x)$
by *arith*

lemma $(x + y) - x = (y::real)$
by *arith*

lemma $(-x = y) = (x = (-y::real))$
by *arith*

lemma $x < (y::real) ==> \neg(x = y)$
by *arith*

lemma $(x \leq x + y) = ((0::real) \leq y)$
by *arith*

lemma $(y \leq x + y) = ((0::real) \leq x)$
by *arith*

lemma $(x < x + y) = ((0::real) < y)$
by *arith*

lemma $(y < x + y) = ((0::real) < x)$
by *arith*

lemma $(x - y) - x = (-y::real)$
by *arith*

lemma $(x + y < z) = (x < z - (y::real))$
by *arith*

lemma $(x - y < z) = (x < z + (y::real))$
by *arith*

lemma $(x < y - z) = (x + z < (y::real))$
by *arith*

lemma $(x \leq y - z) = (x + z \leq (y::real))$
by *arith*

lemma $(x - y \leq z) = (x \leq z + (y::real))$
by *arith*

lemma $(-x < -y) = (y < (x::real))$
by *arith*

lemma $(-x \leq -y) = (y \leq (x::real))$
by *arith*

lemma $(a + b) - (c + d) = (a - c) + (b - (d::real))$
by *arith*

lemma $(0::real) - x = -x$
by *arith*

lemma $x - (0::real) = x$
by *arith*

lemma $w \leq x \wedge y < z ==> w + y < x + (z::real)$
by *arith*

lemma $w < x \wedge y \leq z ==> w + y < x + (z::real)$
by *arith*

lemma $(0::real) \leq x \wedge 0 < y ==> 0 < x + (y::real)$
by *arith*

lemma $(0::real) < x \wedge 0 \leq y ==> 0 < x + y$
by *arith*

lemma $-x - y = -(x + (y::real))$
by *arith*

lemma $x - (-y) = x + (y::real)$
by *arith*

lemma $-x - -y = y - (x::real)$
by *arith*

lemma $(a - b) + (b - c) = a - (c::real)$
by *arith*

lemma $(x = y - z) = (x + z = (y::real))$
by *arith*

lemma $(x - y = z) = (x = z + (y::real))$
by *arith*

lemma $x - (x - y) = (y::real)$
by *arith*

lemma $x - (x + y) = -(y::real)$
by *arith*

lemma $x = y ==> x \leq (y::real)$
by *arith*

lemma $(0::real) < x ==> \neg(x = 0)$
by *arith*

lemma $(x + y) * (x - y) = (x * x) - (y * y)$
oops

lemma $(-x = -y) = (x = (y::real))$
by *arith*

lemma $(-x < -y) = (y < (x::real))$
by *arith*

lemma $!!a::real. a \leq b ==> c \leq d ==> x + y < z ==> a + c \leq b + d$
by *linarith*

lemma $!!a::real. a < b ==> c < d ==> a - d \leq b + (-c)$
by *linarith*

lemma $!!a::real. a \leq b ==> b + b \leq c ==> a + a \leq c$
by *linarith*

lemma $!!a::real. a + b \leq i + j ==> a \leq b ==> i \leq j ==> a + a \leq j + j$
by *linarith*

lemma $!!a::real. a + b < i + j ==> a < b ==> i < j ==> a + a < j + j$
by *linarith*

lemma $!!a::real. a + b + c \leq i + j + k \wedge a \leq b \wedge b \leq c \wedge i \leq j \wedge j \leq k -->$
 $a + a + a \leq k + k + k$
by *arith*

lemma $!!a::real. a + b + c + d \leq i + j + k + l ==> a \leq b ==> b \leq c$
 $==> c \leq d ==> i \leq j ==> j \leq k ==> k \leq l ==> a \leq l$
by *linarith*

lemma !!*a::real*. $a + b + c + d \leq i + j + k + l \implies a \leq b \implies b \leq c$
 $\implies c \leq d \implies i \leq j \implies j \leq k \implies k \leq l \implies a + a + a + a \leq l +$
 $l + l + l$
by *linarith*

lemma !!*a::real*. $a + b + c + d \leq i + j + k + l \implies a \leq b \implies b \leq c$
 $\implies c \leq d \implies i \leq j \implies j \leq k \implies k \leq l \implies a + a + a + a + a \leq$
 $l + l + l + l + i$
by *linarith*

lemma !!*a::real*. $a + b + c + d \leq i + j + k + l \implies a \leq b \implies b \leq c$
 $\implies c \leq d \implies i \leq j \implies j \leq k \implies k \leq l \implies a + a + a + a + a +$
 $a \leq l + l + l + l + i + l$
by *linarith*

14.6 Complex Arithmetic

lemma $(1359 + 93746*ii) - (2468 + 46375*ii) = -1109 + 47371*ii$
by *simp*

lemma $-(65745 + -47371*ii) = -65745 + 47371*ii$
by *simp*

Multiplication requires distributive laws. Perhaps versions instantiated to literal constants should be added to the simpset.

lemma $(1 + ii) * (1 - ii) = 2$
by (*simp add: ring-distribs*)

lemma $(1 + 2*ii) * (1 + 3*ii) = -5 + 5*ii$
by (*simp add: ring-distribs*)

lemma $(-84 + 255*ii) + (51 * 255*ii) = -84 + 13260 * ii$
by (*simp add: ring-distribs*)

No inequalities or linear arithmetic: the complex numbers are unordered!

No powers (not supported yet)

end

15 Examples for hexadecimal and binary numerals

theory *Hex-Bin-Examples* **imports** *Main*
begin

Hex and bin numerals can be used like normal decimal numerals in input

lemma $0xFF = 255$ **by** (*rule refl*)

lemma $0xF = 0b1111$ **by** (*rule refl*)

Just like decimal numeral they are polymorphic, for arithmetic they need to be constrained

lemma $0x0A + 0x10 = (0x1A :: \text{nat})$ **by** *simp*

The number of leading zeros is irrelevant

lemma $0b00010000 = 0x10$ **by** (*rule refl*)

Unary minus works as for decimal numerals

lemma $- 0x0A = - 10$ **by** (*rule refl*)

Hex and bin numerals are printed as decimal: $2 :: 'a$

term $0b10$

term $0x0A$

The numerals 0 and 1 are syntactically different from the constants 0 and 1. For the usual numeric types, their values are the same, though.

lemma $0x01 = 1$ **oops**

lemma $0x00 = 0$ **oops**

lemma $0x01 = (1 :: \text{nat})$ **by** *simp*

lemma $0b0000 = (0 :: \text{int})$ **by** *simp*

end

16 Antiquotations

theory *Antiquote*

imports *Main*

begin

A simple example on quote / antiquote in higher-order abstract syntax.

syntax

$\text{-Expr} :: 'a \Rightarrow 'a \quad (\text{EXPR} - [1000] \ 999)$

definition

$\text{var} :: 'a \Rightarrow ('a \Rightarrow \text{nat}) \Rightarrow \text{nat} \quad (\text{VAR} - [1000] \ 999)$

where $\text{var } x \ \text{env} = \text{env } x$

definition

$\text{Expr} :: (('a \Rightarrow \text{nat}) \Rightarrow \text{nat}) \Rightarrow ('a \Rightarrow \text{nat}) \Rightarrow \text{nat}$

where $\text{Expr } \text{exp } \text{env} = \text{exp } \text{env}$

```

parse-translation ⟨⟨
  [Syntax.quote-antiquote-tr' @ {syntax-const -Expr} @ {const-syntax var} @ {const-syntax
    Expr}]
  ⟩⟩

print-translation ⟨⟨
  [Syntax.quote-antiquote-tr' @ {syntax-const -Expr} @ {const-syntax var} @ {const-syntax
    Expr}]
  ⟩⟩

term EXPR (a + b + c)
term EXPR (a + b + c + VAR x + VAR y + 1)
term EXPR (VAR (f w) + VAR x)

term Expr (λenv. env x) — improper
term Expr (λenv. f env)
term Expr (λenv. f env + env x) — improper
term Expr (λenv. f env y z)
term Expr (λenv. f env + g y env)
term Expr (λenv. f env + g env y + h a env z)

end

```

17 Multiple nested quotations and anti-quotations

```

theory Multiquote
imports Main
begin

```

Multiple nested quotations and anti-quotations – basically a generalized version of de-Bruijn representation.

```

syntax
  -quote :: 'b => ('a => 'b)    (<-> [0] 1000)
  -antiquote :: ('a => 'b) => 'b  ('- [1000] 1000)

```

```

parse-translation ⟨⟨
  let
    fun antiquote-tr i (Const (@{syntax-const -antiquote}, -) $
      (t as Const (@{syntax-const -antiquote}, -) $ -)) = skip-antiquote-tr i t
    | antiquote-tr i (Const (@{syntax-const -antiquote}, -) $ t) =
      antiquote-tr i t $ Bound i
    | antiquote-tr i (t $ u) = antiquote-tr i t $ antiquote-tr i u
    | antiquote-tr i (Abs (x, T, t)) = Abs (x, T, antiquote-tr (i + 1) t)
    | antiquote-tr - a = a
  and skip-antiquote-tr i ((c as Const (@{syntax-const -antiquote}, -)) $ t) =
    c $ skip-antiquote-tr i t
    | skip-antiquote-tr i t = antiquote-tr i t;

```

```

    fun quote-tr [t] = Abs (s, dummyT, antiquote-tr 0 (Term.incr-boundvars 1 t))
      | quote-tr ts = raise TERM (quote-tr, ts);
  in [(@{syntax-const -quote}, quote-tr)] end
>>

```

basic examples

```

term <<a + b + c>>
term <<a + b + c + 'x + 'y + 1>>
term <<'(f w) + 'x>>
term <<f 'x 'y z>>

```

advanced examples

```

term <<<'x + 'y>>>
term <<<'x + 'y>> o 'f>>
term <<'(f o 'g)>>
term <<<'(f o 'g)>>>

```

end

18 Partial equivalence relations

theory PER imports Main begin

Higher-order quotients are defined over partial equivalence relations (PERs) instead of total ones. We provide axiomatic type classes *equiv* < *partial-equiv* and a type constructor *'a quot* with basic operations. This development is based on:

Oscar Slotosch: *Higher Order Quotients and their Implementation in Isabelle HOL*. Elsa L. Gunter and Amy Felty, editors, Theorem Proving in Higher Order Logics: TPHOLs '97, Springer LNCS 1275, 1997.

18.1 Partial equivalence

Type class *partial-equiv* models partial equivalence relations (PERs) using the polymorphic $\sim :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ relation, which is required to be symmetric and transitive, but not necessarily reflexive.

```

class partial-equiv =
  fixes eqv :: 'a => 'a => bool    (infixl ~ 50)
  assumes partial-equiv-sym [elim?]: x ~ y ==> y ~ x
  assumes partial-equiv-trans [trans]: x ~ y ==> y ~ z ==> x ~ z

```

The domain of a partial equivalence relation is the set of reflexive elements. Due to symmetry and transitivity this characterizes exactly those elements that are connected with *any* other one.

definition

```
domain :: 'a::partial-equiv set where
domain = {x. x ~ x}
```

lemma *domainI* [intro]: $x \sim x \implies x \in \text{domain}$
unfolding *domain-def* **by** *blast*

lemma *domainD* [dest]: $x \in \text{domain} \implies x \sim x$
unfolding *domain-def* **by** *blast*

theorem *domainI'* [elim?]: $x \sim y \implies x \in \text{domain}$
proof

```
  assume xy: x ~ y
  also from xy have y ~ x ..
  finally show x ~ x .
```

qed

18.2 Equivalence on function spaces

The \sim relation is lifted to function spaces. It is important to note that this is *not* the direct product, but a structural one corresponding to the congruence property.

instantiation *fun* :: (*partial-equiv*, *partial-equiv*) *partial-equiv*
begin

definition

```
eqv-fun-def: f ~ g ==  $\forall x \in \text{domain}. \forall y \in \text{domain}. x \sim y \implies f x \sim g y$ 
```

lemma *partial-equiv-funI* [intro?]:
 $(!!x y. x \in \text{domain} \implies y \in \text{domain} \implies x \sim y \implies f x \sim g y) \implies f \sim g$
unfolding *eqv-fun-def* **by** *blast*

lemma *partial-equiv-funD* [dest?]:
 $f \sim g \implies x \in \text{domain} \implies y \in \text{domain} \implies x \sim y \implies f x \sim g y$
unfolding *eqv-fun-def* **by** *blast*

The class of partial equivalence relations is closed under function spaces (in *both* argument positions).

instance proof

```
fix f g h :: 'a::partial-equiv => 'b::partial-equiv
assume fg: f ~ g
show g ~ f
proof
  fix x y :: 'a
  assume x: x ~ x and y: y ~ y
  assume x ~ y then have y ~ x ..
  with fg y x have f y ~ g x ..
  then show g x ~ f y ..
```

```

qed
assume gh:  $g \sim h$ 
show  $f \sim h$ 
proof
  fix  $x\ y :: 'a$ 
  assume  $x: x \in \text{domain}$  and  $y: y \in \text{domain}$  and  $x \sim y$ 
  with fg have  $f\ x \sim g\ y$  ..
  also from y have  $y \sim y$  ..
  with gh y y have  $g\ y \sim h\ y$  ..
  finally show  $f\ x \sim h\ y$  .
qed
qed

end

```

18.3 Total equivalence

The class of total equivalence relations on top of PERs. It coincides with the standard notion of equivalence, i.e. $\sim :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ is required to be reflexive, transitive and symmetric.

```

class equiv =
  assumes equiv-refl [intro]:  $x \sim x$ 

```

On total equivalences all elements are reflexive, and congruence holds unconditionally.

```

theorem equiv-domain [intro]:  $(x::'a::\text{equiv}) \in \text{domain}$ 
proof
  show  $x \sim x$  ..
qed

```

```

theorem equiv-cong [dest?]:  $f \sim g \implies x \sim y \implies f\ x \sim g\ (y::'a::\text{equiv})$ 
proof -
  assume  $f \sim g$ 
  moreover have  $x \in \text{domain}$  ..
  moreover have  $y \in \text{domain}$  ..
  moreover assume  $x \sim y$ 
  ultimately show ?thesis ..
qed

```

18.4 Quotient types

The quotient type $'a\ \text{quot}$ consists of all *equivalence classes* over elements of the base type $'a$.

```

typedef 'a quot =  $\{\{x. a \sim x\} \mid a::'a::\text{partial-equiv. True}\}$ 
  by blast

```

```

lemma quotI [intro]:  $\{x. a \sim x\} \in \text{quot}$ 

```

unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]: $R \in \text{quot} \implies (!a. R = \{x. a \sim x\} \implies C) \implies C$
unfolding *quot-def* **by** *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition

eqv-class :: (*a*::*partial-equiv*) => 'a *quot* ($\lfloor \cdot \rfloor$) **where**
 $\lfloor a \rfloor = \text{Abs-quot } \{x. a \sim x\}$

theorem *quot-rep*: $\exists a. A = \lfloor a \rfloor$

proof (*cases A*)

fix *R* **assume** *R*: $A = \text{Abs-quot } R$

assume $R \in \text{quot}$ **then have** $\exists a. R = \{x. a \sim x\}$ **by** *blast*

with *R* **have** $\exists a. A = \text{Abs-quot } \{x. a \sim x\}$ **by** *blast*

then show *?thesis* **by** (*unfold eqv-class-def*)

qed

lemma *quot-cases* [*cases type: quot*]:

obtains (*rep*) *a* **where** $A = \lfloor a \rfloor$

using *quot-rep* **by** *blast*

18.5 Equality on quotients

Equality of canonical quotient elements corresponds to the original relation as follows.

theorem *eqv-class-eqI* [*intro*]: $a \sim b \implies \lfloor a \rfloor = \lfloor b \rfloor$

proof –

assume *ab*: $a \sim b$

have $\{x. a \sim x\} = \{x. b \sim x\}$

proof (*rule Collect-cong*)

fix *x* **show** $(a \sim x) = (b \sim x)$

proof

from *ab* **have** $b \sim a$..

also assume $a \sim x$

finally show $b \sim x$.

next

note *ab*

also assume $b \sim x$

finally show $a \sim x$.

qed

qed

then show *?thesis* **by** (*simp only: eqv-class-def*)

qed

theorem *eqv-class-eqD'* [*dest?*]: $\lfloor a \rfloor = \lfloor b \rfloor \implies a \in \text{domain} \implies a \sim b$

proof (*unfold eqv-class-def*)

assume $Abs-quot \{x. a \sim x\} = Abs-quot \{x. b \sim x\}$
 then have $\{x. a \sim x\} = \{x. b \sim x\}$ by (*simp only: Abs-quot-inject quotI*)
 moreover assume $a \in domain$ then have $a \sim a$..
 ultimately have $a \in \{x. b \sim x\}$ by *blast*
 then have $b \sim a$ by *blast*
 then show $a \sim b$..
 qed

theorem *eqv-class-eqD* [*dest?*]: $\lfloor a \rfloor = \lfloor b \rfloor \implies a \sim (b::'a::equiv)$
proof (*rule eqv-class-eqD'*)
 show $a \in domain$..
 qed

lemma *eqv-class-eq'* [*simp*]: $a \in domain \implies (\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
 using *eqv-class-eqI eqv-class-eqD' by (blast del: eqv-refl)*

lemma *eqv-class-eq* [*simp*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim (b::'a::equiv))$
 using *eqv-class-eqI eqv-class-eqD by blast*

18.6 Picking representing elements

definition

pick :: $'a::partial-equiv \text{ quot} \implies 'a$ **where**
pick $A = (SOME\ a. A = \lfloor a \rfloor)$

theorem *pick-eqv'* [*intro?*, *simp*]: $a \in domain \implies pick\ \lfloor a \rfloor \sim a$
proof (*unfold pick-def*)
 assume $a: a \in domain$
 show $(SOME\ x. \lfloor a \rfloor = \lfloor x \rfloor) \sim a$
proof (*rule someI2*)
 show $\lfloor a \rfloor = \lfloor a \rfloor$..
 fix x assume $\lfloor a \rfloor = \lfloor x \rfloor$
 from *this* and a have $a \sim x$..
 then show $x \sim a$..
 qed
 qed

theorem *pick-eqv* [*intro*, *simp*]: $pick\ \lfloor a \rfloor \sim (a::'a::equiv)$
proof (*rule pick-eqv'*)
 show $a \in domain$..
 qed

theorem *pick-inverse*: $\lfloor pick\ A \rfloor = (A::'a::equiv \text{ quot})$
proof (*cases A*)
 fix a assume $a: A = \lfloor a \rfloor$
 then have $pick\ A \sim a$ by *simp*
 then have $\lfloor pick\ A \rfloor = \lfloor a \rfloor$ by *simp*
 with a show *?thesis* by *simp*
 qed

end

19 Summing natural numbers

theory *NatSum* **imports** *Main Parity* **begin**

Summing natural numbers, squares, cubes, etc.

Thanks to Sloane's On-Line Encyclopedia of Integer Sequences, <http://www.research.att.com/~njas/sequences/>.

lemmas [*simp*] =
 ring-distrib
 diff-mult-distrib diff-mult-distrib2 — for type nat

The sum of the first n odd numbers equals n squared.

lemma *sum-of-odds*: $(\sum i=0..<n. \text{Suc } (i + i)) = n * n$
by (*induct n*) *auto*

The sum of the first n odd squares.

lemma *sum-of-odd-squares*:
 $3 * (\sum i=0..<n. \text{Suc}(2*i) * \text{Suc}(2*i)) = n * (4 * n * n - 1)$
by (*induct n*) *auto*

The sum of the first n odd cubes

lemma *sum-of-odd-cubes*:
 $(\sum i=0..<n. \text{Suc } (2*i) * \text{Suc } (2*i) * \text{Suc } (2*i)) =$
 $n * n * (2 * n * n - 1)$
by (*induct n*) *auto*

The sum of the first n positive integers equals $n (n + 1) / 2$.

lemma *sum-of-naturals*:
 $2 * (\sum i=0..n. i) = n * \text{Suc } n$
by (*induct n*) *auto*

lemma *sum-of-squares*:
 $6 * (\sum i=0..n. i * i) = n * \text{Suc } n * \text{Suc } (2 * n)$
by (*induct n*) *auto*

lemma *sum-of-cubes*:
 $4 * (\sum i=0..n. i * i * i) = n * n * \text{Suc } n * \text{Suc } n$
by (*induct n*) *auto*

A cute identity:

```

lemma sum-squared:  $(\sum i=0..n. i)^2 = (\sum i=0..n::nat. i^3)$ 
proof(induct n)
  case 0 show ?case by simp
next
  case (Suc n)
  have  $(\sum i = 0..Suc\ n. i)^2 =$ 
     $(\sum i = 0..n. i^3) + (2 * (\sum i = 0..n. i) * (n+1) + (n+1)^2)$ 
    (is - = ?A + ?B)
    using Suc by(simp add:nat-number)
  also have ?B =  $(n+1)^3$ 
    using sum-of-naturals by(simp add:nat-number)
  also have ?A +  $(n+1)^3 = (\sum i=0..Suc\ n. i^3)$  by simp
  finally show ?case .
qed

```

Sum of fourth powers: three versions.

```

lemma sum-of-fourth-powers:
   $30 * (\sum i=0..n. i * i * i * i) =$ 
     $n * Suc\ n * Suc\ (2 * n) * (3 * n * n + 3 * n - 1)$ 
  apply (induct n)
  apply simp-all
  apply (case-tac n) — eliminates the subtraction
  apply (simp-all (no-asm-simp))
  done

```

Two alternative proofs, with a change of variables and much more subtraction, performed using the integers.

```

lemma int-sum-of-fourth-powers:
   $30 * int\ (\sum i=0..<m. i * i * i * i) =$ 
     $int\ m * (int\ m - 1) * (int\ (2 * m) - 1) *$ 
     $(int\ (3 * m * m) - int\ (3 * m) - 1)$ 
  by (induct m) (simp-all add: int-mult)

```

```

lemma of-nat-sum-of-fourth-powers:
   $30 * of-nat\ (\sum i=0..<m. i * i * i * i) =$ 
     $of-nat\ m * (of-nat\ m - 1) * (of-nat\ (2 * m) - 1) *$ 
     $(of-nat\ (3 * m * m) - of-nat\ (3 * m) - (1::int))$ 
  by (induct m) (simp-all add: of-nat-mult)

```

Sums of geometric series: 2, 3 and the general case.

```

lemma sum-of-2-powers:  $(\sum i=0..<n. 2^i) = 2^n - (1::nat)$ 
  by (induct n) (auto split: nat-diff-split)

```

```

lemma sum-of-3-powers:  $2 * (\sum i=0..<n. 3^i) = 3^n - (1::nat)$ 
  by (induct n) auto

```

```

lemma sum-of-powers:  $0 < k ==> (k - 1) * (\sum i=0..<n. k^i) = k^n - (1::nat)$ 
  by (induct n) auto

```

end

20 Three Divides Theorem

theory *ThreeDivides*
imports *Main LaTeXsugar*
begin

20.1 Abstract

The following document presents a proof of the Three Divides N theorem formalised in the Isabelle/Isar theorem proving system.

Theorem: 3 divides n if and only if 3 divides the sum of all digits in n .

Informal Proof: Take $n = \sum n_j * 10^j$ where n_j is the j 'th least significant digit of the decimal denotation of the number n and the sum ranges over all digits. Then

$$(n - \sum n_j) = \sum n_j * (10^j - 1)$$

We know $\forall j \ 3|(10^j - 1)$ and hence $3|LHS$, therefore

$$\forall n \ 3|n \iff 3|\sum n_j$$

□

20.2 Formal proof

20.2.1 Miscellaneous summation lemmas

If a divides $A \ x$ for all x then a divides any sum over terms of the form $(A \ x) * (P \ x)$ for arbitrary P .

lemma *div-sum:*

fixes $a::nat$ **and** $n::nat$

shows $\forall x. \ a \ dvd \ A \ x \implies a \ dvd \ (\sum x < n. \ A \ x * D \ x)$

proof (*induct n*)

case 0 **show** ?case **by** *simp*

next

case (*Suc n*)

from *Suc*

have $a \ dvd \ (A \ n * D \ n)$ **by** (*simp add: dvd-mult2*)

with *Suc*

have $a \ dvd \ ((\sum x < n. \ A \ x * D \ x) + (A \ n * D \ n))$ **by** (*simp add: dvd-add*)

thus ?case **by** *simp*

qed

20.2.2 Generalised Three Divides

This section solves a generalised form of the three divides problem. Here we show that for any sequence of numbers the theorem holds. In the next section we specialise this theorem to apply directly to the decimal expansion of the natural numbers.

Here we show that the first statement in the informal proof is true for all natural numbers. Note we are using $D\ i$ to denote the i 'th element in a sequence of numbers.

lemma *digit-diff-split*:

fixes $n::nat$ **and** $nd::nat$ **and** $x::nat$
shows $n = (\sum x \in \{..<nd\}. (D\ x) * ((10::nat) ^ x)) \implies$
 $(n - (\sum x < nd. (D\ x))) = (\sum x < nd. (D\ x) * (10 ^ x - 1))$
by (*simp add: sum-diff-distrib diff-mult-distrib2*)

Now we prove that 3 always divides numbers of the form $10^x - 1$.

lemma *three-divs-0*:

shows $(3::nat) \text{ dvd } (10^x - 1)$
proof (*induct x*)
case 0 **show** ?case **by** *simp*
next
case (*Suc n*)
let ?thr = $(3::nat)$
have ?thr *dvd* 9 **by** *simp*
moreover
have ?thr *dvd* $(10 * (10^n - 1))$ **by** (*rule dvd-mult*) (*rule Suc*)
hence ?thr *dvd* $(10^{n+1} - 10)$ **by** (*simp add: nat-distrib*)
ultimately
have ?thr *dvd* $((10^{n+1} - 10) + 9)$
by (*simp only: add-ac*) (*rule dvd-add*)
thus ?case **by** *simp*
qed

Expanding on the previous lemma and lemma *div-sum*.

lemma *three-divs-1*:

fixes $D :: nat \Rightarrow nat$
shows $3 \text{ dvd } (\sum x < nd. D\ x * (10^x - 1))$
by (*subst nat-mult-commute, rule div-sum*) (*simp add: three-divs-0 [simplified]*)

Using lemmas *digit-diff-split* and *three-divs-1* we now prove the following lemma.

lemma *three-divs-2*:

fixes $nd::nat$ **and** $D::nat \Rightarrow nat$
shows $3 \text{ dvd } ((\sum x < nd. (D\ x) * (10^x)) - (\sum x < nd. (D\ x)))$
proof –
from *three-divs-1* **have** $3 \text{ dvd } (\sum x < nd. D\ x * (10^x - 1))$.
thus ?thesis **by** (*simp only: digit-diff-split*)

qed

We now present the final theorem of this section. For any sequence of numbers (defined by a function D), we show that 3 divides the expansive sum $\sum (D\ x) * 10^x$ over x if and only if 3 divides the sum of the individual numbers $\sum D\ x$.

lemma *three-div-general*:

fixes $D :: nat \Rightarrow nat$

shows $(3\ dvd\ (\sum x < nd. D\ x * 10^x)) = (3\ dvd\ (\sum x < nd. D\ x))$

proof

have *mono*: $(\sum x < nd. D\ x) \leq (\sum x < nd. D\ x * 10^x)$

by (*rule setsum-mono*) *simp*

This lets us form the term $(\sum x < nd. D\ x * 10^x) - \text{setsum } D\ \{..<nd\}$

```
{
  assume 3 dvd (sum x < nd. D x)
  with three-divs-2 mono
  show 3 dvd (sum x < nd. D x * 10^x)
    by (blast intro: dvd-diffD)
}
{
  assume 3 dvd (sum x < nd. D x * 10^x)
  with three-divs-2 mono
  show 3 dvd (sum x < nd. D x)
    by (blast intro: dvd-diffD1)
}

```

qed

20.2.3 Three Divides Natural

This section shows that for all natural numbers we can generate a sequence of digits less than ten that represent the decimal expansion of the number. We then use the lemma *three-div-general* to prove our final theorem.

Definitions of length and digit sum.

This section introduces some functions to calculate the required properties of natural numbers. We then proceed to prove some properties of these functions.

The function *nlen* returns the number of digits in a natural number n .

fun *nlen* :: $nat \Rightarrow nat$

where

nlen 0 = 0

| *nlen* $x = 1 + \text{nlen } (x\ \text{div } 10)$

The function *sumdig* returns the sum of all digits in some number n .

definition

```

sumdig :: nat ⇒ nat where
sumdig n = (∑ x < nlen n. n div 10x mod 10)

```

Some properties of these functions follow.

```

lemma nlen-zero:
  0 = nlen x ⇒ x = 0
by (induct x rule: nlen.induct) auto

```

```

lemma nlen-suc:
  Suc m = nlen n ⇒ m = nlen (n div 10)
by (induct n rule: nlen.induct) simp-all

```

The following lemma is the principle lemma required to prove our theorem. It states that an expansion of some natural number n into a sequence of its individual digits is always possible.

```

lemma exp-exists:
  m = (∑ x<nlen m. (m div (10::nat)x mod 10) * 10x)
proof (induct nlen m arbitrary: m)
  case 0 thus ?case by (simp add: nlen-zero)
next
  case (Suc nd)
  obtain c where mexp: m = 10*(m div 10) + c ∧ c < 10
  and cdef: c = m mod 10 by simp
  show m = (∑ x<nlen m. m div 10x mod 10 * 10x)
  proof -
    from (Suc nd = nlen m)
    have nd = nlen (m div 10) by (rule nlen-suc)
    with Suc have
      m div 10 = (∑ x<nd. m div 10 div 10x mod 10 * 10x) by simp
    with mexp have
      m = 10*(∑ x<nd. m div 10 div 10x mod 10 * 10x) + c by simp
    also have
      ... = (∑ x<nd. m div 10 div 10x mod 10 * 10(x+1)) + c
      by (subst setsum-right-distrib) (simp add: mult-ac)
    also have
      ... = (∑ x<nd. m div 10(Suc x) mod 10 * 10(Suc x)) + c
      by (simp add: div-mult2-eq[symmetric])
    also have
      ... = (∑ x∈{Suc 0..Suc nd}. m div 10x mod 10 * 10x) + c
      by (simp only: setsum-shift-bounds-Suc-ivl)
      (simp add: atLeast0LessThan)
    also have
      ... = (∑ x<Suc nd. m div 10x mod 10 * 10x)
      by (simp add: atLeast0LessThan[symmetric] setsum-head-upt-Suc cdef)
    also note (Suc nd = nlen m)
    finally
    show m = (∑ x<nlen m. m div 10x mod 10 * 10x) .
  qed
qed

```

Final theorem.

We now combine the general theorem *three-div-general* and existence result of *exp-exists* to prove our final theorem.

```

theorem three-divides-nat:
  shows  $(3 \text{ dvd } n) = (3 \text{ dvd } \text{sumdig } n)$ 
proof (unfold sumdig-def)
  have  $n = (\sum x < n \text{ len } n. (n \text{ div } (10 :: \text{nat})^x \text{ mod } 10) * 10^x)$ 
    by (rule exp-exists)
  moreover
  have  $3 \text{ dvd } (\sum x < n \text{ len } n. (n \text{ div } (10 :: \text{nat})^x \text{ mod } 10) * 10^x) =$ 
     $(3 \text{ dvd } (\sum x < n \text{ len } n. n \text{ div } 10^x \text{ mod } 10))$ 
    by (rule three-div-general)
  ultimately
  show  $3 \text{ dvd } n = (3 \text{ dvd } (\sum x < n \text{ len } n. n \text{ div } 10^x \text{ mod } 10))$  by simp
qed

end

```

21 Higher-Order Logic: Intuitionistic predicate calculus problems

theory *Intuitionistic* **imports** *Main* **begin**

```

lemma  $(\sim\sim(P \& Q)) = ((\sim\sim P) \& (\sim\sim Q))$ 
by iprover

```

```

lemma  $\sim\sim((\sim P \longrightarrow Q) \longrightarrow (\sim P \longrightarrow \sim Q) \longrightarrow P)$ 
by iprover

```

```

lemma  $(\sim\sim(P \longrightarrow Q)) = (\sim\sim P \longrightarrow \sim\sim Q)$ 
by iprover

```

```

lemma  $(\sim\sim\sim P) = (\sim P)$ 
by iprover

```

```

lemma  $\sim\sim((P \longrightarrow Q \mid R) \longrightarrow (P \longrightarrow Q) \mid (P \longrightarrow R))$ 
by iprover

```

```

lemma  $(P = Q) = (Q = P)$ 
by iprover

```

lemma $((P \multimap (Q \mid (Q \multimap R))) \multimap R) \multimap R$
by *iprover*

lemma $((G \multimap A) \multimap J) \multimap D \multimap E \multimap (((H \multimap B) \multimap I) \multimap C \multimap J)$
 $\multimap (A \multimap H) \multimap F \multimap G \multimap (((C \multimap B) \multimap I) \multimap D) \multimap (A \multimap C)$
 $\multimap (((F \multimap A) \multimap B) \multimap I) \multimap E$
by *iprover*

lemma $P \multimap \sim\sim P$
by *iprover*

lemma $\sim\sim(\sim\sim P \multimap P)$
by *iprover*

lemma $\sim\sim P \ \& \ \sim\sim(P \multimap Q) \multimap \sim\sim Q$
by *iprover*

lemma $((P=Q) \multimap P \& Q \& R) \ \& \$
 $((Q=R) \multimap P \& Q \& R) \ \& \$
 $((R=P) \multimap P \& Q \& R) \multimap P \& Q \& R$
by *iprover*

lemma $((P=Q) \multimap P \& Q \& R \& S \& T) \ \& \$
 $((Q=R) \multimap P \& Q \& R \& S \& T) \ \& \$
 $((R=S) \multimap P \& Q \& R \& S \& T) \ \& \$
 $((S=T) \multimap P \& Q \& R \& S \& T) \ \& \$
 $((T=P) \multimap P \& Q \& R \& S \& T) \multimap P \& Q \& R \& S \& T$
by *iprover*

lemma $(ALL \ x. \ EX \ y. \ ALL \ z. \ p(x) \ \& \ q(y) \ \& \ r(z)) =$
 $(ALL \ z. \ EX \ y. \ ALL \ x. \ p(x) \ \& \ q(y) \ \& \ r(z))$
by (*iprover del: allE elim 2: allE'*)

lemma $\sim (EX \ x. \ ALL \ y. \ p \ y \ x = (\sim \ p \ x \ x))$
by *iprover*

lemma $\sim\sim((P \dashrightarrow Q) = (\sim Q \dashrightarrow \sim P))$
by *iprover*

lemma $\sim\sim(\sim\sim P = P)$
by *iprover*

lemma $\sim(P \dashrightarrow Q) \dashrightarrow (Q \dashrightarrow P)$
by *iprover*

lemma $\sim\sim((\sim P \dashrightarrow Q) = (\sim Q \dashrightarrow P))$
by *iprover*

lemma $\sim\sim((P|Q \dashrightarrow P|R) \dashrightarrow P|(Q \dashrightarrow R))$
by *iprover*

lemma $\sim\sim(P | \sim P)$
by *iprover*

lemma $\sim\sim(P | \sim\sim P)$
by *iprover*

lemma $\sim\sim(((P \dashrightarrow Q) \dashrightarrow P) \dashrightarrow P)$
by *iprover*

lemma $((P|Q) \& (\sim P|Q) \& (P|\sim Q)) \dashrightarrow \sim(\sim P | \sim Q)$
by *iprover*

lemma $(Q \dashrightarrow R) \dashrightarrow (R \dashrightarrow P \& Q) \dashrightarrow (P \dashrightarrow (Q|R)) \dashrightarrow (P=Q)$
by *iprover*

lemma $P=P$
by *iprover*

lemma $\sim\sim(((P = Q) = R) = (P = (Q = R)))$

$$\begin{array}{l}
\textbf{lemma } ((P = Q) = R) \quad \dashv\dashv \rightarrow \quad \sim\sim(P = (Q = R)) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } (P \mid (Q \& R)) = ((P \mid Q) \& (P \mid R)) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } \sim\sim((P = Q) = ((Q \mid \sim P) \& (\sim Q \mid P))) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } \sim\sim((P \dashv\dashv \rightarrow Q) = (\sim P \mid Q)) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } \sim\sim((P \dashv\dashv \rightarrow Q) \mid (Q \dashv\dashv \rightarrow P)) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } \sim\sim(((P \& (Q \dashv\dashv \rightarrow R)) \dashv\dashv \rightarrow S) = ((\sim P \mid Q \mid S) \& (\sim P \mid \sim R \mid S))) \\
\textbf{oops} \\[10pt]
\textbf{lemma } (P \& Q) = (P = (Q = (P \mid Q))) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } (EX x. P(x) \dashv\dashv \rightarrow Q) \quad \dashv\dashv \rightarrow \quad (ALL x. P(x)) \dashv\dashv \rightarrow Q \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } ((ALL x. P(x)) \dashv\dashv \rightarrow Q) \dashv\dashv \rightarrow \sim (ALL x. P(x) \& \sim Q) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } ((ALL x. \sim P(x)) \dashv\dashv \rightarrow Q) \dashv\dashv \rightarrow \sim (ALL x. \sim (P(x) \mid Q)) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } (ALL x. P(x)) \mid Q \dashv\dashv \rightarrow (ALL x. P(x) \mid Q) \\
\textbf{by } iprover \\[10pt]
\textbf{lemma } (EX x. P \dashv\dashv \rightarrow Q(x)) \dashv\dashv \rightarrow (P \dashv\dashv \rightarrow (EX x. Q(x))) \\
\textbf{by } iprover
\end{array}$$

lemma $\sim\sim(EX\ x.\ ALL\ y\ z.\ (P(y) \dashrightarrow Q(z)) \dashrightarrow (P(x) \dashrightarrow Q(x)))$
by *iprover*

lemma $(ALL\ x\ y.\ EX\ z.\ ALL\ w.\ (P(x) \& Q(y) \dashrightarrow R(z) \& S(w)))$
 $\dashrightarrow (EX\ x\ y.\ P(x) \& Q(y)) \dashrightarrow (EX\ z.\ R(z))$
by *iprover*

lemma $(EX\ x.\ P \dashrightarrow Q(x)) \& (EX\ x.\ Q(x) \dashrightarrow P) \dashrightarrow \sim\sim(EX\ x.\ P = Q(x))$
by *iprover*

lemma $(ALL\ x.\ P = Q(x)) \dashrightarrow (P = (ALL\ x.\ Q(x)))$
by *iprover*

lemma $\sim\sim((ALL\ x.\ P \mid Q(x)) = (P \mid (ALL\ x.\ Q(x))))$
by *iprover*

lemma $(EX\ x.\ P(x)) \&$
 $(ALL\ x.\ L(x) \dashrightarrow \sim(M(x) \& R(x))) \&$
 $(ALL\ x.\ P(x) \dashrightarrow (M(x) \& L(x))) \&$
 $((ALL\ x.\ P(x) \dashrightarrow Q(x)) \mid (EX\ x.\ P(x) \& R(x)))$
 $\dashrightarrow (EX\ x.\ Q(x) \& P(x))$
by *iprover*

lemma $(EX\ x.\ P(x) \& \sim Q(x)) \&$
 $(ALL\ x.\ P(x) \dashrightarrow R(x)) \&$
 $(ALL\ x.\ M(x) \& L(x) \dashrightarrow P(x)) \&$
 $((EX\ x.\ R(x) \& \sim Q(x)) \dashrightarrow (ALL\ x.\ L(x) \dashrightarrow \sim R(x)))$
 $\dashrightarrow (ALL\ x.\ M(x) \dashrightarrow \sim L(x))$
by *iprover*

lemma $(ALL\ x.\ P(x) \dashrightarrow (ALL\ x.\ Q(x))) \&$
 $(\sim\sim(ALL\ x.\ Q(x) \mid R(x)) \dashrightarrow (EX\ x.\ Q(x) \& S(x))) \&$
 $(\sim\sim(EX\ x.\ S(x)) \dashrightarrow (ALL\ x.\ L(x) \dashrightarrow M(x)))$
 $\dashrightarrow (ALL\ x.\ P(x) \& L(x) \dashrightarrow M(x))$
by *iprover*

lemma $((EX\ x.\ P(x)) \ \&\ (EX\ y.\ Q(y))) \dashv\vdash$
 $((ALL\ x.\ (P(x) \dashv\vdash R(x))) \ \&\ (ALL\ y.\ (Q(y) \dashv\vdash S(y)))) =$
 $(ALL\ x\ y.\ ((P(x) \ \&\ Q(y)) \dashv\vdash (R(x) \ \&\ S(y))))$
by *iprover*

lemma $(ALL\ x.\ (P(x) \mid Q(x)) \dashv\vdash \sim R(x)) \ \&$
 $(ALL\ x.\ (Q(x) \dashv\vdash \sim S(x)) \dashv\vdash P(x) \ \&\ R(x))$
 $\dashv\vdash (ALL\ x.\ \sim\sim S(x))$
by *iprover*

lemma $\sim(EX\ x.\ P(x) \ \&\ (Q(x) \mid R(x))) \ \&$
 $(EX\ x.\ L(x) \ \&\ P(x)) \ \&$
 $(ALL\ x.\ \sim R(x) \dashv\vdash M(x))$
 $\dashv\vdash (EX\ x.\ L(x) \ \&\ M(x))$
by *iprover*

lemma $(ALL\ x.\ P(x) \ \&\ (Q(x) \mid R(x)) \dashv\vdash S(x)) \ \&$
 $(ALL\ x.\ S(x) \ \&\ R(x) \dashv\vdash L(x)) \ \&$
 $(ALL\ x.\ M(x) \dashv\vdash R(x))$
 $\dashv\vdash (ALL\ x.\ P(x) \ \&\ M(x) \dashv\vdash L(x))$
by *iprover*

lemma $(ALL\ x.\ \sim\sim(P(a) \ \&\ (P(x) \dashv\vdash P(b)) \dashv\vdash P(c))) =$
 $(ALL\ x.\ \sim\sim((\sim P(a) \mid P(x) \mid P(c)) \ \&\ (\sim P(a) \mid \sim P(b) \mid P(c))))$
oops

lemma
 $(ALL\ x.\ EX\ y.\ J\ x\ y) \ \&$
 $(ALL\ x.\ EX\ y.\ G\ x\ y) \ \&$
 $(ALL\ x\ y.\ J\ x\ y \mid G\ x\ y \dashv\vdash (ALL\ z.\ J\ y\ z \mid G\ y\ z \dashv\vdash H\ x\ z))$
 $\dashv\vdash (ALL\ x.\ EX\ y.\ H\ x\ y)$
by *iprover*

lemma $\sim(EX\ x.\ ALL\ y.\ F\ y\ x = (\sim F\ y\ y))$
by *iprover*

lemma $(EX\ y.\ ALL\ x.\ F\ x\ y = F\ x\ x) \dashv\vdash$
 $\sim(ALL\ x.\ EX\ y.\ ALL\ z.\ F\ z\ y = (\sim F\ z\ x))$
by *iprover*


```

lemma ( $ALL\ x.\ f(x) \dashv\vdash$ 
  ( $EX\ y.\ g(y) \ \&\ h\ x\ y \ \&\ (EX\ y.\ g(y) \ \&\ \sim\ h\ x\ y)$ )  $\ \&$ 
  ( $EX\ x.\ j(x) \ \&\ (ALL\ y.\ g(y) \dashv\vdash\ h\ x\ y)$ )
   $\dashv\vdash\ (EX\ x.\ j(x) \ \&\ \sim f(x))$ )
by iprover

lemma ( $a=b \mid c=d \ \&\ (a=c \mid b=d) \dashv\vdash a=d \mid b=c$ )
by iprover

lemma ( $(EX\ z\ w.\ (ALL\ x\ y.\ (P\ x\ y = ((x = z) \ \&\ (y = w)))) \dashv\vdash$ 
  ( $EX\ z.\ (ALL\ x.\ (EX\ w.\ ((ALL\ y.\ (P\ x\ y = (y = w))) = (x = z))))$ )
by iprover

lemma ( $(EX\ z\ w.\ (ALL\ x\ y.\ (P\ x\ y = ((x = z) \ \&\ (y = w)))) \dashv\vdash$ 
  ( $EX\ w.\ (ALL\ y.\ (EX\ z.\ ((ALL\ x.\ (P\ x\ y = (x = z))) = (y = w))))$ )
by iprover

lemma ( $ALL\ x.\ (EX\ y.\ P(y) \ \&\ x=f(y)) \dashv\vdash P(x) = (ALL\ x.\ P(x) \dashv\vdash$ 
   $P(f(x)))$ )
by iprover

lemma  $P\ (f\ a\ b)\ (f\ b\ c) \ \&\ P\ (f\ b\ c)\ (f\ a\ c) \ \&$ 
  ( $ALL\ x\ y\ z.\ P\ x\ y \ \&\ P\ y\ z \dashv\vdash P\ x\ z \dashv\vdash P\ (f\ a\ b)\ (f\ a\ c)$ )
by iprover

lemma  $ALL\ x.\ P\ x\ (f\ x) = (EX\ y.\ (ALL\ z.\ P\ z\ y \dashv\vdash P\ z\ (f\ x)) \ \&\ P\ x\ y)$ 
by iprover

end

```

22 CTL formulae

theory *CTL* **imports** *Main* **begin**

We formalize basic concepts of Computational Tree Logic (CTL) [3, 2] within the simply-typed set theory of HOL.

By using the common technique of “shallow embedding”, a CTL formula is identified with the corresponding set of states where it holds. Consequently, CTL operations such as negation, conjunction, disjunction simply become complement, intersection, union of sets. We only require a separate oper-

ation for implication, as point-wise inclusion is usually not encountered in plain set-theory.

lemmas [intro!] = *Int-greatest Un-upper2 Un-upper1 Int-lower1 Int-lower2*

types 'a ctl = 'a set

definition

imp :: 'a ctl \Rightarrow 'a ctl \Rightarrow 'a ctl (infixr \rightarrow 75) **where**
 $p \rightarrow q = - p \cup q$

lemma [intro!]: $p \cap p \rightarrow q \subseteq q$ **unfolding** *imp-def* **by** *auto*

lemma [intro!]: $p \subseteq (q \rightarrow p)$ **unfolding** *imp-def* **by** *rule*

The CTL path operators are more interesting; they are based on an arbitrary, but fixed model \mathcal{M} , which is simply a transition relation over states 'a.

axiomatization $\mathcal{M} :: ('a \times 'a) \text{ set}$

The operators EX, EF, EG are taken as primitives, while AX, AF, AG are defined as derived ones. The formula EX p holds in a state s , iff there is a successor state s' (with respect to the model \mathcal{M}), such that p holds in s' . The formula EF p holds in a state s , iff there is a path in \mathcal{M} , starting from s , such that there exists a state s' on the path, such that p holds in s' . The formula EG p holds in a state s , iff there is a path, starting from s , such that for all states s' on the path, p holds in s' . It is easy to see that EF p and EG p may be expressed using least and greatest fixed points [3].

definition

EX (EX - [80] 90) **where** EX $p = \{s. \exists s'. (s, s') \in \mathcal{M} \wedge s' \in p\}$

definition

EF (EF - [80] 90) **where** EF $p = lfp (\lambda s. p \cup EX s)$

definition

EG (EG - [80] 90) **where** EG $p = gfp (\lambda s. p \cap EX s)$

AX, AF and AG are now defined dually in terms of EX, EF and EG.

definition

AX (AX - [80] 90) **where** AX $p = - EX - p$

definition

AF (AF - [80] 90) **where** AF $p = - EG - p$

definition

AG (AG - [80] 90) **where** AG $p = - EF - p$

lemmas [simp] = *EX-def EG-def AX-def EF-def AF-def AG-def*

22.1 Basic fixed point properties

First of all, we use the de-Morgan property of fixed points

```

lemma lfp-gfp:  $\text{lfp } f = - \text{gfp } (\lambda s. :a \text{ set. } - (f (- s)))$ 
proof
  show  $\text{lfp } f \subseteq - \text{gfp } (\lambda s. - f (- s))$ 
  proof
    fix  $x$  assume  $l: x \in \text{lfp } f$ 
    show  $x \in - \text{gfp } (\lambda s. - f (- s))$ 
    proof
      assume  $x \in \text{gfp } (\lambda s. - f (- s))$ 
      then obtain  $u$  where  $x \in u$  and  $u \subseteq - f (- u)$ 
      by (auto simp add: gfp-def)
      then have  $f (- u) \subseteq - u$  by auto
      then have  $\text{lfp } f \subseteq - u$  by (rule lfp-lowerbound)
      from  $l$  and this have  $x \notin u$  by auto
      with  $(x \in u)$  show False by contradiction
    qed
  qed
  show  $- \text{gfp } (\lambda s. - f (- s)) \subseteq \text{lfp } f$ 
  proof (rule lfp-greatest)
    fix  $u$  assume  $f u \subseteq u$ 
    then have  $- u \subseteq - f u$  by auto
    then have  $- u \subseteq - f (- (- u))$  by simp
    then have  $- u \subseteq \text{gfp } (\lambda s. - f (- s))$  by (rule gfp-upperbound)
    then show  $- \text{gfp } (\lambda s. - f (- s)) \subseteq u$  by auto
  qed
qed

lemma lfp-gfp':  $-\text{lfp } f = \text{gfp } (\lambda s. :a \text{ set. } - (f (- s)))$ 
by (simp add: lfp-gfp)

lemma gfp-lfp':  $-\text{gfp } f = \text{lfp } (\lambda s. :a \text{ set. } - (f (- s)))$ 
by (simp add: lfp-gfp)

```

in order to give dual fixed point representations of AF p and AG p :

```

lemma AF-lfp:  $\text{AF } p = \text{lfp } (\lambda s. p \cup \text{AX } s)$  by (simp add: lfp-gfp)
lemma AG-gfp:  $\text{AG } p = \text{gfp } (\lambda s. p \cap \text{AX } s)$  by (simp add: lfp-gfp)

lemma EF-fp:  $\text{EF } p = p \cup \text{EX } \text{EF } p$ 
proof -
  have mono  $(\lambda s. p \cup \text{EX } s)$  by rule (auto simp add: EX-def)
  then show ?thesis by (simp only: EF-def) (rule lfp-unfold)
qed

lemma AF-fp:  $\text{AF } p = p \cup \text{AX } \text{AF } p$ 
proof -
  have mono  $(\lambda s. p \cup \text{AX } s)$  by rule (auto simp add: AX-def EX-def)
  then show ?thesis by (simp only: AF-lfp) (rule lfp-unfold)
qed

lemma EG-fp:  $\text{EG } p = p \cap \text{EX } \text{EG } p$ 

```

proof –
 have *mono* ($\lambda s. p \cap \text{EX } s$) **by** *rule* (*auto simp add: EX-def*)
 then **show** *?thesis* **by** (*simp only: EG-def*) (*rule gfp-unfold*)
qed

From the greatest fixed point definition of $\text{AG } p$, we derive as a consequence of the Knaster-Tarski theorem on the one hand that $\text{AG } p$ is a fixed point of the monotonic function $\lambda s. p \cap \text{AX } s$.

lemma *AG-fp*: $\text{AG } p = p \cap \text{AX } \text{AG } p$

proof –
 have *mono* ($\lambda s. p \cap \text{AX } s$) **by** *rule* (*auto simp add: AX-def EX-def*)
 then **show** *?thesis* **by** (*simp only: AG-gfp*) (*rule gfp-unfold*)
qed

This fact may be split up into two inequalities (merely using transitivity of \subseteq , which is an instance of the overloaded \leq in Isabelle/HOL).

lemma *AG-fp-1*: $\text{AG } p \subseteq p$

proof –
 note *AG-fp* **also have** $p \cap \text{AX } \text{AG } p \subseteq p$ **by** *auto*
 finally **show** *?thesis* .
qed

lemma *AG-fp-2*: $\text{AG } p \subseteq \text{AX } \text{AG } p$

proof –
 note *AG-fp* **also have** $p \cap \text{AX } \text{AG } p \subseteq \text{AX } \text{AG } p$ **by** *auto*
 finally **show** *?thesis* .
qed

On the other hand, we have from the Knaster-Tarski fixed point theorem that any other post-fixed point of $\lambda s. p \cap \text{AX } s$ is smaller than $\text{AG } p$. A post-fixed point is a set of states q such that $q \subseteq p \cap \text{AX } q$. This leads to the following co-induction principle for $\text{AG } p$.

lemma *AG-I*: $q \subseteq p \cap \text{AX } q \implies q \subseteq \text{AG } p$
by (*simp only: AG-gfp*) (*rule gfp-upperbound*)

22.2 The tree induction principle

With the most basic facts available, we are now able to establish a few more interesting results, leading to the *tree induction* principle for AG (see below). We will use some elementary monotonicity and distributivity rules.

lemma *AX-int*: $\text{AX } (p \cap q) = \text{AX } p \cap \text{AX } q$ **by** *auto*

lemma *AX-mono*: $p \subseteq q \implies \text{AX } p \subseteq \text{AX } q$ **by** *auto*

lemma *AG-mono*: $p \subseteq q \implies \text{AG } p \subseteq \text{AG } q$
by (*simp only: AG-gfp, rule gfp-mono*) *auto*

The formula $\text{AG } p$ implies $\text{AX } p$ (we use substitution of \subseteq with monotonicity).

lemma *AG-AX*: $\text{AG } p \subseteq \text{AX } p$
proof –
 have $\text{AG } p \subseteq \text{AX AG } p$ **by** (*rule AG-fp-2*)
 also have $\text{AG } p \subseteq p$ **by** (*rule AG-fp-1*) **moreover note** *AX-mono*
 finally show *?thesis* .
qed

Furthermore we show idempotency of the AG operator. The proof is a good example of how accumulated facts may get used to feed a single rule step.

lemma *AG-AG*: $\text{AG AG } p = \text{AG } p$
proof
 show $\text{AG AG } p \subseteq \text{AG } p$ **by** (*rule AG-fp-1*)
next
 show $\text{AG } p \subseteq \text{AG AG } p$
 proof (*rule AG-I*)
 have $\text{AG } p \subseteq \text{AG } p$..
 moreover have $\text{AG } p \subseteq \text{AX AG } p$ **by** (*rule AG-fp-2*)
 ultimately show $\text{AG } p \subseteq \text{AG } p \cap \text{AX AG } p$..
 qed
qed

We now give an alternative characterization of the AG operator, which describes the AG operator in an “operational” way by tree induction: In a state holds AG p iff in that state holds p , and in all reachable states s follows from the fact that p holds in s , that p also holds in all successor states of s . We use the co-induction principle *AG-I* to establish this in a purely algebraic manner.

theorem *AG-induct*: $p \cap \text{AG } (p \rightarrow \text{AX } p) = \text{AG } p$
proof
 show $p \cap \text{AG } (p \rightarrow \text{AX } p) \subseteq \text{AG } p$ (**is** *?lhs* \subseteq -)
 proof (*rule AG-I*)
 show *?lhs* $\subseteq p \cap \text{AX } ?lhs$
 proof
 show *?lhs* $\subseteq p$..
 show *?lhs* $\subseteq \text{AX } ?lhs$
 proof –
 {
 have $\text{AG } (p \rightarrow \text{AX } p) \subseteq p \rightarrow \text{AX } p$ **by** (*rule AG-fp-1*)
 moreover have $p \cap p \rightarrow \text{AX } p \subseteq \text{AX } p$..
 ultimately have *?lhs* $\subseteq \text{AX } p$ **by** *auto*
 }
 moreover
 {
 have $p \cap \text{AG } (p \rightarrow \text{AX } p) \subseteq \text{AG } (p \rightarrow \text{AX } p)$..
 also have $\dots \subseteq \text{AX } \dots$ **by** (*rule AG-fp-2*)
 finally have *?lhs* $\subseteq \text{AX AG } (p \rightarrow \text{AX } p)$.
 }
 ultimately have *?lhs* $\subseteq \text{AX } p \cap \text{AX AG } (p \rightarrow \text{AX } p)$

```

      by (rule Int-greatest)
    also have ... = AX ?lhs by (simp only: AX-int)
    finally show ?thesis .
  qed
qed
qed
next
show AG p ⊆ p ∩ AG (p → AX p)
proof
  show AG p ⊆ p by (rule AG-fp-1)
  show AG p ⊆ AG (p → AX p)
  proof -
    have AG p = AG AG p by (simp only: AG-AG)
    also have AG p ⊆ AX p by (rule AG-AX) moreover note AG-mono
    also have AX p ⊆ (p → AX p) .. moreover note AG-mono
    finally show ?thesis .
  qed
qed
qed

```

22.3 An application of tree induction

Further interesting properties of CTL expressions may be demonstrated with the help of tree induction; here we show that AX and AG commute.

theorem *AG-AX-commute*: AG AX p = AX AG p

```

proof -
  have AG AX p = AX p ∩ AX AG AX p by (rule AG-fp)
  also have ... = AX (p ∩ AG AX p) by (simp only: AX-int)
  also have p ∩ AG AX p = AG p (is ?lhs = -)
  proof
    have AX p ⊆ p → AX p ..
    also have p ∩ AG (p → AX p) = AG p by (rule AG-induct)
    also note Int-mono AG-mono
    ultimately show ?lhs ⊆ AG p by fast
  next
    have AG p ⊆ p by (rule AG-fp-1)
    moreover
    {
      have AG p = AG AG p by (simp only: AG-AG)
      also have AG p ⊆ AX p by (rule AG-AX)
      also note AG-mono
      ultimately have AG p ⊆ AG AX p .
    }
    ultimately show AG p ⊆ ?lhs ..
  qed
  finally show ?thesis .
qed
end

```

23 Arithmetic

```
theory Arith-Examples
imports Main
begin
```

The *arith* method is used frequently throughout the Isabelle distribution. This file merely contains some additional tests and special corner cases. Some rather technical remarks:

`Lin_Arith.simple_tac` is a very basic version of the tactic. It performs no meta-to-object-logic conversion, and only some splitting of operators. `Lin_Arith.tac` performs meta-to-object-logic conversion, full splitting of operators, and NNF normalization of the goal. The *arith* method combines them both, and tries other methods (e.g. *presburger*) as well. This is the one that you should use in your proofs!

An *arith*-based simproc is available as well (see `Lin_Arith.simproc`), which—for performance reasons—however does even less splitting than `Lin_Arith.simple_tac` at the moment (namely inequalities only). (On the other hand, it does take apart conjunctions, which `Lin_Arith.simple_tac` currently does not do.)

23.1 Splitting of Operators: *max*, *min*, *abs*, *op −*, *nat*, *op mod*, *op div*

```
lemma (i::nat) <= max i j
  by linarith
```

```
lemma (i::int) <= max i j
  by linarith
```

```
lemma min i j <= (i::nat)
  by linarith
```

```
lemma min i j <= (i::int)
  by linarith
```

```
lemma min (i::nat) j <= max i j
  by linarith
```

```
lemma min (i::int) j <= max i j
  by linarith
```

```
lemma min (i::nat) j + max i j = i + j
  by linarith
```

```
lemma min (i::int) j + max i j = i + j
```

```

    by linarith

lemma (i::nat) < j ==> min i j < max i j
  by linarith

lemma (i::int) < j ==> min i j < max i j
  by linarith

lemma (0::int) <= abs i
  by linarith

lemma (i::int) <= abs i
  by linarith

lemma abs (abs (i::int)) = abs i
  by linarith

Also testing subgoals with bound variables.

lemma !!x. (x::nat) <= y ==> x - y = 0
  by linarith

lemma !!x. (x::nat) - y = 0 ==> x <= y
  by linarith

lemma !!x. ((x::nat) <= y) = (x - y = 0)
  by linarith

lemma [| (x::nat) < y; d < 1 |] ==> x - y = d
  by linarith

lemma [| (x::nat) < y; d < 1 |] ==> x - y - x = d - x
  by linarith

lemma (x::int) < y ==> x - y < 0
  by linarith

lemma nat (i + j) <= nat i + nat j
  by linarith

lemma i < j ==> nat (i - j) = 0
  by linarith

lemma (i::nat) mod 0 = i

  apply (subst nat-numeral-0-eq-0 [symmetric])
  by linarith

lemma (i::nat) mod 1 = 0

```



```

apply (subst nat-numeral-1-eq-1 [symmetric])
by linarith

lemma (i::nat) mod 42 <= 41
by linarith

lemma (i::int) mod 0 = i

apply (subst numeral-0-eq-0 [symmetric])
by linarith

lemma (i::int) mod 1 = 0

apply (subst numeral-1-eq-1 [symmetric])

apply (tactic << Lin-Arith.pre-tac @{simpset} 1 >>)
oops

lemma (i::int) mod 42 <= 41

apply (tactic << Lin-Arith.pre-tac @{simpset} 1 >>)
oops

lemma -(i::int) * 1 = 0 ==> i = 0
by linarith

lemma [| (0::int) < abs i; abs i * 1 < abs i * j |] ==> 1 < abs i * j
by linarith

```

23.2 Meta-Logic

```

lemma x < Suc y == x <= y
by linarith

lemma ((x::nat) == z ==> x ~ y) ==> x ~ y | z ~ y
by linarith

```

23.3 Various Other Examples

```

lemma (x < Suc y) = (x <= y)
by linarith

lemma [| (x::nat) < y; y < z |] ==> x < z
by linarith

lemma (x::nat) < y & y < z ==> x < z
by linarith

```

This example involves no arithmetic at all, but is solved by preprocessing (i.e. NNF normalization) alone.

lemma $(P::\text{bool}) = Q \implies Q = P$
by *linarith*

lemma $[| P = (x = 0); (\sim P) = (y = 0) |] \implies \min (x::\text{nat}) y = 0$
by *linarith*

lemma $[| P = (x = 0); (\sim P) = (y = 0) |] \implies \max (x::\text{nat}) y = x + y$
by *linarith*

lemma $[| (x::\text{nat}) \sim = y; a + 2 = b; a < y; y < b; a < x; x < b |] \implies \text{False}$
by *linarith*

lemma $[| (x::\text{nat}) > y; y > z; z > x |] \implies \text{False}$
by *linarith*

lemma $(x::\text{nat}) - 5 > y \implies y < x$
by *linarith*

lemma $(x::\text{nat}) \sim = 0 \implies 0 < x$
by *linarith*

lemma $[| (x::\text{nat}) \sim = y; x \leq y |] \implies x < y$
by *linarith*

lemma $[| (x::\text{nat}) < y; P (x - y) |] \implies P 0$
by *linarith*

lemma $(x - y) - (x::\text{nat}) = (x - x) - y$
by *linarith*

lemma $[| (a::\text{nat}) < b; c < d |] \implies (a - b) = (c - d)$
by *linarith*

lemma $((a::\text{nat}) - (b - (c - (d - e)))) = (a - (b - (c - (d - e))))$
by *linarith*

lemma $(n < m \ \& \ m < n') \mid (n < m \ \& \ m = n') \mid (n < n' \ \& \ n' < m) \mid$
 $(n = n' \ \& \ n' < m) \mid (n = m \ \& \ m < n') \mid$
 $(n' < m \ \& \ m < n) \mid (n' < m \ \& \ m = n) \mid$
 $(n' < n \ \& \ n < m) \mid (n' = n \ \& \ n < m) \mid (n' = m \ \& \ m < n) \mid$
 $(m < n \ \& \ n < n') \mid (m < n \ \& \ n' = n) \mid (m < n' \ \& \ n' < n) \mid$
 $(m = n \ \& \ n < n') \mid (m = n' \ \& \ n' < n) \mid$
 $(n' = m \ \& \ m = (n::\text{nat}))$

oops

```
lemma  $2 * (x::nat) \sim= 1$ 
```

oops

Constants.

```
lemma  $(0::nat) < 1$   
  by linarith
```

```
lemma  $(0::int) < 1$   
  by linarith
```

```
lemma  $(47::nat) + 11 < 08 * 15$   
  by linarith
```

```
lemma  $(47::int) + 11 < 08 * 15$   
  by linarith
```

Splitting of inequalities of different type.

```
lemma  $[(a::nat) \sim= b; (i::int) \sim= j; a < 2; b < 2] ==>$   
   $a + b \leq nat (max (abs i) (abs j))$   
  by linarith
```

Again, but different order.

```
lemma  $[(i::int) \sim= j; (a::nat) \sim= b; a < 2; b < 2] ==>$   
   $a + b \leq nat (max (abs i) (abs j))$   
  by linarith
```

end

24 Binary trees

theory *BT* **imports** *Main* **begin**

```
datatype 'a bt =  
  Lf  
  | Br 'a 'a bt 'a bt
```

consts

```
n-nodes  :: 'a bt => nat  
n-leaves :: 'a bt => nat  
depth    :: 'a bt => nat  
reflect  :: 'a bt => 'a bt
```

$bt_map \quad :: ('a \Rightarrow 'b) \Rightarrow ('a \rightarrow bt \Rightarrow 'b \rightarrow bt)$
 $preorder \quad :: 'a \rightarrow bt \Rightarrow 'a \rightarrow list$
 $inorder \quad \quad :: 'a \rightarrow bt \Rightarrow 'a \rightarrow list$
 $postorder \quad :: 'a \rightarrow bt \Rightarrow 'a \rightarrow list$
 $append \quad \quad :: 'a \rightarrow bt \Rightarrow 'a \rightarrow bt \Rightarrow 'a \rightarrow bt$

primrec

$n_nodes \ Lf = 0$
 $n_nodes \ (Br \ a \ t1 \ t2) = Suc \ (n_nodes \ t1 + n_nodes \ t2)$

primrec

$n_leaves \ Lf = Suc \ 0$
 $n_leaves \ (Br \ a \ t1 \ t2) = n_leaves \ t1 + n_leaves \ t2$

primrec

$depth \ Lf = 0$
 $depth \ (Br \ a \ t1 \ t2) = Suc \ (max \ (depth \ t1) \ (depth \ t2))$

primrec

$reflect \ Lf = Lf$
 $reflect \ (Br \ a \ t1 \ t2) = Br \ a \ (reflect \ t2) \ (reflect \ t1)$

primrec

$bt_map \ f \ Lf = Lf$
 $bt_map \ f \ (Br \ a \ t1 \ t2) = Br \ (f \ a) \ (bt_map \ f \ t1) \ (bt_map \ f \ t2)$

primrec

$preorder \ Lf = []$
 $preorder \ (Br \ a \ t1 \ t2) = [a] @ (preorder \ t1) @ (preorder \ t2)$

primrec

$inorder \ Lf = []$
 $inorder \ (Br \ a \ t1 \ t2) = (inorder \ t1) @ [a] @ (inorder \ t2)$

primrec

$postorder \ Lf = []$
 $postorder \ (Br \ a \ t1 \ t2) = (postorder \ t1) @ (postorder \ t2) @ [a]$

primrec

$append \ Lf \ t = t$
 $append \ (Br \ a \ t1 \ t2) \ t = Br \ a \ (append \ t1 \ t) \ (append \ t2 \ t)$

BT simplification

lemma $n_leaves_reflect$: $n_leaves \ (reflect \ t) = n_leaves \ t$
apply $(induct \ t)$
apply $auto$
done

lemma $n_nodes_reflect$: $n_nodes \ (reflect \ t) = n_nodes \ t$

```

apply (induct t)
apply auto
done

```

```

lemma depth-reflect: depth (reflect t) = depth t
apply (induct t)
apply auto
done

```

The famous relationship between the numbers of leaves and nodes.

```

lemma n-leaves-nodes: n-leaves t = Suc (n-nodes t)
apply (induct t)
apply auto
done

```

```

lemma reflect-reflect-ident: reflect (reflect t) = t
apply (induct t)
apply auto
done

```

```

lemma bt-map-reflect: bt-map f (reflect t) = reflect (bt-map f t)
apply (induct t)
apply simp-all
done

```

```

lemma preorder-bt-map: preorder (bt-map f t) = map f (preorder t)
apply (induct t)
apply simp-all
done

```

```

lemma inorder-bt-map: inorder (bt-map f t) = map f (inorder t)
apply (induct t)
apply simp-all
done

```

```

lemma postorder-bt-map: postorder (bt-map f t) = map f (postorder t)
apply (induct t)
apply simp-all
done

```

```

lemma depth-bt-map [simp]: depth (bt-map f t) = depth t
apply (induct t)
apply simp-all
done

```

```

lemma n-leaves-bt-map [simp]: n-leaves (bt-map f t) = n-leaves t
apply (induct t)
apply (simp-all add: left-distrib)
done

```

```

lemma preorder-reflect: preorder (reflect t) = rev (postorder t)
  apply (induct t)
  apply simp-all
done

```

```

lemma inorder-reflect: inorder (reflect t) = rev (inorder t)
  apply (induct t)
  apply simp-all
done

```

```

lemma postorder-reflect: postorder (reflect t) = rev (preorder t)
  apply (induct t)
  apply simp-all
done

```

Analogues of the standard properties of the append function for lists.

```

lemma append-assoc [simp]:
  append (append t1 t2) t3 = append t1 (append t2 t3)
  apply (induct t1)
  apply simp-all
done

```

```

lemma append-Lf2 [simp]: append t Lf = t
  apply (induct t)
  apply simp-all
done

```

```

lemma depth-append [simp]: depth (append t1 t2) = depth t1 + depth t2
  apply (induct t1)
  apply (simp-all add: max-add-distrib-left)
done

```

```

lemma n-leaves-append [simp]:
  n-leaves (append t1 t2) = n-leaves t1 * n-leaves t2
  apply (induct t1)
  apply (simp-all add: left-distrib)
done

```

```

lemma bt-map-append:
  bt-map f (append t1 t2) = append (bt-map f t1) (bt-map f t2)
  apply (induct t1)
  apply simp-all
done

```

end

25 2-3 Trees

```
theory Tree23
imports Main
begin
```

This is a very direct translation of some of the functions in table.ML in the Isabelle source code. That source is due to Makarius Wenzel and Stefan Berghofer.

So far this file contains only data types and functions, but no proofs. Feel free to have a go at the latter!

Note that because of complicated patterns and mutual recursion, these function definitions take a few minutes and can also be seen as stress tests for the function definition facility.

types *key* = *int* — for simplicity, should be a type class

datatype *ord* = *LESS* | *EQUAL* | *GREATER*

definition *ord i j* = (if *i*<*j* then *LESS* else if *i*=*j* then *EQUAL* else *GREATER*)

datatype 'a *tree23* =
 Empty |
 Branch2 'a *tree23* *key* * 'a 'a *tree23* |
 Branch3 'a *tree23* *key* * 'a 'a *tree23* *key* * 'a 'a *tree23*

datatype 'a *growth* =
 Stay 'a *tree23* |
 Sprout 'a *tree23* *key* * 'a 'a *tree23*

fun *add* :: *key* ⇒ 'a ⇒ 'a *tree23* ⇒ 'a *growth* **where**
add key y Empty = *Sprout Empty (key,y) Empty* |
add key y (Branch2 left (k,x) right) =
 (case *ord key k* of
 LESS =>
 (case *add key y left* of
 Stay left' => *Stay (Branch2 left' (k,x) right)*
 | *Sprout left1 q left2*
 => *Stay (Branch3 left1 q left2 (k,x) right)*)
 | *EQUAL* => *Stay (Branch2 left (k,y) right)*
 | *GREATER* =>
 (case *add key y right* of
 Stay right' => *Stay (Branch2 left (k,x) right')*
 | *Sprout right1 q right2*
 => *Stay (Branch3 left (k,x) right1 q right2)))* |
add key y (Branch3 left (k1,x1) mid (k2,x2) right) =
 (case *ord key k1* of
 LESS =>
 (case *add key y left* of

```

    Stay left' => Stay (Branch3 left' (k1,x1) mid (k2,x2) right)
  | Sprout left1 q left2
    => Sprout (Branch2 left1 q left2) (k1,x1) (Branch2 mid (k2,x2) right))
  | EQUAL => Stay (Branch3 left (k1,y) mid (k2,x2) right)
  | GREATER =>
    (case ord key k2 of
      LESS =>
        (case add key y mid of
          Stay mid' => Stay (Branch3 left (k1,x1) mid' (k2,x2) right)
        | Sprout mid1 q mid2
          => Sprout (Branch2 left (k1,x1) mid1) q (Branch2 mid2 (k2,x2)
right))
        | EQUAL => Stay (Branch3 left (k1,x1) mid (k2,y) right)
        | GREATER =>
          (case add key y right of
            Stay right' => Stay (Branch3 left (k1,x1) mid (k2,x2) right')
          | Sprout right1 q right2
            => Sprout (Branch2 left (k1,x1) mid) (k2,x2) (Branch2 right1 q
right2))))))

```

definition *add0* :: *key* \Rightarrow '*a* \Rightarrow '*a tree23* \Rightarrow '*a tree23* **where**

```

add0 k y t =
  (case add k y t of Stay t' => t' | Sprout l p r => Branch2 l p r)

```

value *add0 5 e (add0 4 d (add0 3 c (add0 2 b (add0 1 a Empty))))*

fun *compare* **where**

```

compare None (k2, -) = LESS |
compare (Some k1) (k2, -) = ord k1 k2

```

fun *if-eq* **where**

```

if-eq EQUAL x y = x |
if-eq - x y = y

```

fun *del* :: *key* *option* \Rightarrow '*a tree23* \Rightarrow ((*key* * '*a*) * *bool* * '*a tree23*)*option*

where

```

del (Some k) Empty = None |
del None (Branch2 Empty p Empty) = Some(p, (True, Empty)) |
del None (Branch3 Empty p Empty q Empty) = Some(p, (False, Branch2 Empty
q Empty)) |
del k (Branch2 Empty p Empty) = (case compare k p of
  EQUAL => Some(p, (True, Empty)) | - => None) |
del k (Branch3 Empty p Empty q Empty) = (case compare k p of
  EQUAL => Some(p, (False, Branch2 Empty q Empty))
  | - => (case compare k q of
    EQUAL => Some(q, (False, Branch2 Empty p Empty))
    | - => None)) |
del k (Branch2 l p r) = (case compare k p of
  LESS => (case del k l of None => None |

```


$Some(p', (False, l')) \Rightarrow Some(p', (False, Branch2\ l'\ p\ r))$
 $| Some(p', (True, l')) \Rightarrow Some(p', \text{case } r \text{ of}$
 $\quad Branch2\ rl\ rp\ rr \Rightarrow (True, Branch3\ l'\ p\ rl\ rp\ rr)$
 $\quad | Branch3\ rl\ rp\ rm\ rq\ rr \Rightarrow (False, Branch2$
 $\quad\quad (Branch2\ l'\ p\ rl)\ rp\ (Branch2\ rm\ rq\ rr))$
 $| or \Rightarrow (\text{case } del\ (if\text{-}eq\ or\ None\ k)\ r\ of\ None \Rightarrow None |$
 $\quad Some(p', (False, r')) \Rightarrow Some(p', (False, Branch2\ l\ (if\text{-}eq\ or\ p'\ p)\ r'))$
 $\quad | Some(p', (True, r')) \Rightarrow Some(p', \text{case } l \text{ of}$
 $\quad\quad Branch2\ ll\ lp\ lr \Rightarrow (True, Branch3\ ll\ lp\ lr\ (if\text{-}eq\ or\ p'\ p)\ r')$
 $\quad\quad | Branch3\ ll\ lp\ lm\ lq\ lr \Rightarrow (False, Branch2$
 $\quad\quad\quad (Branch2\ ll\ lp\ lm)\ lq\ (Branch2\ lr\ (if\text{-}eq\ or\ p'\ p)\ r'))$
 $del\ k\ (Branch3\ l\ p\ m\ q\ r) = (\text{case } compare\ k\ q\ of$
 $\quad LESS \Rightarrow (\text{case } compare\ k\ p\ of$
 $\quad\quad LESS \Rightarrow (\text{case } del\ k\ l\ of\ None \Rightarrow None |$
 $\quad\quad\quad Some(p', (False, l')) \Rightarrow Some(p', (False, Branch3\ l'\ p\ m\ q\ r))$
 $\quad\quad | Some(p', (True, l')) \Rightarrow Some(p', (False, \text{case } (m, r) \text{ of}$
 $\quad\quad\quad (Branch2\ ml\ mp\ mr, Branch2\ -\ -\ -) \Rightarrow Branch2\ (Branch3\ l'\ p\ ml\ mp$
 $\quad\quad\quad mr)\ q\ r$
 $\quad\quad | (Branch3\ ml\ mp\ mm\ mq\ mr, -) \Rightarrow Branch3\ (Branch2\ l'\ p\ ml)\ mp$
 $\quad\quad\quad (Branch2\ mm\ mq\ mr)\ q\ r$
 $\quad\quad | (Branch2\ ml\ mp\ mr, Branch3\ rl\ rp\ rm\ rq\ rr) \Rightarrow$
 $\quad\quad\quad Branch3\ (Branch2\ l'\ p\ ml)\ mp\ (Branch2\ mr\ q\ rl)\ rp\ (Branch2\ rm\ rq$
 $\quad\quad\quad rr))$
 $\quad | or \Rightarrow (\text{case } del\ (if\text{-}eq\ or\ None\ k)\ m\ of\ None \Rightarrow None |$
 $\quad\quad Some(p', (False, m')) \Rightarrow Some(p', (False, Branch3\ l\ (if\text{-}eq\ or\ p'\ p)\ m'\ q$
 $\quad\quad r))$
 $\quad | Some(p', (True, m')) \Rightarrow Some(p', (False, \text{case } (l, r) \text{ of}$
 $\quad\quad (Branch2\ ll\ lp\ lr, Branch2\ -\ -\ -) \Rightarrow Branch2\ (Branch3\ ll\ lp\ lr\ (if\text{-}eq\ or$
 $\quad\quad\quad p'\ p)\ m')\ q\ r$
 $\quad\quad | (Branch3\ ll\ lp\ lm\ lq\ lr, -) \Rightarrow Branch3\ (Branch2\ ll\ lp\ lm)\ lq\ (Branch2\ lr$
 $\quad\quad\quad (if\text{-}eq\ or\ p'\ p)\ m')\ q\ r$
 $\quad\quad | (-, Branch3\ rl\ rp\ rm\ rq\ rr) \Rightarrow Branch3\ l\ (if\text{-}eq\ or\ p'\ p)\ (Branch2\ m'\ q$
 $\quad\quad\quad rl)\ rp\ (Branch2\ rm\ rq\ rr))$
 $\quad | or \Rightarrow (\text{case } del\ (if\text{-}eq\ or\ None\ k)\ r\ of\ None \Rightarrow None |$
 $\quad\quad Some(q', (False, r')) \Rightarrow Some(q', (False, Branch3\ l\ p\ m\ (if\text{-}eq\ or\ q'\ q)\ r'))$
 $\quad\quad | Some(q', (True, r')) \Rightarrow Some(q', (False, \text{case } (l, m) \text{ of}$
 $\quad\quad\quad (Branch2\ -\ -\ -, Branch2\ ml\ mp\ mr) \Rightarrow Branch2\ l\ p\ (Branch3\ ml\ mp\ mr$
 $\quad\quad\quad (if\text{-}eq\ or\ q'\ q)\ r')$
 $\quad\quad | (-, Branch3\ ml\ mp\ mm\ mq\ mr) \Rightarrow Branch3\ l\ p\ (Branch2\ ml\ mp\ mm)\ mq$
 $\quad\quad\quad (Branch2\ mr\ (if\text{-}eq\ or\ q'\ q)\ r')$
 $\quad\quad | (Branch3\ ll\ lp\ lm\ lq\ lr, Branch2\ ml\ mp\ mr) \Rightarrow$
 $\quad\quad\quad Branch3\ (Branch2\ ll\ lp\ lm)\ lq\ (Branch2\ lr\ p\ ml)\ mp\ (Branch2\ mr\ (if\text{-}eq$
 $\quad\quad\quad or\ q'\ q)\ r'))$

definition $del0 :: key \Rightarrow 'a\ tree23 \Rightarrow 'a\ tree23$ **where**

$del0\ k\ t = (\text{case } del\ (Some\ k)\ t\ of\ None \Rightarrow t\ | Some\ (-, t') \Rightarrow t')$

```

fun ord0 :: 'a tree23 ⇒ bool
and ordl :: key ⇒ 'a tree23 ⇒ bool
and ordr :: 'a tree23 ⇒ key ⇒ bool
and ordlr :: key ⇒ 'a tree23 ⇒ key ⇒ bool
where
ord0 Empty = True |
ord0 (Branch2 l p r) = (ordr l (fst p) & ordl (fst p) r) |
ord0 (Branch3 l p m q r) = (ordr l (fst p) & ordlr (fst p) m (fst q) & ordl (fst
q) r) |

ordl - Empty = True |
ordl x (Branch2 l p r) = (ordlr x l (fst p) & ordl (fst p) r) |
ordl x (Branch3 l p m q r) = (ordlr x l (fst p) & ordlr (fst p) m (fst q) & ordl
(fst q) r) |

ordr Empty - = True |
ordr (Branch2 l p r) x = (ordr l (fst p) & ordlr (fst p) r x) |
ordr (Branch3 l p m q r) x = (ordr l (fst p) & ordlr (fst p) m (fst q) & ordlr (fst
q) r x) |

ordlr x Empty y = (x < y) |
ordlr x (Branch2 l p r) y = (ordlr x l (fst p) & ordlr (fst p) r y) |
ordlr x (Branch3 l p m q r) y = (ordlr x l (fst p) & ordlr (fst p) m (fst q) & ordlr
(fst q) r y)

fun height :: 'a tree23 ⇒ nat where
height Empty = 0 |
height (Branch2 l - r) = Suc(max (height l) (height r)) |
height (Branch3 l - m - r) = Suc(max (height l) (max (height m) (height r)))

```

Is a tree balanced?

```

fun bal :: 'a tree23 ⇒ bool where
bal Empty = True |
bal (Branch2 l - r) = (bal l & bal r & height l = height r) |
bal (Branch3 l - m - r) = (bal l & bal m & bal r & height l = height m & height
m = height r)

```

This is a little test harness and should be commented out once the above functions have been proved correct.

```

datatype 'a act = Add int 'a | Del int

```

```

fun exec where
exec [] t = t |
exec (Add k x # as) t = exec as (add0 k x t) |
exec (Del k # as) t = exec as (del0 k t)

```

Some quick checks:

```

lemma ord0(exec as Empty)
quickcheck

```

oops

lemma *bal*(*exec as Empty*)
quickcheck
oops

end

26 Merge Sort

theory *MergeSort*
imports *Multiset*
begin

context *linorder*
begin

fun *merge* :: 'a list \Rightarrow 'a list \Rightarrow 'a list

where

merge (*x#xs*) (*y#ys*) =
 (if $x \leq y$ then *x # merge xs (y#ys)* else *y # merge (x#xs) ys*)
| *merge xs []* = *xs*
| *merge [] ys* = *ys*

lemma *multiset-of-merge* [*simp*]:

multiset-of (*merge xs ys*) = *multiset-of xs* + *multiset-of ys*
 by (induct *xs ys* rule: *merge.induct*) (*simp-all add: ac-simps*)

lemma *set-merge* [*simp*]:

set (*merge xs ys*) = *set xs* \cup *set ys*
 by (induct *xs ys* rule: *merge.induct*) *auto*

lemma *sorted-merge* [*simp*]:

sorted (*merge xs ys*) \longleftrightarrow *sorted xs* \wedge *sorted ys*
 by (induct *xs ys* rule: *merge.induct*) (*auto simp add: ball-Un not-le less-le sorted-Cons*)

fun *msort* :: 'a list \Rightarrow 'a list

where

msort [] = []
| *msort [x]* = [x]
| *msort xs* = *merge* (*msort* (*take* (*size xs* *div* 2) *xs*))
 (*msort* (*drop* (*size xs* *div* 2) *xs*))

lemma *sorted-msort*:

sorted (*msort xs*)
 by (induct *xs* rule: *msort.induct*) *simp-all*

lemma *multiset-of-msort*:

multiset-of (*msort xs*) = *multiset-of xs*

by (*induct xs rule: msort.induct*)
(simp-all, metis append-take-drop-id drop-Suc-Cons multiset-of.simps(2) multiset-of-append take-Suc-Cons)

theorem *msort-sort*:

sort = msort

by (*rule ext, rule properties-for-sort*) (*fact multiset-of-msort sorted-msort*)+

end

end

27 A lemma for Lagrange's theorem

theory *Lagrange* **imports** *Main* **begin**

This theory only contains a single theorem, which is a lemma in Lagrange's proof that every natural number is the sum of 4 squares. Its sole purpose is to demonstrate ordered rewriting for commutative rings.

The enterprising reader might consider proving all of Lagrange's theorem.

definition *sq* :: '*a::times*' => '*a*' **where** *sq x* == *x*x*

The following lemma essentially shows that every natural number is the sum of four squares, provided all prime numbers are. However, this is an abstract theorem about commutative rings. It has, a priori, nothing to do with nat.

ML <<

Delsimprocs [*ab-group-add-cancel.sum-conv*, *ab-group-add-cancel.rel-conv*]

>>

lemma *Lagrange-lemma*: **fixes** *x1* :: '*a::comm-ring*' **shows**

$(sq\ x1 + sq\ x2 + sq\ x3 + sq\ x4) * (sq\ y1 + sq\ y2 + sq\ y3 + sq\ y4) =$
 $sq\ (x1*y1 - x2*y2 - x3*y3 - x4*y4) +$
 $sq\ (x1*y2 + x2*y1 + x3*y4 - x4*y3) +$
 $sq\ (x1*y3 - x2*y4 + x3*y1 + x4*y2) +$
 $sq\ (x1*y4 + x2*y3 - x3*y2 + x4*y1)$

by (*simp only: sq-def field-simps*)

A challenge by John Harrison. Takes about 12s on a 1.6GHz machine.

lemma **fixes** *p1* :: '*a::comm-ring*' **shows**

$(sq\ p1 + sq\ q1 + sq\ r1 + sq\ s1 + sq\ t1 + sq\ u1 + sq\ v1 + sq\ w1) *$
 $(sq\ p2 + sq\ q2 + sq\ r2 + sq\ s2 + sq\ t2 + sq\ u2 + sq\ v2 + sq\ w2)$
 $= sq\ (p1*p2 - q1*q2 - r1*r2 - s1*s2 - t1*t2 - u1*u2 - v1*v2 - w1*w2)$
 $+$
 $sq\ (p1*q2 + q1*p2 + r1*s2 - s1*r2 + t1*u2 - u1*t2 - v1*w2 + w1*v2)$
 $+$
 $sq\ (p1*r2 - q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 - v1*t2 - w1*u2)$
 $+$

```

      sq (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)
+
      sq (p1*t2 - q1*u2 - r1*v2 - s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)
+
      sq (p1*u2 + q1*t2 - r1*w2 + s1*v2 - t1*q2 + u1*p2 - v1*s2 + w1*r2)
+
      sq (p1*v2 + q1*w2 + r1*t2 - s1*u2 - t1*r2 + u1*s2 + v1*p2 - w1*q2)
+
      sq (p1*w2 - q1*v2 + r1*u2 + s1*t2 - t1*s2 - u1*r2 + v1*q2 + w1*p2)
by (simp only: sq-def field-simps)

end

```

28 Groebner Basis Examples

```

theory Groebner-Examples
imports Groebner-Basis
begin

```

28.1 Basic examples

```

lemma
  fixes x :: int
  shows  $x^3 = x^3$ 
  apply (tactic ⟨ ALLGOALS (CONVERSION
    (Conv.arg-conv (Conv.arg1-conv (Semiring-Normalizer.semiring-normalize-conv
@{context})))) ⟩)
  by (rule refl)

```

```

lemma
  fixes x :: int
  shows  $(x - (-2))^5 = x^5 + (10 * x^4 + (40 * x^3 + (80 * x^2 + (80 * x + 32))))$ 
  apply (tactic ⟨ ALLGOALS (CONVERSION
    (Conv.arg-conv (Conv.arg1-conv (Semiring-Normalizer.semiring-normalize-conv
@{context})))) ⟩)
  by (rule refl)

```

```

schematic-lemma
  fixes x :: int
  shows  $(x - (-2))^5 * (y - 78)^8 = ?X$ 
  apply (tactic ⟨ ALLGOALS (CONVERSION
    (Conv.arg-conv (Conv.arg1-conv (Semiring-Normalizer.semiring-normalize-conv
@{context})))) ⟩)
  by (rule refl)

```

```

lemma ((-3) ^ (Suc (Suc (Suc 0)))) == (X::'a::{number-ring})
  apply (simp only: power-Suc power-0)

```

```

apply (simp only: semiring-norm)
oops

lemma ((x::int) + y)^3 - 1 = (x - z)^2 - 10  $\implies$  x = z + 3  $\implies$  x = - y
by algebra

lemma (4::nat) + 4 = 3 + 5
by algebra

lemma (4::int) + 0 = 4
apply algebra?
by simp

lemma
  assumes a * x^2 + b * x + c = (0::int) and d * x^2 + e * x + f = 0
  shows d^2 * c^2 - 2 * d * c * a * f + a^2 * f^2 - e * d * b * c - e * b * a * f + a * e^2 * c +
f * d * b^2 = 0
  using assms by algebra

lemma (x::int)^3 - x^2 - 5 * x - 3 = 0  $\longleftrightarrow$  (x = 3  $\vee$  x = -1)
by algebra

theorem x * (x^2 - x - 5) - 3 = (0::int)  $\longleftrightarrow$  (x = 3  $\vee$  x = -1)
by algebra

lemma
  fixes x::'a::{idom,number-ring}
  shows x^2 * y = x^2 & x * y^2 = y^2  $\longleftrightarrow$  x=1 & y=1 | x=0 & y=0
  by algebra

```

28.2 Lemmas for Lagrange's theorem

```

definition
  sq :: 'a::times => 'a where
    sq x == x * x

lemma
  fixes x1 :: 'a::{idom,number-ring}
  shows
    (sq x1 + sq x2 + sq x3 + sq x4) * (sq y1 + sq y2 + sq y3 + sq y4) =
      sq (x1 * y1 - x2 * y2 - x3 * y3 - x4 * y4) +
      sq (x1 * y2 + x2 * y1 + x3 * y4 - x4 * y3) +
      sq (x1 * y3 - x2 * y4 + x3 * y1 + x4 * y2) +
      sq (x1 * y4 + x2 * y3 - x3 * y2 + x4 * y1)
  by (algebra add: sq-def)

lemma
  fixes p1 :: 'a::{idom,number-ring}
  shows

```

```

    (sq p1 + sq q1 + sq r1 + sq s1 + sq t1 + sq u1 + sq v1 + sq w1) *
    (sq p2 + sq q2 + sq r2 + sq s2 + sq t2 + sq u2 + sq v2 + sq w2)
    = sq (p1*p2 - q1*q2 - r1*r2 - s1*s2 - t1*t2 - u1*u2 - v1*v2 - w1*w2)
+
    sq (p1*q2 + q1*p2 + r1*s2 - s1*r2 + t1*u2 - u1*t2 - v1*w2 + w1*v2)
+
    sq (p1*r2 - q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 - v1*t2 - w1*u2)
+
    sq (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)
+
    sq (p1*t2 - q1*u2 - r1*v2 - s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)
+
    sq (p1*u2 + q1*t2 - r1*w2 + s1*v2 - t1*q2 + u1*p2 - v1*s2 + w1*r2)
+
    sq (p1*v2 + q1*w2 + r1*t2 - s1*u2 - t1*r2 + u1*s2 + v1*p2 - w1*q2)
+
    sq (p1*w2 - q1*v2 + r1*u2 + s1*t2 - t1*s2 - u1*r2 + v1*q2 + w1*p2)
  by (algebra add: sq-def)

```

28.3 Colinearity is invariant by rotation

types *point* = *int* × *int*

definition *collinear* :: *point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**

```

  collinear ≡ λ(Ax,Ay) (Bx,By) (Cx,Cy).
    ((Ax - Bx) * (By - Cy) = (Ay - By) * (Bx - Cx))

```

lemma *collinear-inv-rotation*:

assumes *collinear* (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) **and** $c^2 + s^2 = 1$

shows *collinear* (*Ax* * *c* - *Ay* * *s*, *Ay* * *c* + *Ax* * *s*)

(*Bx* * *c* - *By* * *s*, *By* * *c* + *Bx* * *s*) (*Cx* * *c* - *Cy* * *s*, *Cy* * *c* + *Cx* * *s*)

using *assms*

by (*algebra add: collinear-def split-def fst-conv snd-conv*)

lemma *EX* (*d*::*int*). $a*y - a*x = n*d \implies EX\ u\ v.\ a*u + n*v = 1 \implies EX\ e.$
 $y - x = n*e$

by *algebra*

end

29 Milner-Tofte: Co-induction in Relational Semantics

theory *MT*

imports *Main*

begin

typedecl *Const*

typedecl *ExVar*

typedecl *Ex*

typedecl *TyConst*

typedecl *Ty*

typedecl *Clos*

typedecl *Val*

typedecl *ValEnv*

typedecl *TyEnv*

consts

c-app :: [*Const*, *Const*] => *Const*

e-const :: *Const* => *Ex*

e-var :: *ExVar* => *Ex*

e-fn :: [*ExVar*, *Ex*] => *Ex* (*fn* - => - [0,51] 1000)

e-fix :: [*ExVar*, *ExVar*, *Ex*] => *Ex* (*fix* - (-) = - [0,51,51] 1000)

e-app :: [*Ex*, *Ex*] => *Ex* (- @@ - [51,51] 1000)

e-const-fst :: *Ex* => *Const*

t-const :: *TyConst* => *Ty*

t-fun :: [*Ty*, *Ty*] => *Ty* (- -> - [51,51] 1000)

v-const :: *Const* => *Val*

v-clos :: *Clos* => *Val*

ve-emp :: *ValEnv*

ve-owr :: [*ValEnv*, *ExVar*, *Val*] => *ValEnv* (- + { - | -> - } [36,0,0] 50)

ve-dom :: *ValEnv* => *ExVar* set

ve-app :: [*ValEnv*, *ExVar*] => *Val*

clos-mk :: [*ExVar*, *Ex*, *ValEnv*] => *Clos* (<| - , - , - |> [0,0,0] 1000)

te-emp :: *TyEnv*

te-owr :: [*TyEnv*, *ExVar*, *Ty*] => *TyEnv* (- + { - | => - } [36,0,0] 50)

te-app :: [*TyEnv*, *ExVar*] => *Ty*

te-dom :: *TyEnv* => *ExVar* set

eval-fun :: ((*ValEnv* * *Ex*) * *Val*) set => ((*ValEnv* * *Ex*) * *Val*) set

eval-rel :: ((*ValEnv* * *Ex*) * *Val*) set

eval :: [*ValEnv*, *Ex*, *Val*] => *bool* (- | - - - -> - [36,0,36] 50)

elab-fun :: ((*TyEnv* * *Ex*) * *Ty*) set => ((*TyEnv* * *Ex*) * *Ty*) set

elab-rel :: ((*TyEnv* * *Ex*) * *Ty*) set

elab :: [*TyEnv*, *Ex*, *Ty*] => *bool* (- | - - - -> - [36,0,36] 50)

isof :: [Const, Ty] => bool (- isof - [36,36] 50)
isof-env :: [ValEnv, TyEnv] => bool (- isofenv -)

hasty-fun :: (Val * Ty) set => (Val * Ty) set
hasty-rel :: (Val * Ty) set
hasty :: [Val, Ty] => bool (- hasty - [36,36] 50)
hasty-env :: [ValEnv, TyEnv] => bool (- hastyenv - [36,36] 35)

axioms

e-const-inj: $e\text{-const}(c1) = e\text{-const}(c2) \implies c1 = c2$
e-var-inj: $e\text{-var}(ev1) = e\text{-var}(ev2) \implies ev1 = ev2$
e-fn-inj: $fn\ ev1 \Rightarrow e1 = fn\ ev2 \Rightarrow e2 \implies ev1 = ev2 \ \& \ e1 = e2$
e-fix-inj:
 $fix\ ev11e(v12) = e1 = fix\ ev21(ev22) = e2 \implies$
 $ev11 = ev21 \ \& \ ev12 = ev22 \ \& \ e1 = e2$

e-app-inj: $e11 \ @\@ \ e12 = e21 \ @\@ \ e22 \implies e11 = e21 \ \& \ e12 = e22$

e-disj-const-var: $\sim e\text{-const}(c) = e\text{-var}(ev)$
e-disj-const-fn: $\sim e\text{-const}(c) = fn\ ev \Rightarrow e$
e-disj-const-fix: $\sim e\text{-const}(c) = fix\ ev1(ev2) = e$
e-disj-const-app: $\sim e\text{-const}(c) = e1 \ @\@ \ e2$
e-disj-var-fn: $\sim e\text{-var}(ev1) = fn\ ev2 \Rightarrow e$
e-disj-var-fix: $\sim e\text{-var}(ev) = fix\ ev1(ev2) = e$
e-disj-var-app: $\sim e\text{-var}(ev) = e1 \ @\@ \ e2$
e-disj-fn-fix: $\sim fn\ ev1 \Rightarrow e1 = fix\ ev21(ev22) = e2$
e-disj-fn-app: $\sim fn\ ev1 \Rightarrow e1 = e21 \ @\@ \ e22$
e-disj-fix-app: $\sim fix\ ev11(ev12) = e1 = e21 \ @\@ \ e22$

e-ind:

$$\begin{aligned}
 & [\quad !!ev. P(e\text{-var}(ev)); \\
 & \quad !!c. P(e\text{-const}(c)); \\
 & \quad !!ev\ e. P(e) \implies P(fn\ ev \Rightarrow e); \\
 & \quad !!ev1\ ev2\ e. P(e) \implies P(fix\ ev1(ev2) = e); \\
 & \quad !!e1\ e2. P(e1) \implies P(e2) \implies P(e1 \ @\@ \ e2) \\
 &] \implies \\
 & P(e)
 \end{aligned}$$

$t\text{-const-inj}: t\text{-const}(c1) = t\text{-const}(c2) \implies c1 = c2$
 $t\text{-fun-inj}: t11 \rightarrow t12 = t21 \rightarrow t22 \implies t11 = t21 \ \& \ t12 = t22$

$t\text{-ind}:$
 $[[!!p. P(t\text{-const } p); !!t1 \ t2. P(t1) \implies P(t2) \implies P(t\text{-fun } t1 \ t2)]]$
 $\implies P(t)$

$v\text{-const-inj}: v\text{-const}(c1) = v\text{-const}(c2) \implies c1 = c2$
 $v\text{-clos-inj}:$
 $v\text{-clos}(<|ev1, e1, ve1|>) = v\text{-clos}(<|ev2, e2, ve2|>) \implies$
 $ev1 = ev2 \ \& \ e1 = e2 \ \& \ ve1 = ve2$

$v\text{-disj-const-clos}: \sim v\text{-const}(c) = v\text{-clos}(cl)$

$ve\text{-dom-owr}: ve\text{-dom}(ve + \{ev \mid \rightarrow v\}) = ve\text{-dom}(ve) \cup \{ev\}$

$ve\text{-app-owr1}: ve\text{-app } (ve + \{ev \mid \rightarrow v\}) \ ev = v$
 $ve\text{-app-owr2}: \sim ev1 = ev2 \implies ve\text{-app } (ve + \{ev1 \mid \rightarrow v\}) \ ev2 = ve\text{-app } ve \ ev2$

$te\text{-dom-owr}: te\text{-dom}(te + \{ev \mid \Rightarrow t\}) = te\text{-dom}(te) \cup \{ev\}$

$te\text{-app-owr1}: te\text{-app } (te + \{ev \mid \Rightarrow t\}) \ ev = t$
 $te\text{-app-owr2}: \sim ev1 = ev2 \implies te\text{-app } (te + \{ev1 \mid \Rightarrow t\}) \ ev2 = te\text{-app } te \ ev2$

defs

```

eval-fun-def:
  eval-fun(s) ==
  { pp.
    (? ve c. pp=((ve,e-const(c)),v-const(c))) |
    (? ve x. pp=((ve,e-var(x)),ve-app ve x) & x:ve-dom(ve)) |
    (? ve e x. pp=((ve,fn x => e),v-clos(<|x,e,ve|>))) |
    (? ve e x f cl.
      pp=((ve,fix f(x) = e),v-clos(cl)) &
      cl=<|x, e, ve+{f |-> v-clos(cl)} |>
    ) |
    (? ve e1 e2 c1 c2.
      pp=((ve,e1 @@ e2),v-const(c-app c1 c2)) &
      ((ve,e1),v-const(c1)):s & ((ve,e2),v-const(c2)):s
    ) |
    (? ve vem e1 e2 em xm v v2.
      pp=((ve,e1 @@ e2),v) &
      ((ve,e1),v-clos(<|xm,em,vem|>)):s &
      ((ve,e2),v2):s &
      ((vem+{xm |-> v2},em),v):s
    )
  }

```

eval-rel-def: eval-rel == lfp(eval-fun)
 eval-def: ve |- e ----> v == ((ve,e),v):eval-rel

```

elab-fun-def:
  elab-fun(s) ==
  { pp.
    (? te c t. pp=((te,e-const(c)),t) & c isof t) |
    (? te x. pp=((te,e-var(x)),te-app te x) & x:te-dom(te)) |
    (? te x e t1 t2. pp=((te,fn x => e),t1->t2) & ((te+{x |=> t1},e),t2):s) |
    (? te f x e t1 t2.
      pp=((te,fix f(x)=e),t1->t2) & ((te+{f |=> t1->t2}+{x |=> t1},e),t2):s
    ) |
    (? te e1 e2 t1 t2.
      pp=((te,e1 @@ e2),t2) & ((te,e1),t1->t2):s & ((te,e2),t1):s
    )
  }

```

elab-rel-def: elab-rel == lfp(elab-fun)
 elab-def: te |- e ==> t == ((te,e),t):elab-rel

isof-env-def:

```

ve isofenv te ==
ve-dom(ve) = te-dom(te) &
( ! x.
  x:ve-dom(ve) -->
  (? c. ve-app ve x = v-const(c) & c isof te-app te x)
)

```

axioms

isof-app: $[[\ c1\ isof\ t1 \rightarrow t2;\ c2\ isof\ t1\]]\ ==>\ c\text{-app}\ c1\ c2\ isof\ t2$

defs

```

hasty-fun-def:
hasty-fun(r) ==
{ p.
  ( ? c t. p = (v-const(c),t) & c isof t ) |
  ( ? ev e ve t te.
    p = (v-clos(<|ev,e,ve|>),t) &
    te |- fn ev => e ==> t &
    ve-dom(ve) = te-dom(te) &
    (! ev1. ev1:ve-dom(ve) --> (ve-app ve ev1,te-app te ev1) : r)
  )
}

```

```

hasty-rel-def: hasty-rel == gfp(hasty-fun)
hasty-def: v hasty t == (v,t) : hasty-rel
hasty-env-def:
ve hastyenv te ==
ve-dom(ve) = te-dom(te) &
(! x. x: ve-dom(ve) --> ve-app ve x hasty te-app te x)

```

ML \ll
val infsys-mono-tac = *REPEAT* (*ares-tac* (@{*thms basic-monos*} @ [*allI*, *impI*])
1)
 \gg

lemma *infsys-p1*: $P\ a\ b \implies P\ (fst\ (a,b))\ (snd\ (a,b))$
by *simp*

lemma *infsys-p2*: $P\ (fst\ (a,b))\ (snd\ (a,b)) \implies P\ a\ b$
by *simp*

```

lemma infsys-pp1:  $P\ a\ b\ c \implies P\ (fst(fst((a,b),c)))\ (snd(fst\ ((a,b),c)))\ (snd\ ((a,b),c))$ 
  by simp

```

```

lemma infsys-pp2:  $P\ (fst(fst((a,b),c)))\ (snd(fst((a,b),c)))\ (snd((a,b),c)) \implies P\ a\ b\ c$ 
  by simp

```

```

lemma lfp-intro2:  $[| \text{mono}(f); x:f(lfp(f)) |] \implies x:lfp(f)$ 
apply (rule subsetD)
apply (rule lfp-lemma2)
apply assumption+
done

```

```

lemma lfp-elim2:
  assumes lfp:  $x:lfp(f)$ 
  and mono:  $\text{mono}(f)$ 
  and r:  $!!y. y:f(lfp(f)) \implies P(y)$ 
  shows  $P(x)$ 
apply (rule r)
apply (rule subsetD)
apply (rule lfp-lemma3)
apply (rule mono)
apply (rule lfp)
done

```

```

lemma lfp-ind2:
  assumes lfp:  $x:lfp(f)$ 
  and mono:  $\text{mono}(f)$ 
  and r:  $!!y. y:f(lfp(f))\ Int\ \{x. P(x)\} \implies P(y)$ 
  shows  $P(x)$ 
apply (rule lfp-induct-set [OF lfp mono])
apply (erule r)
done

```

```

lemma gfp-coind2:
  assumes cih:  $x:f(\{x\}\ Un\ gfp(f))$ 

```

```

    and monoh: mono(f)
  shows x:gfp(f)
apply (rule cih [THEN [2] gfp-upperbound [THEN subsetD]])
apply (rule monoh [THEN monoD])
apply (rule UnE [THEN subsetI])
apply assumption
apply (blast intro!: cih)
apply (rule monoh [THEN monoD [THEN subsetD]])
apply (rule Un-upper2)
apply (erule monoh [THEN gfp-lemma2, THEN subsetD])
done

```

```

lemma gfp-elim2:
  assumes gfph: x:gfp(f)
    and monoh: mono(f)
    and caseh: !!y. y:f(gfp(f)) ==> P(y)
  shows P(x)
apply (rule caseh)
apply (rule subsetD)
apply (rule gfp-lemma2)
apply (rule monoh)
apply (rule gfph)
done

```

lemmas *e-injs* = *e-const-inj* *e-var-inj* *e-fn-inj* *e-fix-inj* *e-app-inj*

lemmas *e-disjs* =
e-disj-const-var
e-disj-const-fn
e-disj-const-fix
e-disj-const-app
e-disj-var-fn
e-disj-var-fix
e-disj-var-app
e-disj-fn-fix
e-disj-fn-app
e-disj-fix-app

lemmas *e-disj-si* = *e-disjs* *e-disjs* [*symmetric*]

lemmas *e-disj-se* = *e-disj-si* [*THEN notE*]

```

lemmas  $v\text{-disjs} = v\text{-disj-const-clos}$ 
lemmas  $v\text{-disj-si} = v\text{-disjs } v\text{-disjs [symmetric]}$ 
lemmas  $v\text{-disj-se} = v\text{-disj-si [THEN notE]}$ 

lemmas  $v\text{-injs} = v\text{-const-inj } v\text{-clos-inj}$ 

```

```

lemma  $\text{eval-fun-mono}$ :  $\text{mono}(\text{eval-fun})$ 
unfolding  $\text{mono-def eval-fun-def}$ 
apply ( $\text{tactic infsys-mono-tac}$ )
done

```

```

lemma  $\text{eval-const}$ :  $ve \mid\!-\! e\text{-const}(c) \text{ ---} > v\text{-const}(c)$ 
unfolding  $\text{eval-def eval-rel-def}$ 
apply ( $\text{rule lfp-intro2}$ )
apply ( $\text{rule eval-fun-mono}$ )
apply ( $\text{unfold eval-fun-def}$ )

apply ( $\text{blast intro!} \text{: exI}$ )
done

```

```

lemma  $\text{eval-var2}$ :
   $ev \text{: } ve\text{-dom}(ve) \implies ve \mid\!-\! e\text{-var}(ev) \text{ ---} > ve\text{-app } ve \text{ } ev$ 
apply ( $\text{unfold eval-def eval-rel-def}$ )
apply ( $\text{rule lfp-intro2}$ )
apply ( $\text{rule eval-fun-mono}$ )
apply ( $\text{unfold eval-fun-def}$ )
apply ( $\text{blast intro!} \text{: exI}$ )
done

```

```

lemma  $\text{eval-fn}$ :
   $ve \mid\!-\! fn \text{ } ev \implies e \text{ ---} > v\text{-clos}(<\mid\!ev, e, ve\mid\!>)$ 
apply ( $\text{unfold eval-def eval-rel-def}$ )
apply ( $\text{rule lfp-intro2}$ )
apply ( $\text{rule eval-fun-mono}$ )
apply ( $\text{unfold eval-fun-def}$ )
apply ( $\text{blast intro!} \text{: exI}$ )
done

```

```

lemma  $\text{eval-fix}$ :
   $cl = <\mid\! ev1, e, ve + \{ev2 \mid\!-\!> v\text{-clos}(cl)\} \mid\!> \implies$ 

```

```

    ve |- fix ev2(ev1) = e ----> v-clos(cl)
  apply (unfold eval-def eval-rel-def)
  apply (rule lfp-intro2)
  apply (rule eval-fun-mono)
  apply (unfold eval-fun-def)
  apply (blast intro!: exI)
done

lemma eval-app1:
  [| ve |- e1 ----> v-const(c1); ve |- e2 ----> v-const(c2) |] ==>
    ve |- e1 @@ e2 ----> v-const(c-app c1 c2)
  apply (unfold eval-def eval-rel-def)
  apply (rule lfp-intro2)
  apply (rule eval-fun-mono)
  apply (unfold eval-fun-def)
  apply (blast intro!: exI)
done

lemma eval-app2:
  [| ve |- e1 ----> v-clos(<|xm,em,vem|>);
    ve |- e2 ----> v2;
    vem + {xm |-> v2} |- em ----> v
  |] ==>
    ve |- e1 @@ e2 ----> v
  apply (unfold eval-def eval-rel-def)
  apply (rule lfp-intro2)
  apply (rule eval-fun-mono)
  apply (unfold eval-fun-def)
  apply (blast intro!: disjI2)
done

lemma eval-ind0:
  [| ve |- e ----> v;
    !!ve c. P(((ve,e-const(c)),v-const(c)));
    !!ev ve. ev:ve-dom(ve) ==> P(((ve,e-var(ev)),ve-app ve ev));
    !!ev ve e. P(((ve,fn ev => e),v-clos(<|ev,e,ve|>)));
    !!ev1 ev2 ve cl e.
      cl = <| ev1, e, ve + {ev2 |-> v-clos(cl)} |> ==>
        P(((ve,fix ev2(ev1) = e),v-clos(cl)));
    !!ve c1 c2 e1 e2.
      [| P(((ve,e1),v-const(c1))); P(((ve,e2),v-const(c2))) |] ==>
        P(((ve,e1 @@ e2),v-const(c-app c1 c2)));
    !!ve vem xm e1 e2 em v v2.
      [| P(((ve,e1),v-clos(<|xm,em,vem|>)));
        P(((ve,e2),v2));
        P(((vem + {xm |-> v2},em),v))
      |] ==>

```



```

      P(((ve,e1 @@ e2),v))
    ]] ==>
      P(((ve,e),v))
unfolding eval-def eval-rel-def
apply (erule lfp-ind2)
apply (rule eval-fun-mono)
apply (unfold eval-fun-def)
apply (drule CollectD)
apply safe
apply auto
done

lemma eval-ind:
  [| ve |- e ----> v;
    !!ve c. P ve (e-const c) (v-const c);
    !!ev ve. ev:ve-dom(ve) ==> P ve (e-var ev) (ve-app ve ev);
    !!ev ve e. P ve (fn ev => e) (v-clos <|ev,e,ve|>);
    !!ev1 ev2 ve cl e.
      cl = <| ev1, e, ve + {ev2 |-> v-clos(cl)} |> ==>
        P ve (fix ev2(ev1) = e) (v-clos cl);
    !!ve c1 c2 e1 e2.
      [| P ve e1 (v-const c1); P ve e2 (v-const c2) |] ==>
        P ve (e1 @@ e2) (v-const(c-app c1 c2));
    !!ve vem evm e1 e2 em v v2.
      [| P ve e1 (v-clos <|evm,em,vem|>);
        P ve e2 v2;
        P (vem + {evm |-> v2}) em v
      |] ==> P ve (e1 @@ e2) v
  |] ==> P ve e v
apply (rule-tac P = P in infsys-pp2)
apply (rule eval-ind0)
apply (rule infsys-pp1)
apply auto
done

```

```

lemma elab-fun-mono: mono(elab-fun)
unfolding mono-def elab-fun-def
apply (tactic infsys-mono-tac)
done

```

```

lemma elab-const:
  c isof ty ==> te |- e-const(c) ==> ty
apply (unfold elab-def elab-rel-def)

```

```

apply (rule lfp-intro2)
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (blast intro!: exI)
done

```

```

lemma elab-var:
   $x:te-dom(te) \implies te \vdash e-var(x) \implies te-app\ te\ x$ 
apply (unfold elab-def elab-rel-def)
apply (rule lfp-intro2)
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (blast intro!: exI)
done

```

```

lemma elab-fn:
   $te + \{x \mid \Rightarrow ty1\} \vdash e \implies ty2 \implies te \vdash fn\ x \Rightarrow e \implies ty1 \multimap ty2$ 
apply (unfold elab-def elab-rel-def)
apply (rule lfp-intro2)
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (blast intro!: exI)
done

```

```

lemma elab-fix:
   $te + \{f \mid \Rightarrow ty1 \multimap ty2\} + \{x \mid \Rightarrow ty1\} \vdash e \implies ty2 \implies$ 
   $te \vdash fix\ f(x) = e \implies ty1 \multimap ty2$ 
apply (unfold elab-def elab-rel-def)
apply (rule lfp-intro2)
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (blast intro!: exI)
done

```

```

lemma elab-app:
   $[te \vdash e1 \implies ty1 \multimap ty2; te \vdash e2 \implies ty1] \implies$ 
   $te \vdash e1\ @@\ e2 \implies ty2$ 
apply (unfold elab-def elab-rel-def)
apply (rule lfp-intro2)
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (blast intro!: disjI2)
done

```

```

lemma elab-ind0:
  assumes 1:  $te \vdash e \implies t$ 
  and 2:  $!!te\ c\ t.\ c\ isof\ t \implies P(((te, e-const(c)), t))$ 

```

```

and 3: !!te x. x:te-dom(te) ==> P(((te,e-var(x)),te-app te x))
and 4: !!te x e t1 t2.
  [| te + {x |=> t1} |- e ==> t2; P(((te + {x |=> t1},e),t2)) |] ==>
  P(((te,fn x => e),t1->t2))
and 5: !!te f x e t1 t2.
  [| te + {f |=> t1->t2} + {x |=> t1} |- e ==> t2;
    P(((te + {f |=> t1->t2} + {x |=> t1},e),t2))
  |] ==>
  P(((te,fix f(x) = e),t1->t2))
and 6: !!te e1 e2 t1 t2.
  [| te |- e1 ==> t1->t2; P(((te,e1),t1->t2));
    te |- e2 ==> t1; P(((te,e2),t1))
  |] ==>
  P(((te,e1 @@ e2),t2))
shows P(((te,e),t))
apply (rule lfp-ind2 [OF 1 [unfolded elab-def elab-rel-def]])
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (drule CollectD)
apply safe
apply (erule 2)
apply (erule 3)
apply (rule 4 [unfolded elab-def elab-rel-def]) apply blast+
apply (rule 5 [unfolded elab-def elab-rel-def]) apply blast+
apply (rule 6 [unfolded elab-def elab-rel-def]) apply blast+
done

lemma elab-ind:
  [| te |- e ==> t;
    !!te c t. c isof t ==> P te (e-const c) t;
    !!te x. x:te-dom(te) ==> P te (e-var x) (te-app te x);
    !!te x e t1 t2.
      [| te + {x |=> t1} |- e ==> t2; P (te + {x |=> t1}) e t2 |] ==>
      P te (fn x => e) (t1->t2);
    !!te f x e t1 t2.
      [| te + {f |=> t1->t2} + {x |=> t1} |- e ==> t2;
        P (te + {f |=> t1->t2} + {x |=> t1}) e t2
      |] ==>
      P te (fix f(x) = e) (t1->t2);
    !!te e1 e2 t1 t2.
      [| te |- e1 ==> t1->t2; P te e1 (t1->t2);
        te |- e2 ==> t1; P te e2 t1
      |] ==>
      P te (e1 @@ e2) t2
  |] ==>
  P te e t
apply (rule-tac P = P in infsys-pp2)
apply (erule elab-ind0)
apply (rule-tac [!] infsys-pp1)

```

apply auto
done

lemma elab-elim0:

assumes 1: $te \vdash e \implies t$
and 2: $\forall te\ c\ t. c \text{ isof } t \implies P((te, e\text{-const}(c)), t)$
and 3: $\forall te\ x. x : te\text{-dom}(te) \implies P((te, e\text{-var}(x)), te\text{-app } te\ x)$
and 4: $\forall te\ x\ e\ t1\ t2.$
 $te + \{x \mid \Rightarrow t1\} \vdash e \implies t2 \implies P((te, \text{fn } x \Rightarrow e), t1 \multimap t2)$
and 5: $\forall te\ f\ x\ e\ t1\ t2.$
 $te + \{f \mid \Rightarrow t1 \multimap t2\} + \{x \mid \Rightarrow t1\} \vdash e \implies t2 \implies$
 $P((te, \text{fix } f(x) = e), t1 \multimap t2)$
and 6: $\forall te\ e1\ e2\ t1\ t2.$
 $\llbracket te \vdash e1 \implies t1 \multimap t2; te \vdash e2 \implies t1 \rrbracket \implies$
 $P((te, e1 \text{ @@ } e2), t2)$
shows $P((te, e), t)$
apply (rule lfp-elim2 [OF 1 [unfolded elab-def elab-rel-def]])
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (drule CollectD)
apply safe
apply (erule 2)
apply (erule 3)
apply (rule 4 [unfolded elab-def elab-rel-def]) **apply** blast+
apply (rule 5 [unfolded elab-def elab-rel-def]) **apply** blast+
apply (rule 6 [unfolded elab-def elab-rel-def]) **apply** blast+
done

lemma elab-elim:

$\llbracket te \vdash e \implies t;$
 $\forall te\ c\ t. c \text{ isof } t \implies P\ te\ (e\text{-const } c)\ t;$
 $\forall te\ x. x : te\text{-dom}(te) \implies P\ te\ (e\text{-var } x)\ (te\text{-app } te\ x);$
 $\forall te\ x\ e\ t1\ t2.$
 $te + \{x \mid \Rightarrow t1\} \vdash e \implies t2 \implies P\ te\ (\text{fn } x \Rightarrow e)\ (t1 \multimap t2);$
 $\forall te\ f\ x\ e\ t1\ t2.$
 $te + \{f \mid \Rightarrow t1 \multimap t2\} + \{x \mid \Rightarrow t1\} \vdash e \implies t2 \implies$
 $P\ te\ (\text{fix } f(x) = e)\ (t1 \multimap t2);$
 $\forall te\ e1\ e2\ t1\ t2.$
 $\llbracket te \vdash e1 \implies t1 \multimap t2; te \vdash e2 \implies t1 \rrbracket \implies$
 $P\ te\ (e1 \text{ @@ } e2)\ t2$
 $\rrbracket \implies$
 $P\ te\ e\ t$
apply (rule-tac $P = P$ in infsys-pp2)
apply (rule elab-elim0)
apply auto
done

lemma *elab-const-elim-lem*:

$te \vdash e \implies t \implies (e = e\text{-const}(c) \dashrightarrow c \text{ isof } t)$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-const-elim*: $te \vdash e\text{-const}(c) \implies t \implies c \text{ isof } t$

apply (*drule elab-const-elim-lem*)
apply *blast*
done

lemma *elab-var-elim-lem*:

$te \vdash e \implies t \implies (e = e\text{-var}(x) \dashrightarrow t = te\text{-app } te \ x \ \& \ x : te\text{-dom}(te))$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-var-elim*: $te \vdash e\text{-var}(ev) \implies t \implies t = te\text{-app } te \ ev \ \& \ ev : te\text{-dom}(te)$

apply (*drule elab-var-elim-lem*)
apply *blast*
done

lemma *elab-fn-elim-lem*:

$te \vdash e \implies t \implies$
 $(e = fn \ x1 \Rightarrow e1 \dashrightarrow$
 $(? \ t1 \ t2. \ t = t\text{-fun } t1 \ t2 \ \& \ te + \{x1 \mid \Rightarrow t1\} \vdash e1 \implies t2))$
 $)$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-fn-elim*: $te \vdash fn \ x1 \Rightarrow e1 \implies t \implies$

$(? \ t1 \ t2. \ t = t1 \text{-}> t2 \ \& \ te + \{x1 \mid \Rightarrow t1\} \vdash e1 \implies t2)$
apply (*drule elab-fn-elim-lem*)
apply *blast*
done

lemma *elab-fix-elim-lem*:

$te \vdash e \implies t \implies$
 $(e = fix \ f(x) = e1 \dashrightarrow$
 $(? \ t1 \ t2. \ t = t1 \text{-}> t2 \ \& \ te + \{f \mid \Rightarrow t1 \text{-}> t2\} + \{x \mid \Rightarrow t1\} \vdash e1 \implies t2))$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-fix-elim*: $te \vdash fix \ ev1(ev2) = e1 \implies t \implies$

```

    (? t1 t2. t=t1->t2 & te + {ev1 |=> t1->t2} + {ev2 |=> t1} |- e1 ===>
t2)
apply (drule elab-fix-elim-lem)
apply blast
done

```

```

lemma elab-app-elim-lem:
  te |- e ===> t2 ==>
  (e = e1 @@ e2 --> (? t1 . te |- e1 ===> t1->t2 & te |- e2 ===> t1))
apply (erule elab-elim)
apply (fast intro!: e-disj-si elim!: e-disj-se dest!: e-injs)+
done

```

```

lemma elab-app-elim: te |- e1 @@ e2 ===> t2 ==> (? t1 . te |- e1 ===>
t1->t2 & te |- e2 ===> t1)
apply (drule elab-app-elim-lem)
apply blast
done

```

```

lemma mono-hasty-fun: mono(hasty-fun)
unfolding mono-def hasty-fun-def
apply (tactic infsys-mono-tac)
apply blast
done

```

```

lemma hasty-rel-const-coind: c isof t ==> (v-const(c),t) : hasty-rel
apply (unfold hasty-rel-def)
apply (rule gfp-coind2)
apply (unfold hasty-fun-def)
apply (rule CollectI)
apply (rule disjI1)
apply blast
apply (rule mono-hasty-fun)
done

```

```

lemma hasty-rel-clos-coind:
  [] te |- fn ev => e ===> t;

```

```

    ve-dom(ve) = te-dom(te);
    ! ev1.
      ev1:ve-dom(ve) -->
        (ve-app ve ev1,te-app te ev1) : {(v-clos(<|ev,e,ve|>),t)} Un hasty-rel
  ]] ==>
    (v-clos(<|ev,e,ve|>),t) : hasty-rel
apply (unfold hasty-rel-def)
apply (rule gfp-coind2)
apply (unfold hasty-fun-def)
apply (rule CollectI)
apply (rule disjI2)
apply blast
apply (rule mono-hasty-fun)
done

```

```

lemma hasty-rel-elim0:
  [[ !! c t. c isof t ==> P((v-const(c),t));
    !! te ev e t ve.
      [[ te |- fn ev => e ==> t;
        ve-dom(ve) = te-dom(te);
        !ev1. ev1:ve-dom(ve) --> (ve-app ve ev1,te-app te ev1) : hasty-rel
      ]] ==> P((v-clos(<|ev,e,ve|>),t));
      (v,t) : hasty-rel
    ]] ==> P(v,t)
unfolding hasty-rel-def
apply (erule gfp-elim2)
apply (rule mono-hasty-fun)
apply (unfold hasty-fun-def)
apply (drule CollectD)
apply (fold hasty-fun-def)
apply auto
done

```

```

lemma hasty-rel-elim:
  [[ (v,t) : hasty-rel;
    !! c t. c isof t ==> P (v-const c) t;
    !! te ev e t ve.
      [[ te |- fn ev => e ==> t;
        ve-dom(ve) = te-dom(te);
        !ev1. ev1:ve-dom(ve) --> (ve-app ve ev1,te-app te ev1) : hasty-rel
      ]] ==> P (v-clos <|ev,e,ve|>) t
    ]] ==> P v t
apply (rule-tac P = P in infsys-p2)
apply (rule hasty-rel-elim0)
apply auto
done

```

```

lemma hasty-const:  $c \text{ isof } t \implies v\text{-const}(c) \text{ hasty } t$ 
apply (unfold hasty-def)
apply (erule hasty-rel-const-coind)
done

```

```

lemma hasty-clos:
   $te \mid\!-\! \text{fn } ev \Rightarrow e \implies t \ \& \ ve \text{ hastyenv } te \implies v\text{-clos}(<\mid\!ev, e, ve\mid\!>) \text{ hasty } t$ 
apply (unfold hasty-def hasty-env-def)
apply (rule hasty-rel-clos-coind)
apply (blast del: equalityI)
done

```

```

lemma hasty-elim-const-lem:
   $v \text{ hasty } t \implies (!c.(v = v\text{-const}(c) \dashrightarrow c \text{ isof } t))$ 
apply (unfold hasty-def)
apply (rule hasty-rel-elim)
apply (blast intro!: v-disj-si elim!: v-disj-se dest!: v-injs)
done

```

```

lemma hasty-elim-const:  $v\text{-const}(c) \text{ hasty } t \implies c \text{ isof } t$ 
apply (drule hasty-elim-const-lem)
apply blast
done

```

```

lemma hasty-elim-clos-lem:
   $v \text{ hasty } t \implies$ 
   $! x e ve.$ 
   $v = v\text{-clos}(<\mid\!x, e, ve\mid\!>) \dashrightarrow (? te. te \mid\!-\! \text{fn } x \Rightarrow e \implies t \ \& \ ve \text{ hastyenv } te)$ 
apply (unfold hasty-env-def hasty-def)
apply (rule hasty-rel-elim)
apply (blast intro!: v-disj-si elim!: v-disj-se dest!: v-injs)
done

```

```

lemma hasty-elim-clos:  $v\text{-clos}(<\mid\!ev, e, ve\mid\!>) \text{ hasty } t \implies$ 
   $? te. te \mid\!-\! \text{fn } ev \Rightarrow e \implies t \ \& \ ve \text{ hastyenv } te$ 
apply (drule hasty-elim-clos-lem)
apply blast
done

```



```

lemma hasty-env1: [| ve hastyenv te; v hasty t |] ==>
  ve + {ev |-> v} hastyenv te + {ev ==> t}
apply (unfold hasty-env-def)
apply (simp del: mem-simps add: ve-dom-owr te-dom-owr)
apply (tactic << safe-tac HOL-cs >>)
apply (case-tac ev=x)
apply (simp (no-asm-simp) add: ve-app-owr1 te-app-owr1)
apply (simp add: ve-app-owr2 te-app-owr2)
done

```

```

lemma consistency-const: [| ve hastyenv te ; te |- e-const(c) ==> t |] ==>
  v-const(c) hasty t
apply (drule elab-const-elim)
apply (erule hasty-const)
done

```

```

lemma consistency-var:
  [| ev : ve-dom(ve); ve hastyenv te ; te |- e-var(ev) ==> t |] ==>
    ve-app ve ev hasty t
apply (unfold hasty-env-def)
apply (drule elab-var-elim)
apply blast
done

```

```

lemma consistency-fn: [| ve hastyenv te ; te |- fn ev ==> e ==> t |] ==>
  v-clos(<| ev, e, ve |>) hasty t
apply (rule hasty-clos)
apply blast
done

```

```

lemma consistency-fix:
  [| cl = <| ev1, e, ve + { ev2 |-> v-clos(cl) } |>;
    ve hastyenv te ;
    te |- fix ev2 ev1 = e ==> t
  |] ==>
    v-clos(cl) hasty t
apply (unfold hasty-env-def hasty-def)
apply (drule elab-fix-elim)
apply (tactic << safe-tac HOL-cs >>)

```

```

apply (frule ssubst) prefer 2 apply assumption
apply (rule hasty-rel-clos-coind)
apply (erule elab-fn)
apply (simp (no-asm-simp) add: ve-dom-owr te-dom-owr)

```

```

apply (simp (no-asm-simp) del: mem-simps add: ve-dom-owr)
apply (tactic ⟨⟨ safe-tac HOL-cs ⟩⟩)
apply (case-tac ev2=ev1a)
apply (simp (no-asm-simp) del: mem-simps add: ve-app-owr1 te-app-owr1)
apply blast
apply (simp add: ve-app-owr2 te-app-owr2)
done

lemma consistency-app1: [| ! t te. ve hastyenv te --> te |- e1 ==> t -->
v-const(c1) hasty t;
    ! t te. ve hastyenv te --> te |- e2 ==> t --> v-const(c2) hasty t;
    ve hastyenv te ; te |- e1 @@ e2 ==> t
|] ==>
    v-const(c-app c1 c2) hasty t
apply (drule elab-app-elim)
apply safe
apply (rule hasty-const)
apply (rule isof-app)
apply (rule hasty-elim-const)
apply blast
apply (rule hasty-elim-const)
apply blast
done

lemma consistency-app2: [| ! t te.
    ve hastyenv te -->
    te |- e1 ==> t --> v-clos(<|evm, em, vem|>) hasty t;
    ! t te. ve hastyenv te --> te |- e2 ==> t --> v2 hasty t;
    ! t te.
    vem + { evm |-> v2 } hastyenv te --> te |- em ==> t --> v hasty
t;
    ve hastyenv te ;
    te |- e1 @@ e2 ==> t
|] ==>
    v hasty t
apply (drule elab-app-elim)
apply safe
apply (erule allE, erule allE, erule impE)
apply assumption
apply (erule impE)
apply assumption
apply (erule allE, erule allE, erule impE)
apply assumption
apply (erule impE)
apply assumption
apply (drule hasty-elim-clos)
apply safe
apply (drule elab-fn-elim)

```

```

apply (blast intro: hasty-env1 dest!: t-fun-inj)
done

```

```

lemma consistency:  $ve \mid - e \dashv\dashv \rightarrow v \implies$ 
  ( $! t te. ve \text{hastyenv } te \dashv\dashv \rightarrow te \mid - e \implies t \dashv\dashv \rightarrow v \text{hasty } t$ )

```

```

apply (erule eval-ind)
apply safe
apply (blast intro: consistency-const consistency-var consistency-fn consistency-fix
  consistency-app1 consistency-app2)+
done

```

```

lemma basic-consistency-lem:
   $ve \text{isofenv } te \implies ve \text{hastyenv } te$ 
apply (unfold isof-env-def hasty-env-def)
apply safe
apply (erule allE)
apply (erule impE)
apply assumption
apply (erule exE)
apply (erule conjE)
apply (drule hasty-const)
apply (simp (no-asm-simp))
done

```

```

lemma basic-consistency:
  ( $[| ve \text{isofenv } te; ve \mid - e \dashv\dashv \rightarrow v\text{-const}(c); te \mid - e \implies t |] \implies c \text{isof } t$ )
apply (rule hasty-elim-const)
apply (drule consistency)
apply (blast intro!: basic-consistency-lem)
done

```

```

end

```

30 Case study: Unification Algorithm

```

theory Unification
imports Main
begin

```

This is a formalization of a first-order unification algorithm. It uses the new "function" package to define recursive functions, which allows a better

treatment of nested recursion.

This is basically a modernized version of a previous formalization by Konrad Slind (see: HOL/Subst/Unify.thy), which itself builds on previous work by Paulson and Manna & Waldinger (for details, see there).

Unlike that formalization, where the proofs of termination and some partial correctness properties are intertwined, we can prove partial correctness and termination separately.

30.1 Basic definitions

```
datatype 'a trm =
  Var 'a
| Const 'a
| App 'a trm 'a trm (infix · 60)
```

```
types
'a subst = ('a × 'a trm) list
```

Applying a substitution to a variable:

```
fun assoc :: 'a ⇒ 'b ⇒ ('a × 'b) list ⇒ 'b
where
  assoc x d [] = d
| assoc x d ((p,q)#t) = (if x = p then q else assoc x d t)
```

Applying a substitution to a term:

```
fun apply-subst :: 'a trm ⇒ 'a subst ⇒ 'a trm (infixl < 60)
where
  (Var v) < s = assoc v (Var v) s
| (Const c) < s = (Const c)
| (M · N) < s = (M < s) · (N < s)
```

Composition of substitutions:

```
fun
  compose :: 'a subst ⇒ 'a subst ⇒ 'a subst (infixl · 80)
where
  [] · bl = bl
| ((a,b) # al) · bl = (a, b < bl) # (al · bl)
```

Equivalence of substitutions:

```
definition eqv (infix =s 50)
where
  s1 =s s2 ≡ ∀ t. t < s1 = t < s2
```

30.2 Basic lemmas

```
lemma apply-empty[simp]: t < [] = t
by (induct t) auto
```

```

lemma compose-empty[simp]:  $\sigma \cdot [] = \sigma$ 
by (induct  $\sigma$ ) auto

lemma apply-compose[simp]:  $t \triangleleft (s1 \cdot s2) = t \triangleleft s1 \triangleleft s2$ 
proof (induct  $t$ )
  case App thus ?case by simp
next
  case Const thus ?case by simp
next
  case (Var  $v$ ) thus ?case
proof (induct  $s1$ )
  case Nil show ?case by simp
next
  case (Cons  $p$   $s1s$ ) thus ?case by (cases  $p$ , simp)
qed
qed

lemma eqv-refl[intro]:  $s =_s s$ 
by (auto simp: eqv-def)

lemma eqv-trans[trans]:  $\llbracket s1 =_s s2; s2 =_s s3 \rrbracket \implies s1 =_s s3$ 
by (auto simp: eqv-def)

lemma eqv-sym[sym]:  $\llbracket s1 =_s s2 \rrbracket \implies s2 =_s s1$ 
by (auto simp: eqv-def)

lemma eqv-intro[intro]:  $(\bigwedge t. t \triangleleft \sigma = t \triangleleft \vartheta) \implies \sigma =_s \vartheta$ 
by (auto simp: eqv-def)

lemma eqv-dest[dest]:  $s1 =_s s2 \implies t \triangleleft s1 = t \triangleleft s2$ 
by (auto simp: eqv-def)

lemma compose-equiv:  $\llbracket \sigma =_s \sigma'; \vartheta =_s \vartheta' \rrbracket \implies (\sigma \cdot \vartheta) =_s (\sigma' \cdot \vartheta')$ 
by (auto simp: eqv-def)

lemma compose-assoc:  $(a \cdot b) \cdot c =_s a \cdot (b \cdot c)$ 
by auto

```

30.3 Specification: Most general unifiers

definition

Unifier σ t $u \equiv (t \triangleleft \sigma = u \triangleleft \sigma)$

definition

MGU σ t $u \equiv \text{Unifier } \sigma$ t $u \wedge (\forall \vartheta. \text{Unifier } \vartheta$ t u
 $\longrightarrow (\exists \gamma. \vartheta =_s \sigma \cdot \gamma))$

lemma *MGUI*[intro]:

$\llbracket t \triangleleft \sigma = u \triangleleft \sigma; \bigwedge \vartheta. t \triangleleft \vartheta = u \triangleleft \vartheta \implies \exists \gamma. \vartheta =_s \sigma \cdot \gamma \rrbracket$
 $\implies MGU \sigma t u$
by (*simp only: Unifier-def MGU-def, auto*)

lemma *MGU-sym[sym]*:
 $MGU \sigma s t \implies MGU \sigma t s$
by (*auto simp: MGU-def Unifier-def*)

30.4 The unification algorithm

Occurs check: Proper subterm relation

fun *occ* :: 'a trm \Rightarrow 'a trm \Rightarrow bool
where
 $occ\ u\ (Var\ v) = False$
 $| occ\ u\ (Const\ c) = False$
 $| occ\ u\ (M \cdot N) = (u = M \vee u = N \vee occ\ u\ M \vee occ\ u\ N)$

The unification algorithm:

function *unify* :: 'a trm \Rightarrow 'a trm \Rightarrow 'a subst option
where
 $unify\ (Const\ c)\ (M \cdot N) = None$
 $| unify\ (M \cdot N)\ (Const\ c) = None$
 $| unify\ (Const\ c)\ (Var\ v) = Some\ [(v, Const\ c)]$
 $| unify\ (M \cdot N)\ (Var\ v) = (if\ (occ\ (Var\ v)\ (M \cdot N))$
 $\quad then\ None$
 $\quad else\ Some\ [(v, M \cdot N)])$
 $| unify\ (Var\ v)\ M = (if\ (occ\ (Var\ v)\ M)$
 $\quad then\ None$
 $\quad else\ Some\ [(v, M)])$
 $| unify\ (Const\ c)\ (Const\ d) = (if\ c=d\ then\ Some\ []\ else\ None)$
 $| unify\ (M \cdot N)\ (M' \cdot N') = (case\ unify\ M\ M'\ of$
 $\quad None \Rightarrow None \mid$
 $\quad Some\ \vartheta \Rightarrow (case\ unify\ (N \triangleleft \vartheta)\ (N' \triangleleft \vartheta)$
 $\quad of\ None \Rightarrow None \mid$
 $\quad Some\ \sigma \Rightarrow Some\ (\vartheta \cdot \sigma)))$
by *pat-completeness auto*

30.5 Partial correctness

Some lemmas about *occ* and *MGU*:

lemma *subst-no-occ*: $\neg occ\ (Var\ v)\ t \implies Var\ v \neq t$
 $\implies t \triangleleft [(v, s)] = t$
by (*induct t auto*)

lemma *MGU-Var[intro]*:
assumes *no-occ*: $\neg occ\ (Var\ v)\ t$
shows $MGU\ [(v, t)]\ (Var\ v)\ t$
proof (*intro MGUI exI*)

```

  show  $\text{Var } v \triangleleft [(v, t)] = t \triangleleft [(v, t)]$  using no-occ
    by (cases  $\text{Var } v = t$ , auto simp:subst-no-occ)
next
  fix  $\vartheta$  assume th:  $\text{Var } v \triangleleft \vartheta = t \triangleleft \vartheta$ 
  show  $\vartheta =_s [(v, t)] \cdot \vartheta$ 
  proof
    fix  $s$  show  $s \triangleleft \vartheta = s \triangleleft [(v, t)] \cdot \vartheta$  using th
    by (induct s) auto
  qed
qed

```

```

declare MGU-Var[symmetric, intro]

```

```

lemma MGU-Const[simp]: MGU [] (Const c) (Const d) = ( $c = d$ )
  unfolding MGU-def Unifier-def
  by auto

```

If unification terminates, then it computes most general unifiers:

```

lemma unify-partial-correctness:
  assumes unify-dom ( $M, N$ )
  assumes unify  $M N = \text{Some } \sigma$ 
  shows MGU  $\sigma M N$ 
using assms
proof (induct M N arbitrary:  $\sigma$ )
  case ( $\gamma M N M' N' \sigma$ ) — The interesting case

```

```

  then obtain  $\vartheta 1 \ \vartheta 2$ 
    where unify  $M M' = \text{Some } \vartheta 1$ 
    and unify ( $N \triangleleft \vartheta 1$ ) ( $N' \triangleleft \vartheta 1$ ) = Some  $\vartheta 2$ 
    and  $\sigma = \vartheta 1 \cdot \vartheta 2$ 
    and MGU-inner: MGU  $\vartheta 1 M M'$ 
    and MGU-outer: MGU  $\vartheta 2 (N \triangleleft \vartheta 1) (N' \triangleleft \vartheta 1)$ 
    by (auto split:option.split-asm)

```

```

  show ?case
  proof
    from MGU-inner and MGU-outer
    have  $M \triangleleft \vartheta 1 = M' \triangleleft \vartheta 1$ 
    and  $N \triangleleft \vartheta 1 \triangleleft \vartheta 2 = N' \triangleleft \vartheta 1 \triangleleft \vartheta 2$ 
    unfolding MGU-def Unifier-def
    by auto
    thus  $M \cdot N \triangleleft \sigma = M' \cdot N' \triangleleft \sigma$  unfolding  $\sigma$ 
    by simp
  next
    fix  $\sigma'$  assume  $M \cdot N \triangleleft \sigma' = M' \cdot N' \triangleleft \sigma'$ 
    hence  $M \triangleleft \sigma' = M' \triangleleft \sigma'$ 
    and  $Ns$ :  $N \triangleleft \sigma' = N' \triangleleft \sigma'$  by auto

    with MGU-inner obtain  $\delta$ 

```

```

where  $eqv: \sigma' =_s \vartheta 1 \cdot \delta$ 
unfolding  $MGU-def$   $Unifier-def$ 
by  $auto$ 

from  $Ns$  have  $N \triangleleft \vartheta 1 \triangleleft \delta = N' \triangleleft \vartheta 1 \triangleleft \delta$ 
by ( $simp$   $add:eqv-dest[OF eqv]$ )

with  $MGU-outer$  obtain  $\varrho$ 
where  $eqv2: \delta =_s \vartheta 2 \cdot \varrho$ 
unfolding  $MGU-def$   $Unifier-def$ 
by  $auto$ 

have  $\sigma' =_s \sigma \cdot \varrho$  unfolding  $\sigma$ 
by ( $rule$   $eqv-intro$ ,  $auto$   $simp:eqv-dest[OF eqv]$   $eqv-dest[OF eqv2]$ )
thus  $\exists \gamma. \sigma' =_s \sigma \cdot \gamma ..$ 
qed
qed ( $auto$   $split:split-if-asm$ ) — Solve the remaining cases automatically

```

30.6 Properties used in termination proof

The variables of a term:

```

fun  $vars-of:: 'a\ term \Rightarrow 'a\ set$ 
where
   $vars-of\ (Var\ v) = \{ v \}$ 
|  $vars-of\ (Const\ c) = \{\}$ 
|  $vars-of\ (M \cdot N) = vars-of\ M \cup vars-of\ N$ 

```

```

lemma  $vars-of-finite[intro]: finite\ (vars-of\ t)$ 
by ( $induct\ t$ )  $simp-all$ 

```

Elimination of variables by a substitution:

```

definition
   $elim\ \sigma\ v \equiv \forall t. v \notin vars-of\ (t \triangleleft \sigma)$ 

```

```

lemma  $elim-intro[intro]: (\bigwedge t. v \notin vars-of\ (t \triangleleft \sigma)) \implies elim\ \sigma\ v$ 
by ( $auto\ simp:elim-def$ )

```

```

lemma  $elim-dest[dest]: elim\ \sigma\ v \implies v \notin vars-of\ (t \triangleleft \sigma)$ 
by ( $auto\ simp:elim-def$ )

```

```

lemma  $elim-eqv: \sigma =_s \vartheta \implies elim\ \sigma\ x = elim\ \vartheta\ x$ 
by ( $auto\ simp:elim-def\ eqv-def$ )

```

Replacing a variable by itself yields an identity substitution:

```

lemma  $var-self[intro]: [(v, Var\ v)] =_s []$ 
proof
  fix  $t$  show  $t \triangleleft [(v, Var\ v)] = t \triangleleft []$ 
  by ( $induct\ t$ )  $simp-all$ 

```


qed

lemma *var-same*: $((v, t)] =_s [] = (t = \text{Var } v)$

proof

assume *t-v*: $t = \text{Var } v$

thus $[(v, t)] =_s []$

by *auto*

next

assume *id*: $[(v, t)] =_s []$

show $t = \text{Var } v$

proof –

have $t = \text{Var } v \triangleleft [(v, t)]$ **by** *simp*

also from *id* **have** $\dots = \text{Var } v \triangleleft []$ **..**

finally show *?thesis* **by** *simp*

qed

qed

A lemma about occ and elim

lemma *remove-var*:

assumes [*simp*]: $v \notin \text{vars-of } s$

shows $v \notin \text{vars-of } (t \triangleleft [(v, s)])$

by (*induct t*) *simp-all*

lemma *occ-elim*: $\neg \text{occ } (\text{Var } v) \ t$

$\implies \text{elim } [(v, t)] \ v \vee [(v, t)] =_s []$

proof (*induct t*)

case (*Var x*)

show *?case*

proof *cases*

assume $v = x$

thus *?thesis*

by (*simp add:var-same*)

next

assume *neq*: $v \neq x$

have $\text{elim } [(v, \text{Var } x)] \ v$

by (*auto intro!:remove-var simp:neq*)

thus *?thesis* **..**

qed

next

case (*Const c*)

have $\text{elim } [(v, \text{Const } c)] \ v$

by (*auto intro!:remove-var*)

thus *?case* **..**

next

case (*App M N*)

hence *ih1*: $\text{elim } [(v, M)] \ v \vee [(v, M)] =_s []$

and *ih2*: $\text{elim } [(v, N)] \ v \vee [(v, N)] =_s []$

and *nonocc*: $\text{Var } v \neq M \ \text{Var } v \neq N$

```

    by auto

  from nonocc have  $\neg [(v, M)] =_s []$ 
    by (simp add: var-same)
  with ih1 have elim  $[(v, M)] v$  by blast
  hence  $v \notin \text{vars-of } (Var\ v \triangleleft [(v, M)])$  ..
  hence not-in-M:  $v \notin \text{vars-of } M$  by simp

  from nonocc have  $\neg [(v, N)] =_s []$ 
    by (simp add: var-same)
  with ih2 have elim  $[(v, N)] v$  by blast
  hence  $v \notin \text{vars-of } (Var\ v \triangleleft [(v, N)])$  ..
  hence not-in-N:  $v \notin \text{vars-of } N$  by simp

  have elim  $[(v, M \cdot N)] v$ 
  proof
    fix t
    show  $v \notin \text{vars-of } (t \triangleleft [(v, M \cdot N)])$ 
    proof (induct t)
      case (Var x) thus ?case by (simp add: not-in-M not-in-N)
    qed auto
  qed
  thus ?case ..
qed

```

The result of a unification never introduces new variables:

```

lemma unify-vars:
  assumes unify-dom (M, N)
  assumes unify M N = Some  $\sigma$ 
  shows  $\text{vars-of } (t \triangleleft \sigma) \subseteq \text{vars-of } M \cup \text{vars-of } N \cup \text{vars-of } t$ 
  (is ?P M N  $\sigma$  t)
using assms
proof (induct M N arbitrary:  $\sigma$  t)
  case (3 c v)
  hence  $\sigma = [(v, Const\ c)]$  by simp
  thus ?case by (induct t) auto
next
  case (4 M N v)
  hence  $\neg \text{occ } (Var\ v) (M \cdot N)$  by (cases occ (Var v) (M · N), auto)
  with 4 have  $\sigma = [(v, M \cdot N)]$  by simp
  thus ?case by (induct t) auto
next
  case (5 v M)
  hence  $\neg \text{occ } (Var\ v) M$  by (cases occ (Var v) M, auto)
  with 5 have  $\sigma = [(v, M)]$  by simp
  thus ?case by (induct t) auto
next
  case (7 M N M' N'  $\sigma$ )
  then obtain  $\vartheta 1\ \vartheta 2$ 

```

```

where unify  $M M' = \text{Some } \vartheta 1$ 
and unify  $(N \triangleleft \vartheta 1) (N' \triangleleft \vartheta 1) = \text{Some } \vartheta 2$ 
and  $\sigma: \sigma = \vartheta 1 \cdot \vartheta 2$ 
and  $ih1: \bigwedge t. ?P M M' \vartheta 1 t$ 
and  $ih2: \bigwedge t. ?P (N \triangleleft \vartheta 1) (N' \triangleleft \vartheta 1) \vartheta 2 t$ 
by (auto split:option.split-asm)

show ?case
proof
  fix  $v$  assume  $a: v \in \text{vars-of } (t \triangleleft \sigma)$ 

  show  $v \in \text{vars-of } (M \cdot N) \cup \text{vars-of } (M' \cdot N') \cup \text{vars-of } t$ 
  proof (cases  $v \notin \text{vars-of } M \wedge v \notin \text{vars-of } M'$ 
     $\wedge v \notin \text{vars-of } N \wedge v \notin \text{vars-of } N'$ )
    case True
    with  $ih1$  have  $l: \bigwedge t. v \in \text{vars-of } (t \triangleleft \vartheta 1) \implies v \in \text{vars-of } t$ 
    by auto

    from  $a$  and  $ih2$  [where  $t = t \triangleleft \vartheta 1$ ]
    have  $v \in \text{vars-of } (N \triangleleft \vartheta 1) \cup \text{vars-of } (N' \triangleleft \vartheta 1)$ 
     $\vee v \in \text{vars-of } (t \triangleleft \vartheta 1)$  unfolding  $\sigma$ 
    by auto
    hence  $v \in \text{vars-of } t$ 
  proof
    assume  $v \in \text{vars-of } (N \triangleleft \vartheta 1) \cup \text{vars-of } (N' \triangleleft \vartheta 1)$ 
    with True show ?thesis by (auto dest:l)
  next
    assume  $v \in \text{vars-of } (t \triangleleft \vartheta 1)$ 
    thus ?thesis by (rule l)
  qed

  thus ?thesis by auto
qed auto
qed
qed (auto split: split-if-asm)

```

The result of a unification is either the identity substitution or it eliminates a variable from one of the terms:

```

lemma unify-eliminates:
  assumes unify-dom  $(M, N)$ 
  assumes unify  $M N = \text{Some } \sigma$ 
  shows  $(\exists v \in \text{vars-of } M \cup \text{vars-of } N. \text{elim } \sigma v) \vee \sigma =_s []$ 
  (is ?P  $M N \sigma$ )
using assms
proof (induct  $M N$  arbitrary:  $\sigma$ )
  case 1 thus ?case by simp
next
  case 2 thus ?case by simp
next

```

```

case (3 c v)
have no-occ:  $\neg \text{occ } (\text{Var } v) (\text{Const } c)$  by simp
with 3 have  $\sigma = [(v, \text{Const } c)]$  by simp
with occ-elim[OF no-occ]
show ?case by auto
next
case (4 M N v)
hence no-occ:  $\neg \text{occ } (\text{Var } v) (M \cdot N)$  by (cases occ (Var v) (M · N), auto)
with 4 have  $\sigma = [(v, M \cdot N)]$  by simp
with occ-elim[OF no-occ]
show ?case by auto
next
case (5 v M)
hence no-occ:  $\neg \text{occ } (\text{Var } v) M$  by (cases occ (Var v) M, auto)
with 5 have  $\sigma = [(v, M)]$  by simp
with occ-elim[OF no-occ]
show ?case by auto
next
case (6 c d) thus ?case
by (cases c = d) auto
next
case (7 M N M' N'  $\sigma$ )
then obtain  $\vartheta 1 \ \vartheta 2$ 
where unify M M' = Some  $\vartheta 1$ 
and unify (N  $\triangleleft \vartheta 1$ ) (N'  $\triangleleft \vartheta 1$ ) = Some  $\vartheta 2$ 
and  $\sigma: \sigma = \vartheta 1 \cdot \vartheta 2$ 
and ih1: ?P M M'  $\vartheta 1$ 
and ih2: ?P (N  $\triangleleft \vartheta 1$ ) (N'  $\triangleleft \vartheta 1$ )  $\vartheta 2$ 
by (auto split:option.split-asm)

from  $\langle \text{unify-dom } (M \cdot N, M' \cdot N') \rangle$ 
have unify-dom (M, M')
by (rule accp-downward) (rule unify-rel.intros)
hence no-new-vars:
 $\bigwedge t. \text{vars-of } (t \triangleleft \vartheta 1) \subseteq \text{vars-of } M \cup \text{vars-of } M' \cup \text{vars-of } t$ 
by (rule unify-vars) (rule  $\langle \text{unify } M M' = \text{Some } \vartheta 1 \rangle$ )

from ih2 show ?case
proof
assume  $\exists v \in \text{vars-of } (N \triangleleft \vartheta 1) \cup \text{vars-of } (N' \triangleleft \vartheta 1). \text{elim } \vartheta 2 \ v$ 
then obtain v
where  $v \in \text{vars-of } (N \triangleleft \vartheta 1) \cup \text{vars-of } (N' \triangleleft \vartheta 1)$ 
and el: elim  $\vartheta 2 \ v$  by auto
with no-new-vars show ?thesis unfolding  $\sigma$ 
by (auto simp:elim-def)
next
assume empty[simp]:  $\vartheta 2 =_s []$ 

have  $\sigma =_s (\vartheta 1 \cdot [])$  unfolding  $\sigma$ 

```

```

    by (rule compose-equiv) auto
  also have ... =s  $\vartheta 1$  by auto
  finally have  $\sigma =_s \vartheta 1$  .

from ih1 show ?thesis
proof
  assume  $\exists v \in \text{vars-of } M \cup \text{vars-of } M'. \text{ elim } \vartheta 1 v$ 
  with elim-equiv[OF  $\langle \sigma =_s \vartheta 1 \rangle$ ]
  show ?thesis by auto
next
  note  $\langle \sigma =_s \vartheta 1 \rangle$ 
  also assume  $\vartheta 1 =_s []$ 
  finally show ?thesis ..
qed
qed
qed

```

30.7 Termination proof

```

termination unify
proof
  let ?R = measures [ $\lambda(M, N). \text{ card } (\text{vars-of } M \cup \text{vars-of } N),$ 
                     $\lambda(M, N). \text{ size } M]$ 
  show wf ?R by simp

  fix M N M' N'
  show ((M, M'), (M · N, M' · N')) ∈ ?R — Inner call
    by (rule measures-lesseq) (auto intro: card-mono)

  fix  $\vartheta$  — Outer call
  assume inner: unify-dom (M, M')
    unify M M' = Some  $\vartheta$ 

  from unify-eliminates[OF inner]
  show ((N <  $\vartheta$ , N' <  $\vartheta$ ), (M · N, M' · N')) ∈ ?R
  proof
    — Either a variable is eliminated ...
    assume ( $\exists v \in \text{vars-of } M \cup \text{vars-of } M'. \text{ elim } \vartheta v$ )
    then obtain v
      where elim  $\vartheta v$ 
      and  $v \in \text{vars-of } M \cup \text{vars-of } M'$  by auto
    with unify-vars[OF inner]
    have vars-of (N <  $\vartheta$ ) ∪ vars-of (N' <  $\vartheta$ )
      ⊆ vars-of (M · N) ∪ vars-of (M' · N')
      by auto

    thus ?thesis
      by (auto intro!: measures-less intro: psubset-card-mono)
  next

```

— Or the substitution is empty
assume $\vartheta =_s []$
hence $N \triangleleft \vartheta = N$
and $N' \triangleleft \vartheta = N'$ **by** *auto*
thus *?thesis*
by (*auto intro!:: measures-less intro: psubset-card-mono*)
qed
qed
end

31 Primitive Recursive Functions

theory *Primrec* **imports** *Main* **begin**

Proof adopted from

Nora Szasz, A Machine Checked Proof that Ackermann's Function is not Primitive Recursive, In: Huet & Plotkin, eds., Logical Environments (CUP, 1993), 317-338.

See also E. Mendelson, Introduction to Mathematical Logic. (Van Nostrand, 1964), page 250, exercise 11.

31.1 Ackermann's Function

fun *ack* :: *nat* => *nat* => *nat* **where**
ack 0 *n* = *Suc* *n* |
ack (*Suc* *m*) 0 = *ack* *m* 1 |
ack (*Suc* *m*) (*Suc* *n*) = *ack* *m* (*ack* (*Suc* *m*) *n*)

PROPERTY A 4

lemma *less-ack2* [*iff*]: $j < \text{ack } i \ j$
by (*induct* *i j* *rule: ack.induct*) *simp-all*

PROPERTY A 5-, the single-step lemma

lemma *ack-less-ack-Suc2* [*iff*]: $\text{ack } i \ j < \text{ack } i \ (\text{Suc } j)$
by (*induct* *i j* *rule: ack.induct*) *simp-all*

PROPERTY A 5, monotonicity for <

lemma *ack-less-mono2*: $j < k \implies \text{ack } i \ j < \text{ack } i \ k$
using *lift-Suc-mono-less* [**where** $f = \text{ack } i$]
by (*metis* *ack-less-ack-Suc2*)

PROPERTY A 5', monotonicity for \leq

lemma *ack-le-mono2*: $j \leq k \implies \text{ack } i \ j \leq \text{ack } i \ k$
apply (*simp* *add: order-le-less*)

apply (*blast intro: ack-less-mono2*)
done

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: $ack\ i\ (Suc\ j) \leq ack\ (Suc\ i)\ j$
proof (*induct j*)
 case 0 **show** ?*case* **by** *simp*
next
 case (*Suc j*) **show** ?*case*
 by (*auto intro!: ack-le-mono2*)
 (*metis Suc Suc-leI Suc-lessI less-ack2 linorder-not-less*)
qed

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: $ack\ i\ j < ack\ (Suc\ i)\ j$
by (*blast intro: ack-less-mono2 less-le-trans*)

PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions

lemma *less-ack1* [*iff*]: $i < ack\ i\ j$
apply (*induct i*)
 apply *simp-all*
apply (*blast intro: Suc-leI le-less-trans*)
done

PROPERTY A 8

lemma *ack-1* [*simp*]: $ack\ (Suc\ 0)\ j = j + 2$
by (*induct j*) *simp-all*

PROPERTY A 9. The unary 1 and 2 in *ack* is essential for the rewriting.

lemma *ack-2* [*simp*]: $ack\ (Suc\ (Suc\ 0))\ j = 2 * j + 3$
by (*induct j*) *simp-all*

PROPERTY A 7, monotonicity for $<$ [not clear why *ack-1* is now needed first!]

lemma *ack-less-mono1-aux*: $ack\ i\ k < ack\ (Suc\ (i + i'))\ k$
proof (*induct i k rule: ack.induct*)
 case (1 *n*) **show** ?*case*
 by (*simp, metis ack-less-ack-Suc1 less-ack2 less-trans-Suc*)
next
 case (2 *m*) **thus** ?*case* **by** *simp*
next
 case (3 *m n*) **thus** ?*case*
 by (*simp, blast intro: less-trans ack-less-mono2*)
qed

lemma *ack-less-mono1*: $i < j ==> ack\ i\ k < ack\ j\ k$

```

apply (drule less-imp-Suc-add)
apply (blast intro!: ack-less-mono1-aux)
done

```

PROPERTY A 7', monotonicity for \leq

```

lemma ack-le-mono1:  $i \leq j \implies \text{ack } i \ k \leq \text{ack } j \ k$ 
apply (simp add: order-le-less)
apply (blast intro: ack-less-mono1)
done

```

PROPERTY A 10

```

lemma ack-nest-bound:  $\text{ack } i1 \ (\text{ack } i2 \ j) < \text{ack } (2 + (i1 + i2)) \ j$ 
apply (simp add: numerals)
apply (rule ack2-le-ack1 [THEN [2] less-le-trans])
apply simp
apply (rule le-add1 [THEN ack-le-mono1, THEN le-less-trans])
apply (rule ack-less-mono1 [THEN ack-less-mono2])
apply (simp add: le-imp-less-Suc le-add2)
done

```

PROPERTY A 11

```

lemma ack-add-bound:  $\text{ack } i1 \ j + \text{ack } i2 \ j < \text{ack } (4 + (i1 + i2)) \ j$ 
apply (rule less-trans [of - ack (Suc (Suc 0)) (ack (i1 + i2) j)])
prefer 2
apply (rule ack-nest-bound [THEN less-le-trans])
apply (simp add: Suc3-eq-add-3)
apply simp
apply (cut-tac i = i1 and m1 = i2 and k = j in le-add1 [THEN ack-le-mono1])
apply (cut-tac i = i2 and m1 = i1 and k = j in le-add2 [THEN ack-le-mono1])
apply auto
done

```

PROPERTY A 12. Article uses existential quantifier but the ALF proof used $k + 4$. Quantified version must be nested $\exists k'. \forall i \ j. \dots$

```

lemma ack-add-bound2:  $i < \text{ack } k \ j \implies i + j < \text{ack } (4 + k) \ j$ 
apply (rule less-trans [of - ack k j + ack 0 j])
apply (blast intro: add-less-mono less-ack2)
apply (rule ack-add-bound [THEN less-le-trans])
apply simp
done

```

31.2 Primitive Recursive Functions

```

primrec hd0 :: nat list => nat where
  hd0 [] = 0 |
  hd0 (m # ms) = m

```

Inductive definition of the set of primitive recursive functions of type *nat list* \Rightarrow *nat*.

definition $SC :: nat \Rightarrow nat$ **where**
 $SC\ l = Suc\ (hd0\ l)$

definition $CONSTANT :: nat \Rightarrow nat \Rightarrow nat$ **where**
 $CONSTANT\ k\ l = k$

definition $PROJ :: nat \Rightarrow nat \Rightarrow nat$ **where**
 $PROJ\ i\ l = hd0\ (drop\ i\ l)$

definition
 $COMP :: (nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$
where $COMP\ g\ fs\ l = g\ (map\ (\lambda f. f\ l)\ fs)$

definition $PREC :: (nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$
where
 $PREC\ f\ g\ l =$
 $(case\ l\ of$
 $\quad [] \Rightarrow 0$
 $\quad | x \# l' \Rightarrow nat-rec\ (f\ l')\ (\lambda y\ r. g\ (r \# y \# l'))\ x)$
 — Note that g is applied first to $PREC\ f\ g\ y$ and then to y !

inductive $PRIMREC :: (nat \Rightarrow nat) \Rightarrow bool$ **where**
 $SC: PRIMREC\ SC \mid$
 $CONSTANT: PRIMREC\ (CONSTANT\ k) \mid$
 $PROJ: PRIMREC\ (PROJ\ i) \mid$
 $COMP: PRIMREC\ g \implies \forall f \in set\ fs. PRIMREC\ f \implies PRIMREC\ (COMP\ g\ fs) \mid$
 $PREC: PRIMREC\ f \implies PRIMREC\ g \implies PRIMREC\ (PREC\ f\ g)$

Useful special cases of evaluation

lemma $SC\ [simp]: SC\ (x \# l) = Suc\ x$
by ($simp\ add: SC-def$)

lemma $CONSTANT\ [simp]: CONSTANT\ k\ l = k$
by ($simp\ add: CONSTANT-def$)

lemma $PROJ-0\ [simp]: PROJ\ 0\ (x \# l) = x$
by ($simp\ add: PROJ-def$)

lemma $COMP-1\ [simp]: COMP\ g\ [f]\ l = g\ [f\ l]$
by ($simp\ add: COMP-def$)

lemma $PREC-0\ [simp]: PREC\ f\ g\ (0 \# l) = f\ l$
by ($simp\ add: PREC-def$)

lemma $PREC-Suc\ [simp]: PREC\ f\ g\ (Suc\ x \# l) = g\ (PREC\ f\ g\ (x \# l) \# x \# l)$
by ($simp\ add: PREC-def$)

MAIN RESULT

```

lemma SC-case:  $SC\ l < ack\ 1\ (listsum\ l)$ 
apply (unfold SC-def)
apply (induct l)
apply (simp-all add: le-add1 le-imp-less-Suc)
done

```

```

lemma CONSTANT-case:  $CONSTANT\ k\ l < ack\ k\ (listsum\ l)$ 
by simp

```

```

lemma PROJ-case:  $PROJ\ i\ l < ack\ 0\ (listsum\ l)$ 
apply (simp add: PROJ-def)
apply (induct l arbitrary:i)
apply (auto simp add: drop-Cons split: nat.split)
apply (blast intro: less-le-trans le-add2)
done

```

COMP case

```

lemma COMP-map-aux:  $\forall f \in set\ fs. PRIMREC\ f \wedge (\exists kf. \forall l. f\ l < ack\ kf\ (listsum\ l))$ 
 $\implies \exists k. \forall l. listsum\ (map\ (\lambda f. f\ l)\ fs) < ack\ k\ (listsum\ l)$ 
apply (induct fs)
apply (rule-tac x = 0 in exI)
apply simp
apply simp
apply (blast intro: add-less-mono ack-add-bound less-trans)
done

```

```

lemma COMP-case:
 $\forall l. g\ l < ack\ kg\ (listsum\ l) \implies$ 
 $\forall f \in set\ fs. PRIMREC\ f \wedge (\exists kf. \forall l. f\ l < ack\ kf\ (listsum\ l))$ 
 $\implies \exists k. \forall l. COMP\ g\ fs\ l < ack\ k\ (listsum\ l)$ 
apply (unfold COMP-def)
apply (drule COMP-map-aux)
apply (meson ack-less-mono2 ack-nest-bound less-trans)
done

```

PREC case

```

lemma PREC-case-aux:
 $\forall l. f\ l + listsum\ l < ack\ kf\ (listsum\ l) \implies$ 
 $\forall l. g\ l + listsum\ l < ack\ kg\ (listsum\ l) \implies$ 
 $PREC\ f\ g\ l + listsum\ l < ack\ (Suc\ (kf + kg))\ (listsum\ l)$ 
apply (unfold PREC-def)
apply (case-tac l)
apply simp-all
apply (blast intro: less-trans)
apply (erule ssubst) — get rid of the needless assumption
apply (induct-tac a)
apply simp-all

```

base case

```

apply (blast intro: le-add1 [THEN le-imp-less-Suc, THEN ack-less-mono1] less-trans)

induction step

apply (rule Suc-leI [THEN le-less-trans])
apply (rule le-refl [THEN add-le-mono, THEN le-less-trans])
  prefer 2
  apply (erule spec)
apply (simp add: le-add2)

final part of the simplification

apply simp
apply (rule le-add2 [THEN ack-le-mono1, THEN le-less-trans])
apply (erule ack-less-mono2)
done

lemma PREC-case:
   $\forall l. f\ l < \text{ack}\ kf\ (\text{listsum}\ l) ==>$ 
   $\forall l. g\ l < \text{ack}\ kg\ (\text{listsum}\ l) ==>$ 
   $\exists k. \forall l. \text{PREC}\ f\ g\ l < \text{ack}\ k\ (\text{listsum}\ l)$ 
by (metis le-less-trans [OF le-add1 PREC-case-aux] ack-add-bound2)

lemma ack-bounds-PRIMREC: PRIMREC  $f ==> \exists k. \forall l. f\ l < \text{ack}\ k\ (\text{listsum}\ l)$ 
apply (erule PRIMREC.induct)
  apply (blast intro: SC-case CONSTANT-case PROJ-case COMP-case PREC-case)+
done

theorem ack-not-PRIMREC:
   $\neg \text{PRIMREC}\ (\lambda l. \text{case}\ l\ \text{of}\ [] ==> 0 \mid x \# l' ==> \text{ack}\ x\ x)$ 
apply (rule notI)
apply (erule ack-bounds-PRIMREC [THEN exE])
apply (rule less-irrefl [THEN notE])
apply (drule-tac  $x = [x]$  in spec)
apply simp
done

end

```

32 The Full Theorem of Tarski

```

theory Tarski
imports Main FuncSet
begin

```

Minimal version of lattice theory plus the full theorem of Tarski: The fixed-points of a complete lattice themselves form a complete lattice.

Illustrates first-class theories, using the Sigma representation of structures. Tidied and converted to Isar by lcp.

record 'a potype =
 pset :: 'a set
 order :: ('a * 'a) set

definition

monotone :: ['a => 'a, 'a set, ('a * 'a) set] => bool **where**
 monotone f A r = ($\forall x \in A. \forall y \in A. (x, y): r \longrightarrow ((f\ x), (f\ y)) : r$)

definition

least :: ['a => bool, 'a potype] => 'a **where**
 least P po = (SOME x. x: pset po & P x &
 ($\forall y \in pset\ po. P\ y \longrightarrow (x, y): order\ po$))

definition

greatest :: ['a => bool, 'a potype] => 'a **where**
 greatest P po = (SOME x. x: pset po & P x &
 ($\forall y \in pset\ po. P\ y \longrightarrow (y, x): order\ po$))

definition

lub :: ['a set, 'a potype] => 'a **where**
 lub S po = least (%x. $\forall y \in S. (y, x): order\ po$) po

definition

glb :: ['a set, 'a potype] => 'a **where**
 glb S po = greatest (%x. $\forall y \in S. (x, y): order\ po$) po

definition

isLub :: ['a set, 'a potype, 'a] => bool **where**
 isLub S po = (%L. (L: pset po & ($\forall y \in S. (y, L): order\ po$) &
 ($\forall z \in pset\ po. (\forall y \in S. (y, z): order\ po) \longrightarrow (L, z): order\ po$)))

definition

isGlb :: ['a set, 'a potype, 'a] => bool **where**
 isGlb S po = (%G. (G: pset po & ($\forall y \in S. (G, y): order\ po$) &
 ($\forall z \in pset\ po. (\forall y \in S. (z, y): order\ po) \longrightarrow (z, G): order\ po$)))

definition

fix :: [('a => 'a), 'a set] => 'a set **where**
 fix f A = {x. x: A & f x = x}

definition

interval :: [('a * 'a) set, 'a, 'a] => 'a set **where**
 interval r a b = {x. (a, x): r & (x, b): r}

definition

Bot :: 'a potype => 'a **where**
 Bot po = least (%x. True) po

definition

Top :: 'a potype => 'a **where**
Top po = *greatest* (%x. *True*) *po*

definition

PartialOrder :: ('a potype) set **where**
PartialOrder = {*P*. *refl-on* (*pset P*) (*order P*) & *antisym* (*order P*) &
trans (*order P*)}

definition

CompleteLattice :: ('a potype) set **where**
CompleteLattice = {*cl*. *cl*: *PartialOrder* &
($\forall S. S \subseteq \text{pset } cl \longrightarrow (\exists L. \text{isLub } S \text{ } cl \text{ } L)$) &
($\forall S. S \subseteq \text{pset } cl \longrightarrow (\exists G. \text{isGlb } S \text{ } cl \text{ } G)$)}

definition

CLF-set :: ('a potype * ('a => 'a)) set **where**
CLF-set = (*SIGMA cl*: *CompleteLattice*.
{*f*. *f*: *pset cl* -> *pset cl* & *monotone f* (*pset cl*) (*order cl*)})

definition

induced :: ['a set, ('a * 'a) set] => ('a * 'a) set **where**
induced A r = {(*a*,*b*). *a* : *A* & *b*: *A* & (*a*,*b*): *r*}

definition

sublattice :: ('a potype * 'a set) set **where**
sublattice =
(*SIGMA cl*: *CompleteLattice*.
{*S*. *S* \subseteq *pset cl* &
(\mid *pset* = *S*, *order* = *induced S* (*order cl*) \mid): *CompleteLattice*})

abbreviation

sublat :: ['a set, 'a potype] => bool (- <=<= - [51,50]50) **where**
S <=<= *cl* == *S* : *sublattice* “ {*cl*}

definition

dual :: 'a potype => 'a potype **where**
dual po = (\mid *pset* = *pset po*, *order* = *converse* (*order po*) \mid)

locale S =

fixes *cl* :: 'a potype
and *A* :: 'a set
and *r* :: ('a * 'a) set
defines *A-def*: *A* == *pset cl*
and *r-def*: *r* == *order cl*

locale PO = S +

assumes *cl-po*: *cl* : *PartialOrder*

```

locale  $CL = S +$ 
  assumes  $cl-co$ :  $cl : CompleteLattice$ 

sublocale  $CL < PO$ 
apply ( $simp-all$   $add: A-def\ r-def$ )
apply  $unfold-locales$ 
using  $cl-co$  unfolding  $CompleteLattice-def$  by  $auto$ 

locale  $CLF = S +$ 
  fixes  $f :: 'a ==> 'a$ 
  and  $P :: 'a\ set$ 
  assumes  $f-cl$ :  $(cl, f) : CLF-set$ 
  defines  $P-def$ :  $P == fix\ f\ A$ 

sublocale  $CLF < CL$ 
apply ( $simp-all$   $add: A-def\ r-def$ )
apply  $unfold-locales$ 
using  $f-cl$  unfolding  $CLF-set-def$  by  $auto$ 

locale  $Tarski = CLF +$ 
  fixes  $Y :: 'a\ set$ 
  and  $intY1 :: 'a\ set$ 
  and  $v :: 'a$ 
  assumes
     $Y-ss$ :  $Y \subseteq P$ 
  defines
     $intY1-def$ :  $intY1 == interval\ r\ (lub\ Y\ cl)\ (Top\ cl)$ 
    and  $v-def$ :  $v == glb\ \{x. ((\%x: intY1. f\ x)\ x, x): induced\ intY1\ r\ \&$ 
       $x: intY1\}$ 
       $(\mid\ pset=intY1, order=induced\ intY1\ r\ \mid)$ 

```

32.1 Partial Order

```

lemma (in  $PO$ )  $dual$ :
   $PO\ (dual\ cl)$ 
apply  $unfold-locales$ 
using  $cl-po$ 
unfolding  $PartialOrder-def\ dual-def$ 
by  $auto$ 

lemma (in  $PO$ )  $PO-imp-refl-on$  [ $simp$ ]:  $refl-on\ A\ r$ 
apply ( $insert\ cl-po$ )
apply ( $simp\ add: PartialOrder-def\ A-def\ r-def$ )
done

lemma (in  $PO$ )  $PO-imp-sym$  [ $simp$ ]:  $antisym\ r$ 
apply ( $insert\ cl-po$ )
apply ( $simp\ add: PartialOrder-def\ r-def$ )

```

done

lemma (in *PO*) *PO-imp-trans* [*simp*]: *trans r*
apply (*insert cl-po*)
apply (*simp add: PartialOrder-def r-def*)
done

lemma (in *PO*) *reflE*: $x \in A \implies (x, x) \in r$
apply (*insert cl-po*)
apply (*simp add: PartialOrder-def refl-on-def A-def r-def*)
done

lemma (in *PO*) *antisymE*: $[(a, b) \in r; (b, a) \in r] \implies a = b$
apply (*insert cl-po*)
apply (*simp add: PartialOrder-def antisym-def r-def*)
done

lemma (in *PO*) *transE*: $[(a, b) \in r; (b, c) \in r] \implies (a, c) \in r$
apply (*insert cl-po*)
apply (*simp add: PartialOrder-def r-def*)
apply (*unfold trans-def, fast*)
done

lemma (in *PO*) *monotoneE*:
 $[\text{monotone } f \ A \ r; \ x \in A; \ y \in A; \ (x, y) \in r] \implies (f \ x, f \ y) \in r$
by (*simp add: monotone-def*)

lemma (in *PO*) *po-subset-po*:
 $S \subseteq A \implies (| \text{pset} = S, \text{order} = \text{induced } S \ r |) \in \text{PartialOrder}$
apply (*simp (no-asm) add: PartialOrder-def*)
apply *auto*
— *refl*
apply (*simp add: refl-on-def induced-def*)
apply (*blast intro: reflE*)
— *antisym*
apply (*simp add: antisym-def induced-def*)
apply (*blast intro: antisymE*)
— *trans*
apply (*simp add: trans-def induced-def*)
apply (*blast intro: transE*)
done

lemma (in *PO*) *indE*: $[(x, y) \in \text{induced } S \ r; \ S \subseteq A] \implies (x, y) \in r$
by (*simp add: add: induced-def*)

lemma (in *PO*) *indI*: $[(x, y) \in r; \ x \in S; \ y \in S] \implies (x, y) \in \text{induced } S \ r$
by (*simp add: add: induced-def*)

lemma (in *CL*) *CL-imp-ex-isLub*: $S \subseteq A \implies \exists L. \text{isLub } S \ cl \ L$

```

apply (insert cl-co)
apply (simp add: CompleteLattice-def A-def)
done

declare (in CL) cl-co [simp]

lemma isLub-lub:  $(\exists L. \text{isLub } S \text{ cl } L) = \text{isLub } S \text{ cl } (\text{lub } S \text{ cl})$ 
by (simp add: lub-def least-def isLub-def some-eq-ex [symmetric])

lemma isGlb-glb:  $(\exists G. \text{isGlb } S \text{ cl } G) = \text{isGlb } S \text{ cl } (\text{glb } S \text{ cl})$ 
by (simp add: glb-def greatest-def isGlb-def some-eq-ex [symmetric])

lemma isGlb-dual-isLub:  $\text{isGlb } S \text{ cl} = \text{isLub } S \text{ (dual cl)}$ 
by (simp add: isLub-def isGlb-def dual-def converse-def)

lemma isLub-dual-isGlb:  $\text{isLub } S \text{ cl} = \text{isGlb } S \text{ (dual cl)}$ 
by (simp add: isLub-def isGlb-def dual-def converse-def)

lemma (in PO) dualPO:  $\text{dual cl} \in \text{PartialOrder}$ 
apply (insert cl-po)
apply (simp add: PartialOrder-def dual-def refl-on-converse
      trans-converse antisym-converse)
done

lemma Rdual:
   $\forall S. (S \subseteq A \dashv\dashv (\exists L. \text{isLub } S \text{ (| pset = A, order = r|) } L))$ 
   $\implies \forall S. (S \subseteq A \dashv\dashv (\exists G. \text{isGlb } S \text{ (| pset = A, order = r|) } G))$ 
apply safe
apply (rule-tac  $x = \text{lub } \{y. y \in A \ \& \ (\forall k \in S. (y, k) \in r)\}$ 
       $(| \text{pset} = A, \text{order} = r|)$  in exI)
apply (drule-tac  $x = \{y. y \in A \ \& \ (\forall k \in S. (y, k) \in r)\}$  in spec)
apply (drule mp, fast)
apply (simp add: isLub-lub isGlb-def)
apply (simp add: isLub-def, blast)
done

lemma lub-dual-glb:  $\text{lub } S \text{ cl} = \text{glb } S \text{ (dual cl)}$ 
by (simp add: lub-def glb-def least-def greatest-def dual-def converse-def)

lemma glb-dual-lub:  $\text{glb } S \text{ cl} = \text{lub } S \text{ (dual cl)}$ 
by (simp add: lub-def glb-def least-def greatest-def dual-def converse-def)

lemma CL-subset-PO:  $\text{CompleteLattice} \subseteq \text{PartialOrder}$ 
by (simp add: PartialOrder-def CompleteLattice-def, fast)

lemmas CL-imp-PO = CL-subset-PO [THEN subsetD]

```



```

lemma (in CL) CO-refl-on: refl-on A r
by (rule PO-imp-refl-on)

lemma (in CL) CO-antisym: antisym r
by (rule PO-imp-sym)

lemma (in CL) CO-trans: trans r
by (rule PO-imp-trans)

lemma CompleteLatticeI:
  [| po ∈ PartialOrder; (∀ S. S ⊆ pset po --> (∃ L. isLub S po L));
    (∀ S. S ⊆ pset po --> (∃ G. isGlb S po G)) |]
  ==> po ∈ CompleteLattice
apply (unfold CompleteLattice-def, blast)
done

lemma (in CL) CL-dualCL: dual cl ∈ CompleteLattice
apply (insert cl-co)
apply (simp add: CompleteLattice-def dual-def)
apply (fold dual-def)
apply (simp add: isLub-dual-isGlb [symmetric] isGlb-dual-isLub [symmetric]
        dualPO)
done

lemma (in PO) dualA-iff: pset (dual cl) = pset cl
by (simp add: dual-def)

lemma (in PO) dualr-iff: ((x, y) ∈ (order(dual cl))) = ((y, x) ∈ order cl)
by (simp add: dual-def)

lemma (in PO) monotone-dual:
  monotone f (pset cl) (order cl)
  ==> monotone f (pset (dual cl)) (order(dual cl))
by (simp add: monotone-def dualA-iff dualr-iff)

lemma (in PO) interval-dual:
  [| x ∈ A; y ∈ A |] ==> interval r x y = interval (order(dual cl)) y x
apply (simp add: interval-def dualr-iff)
apply (fold r-def, fast)
done

lemma (in PO) trans:
  (x, y) ∈ r ==> (y, z) ∈ r ==> (x, z) ∈ r
using cl-po apply (auto simp add: PartialOrder-def r-def)
unfolding trans-def by blast

lemma (in PO) interval-not-empty:
  interval r a b ≠ {} ==> (a, b) ∈ r
apply (simp add: interval-def)

```

using *trans* **by** *blast*

lemma (in *PO*) *interval-imp-mem*: $x \in \text{interval } r \ a \ b \implies (a, x) \in r$
by (*simp add: interval-def*)

lemma (in *PO*) *left-in-interval*:
 $[| a \in A; b \in A; \text{interval } r \ a \ b \neq \{\} |] \implies a \in \text{interval } r \ a \ b$
apply (*simp (no-asm-simp) add: interval-def*)
apply (*simp add: PO-imp-trans interval-not-empty*)
apply (*simp add: reflE*)
done

lemma (in *PO*) *right-in-interval*:
 $[| a \in A; b \in A; \text{interval } r \ a \ b \neq \{\} |] \implies b \in \text{interval } r \ a \ b$
apply (*simp (no-asm-simp) add: interval-def*)
apply (*simp add: PO-imp-trans interval-not-empty*)
apply (*simp add: reflE*)
done

32.2 sublattice

lemma (in *PO*) *sublattice-imp-CL*:
 $S \leqslant cl \implies (| \text{pset} = S, \text{order} = \text{induced } S \ r \ |) \in \text{CompleteLattice}$
by (*simp add: sublattice-def CompleteLattice-def r-def*)

lemma (in *CL*) *sublatticeI*:
 $[| S \subseteq A; (| \text{pset} = S, \text{order} = \text{induced } S \ r \ |) \in \text{CompleteLattice} |]$
 $\implies S \leqslant cl$
by (*simp add: sublattice-def A-def r-def*)

lemma (in *CL*) *dual*:
 $CL \ (\text{dual } cl)$
apply *unfold-locales*
using *cl-co unfolding CompleteLattice-def*
apply (*simp add: dualPO isGlb-dual-isLub [symmetric] isLub-dual-isGlb [symmetric]*
dualA-iff)
done

32.3 lub

lemma (in *CL*) *lub-unique*: $[| S \subseteq A; \text{isLub } S \ cl \ x; \text{isLub } S \ cl \ L |] \implies x = L$
apply (*rule antisymE*)
apply (*auto simp add: isLub-def r-def*)
done

lemma (in *CL*) *lub-upper*: $[| S \subseteq A; x \in S |] \implies (x, \text{lub } S \ cl) \in r$
apply (*rule CL-imp-ex-isLub [THEN exE], assumption*)
apply (*unfold lub-def least-def*)
apply (*rule some-equality [THEN ssubst]*)
apply (*simp add: isLub-def*)

```

  apply (simp add: lub-unique A-def isLub-def)
apply (simp add: isLub-def r-def)
done

```

```

lemma (in CL) lub-least:
  [| S ⊆ A; L ∈ A; ∀ x ∈ S. (x,L) ∈ r |] ==> (lub S cl, L) ∈ r
  apply (rule CL-imp-ex-isLub [THEN exE], assumption)
  apply (unfold lub-def least-def)
  apply (rule-tac s=x in some-equality [THEN ssubst])
    apply (simp add: isLub-def)
    apply (simp add: lub-unique A-def isLub-def)
  apply (simp add: isLub-def r-def A-def)
done

```

```

lemma (in CL) lub-in-lattice: S ⊆ A ==> lub S cl ∈ A
  apply (rule CL-imp-ex-isLub [THEN exE], assumption)
  apply (unfold lub-def least-def)
  apply (subst some-equality)
  apply (simp add: isLub-def)
  prefer 2 apply (simp add: isLub-def A-def)
  apply (simp add: lub-unique A-def isLub-def)
done

```

```

lemma (in CL) lubI:
  [| S ⊆ A; L ∈ A; ∀ x ∈ S. (x,L) ∈ r;
    ∀ z ∈ A. (∀ y ∈ S. (y,z) ∈ r) --> (L,z) ∈ r |] ==> L = lub S cl
  apply (rule lub-unique, assumption)
  apply (simp add: isLub-def A-def r-def)
  apply (unfold isLub-def)
  apply (rule conjI)
  apply (fold A-def r-def)
  apply (rule lub-in-lattice, assumption)
  apply (simp add: lub-upper lub-least)
done

```

```

lemma (in CL) lubIa: [| S ⊆ A; isLub S cl L |] ==> L = lub S cl
by (simp add: lubI isLub-def A-def r-def)

```

```

lemma (in CL) isLub-in-lattice: isLub S cl L ==> L ∈ A
by (simp add: isLub-def A-def)

```

```

lemma (in CL) isLub-upper: [| isLub S cl L; y ∈ S |] ==> (y, L) ∈ r
by (simp add: isLub-def r-def)

```

```

lemma (in CL) isLub-least:
  [| isLub S cl L; z ∈ A; ∀ y ∈ S. (y, z) ∈ r |] ==> (L, z) ∈ r
by (simp add: isLub-def A-def r-def)

```

```

lemma (in CL) isLubI:

```

```

    [| L ∈ A; ∀ y ∈ S. (y, L) ∈ r;
      (∀ z ∈ A. (∀ y ∈ S. (y, z):r) --> (L, z) ∈ r) |] ==> isLub S cl L
  by (simp add: isLub-def A-def r-def)

```

32.4 glb

```

lemma (in CL) glb-in-lattice: S ⊆ A ==> glb S cl ∈ A
apply (subst glb-dual-lub)
apply (simp add: A-def)
apply (rule dualA-iff [THEN subst])
apply (rule CL.lub-in-lattice)
apply (rule dual)
apply (simp add: dualA-iff)
done

```

```

lemma (in CL) glb-lower: [| S ⊆ A; x ∈ S |] ==> (glb S cl, x) ∈ r
apply (subst glb-dual-lub)
apply (simp add: r-def)
apply (rule dualr-iff [THEN subst])
apply (rule CL.lub-upper)
apply (rule dual)
apply (simp add: dualA-iff A-def, assumption)
done

```

Reduce the sublattice property by using substructural properties; abandoned
see *Tarski-4.ML*.

```

lemma (in CLF) [simp]:
  f: pset cl -> pset cl & monotone f (pset cl) (order cl)
apply (insert f-cl)
apply (simp add: CLF-set-def)
done

```

```

declare (in CLF) f-cl [simp]

```

```

lemma (in CLF) f-in-funcset: f ∈ A -> A
by (simp add: A-def)

```

```

lemma (in CLF) monotone-f: monotone f A r
by (simp add: A-def r-def)

```

```

lemma (in CLF) CLF-dual: (dual cl, f) ∈ CLF-set
apply (simp add: CLF-set-def CL-dualCL monotone-dual)
apply (simp add: dualA-iff)
done

```

```

lemma (in CLF) dual:
  CLF (dual cl) f
apply (rule CLF.intro)

```

apply (*rule CLF-dual*)
done

32.5 fixed points

lemma *fix-subset*: $\text{fix } f \ A \subseteq A$
by (*simp add: fix-def, fast*)

lemma *fix-imp-eq*: $x \in \text{fix } f \ A \implies f \ x = x$
by (*simp add: fix-def*)

lemma *fixf-subset*:
 $[| \ A \subseteq B; \ x \in \text{fix } (\%y: A. f \ y) \ A \ |] \implies x \in \text{fix } f \ B$
by (*simp add: fix-def, auto*)

32.6 lemmas for Tarski, lub

lemma (*in CLF*) *lubH-le-flubH*:
 $H = \{x. (x, f \ x) \in r \ \& \ x \in A\} \implies (\text{lub } H \ cl, f \ (\text{lub } H \ cl)) \in r$
apply (*rule lub-least, fast*)
apply (*rule f-in-funcset [THEN funcset-mem]*)
apply (*rule lub-in-lattice, fast*)
 $\text{--- } \forall x:H. (x, f \ (\text{lub } H \ r)) \in r$
apply (*rule ballI*)
apply (*rule transE*)
 $\text{--- instantiates } (x, ???z) \in \text{order } cl \text{ to } (x, f \ x),$
 $\text{--- because of the def of } H$
apply *fast*
 $\text{--- so it remains to show } (f \ x, f \ (\text{lub } H \ cl)) \in r$
apply (*rule-tac f = f in monotoneE*)
apply (*rule monotone-f, fast*)
apply (*rule lub-in-lattice, fast*)
apply (*rule lub-upper, fast*)
apply *assumption*
done

lemma (*in CLF*) *flubH-le-lubH*:
 $[| \ H = \{x. (x, f \ x) \in r \ \& \ x \in A\} \ |] \implies (f \ (\text{lub } H \ cl), \text{lub } H \ cl) \in r$
apply (*rule lub-upper, fast*)
apply (*rule-tac t = H in ssubst, assumption*)
apply (*rule CollectI*)
apply (*rule conjI*)
apply (*rule-tac [2] f-in-funcset [THEN funcset-mem]*)
apply (*rule-tac [2] lub-in-lattice*)
prefer 2 apply fast
apply (*rule-tac f = f in monotoneE*)
apply (*rule monotone-f*)
apply (*blast intro: lub-in-lattice*)
apply (*blast intro: lub-in-lattice f-in-funcset [THEN funcset-mem]*)
apply (*simp add: lubH-le-flubH*)

done

lemma (in CLF) *lubH-is-fixp*:

$H = \{x. (x, f x) \in r \ \& \ x \in A\} \implies \text{lub } H \text{ cl} \in \text{fix } f A$

apply (simp add: fix-def)

apply (rule conjI)

apply (rule lub-in-lattice, fast)

apply (rule antisymE)

apply (simp add: flubH-le-lubH)

apply (simp add: lubH-le-flubH)

done

lemma (in CLF) *fix-in-H*:

$[| H = \{x. (x, f x) \in r \ \& \ x \in A\}; \ x \in P \ |] \implies x \in H$

by (simp add: P-def fix-imp-eq [of - f A] reflE CO-refl-on
fix-subset [of f A, THEN subsetD])

lemma (in CLF) *fixf-le-lubH*:

$H = \{x. (x, f x) \in r \ \& \ x \in A\} \implies \forall x \in \text{fix } f A. (x, \text{lub } H \text{ cl}) \in r$

apply (rule ballI)

apply (rule lub-upper, fast)

apply (rule fix-in-H)

apply (simp-all add: P-def)

done

lemma (in CLF) *lubH-least-fixf*:

$H = \{x. (x, f x) \in r \ \& \ x \in A\}$

$\implies \forall L. (\forall y \in \text{fix } f A. (y, L) \in r) \longrightarrow (\text{lub } H \text{ cl}, L) \in r$

apply (rule allI)

apply (rule impI)

apply (erule bspec)

apply (rule lubH-is-fixp, assumption)

done

32.7 Tarski fixpoint theorem 1, first part

lemma (in CLF) *T-thm-1-lub*: $\text{lub } P \text{ cl} = \text{lub } \{x. (x, f x) \in r \ \& \ x \in A\} \text{ cl}$

apply (rule sym)

apply (simp add: P-def)

apply (rule lubI)

apply (rule fix-subset)

apply (rule lub-in-lattice, fast)

apply (simp add: fixf-le-lubH)

apply (simp add: lubH-least-fixf)

done

lemma (in CLF) *glbH-is-fixp*: $H = \{x. (f x, x) \in r \ \& \ x \in A\} \implies \text{glb } H \text{ cl} \in P$

— Tarski for glb

apply (simp add: glb-dual-lub P-def A-def r-def)

```

apply (rule dualA-iff [THEN subst])
apply (rule CLF.lubH-is-fixp)
apply (rule dual)
apply (simp add: dualr-iff dualA-iff)
done

```

```

lemma (in CLF) T-thm-1-glb: glb P cl = glb {x. (f x, x) ∈ r & x ∈ A} cl
apply (simp add: glb-dual-lub P-def A-def r-def)
apply (rule dualA-iff [THEN subst])
apply (simp add: CLF.T-thm-1-lub [of - f, OF dual]
             dualPO CL-dualCL CLF-dual dualr-iff)
done

```

32.8 interval

```

lemma (in CLF) rel-imp-elem: (x, y) ∈ r ==> x ∈ A
apply (insert CO-refl-on)
apply (simp add: refl-on-def, blast)
done

```

```

lemma (in CLF) interval-subset: [| a ∈ A; b ∈ A |] ==> interval r a b ⊆ A
apply (simp add: interval-def)
apply (blast intro: rel-imp-elem)
done

```

```

lemma (in CLF) intervalI:
  [| (a, x) ∈ r; (x, b) ∈ r |] ==> x ∈ interval r a b
by (simp add: interval-def)

```

```

lemma (in CLF) interval-lemma1:
  [| S ⊆ interval r a b; x ∈ S |] ==> (a, x) ∈ r
by (unfold interval-def, fast)

```

```

lemma (in CLF) interval-lemma2:
  [| S ⊆ interval r a b; x ∈ S |] ==> (x, b) ∈ r
by (unfold interval-def, fast)

```

```

lemma (in CLF) a-less-lub:
  [| S ⊆ A; S ≠ {} |]
  ∀ x ∈ S. (a, x) ∈ r; ∀ y ∈ S. (y, L) ∈ r |] ==> (a, L) ∈ r
by (blast intro: transE)

```

```

lemma (in CLF) glb-less-b:
  [| S ⊆ A; S ≠ {} |]
  ∀ x ∈ S. (x, b) ∈ r; ∀ y ∈ S. (G, y) ∈ r |] ==> (G, b) ∈ r
by (blast intro: transE)

```

```

lemma (in CLF) S-intv-cl:
  [| a ∈ A; b ∈ A; S ⊆ interval r a b |] ==> S ⊆ A

```

```

by (simp add: subset-trans [OF - interval-subset])

lemma (in CLF) L-in-interval:
  [| a ∈ A; b ∈ A; S ⊆ interval r a b;
    S ≠ {} ; isLub S cl L; interval r a b ≠ {} |] ==> L ∈ interval r a b
  apply (rule intervalI)
  apply (rule a-less-lub)
  prefer 2 apply assumption
  apply (simp add: S-intv-cl)
  apply (rule ballI)
  apply (simp add: interval-lemma1)
  apply (simp add: isLub-upper)
  — (L, b) ∈ r
  apply (simp add: isLub-least interval-lemma2)
done

lemma (in CLF) G-in-interval:
  [| a ∈ A; b ∈ A; interval r a b ≠ {} ; S ⊆ interval r a b; isGlb S cl G;
    S ≠ {} |] ==> G ∈ interval r a b
  apply (simp add: interval-dual)
  apply (simp add: CLF.L-in-interval [of - f, OF dual]
    dualA-iff A-def isGlb-dual-isLub)
done

lemma (in CLF) intervalPO:
  [| a ∈ A; b ∈ A; interval r a b ≠ {} |]
  ==> (| pset = interval r a b, order = induced (interval r a b) r |)
    ∈ PartialOrder
  apply (rule po-subset-po)
  apply (simp add: interval-subset)
done

lemma (in CLF) intv-CL-lub:
  [| a ∈ A; b ∈ A; interval r a b ≠ {} |]
  ==> ∀ S. S ⊆ interval r a b -->
    (∃ L. isLub S (| pset = interval r a b,
      order = induced (interval r a b) r |) L)
  apply (intro strip)
  apply (frule S-intv-cl [THEN CL-imp-ex-isLub])
  prefer 2 apply assumption
  apply assumption
  apply (erule exE)
  — define the lub for the interval as
  apply (rule-tac x = if S = {} then a else L in exI)
  apply (simp (no-asm-simp) add: isLub-def split del: split-if)
  apply (intro impI conjI)
  — (if S = {} then a else L) ∈ interval r a b
  apply (simp add: CL-imp-PO L-in-interval)
  apply (simp add: left-in-interval)

```



```

— lub prop 1
apply (case-tac  $S = \{\}$ )
—  $S = \{\}$ ,  $y \in S = \text{False} \Rightarrow \text{everything}$ 
apply fast
—  $S \neq \{\}$ 
apply simp
—  $\forall y:S. (y, L) \in \text{induced } (\text{interval } r \ a \ b) \ r$ 
apply (rule ballI)
apply (simp add: induced-def L-in-interval)
apply (rule conjI)
apply (rule subsetD)
apply (simp add: S-intv-cl, assumption)
apply (simp add: isLub-upper)
—  $\forall z:\text{interval } r \ a \ b. (\forall y:S. (y, z) \in \text{induced } (\text{interval } r \ a \ b) \ r \longrightarrow (\text{if } S = \{\} \text{ then } a \text{ else } L, z) \in \text{induced } (\text{interval } r \ a \ b) \ r$ 
apply (rule ballI)
apply (rule impI)
apply (case-tac  $S = \{\}$ )
—  $S = \{\}$ 
apply simp
apply (simp add: induced-def interval-def)
apply (rule conjI)
apply (rule reflE, assumption)
apply (rule interval-not-empty)
apply (simp add: interval-def)
—  $S \neq \{\}$ 
apply simp
apply (simp add: induced-def L-in-interval)
apply (rule isLub-least, assumption)
apply (rule subsetD)
prefer 2 apply assumption
apply (simp add: S-intv-cl, fast)
done

```

lemmas (in CLF) $\text{intv-CL-glb} = \text{intv-CL-lub} \ [\text{THEN } \text{Rdual}]$

lemma (in CLF) *interval-is-sublattice*:
 $[[a \in A; b \in A; \text{interval } r \ a \ b \neq \{\}]]$
 $\Rightarrow \text{interval } r \ a \ b \leqslant \text{cl}$
apply (rule sublatticeI)
apply (simp add: interval-subset)
apply (rule CompleteLatticeI)
apply (simp add: intervalPO)
apply (simp add: intv-CL-lub)
apply (simp add: intv-CL-glb)
done

lemmas (in CLF) *intv-is-compl-latt* =
interval-is-sublattice [*THEN* *sublattice-imp-CL*]

32.9 Top and Bottom

lemma (in CLF) *Top-dual-Bot*: $Top\ cl = Bot\ (dual\ cl)$
by (simp add: Top-def Bot-def least-def greatest-def dualA-iff dualr-iff)

lemma (in CLF) *Bot-dual-Top*: $Bot\ cl = Top\ (dual\ cl)$
by (simp add: Top-def Bot-def least-def greatest-def dualA-iff dualr-iff)

lemma (in CLF) *Bot-in-lattice*: $Bot\ cl \in A$
apply (simp add: Bot-def least-def)
apply (rule-tac a=glb A cl in someI2)
apply (simp-all add: glb-in-lattice glb-lower
r-def [symmetric] A-def [symmetric])
done

lemma (in CLF) *Top-in-lattice*: $Top\ cl \in A$
apply (simp add: Top-dual-Bot A-def)
apply (rule dualA-iff [THEN subst])
apply (rule CLF.Bot-in-lattice [OF dual])
done

lemma (in CLF) *Top-prop*: $x \in A ==> (x, Top\ cl) \in r$
apply (simp add: Top-def greatest-def)
apply (rule-tac a=lub A cl in someI2)
apply (rule someI2)
apply (simp-all add: lub-in-lattice lub-upper
r-def [symmetric] A-def [symmetric])
done

lemma (in CLF) *Bot-prop*: $x \in A ==> (Bot\ cl, x) \in r$
apply (simp add: Bot-dual-Top r-def)
apply (rule dualr-iff [THEN subst])
apply (rule CLF.Top-prop [OF dual])
apply (simp add: dualA-iff A-def)
done

lemma (in CLF) *Top-intv-not-empty*: $x \in A ==> interval\ r\ x\ (Top\ cl) \neq \{\}$
apply (rule notI)
apply (drule-tac a = Top cl in equals0D)
apply (simp add: interval-def)
apply (simp add: refl-on-def Top-in-lattice Top-prop)
done

lemma (in CLF) *Bot-intv-not-empty*: $x \in A ==> interval\ r\ (Bot\ cl)\ x \neq \{\}$
apply (simp add: Bot-dual-Top)
apply (subst interval-dual)
prefer 2 **apply** assumption
apply (simp add: A-def)
apply (rule dualA-iff [THEN subst])
apply (rule CLF.Top-in-lattice [OF dual])

```

apply (rule CLF.Top-intv-not-empty [OF dual])
apply (simp add: dualA-iff A-def)
done

```

32.10 fixed points form a partial order

```

lemma (in CLF) fix-po: ( $| \text{ pset} = P, \text{ order} = \text{induced } P \text{ r}|$ )  $\in \text{PartialOrder}$ 
by (simp add: P-def fix-subset po-subset-po)

```

```

lemma (in Tarski) Y-subset-A:  $Y \subseteq A$ 
apply (rule subset-trans [OF - fix-subset])
apply (rule Y-ss [simplified P-def])
done

```

```

lemma (in Tarski) lubY-in-A:  $\text{lub } Y \text{ cl} \in A$ 
by (rule Y-subset-A [THEN lub-in-lattice])

```

```

lemma (in Tarski) lubY-le-flubY:  $(\text{lub } Y \text{ cl}, f (\text{lub } Y \text{ cl})) \in r$ 
apply (rule lub-least)
apply (rule Y-subset-A)
apply (rule f-in-funcset [THEN funcset-mem])
apply (rule lubY-in-A)
—  $Y \subseteq P \implies f x = x$ 
apply (rule ballI)
apply (rule-tac  $t = x$  in fix-imp-eq [THEN subst])
apply (erule Y-ss [simplified P-def, THEN subsetD])
— reduce  $(f x, f (\text{lub } Y \text{ cl})) \in r$  to  $(x, \text{lub } Y \text{ cl}) \in r$  by monotonicity
apply (rule-tac  $f = f$  in monotoneE)
apply (rule monotone-f)
apply (simp add: Y-subset-A [THEN subsetD])
apply (rule lubY-in-A)
apply (simp add: lub-upper Y-subset-A)
done

```

```

lemma (in Tarski) intY1-subset:  $\text{intY1} \subseteq A$ 
apply (unfold intY1-def)
apply (rule interval-subset)
apply (rule lubY-in-A)
apply (rule Top-in-lattice)
done

```

```

lemmas (in Tarski) intY1-elem = intY1-subset [THEN subsetD]

```

```

lemma (in Tarski) intY1-f-closed:  $x \in \text{intY1} \implies f x \in \text{intY1}$ 
apply (simp add: intY1-def interval-def)
apply (rule conjI)
apply (rule transE)
apply (rule lubY-le-flubY)
—  $(f (\text{lub } Y \text{ cl}), f x) \in r$ 

```

```

apply (rule-tac f=f in monotoneE)
apply (rule monotone-f)
apply (rule lubY-in-A)
apply (simp add: intY1-def interval-def intY1-elem)
apply (simp add: intY1-def interval-def)
— (f x, Top cl) ∈ r
apply (rule Top-prop)
apply (rule f-in-funcset [THEN funcset-mem])
apply (simp add: intY1-def interval-def intY1-elem)
done

```

```

lemma (in Tarski) intY1-mono:
  monotone (λx: intY1. f x) intY1 (induced intY1 r)
apply (auto simp add: monotone-def induced-def intY1-f-closed)
apply (blast intro: intY1-elem monotone-f [THEN monotoneE])
done

```

```

lemma (in Tarski) intY1-is-cl:
  (| pset = intY1, order = induced intY1 r |) ∈ CompleteLattice
apply (unfold intY1-def)
apply (rule interv-is-compl-latt)
apply (rule lubY-in-A)
apply (rule Top-in-lattice)
apply (rule Top-intv-not-empty)
apply (rule lubY-in-A)
done

```

```

lemma (in Tarski) v-in-P: v ∈ P
apply (unfold P-def)
apply (rule-tac A = intY1 in fixf-subset)
apply (rule intY1-subset)
unfolding v-def
apply (rule CLF.glbH-is-fixp [OF CLF.intro, unfolded CLF-set-def, of (|pset =
intY1, order = induced intY1 r|), simplified])
apply auto
apply (rule intY1-is-cl)
apply (erule intY1-f-closed)
apply (rule intY1-mono)
done

```

```

lemma (in Tarski) z-in-interval:
  [| z ∈ P; ∀ y ∈ Y. (y, z) ∈ induced P r |] ==> z ∈ intY1
apply (unfold intY1-def P-def)
apply (rule intervalI)
prefer 2
  apply (erule fix-subset [THEN subsetD, THEN Top-prop])
apply (rule lub-least)
apply (rule Y-subset-A)
apply (fast elim!: fix-subset [THEN subsetD])

```

```

apply (simp add: induced-def)
done

lemma (in Tarski) f'z-in-int-rel: [| z ∈ P; ∀ y ∈ Y. (y, z) ∈ induced P r |]
  ==> ((%x: intY1. f x) z, z) ∈ induced intY1 r
apply (simp add: induced-def intY1-f-closed z-in-interval P-def)
apply (simp add: fix-imp-eq [of - f A] fix-subset [of f A, THEN subsetD]
  reflE)
done

lemma (in Tarski) tarski-full-lemma:
  ∃ L. isLub Y (| pset = P, order = induced P r |) L
apply (rule-tac x = v in exI)
apply (simp add: isLub-def)
  — v ∈ P
apply (simp add: v-in-P)
apply (rule conjI)
  — v is lub
  — 1. ∀ y:Y. (y, v) ∈ induced P r
apply (rule ballI)
apply (simp add: induced-def subsetD v-in-P)
apply (rule conjI)
apply (erule Y-ss [THEN subsetD])
apply (rule-tac b = lub Y cl in transE)
apply (rule lub-upper)
apply (rule Y-subset-A, assumption)
apply (rule-tac b = Top cl in interval-imp-mem)
apply (simp add: v-def)
apply (fold intY1-def)
apply (rule CL.glb-in-lattice [OF CL.intro [OF intY1-is-cl], simplified])
apply auto
apply (rule indI)
  prefer 3 apply assumption
  prefer 2 apply (simp add: v-in-P)
apply (unfold v-def)
apply (rule indE)
apply (rule-tac [2] intY1-subset)
apply (rule CL.glb-lower [OF CL.intro [OF intY1-is-cl], simplified])
  apply (simp add: CL-imp-PO intY1-is-cl)
apply force
apply (simp add: induced-def intY1-f-closed z-in-interval)
apply (simp add: P-def fix-imp-eq [of - f A] reflE
  fix-subset [of f A, THEN subsetD])
done

lemma CompleteLatticeI-simp:
  [| (| pset = A, order = r |) ∈ PartialOrder;
    ∀ S. S ⊆ A --> (∃ L. isLub S (| pset = A, order = r |) L) |]
  ==> (| pset = A, order = r |) ∈ CompleteLattice

```

```

by (simp add: CompleteLatticeI Rdual)

theorem (in CLF) Tarski-full:
  ( $| \text{pset} = P, \text{order} = \text{induced } P \text{ r} |$ )  $\in$  CompleteLattice
apply (rule CompleteLatticeI-simp)
apply (rule fixf-po, clarify)
apply (simp add: P-def A-def r-def)
apply (rule Tarski.tarski-full-lemma [OF Tarski.intro [OF - Tarski-axioms.intro]])
proof - show CLF cl f .. qed

end

```

33 Classical Predicate Calculus Problems

theory *Classical* **imports** *Main* **begin**

33.1 Traditional Classical Reasoner

The machine "griffon" mentioned below is a 2.5GHz Power Mac G5.

Taken from *FOL/Classical.thy*. When porting examples from first-order logic, beware of the precedence of $=$ versus \leftrightarrow .

```

lemma ( $P \dashv\dashv Q \mid R \dashv\dashv (P \dashv\dashv Q) \mid (P \dashv\dashv R)$ )
by blast

```

If and only if

```

lemma ( $P=Q$ ) = ( $Q = (P::\text{bool})$ )
by blast

```

```

lemma  $\sim (P = (\sim P))$ 
by blast

```

Sample problems from F. J. Pelletier, Seventy-Five Problems for Testing Automatic Theorem Provers, J. Automated Reasoning 2 (1986), 191-216. Errata, JAR 4 (1988), 236-236.

The hardest problems – judging by experience with several theorem provers, including matrix ones – are 34 and 43.

33.1.1 Pelletier's examples

1

```

lemma ( $P \dashv\dashv Q$ ) = ( $\sim Q \dashv\dashv \sim P$ )
by blast

```

2

lemma $(\sim \sim P) = P$
by *blast*

3

lemma $\sim(P \multimap Q) \multimap (Q \multimap P)$
by *blast*

4

lemma $(\sim P \multimap Q) = (\sim Q \multimap P)$
by *blast*

5

lemma $((P|Q) \multimap (P|R)) \multimap (P|(Q \multimap R))$
by *blast*

6

lemma $P | \sim P$
by *blast*

7

lemma $P | \sim \sim \sim P$
by *blast*

8. Peirce's law

lemma $((P \multimap Q) \multimap P) \multimap P$
by *blast*

9

lemma $((P|Q) \& (\sim P|Q) \& (P|\sim Q)) \multimap \sim (\sim P | \sim Q)$
by *blast*

10

lemma $(Q \multimap R) \& (R \multimap P \& Q) \& (P \multimap Q|R) \multimap (P=Q)$
by *blast*

11. Proved in each direction (incorrectly, says Pelletier!!)

lemma $P=(P::bool)$
by *blast*

12. "Dijkstra's law"

lemma $((P = Q) = R) = (P = (Q = R))$
by *blast*

13. Distributive law

lemma $(P | (Q \& R)) = ((P | Q) \& (P | R))$
by *blast*

14

lemma $(P = Q) = ((Q \mid \sim P) \& (\sim Q \mid P))$
by *blast*

15

lemma $(P \dashrightarrow Q) = (\sim P \mid Q)$
by *blast*

16

lemma $(P \dashrightarrow Q) \mid (Q \dashrightarrow P)$
by *blast*

17

lemma $((P \& (Q \dashrightarrow R)) \dashrightarrow S) = ((\sim P \mid Q \mid S) \& (\sim P \mid \sim R \mid S))$
by *blast*

33.1.2 Classical Logic: examples with quantifiers

lemma $(\forall x. P(x) \& Q(x)) = ((\forall x. P(x)) \& (\forall x. Q(x)))$
by *blast*

lemma $(\exists x. P \dashrightarrow Q(x)) = (P \dashrightarrow (\exists x. Q(x)))$
by *blast*

lemma $(\exists x. P(x) \dashrightarrow Q) = ((\forall x. P(x)) \dashrightarrow Q)$
by *blast*

lemma $((\forall x. P(x)) \mid Q) = (\forall x. P(x) \mid Q)$
by *blast*

From Wishnu Prasetya

lemma $(\forall s. q(s) \dashrightarrow r(s)) \& \sim r(s) \& (\forall s. \sim r(s) \& \sim q(s) \dashrightarrow p(t) \mid q(t))$
 $\dashrightarrow p(t) \mid r(t)$
by *blast*

33.1.3 Problems requiring quantifier duplication

Theorem B of Peter Andrews, Theorem Proving via General Matings, JACM 28 (1981).

lemma $(\exists x. \forall y. P(x) = P(y)) \dashrightarrow ((\exists x. P(x)) = (\forall y. P(y)))$
by *blast*

Needs multiple instantiation of the quantifier.

lemma $(\forall x. P(x) \dashrightarrow P(f(x))) \& P(d) \dashrightarrow P(f(f(f(d))))$
by *blast*

Needs double instantiation of the quantifier

lemma $\exists x. P(x) \dashv\dashv P(a) \ \& \ P(b)$
by *blast*

lemma $\exists z. P(z) \dashv\dashv (\forall x. P(x))$
by *blast*

lemma $\exists x. (\exists y. P(y)) \dashv\dashv P(x)$
by *blast*

33.1.4 Hard examples with quantifiers

Problem 18

lemma $\exists y. \forall x. P(y) \dashv\dashv P(x)$
by *blast*

Problem 19

lemma $\exists x. \forall y z. (P(y) \dashv\dashv Q(z)) \dashv\dashv (P(x) \dashv\dashv Q(x))$
by *blast*

Problem 20

lemma $(\forall x y. \exists z. \forall w. (P(x) \& Q(y) \dashv\dashv R(z) \& S(w)))$
 $\dashv\dashv (\exists x y. P(x) \ \& \ Q(y)) \dashv\dashv (\exists z. R(z))$
by *blast*

Problem 21

lemma $(\exists x. P \dashv\dashv Q(x)) \ \& \ (\exists x. Q(x) \dashv\dashv P) \dashv\dashv (\exists x. P = Q(x))$
by *blast*

Problem 22

lemma $(\forall x. P = Q(x)) \dashv\dashv (P = (\forall x. Q(x)))$
by *blast*

Problem 23

lemma $(\forall x. P \mid Q(x)) = (P \mid (\forall x. Q(x)))$
by *blast*

Problem 24

lemma $\sim(\exists x. S(x) \& Q(x)) \ \& \ (\forall x. P(x) \dashv\dashv Q(x) \mid R(x)) \ \&$
 $(\sim(\exists x. P(x)) \dashv\dashv (\exists x. Q(x))) \ \& \ (\forall x. Q(x) \mid R(x) \dashv\dashv S(x))$
 $\dashv\dashv (\exists x. P(x) \& R(x))$
by *blast*

Problem 25

lemma $(\exists x. P(x)) \ \&$
 $(\forall x. L(x) \dashv\dashv \sim (M(x) \ \& \ R(x))) \ \&$
 $(\forall x. P(x) \dashv\dashv (M(x) \ \& \ L(x))) \ \&$
 $((\forall x. P(x) \dashv\dashv Q(x)) \mid (\exists x. P(x) \& R(x)))$

$---> (\exists x. Q(x) \& P(x))$
by *blast*

Problem 26

lemma $((\exists x. p(x)) = (\exists x. q(x))) \&$
 $(\forall x. \forall y. p(x) \& q(y) ---> (r(x) = s(y)))$
 $---> ((\forall x. p(x) ---> r(x)) = (\forall x. q(x) ---> s(x)))$
by *blast*

Problem 27

lemma $(\exists x. P(x) \& \sim Q(x)) \&$
 $(\forall x. P(x) ---> R(x)) \&$
 $(\forall x. M(x) \& L(x) ---> P(x)) \&$
 $((\exists x. R(x) \& \sim Q(x)) ---> (\forall x. L(x) ---> \sim R(x)))$
 $---> (\forall x. M(x) ---> \sim L(x))$
by *blast*

Problem 28. AMENDED

lemma $(\forall x. P(x) ---> (\forall x. Q(x))) \&$
 $((\forall x. Q(x) | R(x)) ---> (\exists x. Q(x) \& S(x))) \&$
 $((\exists x. S(x)) ---> (\forall x. L(x) ---> M(x)))$
 $---> (\forall x. P(x) \& L(x) ---> M(x))$
by *blast*

Problem 29. Essentially the same as Principia Mathematica *11.71

lemma $(\exists x. F(x)) \& (\exists y. G(y))$
 $---> (((\forall x. F(x) ---> H(x)) \& (\forall y. G(y) ---> J(y))) =$
 $(\forall x y. F(x) \& G(y) ---> H(x) \& J(y)))$
by *blast*

Problem 30

lemma $(\forall x. P(x) | Q(x) ---> \sim R(x)) \&$
 $(\forall x. (Q(x) ---> \sim S(x)) ---> P(x) \& R(x))$
 $---> (\forall x. S(x))$
by *blast*

Problem 31

lemma $\sim(\exists x. P(x) \& (Q(x) | R(x))) \&$
 $(\exists x. L(x) \& P(x)) \&$
 $(\forall x. \sim R(x) ---> M(x))$
 $---> (\exists x. L(x) \& M(x))$
by *blast*

Problem 32

lemma $(\forall x. P(x) \& (Q(x) | R(x)) ---> S(x)) \&$
 $(\forall x. S(x) \& R(x) ---> L(x)) \&$
 $(\forall x. M(x) ---> R(x))$
 $---> (\forall x. P(x) \& M(x) ---> L(x))$

by *blast*

Problem 33

lemma $(\forall x. P(a) \ \& \ (P(x) \dashv\dashv P(b)) \dashv\dashv P(c)) =$
 $(\forall x. (\sim P(a) \mid P(x) \mid P(c)) \ \& \ (\sim P(a) \mid \sim P(b) \mid P(c)))$
by *blast*

Problem 34 AMENDED (TWICE!!)

Andrews's challenge

lemma $((\exists x. \forall y. p(x) = p(y)) =$
 $((\exists x. q(x)) = (\forall y. p(y)))) =$
 $((\exists x. \forall y. q(x) = q(y)) =$
 $((\exists x. p(x)) = (\forall y. q(y))))$
by *blast*

Problem 35

lemma $\exists x y. P \ x \ y \dashv\dashv (\forall u v. P \ u \ v)$
by *blast*

Problem 36

lemma $(\forall x. \exists y. J \ x \ y) \ \&$
 $(\forall x. \exists y. G \ x \ y) \ \&$
 $(\forall x y. J \ x \ y \mid G \ x \ y \dashv\dashv$
 $(\forall z. J \ y \ z \mid G \ y \ z \dashv\dashv H \ x \ z))$
 $\dashv\dashv (\forall x. \exists y. H \ x \ y)$
by *blast*

Problem 37

lemma $(\forall z. \exists w. \forall x. \exists y.$
 $(P \ x \ z \dashv\dashv P \ y \ w) \ \& \ P \ y \ z \ \& \ (P \ y \ w \dashv\dashv (\exists u. Q \ u \ w))) \ \&$
 $(\forall x z. \sim(P \ x \ z) \dashv\dashv (\exists y. Q \ y \ z)) \ \&$
 $((\exists x y. Q \ x \ y) \dashv\dashv (\forall x. R \ x \ x))$
 $\dashv\dashv (\forall x. \exists y. R \ x \ y)$
by *blast*

Problem 38

lemma $(\forall x. p(a) \ \& \ (p(x) \dashv\dashv (\exists y. p(y) \ \& \ r \ x \ y)) \dashv\dashv$
 $(\exists z. \exists w. p(z) \ \& \ r \ x \ w \ \& \ r \ w \ z)) =$
 $(\forall x. (\sim p(a) \mid p(x) \mid (\exists z. \exists w. p(z) \ \& \ r \ x \ w \ \& \ r \ w \ z)) \ \&$
 $(\sim p(a) \mid \sim(\exists y. p(y) \ \& \ r \ x \ y) \mid$
 $(\exists z. \exists w. p(z) \ \& \ r \ x \ w \ \& \ r \ w \ z)))$
by *blast*

Problem 39

lemma $\sim (\exists x. \forall y. F \ y \ x = (\sim F \ y \ y))$
by *blast*

Problem 40. AMENDED

lemma $(\exists y. \forall x. F x y = F x x)$
 $--> \sim (\forall x. \exists y. \forall z. F z y = (\sim F z x))$
by *blast*

Problem 41

lemma $(\forall z. \exists y. \forall x. f x y = (f x z \ \& \ \sim f x x))$
 $--> \sim (\exists z. \forall x. f x z)$
by *blast*

Problem 42

lemma $\sim (\exists y. \forall x. p x y = (\sim (\exists z. p x z \ \& \ p z x)))$
by *blast*

Problem 43!!

lemma $(\forall x::'a. \forall y::'a. q x y = (\forall z. p z x = (p z y::bool)))$
 $--> (\forall x. (\forall y. q x y = (q y x::bool)))$
by *blast*

Problem 44

lemma $(\forall x. f(x) -->$
 $(\exists y. g(y) \ \& \ h x y \ \& \ (\exists y. g(y) \ \& \ \sim h x y))) \ \&$
 $(\exists x. j(x) \ \& \ (\forall y. g(y) --> h x y))$
 $--> (\exists x. j(x) \ \& \ \sim f(x))$
by *blast*

Problem 45

lemma $(\forall x. f(x) \ \& \ (\forall y. g(y) \ \& \ h x y --> j x y)$
 $--> (\forall y. g(y) \ \& \ h x y --> k(y))) \ \&$
 $\sim (\exists y. l(y) \ \& \ k(y)) \ \&$
 $(\exists x. f(x) \ \& \ (\forall y. h x y --> l(y))$
 $\ \& \ (\forall y. g(y) \ \& \ h x y --> j x y))$
 $--> (\exists x. f(x) \ \& \ \sim (\exists y. g(y) \ \& \ h x y))$
by *blast*

33.1.5 Problems (mainly) involving equality or functions

Problem 48

lemma $(a=b \mid c=d) \ \& \ (a=c \mid b=d) --> a=d \mid b=c$
by *blast*

Problem 49 NOT PROVED AUTOMATICALLY. Hard because it involves substitution for Vars the type constraint ensures that x,y,z have the same type as a,b,u.

lemma $(\exists x y::'a. \forall z. z=x \mid z=y) \ \& \ P(a) \ \& \ P(b) \ \& \ (\sim a=b)$
 $--> (\forall u::'a. P(u))$

by *metis*

Problem 50. (What has this to do with equality?)

lemma $(\forall x. P\ a\ x \mid (\forall y. P\ x\ y)) \dashv\vdash (\exists x. \forall y. P\ x\ y)$
by *blast*

Problem 51

lemma $(\exists z\ w. \forall x\ y. P\ x\ y = (x=z \ \&\ y=w)) \dashv\vdash$
 $(\exists z. \forall x. \exists w. (\forall y. P\ x\ y = (y=w)) = (x=z))$
by *blast*

Problem 52. Almost the same as 51.

lemma $(\exists z\ w. \forall x\ y. P\ x\ y = (x=z \ \&\ y=w)) \dashv\vdash$
 $(\exists w. \forall y. \exists z. (\forall x. P\ x\ y = (x=z)) = (y=w))$
by *blast*

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988).
fast DISCOVERS who killed Agatha.

schematic-lemma $\text{lives}(\text{agatha}) \ \& \ \text{lives}(\text{butler}) \ \& \ \text{lives}(\text{charles}) \ \&$
 $(\text{killed}\ \text{agatha}\ \text{agatha} \mid \text{killed}\ \text{butler}\ \text{agatha} \mid \text{killed}\ \text{charles}\ \text{agatha}) \ \&$
 $(\forall x\ y. \text{killed}\ x\ y \dashv\vdash \text{hates}\ x\ y \ \& \ \sim \text{richer}\ x\ y) \ \&$
 $(\forall x. \text{hates}\ \text{agatha}\ x \dashv\vdash \sim \text{hates}\ \text{charles}\ x) \ \&$
 $(\text{hates}\ \text{agatha}\ \text{agatha} \ \& \ \text{hates}\ \text{agatha}\ \text{charles}) \ \&$
 $(\forall x. \text{lives}(x) \ \& \ \sim \text{richer}\ x\ \text{agatha} \dashv\vdash \text{hates}\ \text{butler}\ x) \ \&$
 $(\forall x. \text{hates}\ \text{agatha}\ x \dashv\vdash \text{hates}\ \text{butler}\ x) \ \&$
 $(\forall x. \sim \text{hates}\ x\ \text{agatha} \mid \sim \text{hates}\ x\ \text{butler} \mid \sim \text{hates}\ x\ \text{charles}) \dashv\vdash$
 $\text{killed}\ ?\text{who}\ \text{agatha}$
by *fast*

Problem 56

lemma $(\forall x. (\exists y. P(y) \ \& \ x=f(y)) \dashv\vdash P(x)) = (\forall x. P(x) \dashv\vdash P(f(x)))$
by *blast*

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c) \ \& \ P\ (f\ b\ c)\ (f\ a\ c) \ \&$
 $(\forall x\ y\ z. P\ x\ y \ \& \ P\ y\ z \dashv\vdash P\ x\ z) \dashv\vdash P\ (f\ a\ b)\ (f\ a\ c)$
by *blast*

Problem 58 NOT PROVED AUTOMATICALLY

lemma $(\forall x\ y. f(x)=g(y)) \dashv\vdash (\forall x\ y. f(f(x))=f(g(y)))$
by (*fast intro: arg-cong [of concl: f]*)

Problem 59

lemma $(\forall x. P(x) = (\sim P(f(x)))) \dashv\vdash (\exists x. P(x) \ \& \ \sim P(f(x)))$
by *blast*

Problem 60

lemma $\forall x. P x (f x) = (\exists y. (\forall z. P z y \longrightarrow P z (f x)) \ \& \ P x y)$
by *blast*

Problem 62 as corrected in JAR 18 (1997), page 135

lemma $(\forall x. p a \ \& \ (p x \longrightarrow p(f x)) \longrightarrow p(f(f x))) =$
 $(\forall x. (\sim p a \mid p x \mid p(f(f x))) \ \& \$
 $(\sim p a \mid \sim p(f x) \mid p(f(f x))))$
by *blast*

From Davis, Obvious Logical Inferences, IJCAI-81, 530-531 fast indeed copes!

lemma $(\forall x. F(x) \ \& \ \sim G(x) \longrightarrow (\exists y. H(x,y) \ \& \ J(y))) \ \& \$
 $(\exists x. K(x) \ \& \ F(x) \ \& \ (\forall y. H(x,y) \longrightarrow K(y))) \ \& \$
 $(\forall x. K(x) \longrightarrow \sim G(x)) \longrightarrow (\exists x. K(x) \ \& \ J(x))$
by *fast*

From Rudnicki, Obvious Inferences, JAR 3 (1987), 383-393. It does seem obvious!

lemma $(\forall x. F(x) \ \& \ \sim G(x) \longrightarrow (\exists y. H(x,y) \ \& \ J(y))) \ \& \$
 $(\exists x. K(x) \ \& \ F(x) \ \& \ (\forall y. H(x,y) \longrightarrow K(y))) \ \& \$
 $(\forall x. K(x) \longrightarrow \sim G(x)) \longrightarrow (\exists x. K(x) \longrightarrow \sim G(x))$
by *fast*

Attributed to Lewis Carroll by S. G. Pulman. The first or last assumption can be deleted.

lemma $(\forall x. honest(x) \ \& \ industrious(x) \longrightarrow healthy(x)) \ \& \$
 $\sim (\exists x. grocer(x) \ \& \ healthy(x)) \ \& \$
 $(\forall x. industrious(x) \ \& \ grocer(x) \longrightarrow honest(x)) \ \& \$
 $(\forall x. cyclist(x) \longrightarrow industrious(x)) \ \& \$
 $(\forall x. \sim healthy(x) \ \& \ cyclist(x) \longrightarrow \sim honest(x))$
 $\longrightarrow (\forall x. grocer(x) \longrightarrow \sim cyclist(x))$
by *blast*

lemma $(\forall x y. R(x,y) \mid R(y,x)) \ \& \$
 $(\forall x y. S(x,y) \ \& \ S(y,x) \longrightarrow x=y) \ \& \$
 $(\forall x y. R(x,y) \longrightarrow S(x,y)) \longrightarrow (\forall x y. S(x,y) \longrightarrow R(x,y))$
by *blast*

33.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y \ \& \ y < z \longrightarrow \sim (z < (x::nat))$
by (*meson order-less-irrefl order-less-trans*)

The "small example" from Bezem, Hendriks and de Nivelle, Automatic Proof Construction in Type Theory Using Resolution, JAR 29: 3-4 (2002), pages 253-275

lemma $(\forall x\ y\ z. R(x,y) \ \& \ R(y,z) \multimap R(x,z)) \ \& \ (\forall x. \exists y. R(x,y)) \multimap \sim (\forall x. P\ x = (\forall y. R(x,y) \multimap \sim P\ y))$
by $(tactic\langle\langle Meson.safe-best-meson-tac \ @\{context\} \ 1 \rangle\rangle)$
— In contrast, *meson* is SLOW: 7.6s on griffon

33.2.1 Pelletier's examples

1

lemma $(P \multimap Q) = (\sim Q \multimap \sim P)$
by *blast*

2

lemma $(\sim \sim P) = P$
by *blast*

3

lemma $\sim(P \multimap Q) \multimap (Q \multimap P)$
by *blast*

4

lemma $(\sim P \multimap Q) = (\sim Q \multimap P)$
by *blast*

5

lemma $((P|Q) \multimap (P|R)) \multimap (P|(Q \multimap R))$
by *blast*

6

lemma $P \mid \sim P$
by *blast*

7

lemma $P \mid \sim \sim \sim P$
by *blast*

8. Peirce's law

lemma $((P \multimap Q) \multimap P) \multimap P$
by *blast*

9

lemma $((P|Q) \ \& \ (\sim P|Q) \ \& \ (P \mid \sim Q)) \multimap \sim (\sim P \mid \sim Q)$
by *blast*

10

lemma $(Q \multimap R) \ \& \ (R \multimap P \ \& \ Q) \ \& \ (P \multimap Q|R) \multimap (P=Q)$

by *blast*

11. Proved in each direction (incorrectly, says Pelletier!!)

lemma $P = (P :: \text{bool})$

by *blast*

12. "Dijkstra's law"

lemma $((P = Q) = R) = (P = (Q = R))$

by *blast*

13. Distributive law

lemma $(P \mid (Q \ \& \ R)) = ((P \mid Q) \ \& \ (P \mid R))$

by *blast*

14

lemma $(P = Q) = ((Q \mid \sim P) \ \& \ (\sim Q \mid P))$

by *blast*

15

lemma $(P \dashrightarrow Q) = (\sim P \mid Q)$

by *blast*

16

lemma $(P \dashrightarrow Q) \mid (Q \dashrightarrow P)$

by *blast*

17

lemma $((P \ \& \ (Q \dashrightarrow R)) \dashrightarrow S) = ((\sim P \mid Q \mid S) \ \& \ (\sim P \mid \sim R \mid S))$

by *blast*

33.2.2 Classical Logic: examples with quantifiers

lemma $(\forall x. P \ x \ \& \ Q \ x) = ((\forall x. P \ x) \ \& \ (\forall x. Q \ x))$

by *blast*

lemma $(\exists x. P \ \dashrightarrow \ Q \ x) = (P \ \dashrightarrow \ (\exists x. Q \ x))$

by *blast*

lemma $(\exists x. P \ x \ \dashrightarrow \ Q) = ((\forall x. P \ x) \ \dashrightarrow \ Q)$

by *blast*

lemma $((\forall x. P \ x) \mid Q) = (\forall x. P \ x \mid Q)$

by *blast*

lemma $(\forall x. P \ x \ \dashrightarrow \ P(f \ x)) \ \& \ P \ d \ \dashrightarrow \ P(f(f \ d))$

by *blast*

Needs double instantiation of EXISTS

lemma $\exists x. P x \longrightarrow P a \ \& \ P b$
by *blast*

lemma $\exists z. P z \longrightarrow (\forall x. P x)$
by *blast*

From a paper by Claire Quigley

lemma $\exists y. ((P c \ \& \ Q y) \mid (\exists z. \sim Q z)) \mid (\exists x. \sim P x \ \& \ Q d)$
by *fast*

33.2.3 Hard examples with quantifiers

Problem 18

lemma $\exists y. \forall x. P y \longrightarrow P x$
by *blast*

Problem 19

lemma $\exists x. \forall y z. (P y \longrightarrow Q z) \longrightarrow (P x \longrightarrow Q x)$
by *blast*

Problem 20

lemma $(\forall x y. \exists z. \forall w. (P x \ \& \ Q y \longrightarrow R z \ \& \ S w))$
 $\longrightarrow (\exists x y. P x \ \& \ Q y) \longrightarrow (\exists z. R z)$
by *blast*

Problem 21

lemma $(\exists x. P \longrightarrow Q x) \ \& \ (\exists x. Q x \longrightarrow P) \longrightarrow (\exists x. P = Q x)$
by *blast*

Problem 22

lemma $(\forall x. P = Q x) \longrightarrow (P = (\forall x. Q x))$
by *blast*

Problem 23

lemma $(\forall x. P \mid Q x) = (P \mid (\forall x. Q x))$
by *blast*

Problem 24

lemma $\sim(\exists x. S x \ \& \ Q x) \ \& \ (\forall x. P x \longrightarrow Q x \mid R x) \ \&$
 $(\sim(\exists x. P x) \longrightarrow (\exists x. Q x)) \ \& \ (\forall x. Q x \mid R x \longrightarrow S x)$
 $\longrightarrow (\exists x. P x \ \& \ R x)$
by *blast*

Problem 25

lemma $(\exists x. P x) \ \&$
 $(\forall x. L x \longrightarrow \sim (M x \ \& \ R x)) \ \&$

$$\begin{aligned}
& (\forall x. P x \longrightarrow (M x \& L x)) \& \\
& ((\forall x. P x \longrightarrow Q x) \mid (\exists x. P x \& R x)) \\
& \longrightarrow (\exists x. Q x \& P x)
\end{aligned}$$

by *blast*

Problem 26; has 24 Horn clauses

$$\begin{aligned}
\textbf{lemma } & ((\exists x. p x) = (\exists x. q x)) \& \\
& (\forall x. \forall y. p x \& q y \longrightarrow (r x = s y)) \\
& \longrightarrow ((\forall x. p x \longrightarrow r x) = (\forall x. q x \longrightarrow s x))
\end{aligned}$$

by *blast*

Problem 27; has 13 Horn clauses

$$\begin{aligned}
\textbf{lemma } & (\exists x. P x \& \sim Q x) \& \\
& (\forall x. P x \longrightarrow R x) \& \\
& (\forall x. M x \& L x \longrightarrow P x) \& \\
& ((\exists x. R x \& \sim Q x) \longrightarrow (\forall x. L x \longrightarrow \sim R x)) \\
& \longrightarrow (\forall x. M x \longrightarrow \sim L x)
\end{aligned}$$

by *blast*

Problem 28. AMENDED; has 14 Horn clauses

$$\begin{aligned}
\textbf{lemma } & (\forall x. P x \longrightarrow (\forall x. Q x)) \& \\
& ((\forall x. Q x \mid R x) \longrightarrow (\exists x. Q x \& S x)) \& \\
& ((\exists x. S x) \longrightarrow (\forall x. L x \longrightarrow M x)) \\
& \longrightarrow (\forall x. P x \& L x \longrightarrow M x)
\end{aligned}$$

by *blast*

Problem 29. Essentially the same as Principia Mathematica *11.71. 62 Horn clauses

$$\begin{aligned}
\textbf{lemma } & (\exists x. F x) \& (\exists y. G y) \\
& \longrightarrow (((\forall x. F x \longrightarrow H x) \& (\forall y. G y \longrightarrow J y)) = \\
& (\forall x y. F x \& G y \longrightarrow H x \& J y))
\end{aligned}$$

by *blast*

Problem 30

$$\begin{aligned}
\textbf{lemma } & (\forall x. P x \mid Q x \longrightarrow \sim R x) \& (\forall x. (Q x \longrightarrow \sim S x) \longrightarrow P x \& R x) \\
& \longrightarrow (\forall x. S x)
\end{aligned}$$

by *blast*

Problem 31; has 10 Horn clauses; first negative clauses is useless

$$\begin{aligned}
\textbf{lemma } & \sim(\exists x. P x \& (Q x \mid R x)) \& \\
& (\exists x. L x \& P x) \& \\
& (\forall x. \sim R x \longrightarrow M x) \\
& \longrightarrow (\exists x. L x \& M x)
\end{aligned}$$

by *blast*

Problem 32

$$\textbf{lemma } (\forall x. P x \& (Q x \mid R x) \longrightarrow S x) \&$$

$$(\forall x. S x \ \& \ R x \ \longrightarrow L x) \ \& \\
(\forall x. M x \ \longrightarrow R x) \\
\longrightarrow (\forall x. P x \ \& \ M x \ \longrightarrow L x) \\
\text{by } \textit{blast}$$

Problem 33; has 55 Horn clauses

lemma $(\forall x. P a \ \& \ (P x \ \longrightarrow P b) \ \longrightarrow P c) =$
 $(\forall x. (\sim P a \mid P x \mid P c) \ \& \ (\sim P a \mid \sim P b \mid P c))$
by *blast*

Problem 34: Andrews's challenge has 924 Horn clauses

lemma $((\exists x. \forall y. p x = p y) = ((\exists x. q x) = (\forall y. p y))) =$
 $((\exists x. \forall y. q x = q y) = ((\exists x. p x) = (\forall y. q y)))$
by *blast*

Problem 35

lemma $\exists x y. P x y \ \longrightarrow \ (\forall u v. P u v)$
by *blast*

Problem 36; has 15 Horn clauses

lemma $(\forall x. \exists y. J x y) \ \& \ (\forall x. \exists y. G x y) \ \& \\
(\forall x y. J x y \mid G x y \ \longrightarrow (\forall z. J y z \mid G y z \ \longrightarrow H x z)) \\
\longrightarrow (\forall x. \exists y. H x y) \\
\text{by } \textit{blast}$

Problem 37; has 10 Horn clauses

lemma $(\forall z. \exists w. \forall x. \exists y. \\
(P x z \ \longrightarrow P y w) \ \& \ P y z \ \& \ (P y w \ \longrightarrow (\exists u. Q u w))) \ \& \\
(\forall x z. \sim P x z \ \longrightarrow (\exists y. Q y z)) \ \& \\
((\exists x y. Q x y) \ \longrightarrow (\forall x. R x x)) \\
\longrightarrow (\forall x. \exists y. R x y) \\
\text{by } \textit{blast} \text{ --- causes unification tracing messages}$

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x. p a \ \& \ (p x \ \longrightarrow (\exists y. p y \ \& \ r x y)) \ \longrightarrow \\
(\exists z. \exists w. p z \ \& \ r x w \ \& \ r w z)) = \\
(\forall x. (\sim p a \mid p x \mid (\exists z. \exists w. p z \ \& \ r x w \ \& \ r w z)) \ \& \\
(\sim p a \mid \sim (\exists y. p y \ \& \ r x y) \mid \\
(\exists z. \exists w. p z \ \& \ r x w \ \& \ r w z))) \\
\text{by } \textit{blast}$

Problem 39

lemma $\sim (\exists x. \forall y. F y x = (\sim F y y))$
by *blast*

Problem 40. AMENDED

lemma $(\exists y. \forall x. F x y = F x x)$
 $--> \sim (\forall x. \exists y. \forall z. F z y = (\sim F z x))$
by *blast*

Problem 41

lemma $(\forall z. (\exists y. (\forall x. f x y = (f x z \ \& \ \sim f x x))))$
 $--> \sim (\exists z. \forall x. f x z)$
by *blast*

Problem 42

lemma $\sim (\exists y. \forall x. p x y = (\sim (\exists z. p x z \ \& \ p z x)))$
by *blast*

Problem 43 NOW PROVED AUTOMATICALLY!!

lemma $(\forall x. \forall y. q x y = (\forall z. p z x = (p z y::bool)))$
 $--> (\forall x. (\forall y. q x y = (q y x::bool)))$
by *blast*

Problem 44: 13 Horn clauses; 7-step proof

lemma $(\forall x. f x --> (\exists y. g y \ \& \ h x y \ \& \ (\exists y. g y \ \& \ \sim h x y))) \ \&$
 $(\exists x. j x \ \& \ (\forall y. g y --> h x y))$
 $--> (\exists x. j x \ \& \ \sim f x)$
by *blast*

Problem 45; has 27 Horn clauses; 54-step proof

lemma $(\forall x. f x \ \& \ (\forall y. g y \ \& \ h x y --> j x y)$
 $--> (\forall y. g y \ \& \ h x y --> k y)) \ \&$
 $\sim (\exists y. l y \ \& \ k y) \ \&$
 $(\exists x. f x \ \& \ (\forall y. h x y --> l y)$
 $\ \& \ (\forall y. g y \ \& \ h x y --> j x y))$
 $--> (\exists x. f x \ \& \ \sim (\exists y. g y \ \& \ h x y))$
by *blast*

Problem 46; has 26 Horn clauses; 21-step proof

lemma $(\forall x. f x \ \& \ (\forall y. f y \ \& \ h y x --> g y) --> g x) \ \&$
 $((\exists x. f x \ \& \ \sim g x) -->$
 $(\exists x. f x \ \& \ \sim g x \ \& \ (\forall y. f y \ \& \ \sim g y --> j x y))) \ \&$
 $(\forall x y. f x \ \& \ f y \ \& \ h x y --> \sim j y x)$
 $--> (\forall x. f x --> g x)$
by *blast*

Problem 47. Schubert's Steamroller. 26 clauses; 63 Horn clauses. 87094 inferences so far. Searching to depth 36

lemma $(\forall x. wolf x \longrightarrow animal x) \ \& \ (\exists x. wolf x) \ \&$
 $(\forall x. fox x \longrightarrow animal x) \ \& \ (\exists x. fox x) \ \&$
 $(\forall x. bird x \longrightarrow animal x) \ \& \ (\exists x. bird x) \ \&$
 $(\forall x. caterpillar x \longrightarrow animal x) \ \& \ (\exists x. caterpillar x) \ \&$

$(\forall x. \text{snail } x \longrightarrow \text{animal } x) \ \& \ (\exists x. \text{snail } x) \ \&$
 $(\forall x. \text{grain } x \longrightarrow \text{plant } x) \ \& \ (\exists x. \text{grain } x) \ \&$
 $(\forall x. \text{animal } x \longrightarrow$
 $\quad ((\forall y. \text{plant } y \longrightarrow \text{eats } x \ y) \ \vee$
 $\quad (\forall y. \text{animal } y \ \& \ \text{smaller-than } y \ x \ \&$
 $\quad \quad (\exists z. \text{plant } z \ \& \ \text{eats } y \ z) \longrightarrow \text{eats } x \ y))) \ \&$
 $(\forall x \ y. \text{bird } y \ \& \ (\text{snail } x \vee \text{caterpillar } x) \longrightarrow \text{smaller-than } x \ y) \ \&$
 $(\forall x \ y. \text{bird } x \ \& \ \text{fox } y \longrightarrow \text{smaller-than } x \ y) \ \&$
 $(\forall x \ y. \text{fox } x \ \& \ \text{wolf } y \longrightarrow \text{smaller-than } x \ y) \ \&$
 $(\forall x \ y. \text{wolf } x \ \& \ (\text{fox } y \vee \text{grain } y) \longrightarrow \sim \text{eats } x \ y) \ \&$
 $(\forall x \ y. \text{bird } x \ \& \ \text{caterpillar } y \longrightarrow \text{eats } x \ y) \ \&$
 $(\forall x \ y. \text{bird } x \ \& \ \text{snail } y \longrightarrow \sim \text{eats } x \ y) \ \&$
 $(\forall x. (\text{caterpillar } x \vee \text{snail } x) \longrightarrow (\exists y. \text{plant } y \ \& \ \text{eats } x \ y))$
 $\longrightarrow (\exists x \ y. \text{animal } x \ \& \ \text{animal } y \ \& \ (\exists z. \text{grain } z \ \& \ \text{eats } y \ z \ \& \ \text{eats } x \ y))$
by (*tactic*⟨⟨*Meson.safe-best-meson-tac* @{context} 1⟩⟩)

— Nearly twice as fast as *meson*, which performs iterative deepening rather than best-first search

The Los problem. Circulated by John Harrison

lemma $(\forall x \ y \ z. P \ x \ y \ \& \ P \ y \ z \longrightarrow P \ x \ z) \ \&$
 $(\forall x \ y \ z. Q \ x \ y \ \& \ Q \ y \ z \longrightarrow Q \ x \ z) \ \&$
 $(\forall x \ y. P \ x \ y \longrightarrow P \ y \ x) \ \&$
 $(\forall x \ y. P \ x \ y \mid Q \ x \ y)$
 $\longrightarrow (\forall x \ y. P \ x \ y) \mid (\forall x \ y. Q \ x \ y)$
by *meson*

A similar example, suggested by Johannes Schumann and credited to Pelletier

lemma $(\forall x \ y \ z. P \ x \ y \longrightarrow P \ y \ z \longrightarrow P \ x \ z) \longrightarrow$
 $(\forall x \ y \ z. Q \ x \ y \longrightarrow Q \ y \ z \longrightarrow Q \ x \ z) \longrightarrow$
 $(\forall x \ y. Q \ x \ y \longrightarrow Q \ y \ x) \longrightarrow (\forall x \ y. P \ x \ y \mid Q \ x \ y) \longrightarrow$
 $(\forall x \ y. P \ x \ y) \mid (\forall x \ y. Q \ x \ y)$
by *meson*

Problem 50. What has this to do with equality?

lemma $(\forall x. P \ a \ x \mid (\forall y. P \ x \ y)) \longrightarrow (\exists x. \forall y. P \ x \ y)$
by *blast*

Problem 54: NOT PROVED

lemma $(\forall y.::'a. \exists z. \forall x. F \ x \ z = (x=y)) \longrightarrow$
 $\sim (\exists w. \forall x. F \ x \ w = (\forall u. F \ x \ u \longrightarrow (\exists y. F \ y \ u \ \& \ \sim (\exists z. F \ z \ u \ \& \ F \ z \ y))))$
oops

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988).
meson cannot report who killed Agatha.

lemma *lives agatha* $\ \& \ \text{lives butler} \ \& \ \text{lives charles} \ \&$

$(killed\ agatha\ agatha \mid killed\ butler\ agatha \mid killed\ charles\ agatha) \ \&$
 $(\forall x\ y. \ killed\ x\ y \dashrightarrow hates\ x\ y \ \& \ \sim richer\ x\ y) \ \&$
 $(\forall x. \ hates\ agatha\ x \dashrightarrow \sim hates\ charles\ x) \ \&$
 $(hates\ agatha\ agatha \ \& \ hates\ agatha\ charles) \ \&$
 $(\forall x. \ lives\ x \ \& \ \sim richer\ x\ agatha \dashrightarrow hates\ butler\ x) \ \&$
 $(\forall x. \ hates\ agatha\ x \dashrightarrow hates\ butler\ x) \ \&$
 $(\forall x. \ \sim hates\ x\ agatha \mid \sim hates\ x\ butler \mid \sim hates\ x\ charles) \dashrightarrow$
 $(\exists x. \ killed\ x\ agatha)$

by *meson*

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c) \ \& \ P\ (f\ b\ c)\ (f\ a\ c) \ \&$
 $(\forall x\ y\ z. \ P\ x\ y \ \& \ P\ y\ z \dashrightarrow P\ x\ z) \dashrightarrow P\ (f\ a\ b)\ (f\ a\ c)$

by *blast*

Problem 58: Challenge found on info-hol

lemma $\forall P\ Q\ R\ x. \ \exists v\ w. \ \forall y\ z. \ P\ x \ \& \ Q\ y \dashrightarrow (P\ v \mid R\ w) \ \& \ (R\ z \dashrightarrow Q\ v)$

by *blast*

Problem 59

lemma $(\forall x. \ P\ x = (\sim P(f\ x))) \dashrightarrow (\exists x. \ P\ x \ \& \ \sim P(f\ x))$

by *blast*

Problem 60

lemma $\forall x. \ P\ x\ (f\ x) = (\exists y. \ (\forall z. \ P\ z\ y \dashrightarrow P\ z\ (f\ x)) \ \& \ P\ x\ y)$

by *blast*

Problem 62 as corrected in JAR 18 (1997), page 135

lemma $(\forall x. \ p\ a \ \& \ (p\ x \dashrightarrow p(f\ x)) \dashrightarrow p(f(f\ x))) =$
 $(\forall x. \ (\sim p\ a \mid p\ x \mid p(f(f\ x))) \ \&$
 $(\sim p\ a \mid \sim p(f\ x) \mid p(f(f\ x))))$

by *blast*

* Charles Morgan's problems *

lemma

assumes $a: \forall x\ y. \ T(i\ x(i\ y\ x))$
and $b: \forall x\ y\ z. \ T(i\ (i\ x\ (i\ y\ z))\ (i\ (i\ x\ y)\ (i\ x\ z)))$
and $c: \forall x\ y. \ T(i\ (i\ (n\ x)\ (n\ y))\ (i\ y\ x))$
and $c': \forall x\ y. \ T(i\ (i\ y\ x)\ (i\ (n\ x)\ (n\ y)))$
and $d: \forall x\ y. \ T(i\ x\ y) \ \& \ T\ x \dashrightarrow T\ y$

shows *True*

proof —

from $a\ b\ d$ **have** $\forall x. \ T(i\ x\ x)$ **by** *blast*

from $a\ b\ c\ d$ **have** $\forall x. \ T(i\ x\ (n(n\ x)))$ — Problem 66

by *metis*

from $a\ b\ c\ d$ **have** $\forall x. \ T(i\ (n(n\ x))\ x)$ — Problem 67

by *meson*

— 4.9s on griffon. 51061 inferences, depth 21

from $a\ b\ c'\ d$ **have** $\forall x. T(i\ x\ (n(n\ x)))$
 — Problem 68: not proved. Listed as satisfiable in TPTP (LCL078-1)
oops
 Problem 71, as found in TPTP (SYN007+1.005)
lemma $p1 = (p2 = (p3 = (p4 = (p5 = (p1 = (p2 = (p3 = (p4 = p5))))))))$
by *blast*
end

34 Set Theory examples: Cantor's Theorem, Schröder-Bernstein Theorem, etc.

theory *set* **imports** *Main* **begin**

These two are cited in Benzmueller and Kohlhase's system description of LEO, CADE-15, 1998 (pages 139-143) as theorems LEO could not prove.

lemma $(X = Y \cup Z) =$
 $(Y \subseteq X \wedge Z \subseteq X \wedge (\forall V. Y \subseteq V \wedge Z \subseteq V \longrightarrow X \subseteq V))$
by *blast*

lemma $(X = Y \cap Z) =$
 $(X \subseteq Y \wedge X \subseteq Z \wedge (\forall V. V \subseteq Y \wedge V \subseteq Z \longrightarrow V \subseteq X))$
by *blast*

Trivial example of term synthesis: apparently hard for some provers!

schematic-lemma $a \neq b \implies a \in ?X \wedge b \notin ?X$
by *blast*

34.1 Examples for the *blast* paper

lemma $(\bigcup x \in C. f\ x \cup g\ x) = \bigcup (f\ ' C) \cup \bigcup (g\ ' C)$
 — Union-image, called *Un-Union-image* in Main HOL
by *blast*

lemma $(\bigcap x \in C. f\ x \cap g\ x) = \bigcap (f\ ' C) \cap \bigcap (g\ ' C)$
 — Inter-image, called *Int-Inter-image* in Main HOL
by *blast*

lemma *singleton-example-1*:
 $\bigwedge S::'a\ set\ set. \forall x \in S. \forall y \in S. x \subseteq y \implies \exists z. S \subseteq \{z\}$
by *blast*

lemma *singleton-example-2*:
 $\forall x \in S. \bigcup S \subseteq x \implies \exists z. S \subseteq \{z\}$
 — Variant of the problem above.
by *blast*

lemma $\exists!x. f (g x) = x \implies \exists!y. g (f y) = y$
 — A unique fixpoint theorem — *fast/best/meson* all fail.
by *metis*

34.2 Cantor's Theorem: There is no surjection from a set to its powerset

lemma *cantor1*: $\neg (\exists f :: 'a \Rightarrow 'a \text{ set}. \forall S. \exists x. f x = S)$
 — Requires best-first search because it is undirectional.
by *best*

schematic-lemma $\forall f :: 'a \Rightarrow 'a \text{ set}. \forall x. f x \neq ?S f$
 — This form displays the diagonal term.
by *best*

schematic-lemma $?S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$
 — This form exploits the set constructs.
by (*rule notI*, *erule rangeE*, *best*)

schematic-lemma $?S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$
 — Or just this!
by *best*

34.3 The Schröder-Berstein Theorem

lemma *disj-lemma*: $-(f ' X) = g ' (-X) \implies f a = g b \implies a \in X \implies b \in X$
by *blast*

lemma *surj-if-then-else*:
 $-(f ' X) = g ' (-X) \implies \text{surj } (\lambda z. \text{if } z \in X \text{ then } f z \text{ else } g z)$
by (*simp add: surj-def*) *blast*

lemma *bij-if-then-else*:
 $\text{inj-on } f X \implies \text{inj-on } g (-X) \implies -(f ' X) = g ' (-X) \implies$
 $h = (\lambda z. \text{if } z \in X \text{ then } f z \text{ else } g z) \implies \text{inj } h \wedge \text{surj } h$
apply (*unfold inj-on-def*)
apply (*simp add: surj-if-then-else*)
apply (*blast dest: disj-lemma sym*)
done

lemma *decomposition*: $\exists X. X = -(g ' (- (f ' X)))$
apply (*rule exI*)
apply (*rule lfp-unfold*)
apply (*rule monoI*, *blast*)
done

theorem *Schroeder-Bernstein*:
 $\text{inj } (f :: 'a \Rightarrow 'b) \implies \text{inj } (g :: 'b \Rightarrow 'a)$


```

     $\implies \exists h:: 'a \Rightarrow 'b. \text{inj } h \wedge \text{surj } h$ 
  apply (rule decomposition [where f=f and g=g, THEN exE])
  apply (rule-tac x = ( $\lambda z. \text{if } z \in x \text{ then } f \ z \text{ else } \text{inv } g \ z$ ) in exI)
  — The term above can be synthesized by a sufficiently detailed proof.
  apply (rule bij-if-then-else)
    apply (rule-tac [4] refl)
    apply (rule-tac [2] inj-on-inv-into)
    apply (erule subset-inj-on [OF - subset-UNIV])
  apply blast
  apply (erule ssbst, subst double-complement, erule inv-image-comp [symmetric])
done

```

34.4 A simple party theorem

At any party there are two people who know the same number of people. Provided the party consists of at least two people and the knows relation is symmetric. Knowing yourself does not count — otherwise knows needs to be reflexive. (From Freek Wiedijk’s talk at TPHOLs 2007.)

```

lemma equal-number-of-acquaintances:
  assumes Domain R <= A and sym R and card A ≥ 2
  shows  $\neg \text{inj-on } (\%a. \text{card}(R \text{ “ } \{a\} - \{a\})) \ A$ 
proof —
  let ?N =  $\%a. \text{card}(R \text{ “ } \{a\} - \{a\})$ 
  let ?n = card A
  have finite A using ⟨card A ≥ 2⟩ by (auto intro: ccontr)
  have 0: R “ A <= A using ⟨sym R⟩ ⟨Domain R <= A⟩
    unfolding Domain-def sym-def by blast
  have h: ALL a:A. R “ {a} <= A using 0 by blast
  hence 1: ALL a:A. finite(R “ {a}) using ⟨finite A⟩
    by (blast intro: finite-subset)
  have sub: ?N ‘ A <= {0.. $\text{?n}$ }
proof —
  have ALL a:A. R “ {a} - {a} < A using h by blast
  thus ?thesis using psubset-card-mono[OF ⟨finite A⟩] by auto
qed
show  $\sim \text{inj-on } ?N \ A$  (is  $\sim ?I$ )
proof
  assume ?I
  hence ?n = card(?N ‘ A) by (rule card-image[symmetric])
  with sub ⟨finite A⟩ have 2[simp]: ?N ‘ A = {0.. $\text{?n}$ }
    using subset-card-intvl-is-intvl[of - 0] by (auto)
  have 0 : ?N ‘ A and ?n - 1 : ?N ‘ A using ⟨card A ≥ 2⟩ by simp+
  then obtain a b where ab: a:A b:A and Na: ?N a = 0 and Nb: ?N b = ?n
  — 1
    by (auto simp del: 2)
  have a ≠ b using Na Nb ⟨card A ≥ 2⟩ by auto
  have R “ {a} - {a} = {} by (metis 1 Na ab card-eq-0-iff finite-Diff)
  hence b  $\notin R \text{ “ } \{a\}$  using ⟨a ≠ b⟩ by blast

```

hence $a \notin R$ “ $\{b\}$ **by** (*metis Image-singleton-iff assms(2) sym-def*)
hence $3: R$ “ $\{b\} - \{b\} \leq A - \{a,b\}$ **using** $0\ ab$ **by** *blast*
have $4: \text{finite } (A - \{a,b\})$ **using** $\langle \text{finite } A \rangle$ **by** *simp*
have $?N\ b \leq ?n - 2$ **using** $ab\ \langle a \neq b \rangle\ \langle \text{finite } A \rangle\ \text{card-mono}[OF\ 4\ 3]$ **by** *simp*
then show *False* **using** $Nb\ \langle \text{card } A \geq 2 \rangle$ **by** *arith*
qed
qed

From W. W. Bledsoe and Guohui Feng, SET-VAR. JAR 11 (3), 1993, pages 293-314.

Isabelle can prove the easy examples without any special mechanisms, but it can't prove the hard ones.

lemma $\exists A. (\forall x \in A. x \leq (0::int))$
— Example 1, page 295.
by *force*

lemma $D \in F \implies \exists G. \forall A \in G. \exists B \in F. A \subseteq B$
— Example 2.
by *force*

lemma $P\ a \implies \exists A. (\forall x \in A. P\ x) \wedge (\exists y. y \in A)$
— Example 3.
by *force*

lemma $a < b \wedge b < (c::int) \implies \exists A. a \notin A \wedge b \in A \wedge c \notin A$
— Example 4.
by *force*

lemma $P\ (f\ b) \implies \exists s\ A. (\forall x \in A. P\ x) \wedge f\ s \in A$
— Example 5, page 298.
by *force*

lemma $P\ (f\ b) \implies \exists s\ A. (\forall x \in A. P\ x) \wedge f\ s \in A$
— Example 6.
by *force*

lemma $\exists A. a \notin A$
— Example 7.
by *force*

lemma $(\forall u\ v. u < (0::int) \longrightarrow u \neq \text{abs } v)$
 $\longrightarrow (\exists A::int\ \text{set}. (\forall y. \text{abs } y \notin A) \wedge -2 \in A)$
— Example 8 now needs a small hint.
by (*simp add: abs-if, force*)
— not *blast*, which can't simplify $-2 < 0$

Example 9 omitted (requires the reals).

The paper has no Example 10!

lemma $(\forall A. 0 \in A \wedge (\forall x \in A. \text{Suc } x \in A) \longrightarrow n \in A) \wedge$
 $P\ 0 \wedge (\forall x. P\ x \longrightarrow P\ (\text{Suc } x)) \longrightarrow P\ n$

— Example 11: needs a hint.

by(metis Nat.induct)

lemma

$(\forall A. (0, 0) \in A \wedge (\forall x\ y. (x, y) \in A \longrightarrow (\text{Suc } x, \text{Suc } y) \in A) \longrightarrow (n, m) \in A)$
 $\wedge P\ n \longrightarrow P\ m$

— Example 12.

by auto

lemma

$(\forall x. (\exists u. x = 2 * u) = (\neg (\exists v. \text{Suc } x = 2 * v))) \longrightarrow$
 $(\exists A. \forall x. (x \in A) = (\text{Suc } x \notin A))$

— Example EO1: typo in article, and with the obvious fix it seems to require arithmetic reasoning.

apply clarify

apply (rule-tac $x = \{x. \exists u. x = 2 * u\}$ in exI, auto)

apply metis+

done

end

35 Meson test cases

theory Meson-Test

imports Main

begin

ML <<

val Goal = OldGoals.Goal;

val by = OldGoals.by;

val gethyps = OldGoals.gethyps;

>>

WARNING: there are many potential conflicts between variables used below and constants declared in HOL!

hide-const (**open**) subset member quotient union inter

Test data for the MESON proof procedure (Excludes the equality problems 51, 52, 56, 58)

35.1 Interactive examples

ML <<

writelnProblem 25;

Goal $(\exists x. P\ x) \ \& \ (\forall x. L\ x \longrightarrow \sim (M\ x \ \& \ R\ x)) \ \& \ (\forall x. P\ x \longrightarrow (M\ x \ \& \ L\ x)) \ \& \ ((\forall x. P\ x \longrightarrow Q\ x) \mid (\exists x. P\ x \ \& \ R\ x)) \longrightarrow (\exists x. Q\ x \ \& \ P\ x);$

```

by (rtac ccontr 1);
val [prem25] = gethyps 1;
val nnf25 = Meson.make-nnf @{context} prem25;
val xsko25 = Meson.skolemize @{context} nnf25;
by (cut-facts-tac [xsko25] 1 THEN REPEAT (etac exE 1));
val [-,sko25] = gethyps 1;
val clauses25 = Meson.make-clauses [sko25]; (*7 clauses*)
val horns25 = Meson.make-horns clauses25; (*16 Horn clauses*)
val go25::= Meson.gocls clauses25;
>>

```

```

ML <<
Goal False;
by (rtac go25 1);
by (Meson.depth-prolog-tac horns25);
>>

```

```

ML <<
writelnProblem 26;
Goal (( $\exists x. p\ x$ ) = ( $\exists x. q\ x$ )) & ( $\forall x. \forall y. p\ x \ \& \ q\ y \longrightarrow (r\ x = s\ y)$ )  $\longrightarrow$  (( $\forall x. p\ x \longrightarrow r\ x$ ) = ( $\forall x. q\ x \longrightarrow s\ x$ ));
by (rtac ccontr 1);
val [prem26] = gethyps 1;
val nnf26 = Meson.make-nnf @{context} prem26;
val xsko26 = Meson.skolemize @{context} nnf26;
by (cut-facts-tac [xsko26] 1 THEN REPEAT (etac exE 1));
val [-,sko26] = gethyps 1;
val clauses26 = Meson.make-clauses [sko26]; (*9 clauses*)
val horns26 = Meson.make-horns clauses26; (*24 Horn clauses*)
val go26::= Meson.gocls clauses26;
>>

```

```

ML <<
Goal False;
by (rtac go26 1);
by (Meson.depth-prolog-tac horns26); (*1.4 secs*)
(*Proof is of length 107!!*)
>>

```

```

ML <<
writelnProblem 43 NOW PROVED AUTOMATICALLY!!; (*16 Horn clauses*)
Goal ( $\forall x. \forall y. q\ x\ y = (\forall z. p\ z\ x = (p\ z\ y::bool)) \longrightarrow (\forall x. (\forall y. q\ x\ y = (q\ y\ x::bool)))$ );
by (rtac ccontr 1);
val [prem43] = gethyps 1;
val nnf43 = Meson.make-nnf @{context} prem43;
val xsko43 = Meson.skolemize @{context} nnf43;
by (cut-facts-tac [xsko43] 1 THEN REPEAT (etac exE 1));
val [-,sko43] = gethyps 1;

```

```

val clauses43 = Meson.make-clauses [sko43];    (*6*)
val horns43 = Meson.make-horns clauses43;    (*16*)
val go43:- = Meson.gocls clauses43;
>>

```

```

ML <<
Goal False;
by (rtac go43 1);
by (Meson.best-prolog-tac Meson.size-of-subgoals horns43);  (*1.6 secs*)
>>

```

MORE and MUCH HARDER test data for the MESON proof procedure
(courtesy John Harrison).

abbreviation *EQ001-0-ax equal* $\equiv (\forall X. \text{equal}(X::'a, X)) \ \&$
 $(\forall Y X. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(Y::'a, X)) \ \&$
 $(\forall Y X Z. \text{equal}(X::'a, Y) \ \& \ \text{equal}(Y::'a, Z) \longrightarrow \text{equal}(X::'a, Z))$

abbreviation *BOO002-0-ax equal INVERSE multiplicative-identity*
additive-identity multiply product add sum \equiv
 $(\forall X Y. \text{sum}(X::'a, Y, \text{add}(X::'a, Y))) \ \&$
 $(\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \ \&$
 $(\forall Y X Z. \text{sum}(X::'a, Y, Z) \longrightarrow \text{sum}(Y::'a, X, Z)) \ \&$
 $(\forall Y X Z. \text{product}(X::'a, Y, Z) \longrightarrow \text{product}(Y::'a, X, Z)) \ \&$
 $(\forall X. \text{sum}(\text{additive-identity}::'a, X, X)) \ \&$
 $(\forall X. \text{sum}(X::'a, \text{additive-identity}, X)) \ \&$
 $(\forall X. \text{product}(\text{multiplicative-identity}::'a, X, X)) \ \&$
 $(\forall X. \text{product}(X::'a, \text{multiplicative-identity}, X)) \ \&$
 $(\forall Y Z X V3 V1 V2 V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(X::'a, V3, V4) \longrightarrow \text{sum}(V1::'a, V2, V4)) \ \&$
 $(\forall Y Z V1 V2 X V3 V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(V1::'a, V2, V4) \longrightarrow \text{product}(X::'a, V3, V4)) \ \&$
 $(\forall Y Z V3 X V1 V2 V4. \text{product}(Y::'a, X, V1) \ \& \ \text{product}(Z::'a, X, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(V3::'a, X, V4) \longrightarrow \text{sum}(V1::'a, V2, V4)) \ \&$
 $(\forall Y Z V1 V2 V3 X V4. \text{product}(Y::'a, X, V1) \ \& \ \text{product}(Z::'a, X, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(V1::'a, V2, V4) \longrightarrow \text{product}(V3::'a, X, V4)) \ \&$
 $(\forall Y Z X V3 V1 V2 V4. \text{sum}(X::'a, Y, V1) \ \& \ \text{sum}(X::'a, Z, V2) \ \& \ \text{product}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(X::'a, V3, V4) \longrightarrow \text{product}(V1::'a, V2, V4)) \ \&$
 $(\forall Y Z V1 V2 X V3 V4. \text{sum}(X::'a, Y, V1) \ \& \ \text{sum}(X::'a, Z, V2) \ \& \ \text{product}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(V1::'a, V2, V4) \longrightarrow \text{sum}(X::'a, V3, V4)) \ \&$
 $(\forall Y Z V3 X V1 V2 V4. \text{sum}(Y::'a, X, V1) \ \& \ \text{sum}(Z::'a, X, V2) \ \& \ \text{product}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(V3::'a, X, V4) \longrightarrow \text{product}(V1::'a, V2, V4)) \ \&$
 $(\forall Y Z V1 V2 V3 X V4. \text{sum}(Y::'a, X, V1) \ \& \ \text{sum}(Z::'a, X, V2) \ \& \ \text{product}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(V1::'a, V2, V4) \longrightarrow \text{sum}(V3::'a, X, V4)) \ \&$
 $(\forall X. \text{sum}(\text{INVERSE}(X), X, \text{multiplicative-identity})) \ \&$
 $(\forall X. \text{sum}(X::'a, \text{INVERSE}(X), \text{multiplicative-identity})) \ \&$
 $(\forall X. \text{product}(\text{INVERSE}(X), X, \text{additive-identity})) \ \&$
 $(\forall X. \text{product}(X::'a, \text{INVERSE}(X), \text{additive-identity})) \ \&$
 $(\forall X Y U V. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(X::'a, Y, V) \longrightarrow \text{equal}(U::'a, V)) \ \&$
 $(\forall X Y U V. \text{product}(X::'a, Y, U) \ \& \ \text{product}(X::'a, Y, V) \longrightarrow \text{equal}(U::'a, V))$

abbreviation *BOO002-0-eq INVERSE multiply add product sum equal* \equiv
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \ \& \ \text{sum}(X::'a, W, Z) \dashrightarrow \text{sum}(Y::'a, W, Z)) \ \& \$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \ \& \ \text{sum}(W::'a, X, Z) \dashrightarrow \text{sum}(W::'a, Y, Z)) \ \& \$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \ \& \ \text{sum}(W::'a, Z, X) \dashrightarrow \text{sum}(W::'a, Z, Y)) \ \& \$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \ \& \ \text{product}(X::'a, W, Z) \dashrightarrow \text{product}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, X, Z) \dashrightarrow \text{product}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, Z, X) \dashrightarrow \text{product}(W::'a, Z, Y))$
 $\&$
 $(\forall X Y W. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{add}(X::'a, W), \text{add}(Y::'a, W))) \ \& \$
 $(\forall X W Y. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{add}(W::'a, X), \text{add}(W::'a, Y))) \ \& \$
 $(\forall X Y W. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{multiply}(X::'a, W), \text{multiply}(Y::'a, W)))$
 $\&$
 $(\forall X W Y. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{multiply}(W::'a, X), \text{multiply}(W::'a, Y)))$
 $\&$
 $(\forall X Y. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{INVERSE}(X), \text{INVERSE}(Y)))$

lemma *BOO003-1:*

EQU001-0-ax equal $\&$
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply
product add sum $\&$
BOO002-0-eq INVERSE multiply add product sum equal $\&$
 $(\sim \text{product}(x::'a, x, x)) \dashrightarrow \text{False}$
oops

lemma *BOO004-1:*

EQU001-0-ax equal $\&$
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply
product add sum $\&$
BOO002-0-eq INVERSE multiply add product sum equal $\&$
 $(\sim \text{sum}(x::'a, x, x)) \dashrightarrow \text{False}$
oops

lemma *BOO005-1:*

EQU001-0-ax equal $\&$
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply
product add sum $\&$
BOO002-0-eq INVERSE multiply add product sum equal $\&$
 $(\sim \text{sum}(x::'a, \text{multiplicative-identity}, \text{multiplicative-identity})) \dashrightarrow \text{False}$
oops

lemma *BOO006-1:*

EQU001-0-ax equal $\&$

BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
(\sim product($x::'a$,additive-identity,additive-identity)) \longrightarrow False
oops

lemma *BOO011-1:*

EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
(\sim equal(INVERSE(additive-identity,multiplicative-identity)) \longrightarrow False
by *meson*

abbreviation *CAT003-0-ax f1 compos codomain domain equal there-exists equivalent \equiv*

$(\forall Y X. \text{equivalent}(X::'a, Y) \longrightarrow \text{there-exists}(X)) \ \&$
 $(\forall X Y. \text{equivalent}(X::'a, Y) \longrightarrow \text{equal}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{there-exists}(X) \ \& \ \text{equal}(X::'a, Y) \longrightarrow \text{equivalent}(X::'a, Y)) \ \&$
 $(\forall X. \text{there-exists}(\text{domain}(X)) \longrightarrow \text{there-exists}(X)) \ \&$
 $(\forall X. \text{there-exists}(\text{codomain}(X)) \longrightarrow \text{there-exists}(X)) \ \&$
 $(\forall Y X. \text{there-exists}(\text{compos}(X::'a, Y)) \longrightarrow \text{there-exists}(\text{domain}(X))) \ \&$
 $(\forall X Y. \text{there-exists}(\text{compos}(X::'a, Y)) \longrightarrow \text{equal}(\text{domain}(X), \text{codomain}(Y)))$
 $\&$
 $(\forall X Y. \text{there-exists}(\text{domain}(X)) \ \& \ \text{equal}(\text{domain}(X), \text{codomain}(Y)) \longrightarrow \text{there-exists}(\text{compos}(X::'a, Y)))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{compos}(X::'a, \text{compos}(Y::'a, Z)), \text{compos}(\text{compos}(X::'a, Y), Z)))$
 $\&$
 $(\forall X. \text{equal}(\text{compos}(X::'a, \text{domain}(X)), X)) \ \&$
 $(\forall X. \text{equal}(\text{compos}(\text{codomain}(X), X), X)) \ \&$
 $(\forall X Y. \text{equivalent}(X::'a, Y) \longrightarrow \text{there-exists}(Y)) \ \&$
 $(\forall X Y. \text{there-exists}(X) \ \& \ \text{there-exists}(Y) \ \& \ \text{equal}(X::'a, Y) \longrightarrow \text{equivalent}(X::'a, Y))$
 $\&$
 $(\forall Y X. \text{there-exists}(\text{compos}(X::'a, Y)) \longrightarrow \text{there-exists}(\text{codomain}(X))) \ \&$
 $(\forall X Y. \text{there-exists}(f1(X::'a, Y)) \mid \text{equal}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{equal}(X::'a, f1(X::'a, Y)) \mid \text{equal}(Y::'a, f1(X::'a, Y)) \mid \text{equal}(X::'a, Y))$
 $\&$
 $(\forall X Y. \text{equal}(X::'a, f1(X::'a, Y)) \ \& \ \text{equal}(Y::'a, f1(X::'a, Y)) \longrightarrow \text{equal}(X::'a, Y))$

abbreviation *CAT003-0-eq f1 compos codomain domain equivalent there-exists equal \equiv*

$(\forall X Y. \text{equal}(X::'a, Y) \ \& \ \text{there-exists}(X) \longrightarrow \text{there-exists}(Y)) \ \&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \ \& \ \text{equivalent}(X::'a, Z) \longrightarrow \text{equivalent}(Y::'a, Z)) \ \&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \ \& \ \text{equivalent}(Z::'a, X) \longrightarrow \text{equivalent}(Z::'a, Y)) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{domain}(X), \text{domain}(Y))) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{codomain}(X), \text{codomain}(Y))) \ \&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{compos}(X::'a, Z), \text{compos}(Y::'a, Z))) \ \&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{compos}(Z::'a, X), \text{compos}(Z::'a, Y))) \ \&$

$(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(f1(A::'a, C), f1(B::'a, C))) \ \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(f1(F'::'a, D), f1(F'::'a, E)))$

lemma *CAT001-3:*

EQU001-0-ax equal &
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
(there-exists(compos(a::'a, b))) &
 $(\forall Y X Z. \text{equal}(\text{compos}(\text{compos}(a::'a, b), X), Y) \ \& \ \text{equal}(\text{compos}(\text{compos}(a::'a, b), Z), Y)$
 $\longrightarrow \text{equal}(X::'a, Z)) \ \&$
(there-exists(compos(b::'a, h))) &
 $(\text{equal}(\text{compos}(b::'a, h), \text{compos}(b::'a, g))) \ \&$
 $(\sim \text{equal}(h::'a, g)) \longrightarrow \text{False}$
by *meson*

lemma *CAT003-3:*

EQU001-0-ax equal &
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
(there-exists(compos(a::'a, b))) &
 $(\forall Y X Z. \text{equal}(\text{compos}(X::'a, \text{compos}(a::'a, b)), Y) \ \& \ \text{equal}(\text{compos}(Z::'a, \text{compos}(a::'a, b)), Y)$
 $\longrightarrow \text{equal}(X::'a, Z)) \ \&$
(there-exists(h)) &
 $(\text{equal}(\text{compos}(h::'a, a), \text{compos}(g::'a, a))) \ \&$
 $(\sim \text{equal}(g::'a, h)) \longrightarrow \text{False}$
by *meson*

abbreviation *CAT001-0-ax equal codomain domain identity-map compos product*
defined \equiv

$(\forall X Y. \text{defined}(X::'a, Y) \longrightarrow \text{product}(X::'a, Y, \text{compos}(X::'a, Y))) \ \&$
 $(\forall Z X Y. \text{product}(X::'a, Y, Z) \longrightarrow \text{defined}(X::'a, Y)) \ \&$
 $(\forall X Xy Y Z. \text{product}(X::'a, Y, Xy) \ \& \ \text{defined}(Xy::'a, Z) \longrightarrow \text{defined}(Y::'a, Z))$
 $\ \&$
 $(\forall Y Xy Z X Yz. \text{product}(X::'a, Y, Xy) \ \& \ \text{product}(Y::'a, Z, Yz) \ \& \ \text{defined}(Xy::'a, Z)$
 $\longrightarrow \text{defined}(X::'a, Yz)) \ \&$
 $(\forall Xy Y Z X Yz Xyz. \text{product}(X::'a, Y, Xy) \ \& \ \text{product}(Xy::'a, Z, Xyz) \ \& \ \text{prod-}$
 $\text{uct}(Y::'a, Z, Yz) \longrightarrow \text{product}(X::'a, Yz, Xyz)) \ \&$
 $(\forall Z Yz X Y. \text{product}(Y::'a, Z, Yz) \ \& \ \text{defined}(X::'a, Yz) \longrightarrow \text{defined}(X::'a, Y))$
 $\ \&$
 $(\forall Y X Yz Xy Z. \text{product}(Y::'a, Z, Yz) \ \& \ \text{product}(X::'a, Y, Xy) \ \& \ \text{defined}(X::'a, Yz)$
 $\longrightarrow \text{defined}(Xy::'a, Z)) \ \&$
 $(\forall Yz X Y Xy Z Xyz. \text{product}(Y::'a, Z, Yz) \ \& \ \text{product}(X::'a, Yz, Xyz) \ \& \ \text{prod-}$
 $\text{uct}(X::'a, Y, Xy) \longrightarrow \text{product}(Xy::'a, Z, Xyz)) \ \&$
 $(\forall Y X Z. \text{defined}(X::'a, Y) \ \& \ \text{defined}(Y::'a, Z) \ \& \ \text{identity-map}(Y) \longrightarrow \text{de-}$
 $\text{fined}(X::'a, Z)) \ \&$
 $(\forall X. \text{identity-map}(\text{domain}(X))) \ \&$
 $(\forall X. \text{identity-map}(\text{codomain}(X))) \ \&$

$(\forall X. \text{defined}(X::'a, \text{domain}(X))) \ \&$
 $(\forall X. \text{defined}(\text{codomain}(X), X)) \ \&$
 $(\forall X. \text{product}(X::'a, \text{domain}(X), X)) \ \&$
 $(\forall X. \text{product}(\text{codomain}(X), X, X)) \ \&$
 $(\forall X \ Y. \text{defined}(X::'a, Y) \ \& \ \text{identity-map}(X) \ \longrightarrow \ \text{product}(X::'a, Y, Y)) \ \&$
 $(\forall Y \ X. \text{defined}(X::'a, Y) \ \& \ \text{identity-map}(Y) \ \longrightarrow \ \text{product}(X::'a, Y, X)) \ \&$
 $(\forall X \ Y \ Z \ W. \text{product}(X::'a, Y, Z) \ \& \ \text{product}(X::'a, Y, W) \ \longrightarrow \ \text{equal}(Z::'a, W))$

abbreviation *CAT001-0-eq compos defined identity-map codomain domain product equal* \equiv

$(\forall X \ Y \ Z \ W. \text{equal}(X::'a, Y) \ \& \ \text{product}(X::'a, Z, W) \ \longrightarrow \ \text{product}(Y::'a, Z, W))$
 $\&$
 $(\forall X \ Z \ Y \ W. \text{equal}(X::'a, Y) \ \& \ \text{product}(Z::'a, X, W) \ \longrightarrow \ \text{product}(Z::'a, Y, W))$
 $\&$
 $(\forall X \ Z \ W \ Y. \text{equal}(X::'a, Y) \ \& \ \text{product}(Z::'a, W, X) \ \longrightarrow \ \text{product}(Z::'a, W, Y))$
 $\&$
 $(\forall X \ Y. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{domain}(X), \text{domain}(Y))) \ \&$
 $(\forall X \ Y. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{codomain}(X), \text{codomain}(Y))) \ \&$
 $(\forall X \ Y. \text{equal}(X::'a, Y) \ \& \ \text{identity-map}(X) \ \longrightarrow \ \text{identity-map}(Y)) \ \&$
 $(\forall X \ Y \ Z. \text{equal}(X::'a, Y) \ \& \ \text{defined}(X::'a, Z) \ \longrightarrow \ \text{defined}(Y::'a, Z)) \ \&$
 $(\forall X \ Z \ Y. \text{equal}(X::'a, Y) \ \& \ \text{defined}(Z::'a, X) \ \longrightarrow \ \text{defined}(Z::'a, Y)) \ \&$
 $(\forall X \ Z \ Y. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{compos}(Z::'a, X), \text{compos}(Z::'a, Y))) \ \&$
 $(\forall X \ Y \ Z. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{compos}(X::'a, Z), \text{compos}(Y::'a, Z)))$

lemma *CAT005-1:*

EQU001-0-ax equal $\&$
CAT001-0-ax equal codomain domain identity-map compos product defined $\&$
CAT001-0-eq compos defined identity-map codomain domain product equal $\&$
 $(\text{defined}(a::'a, d)) \ \&$
 $(\text{identity-map}(d)) \ \&$
 $(\sim \text{equal}(\text{domain}(a), d)) \ \longrightarrow \ \text{False}$
oops

lemma *CAT007-1:*

EQU001-0-ax equal $\&$
CAT001-0-ax equal codomain domain identity-map compos product defined $\&$
CAT001-0-eq compos defined identity-map codomain domain product equal $\&$
 $(\text{equal}(\text{domain}(a), \text{codomain}(b))) \ \&$
 $(\sim \text{defined}(a::'a, b)) \ \longrightarrow \ \text{False}$
by meson

lemma *CAT018-1:*

EQU001-0-ax equal $\&$
CAT001-0-ax equal codomain domain identity-map compos product defined $\&$
CAT001-0-eq compos defined identity-map codomain domain product equal $\&$
 $(\text{defined}(a::'a, b)) \ \&$

$(\text{defined}(b::'a,c)) \ \&$
 $(\sim \text{defined}(a::'a,\text{compos}(b::'a,c))) \longrightarrow \text{False}$
oops

lemma COL001-2:

$\text{EQU001-0-ax equal} \ \&$
 $(\forall X \ Y \ Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(s::'a,X),Y),Z),\text{apply}(\text{apply}(X::'a,Z),\text{apply}(Y::'a,Z))))$
 $\&$
 $(\forall Y \ X. \text{equal}(\text{apply}(\text{apply}(k::'a,X),Y),X)) \ \&$
 $(\forall X \ Y \ Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(b::'a,X),Y),Z),\text{apply}(X::'a,\text{apply}(Y::'a,Z))))$
 $\&$
 $(\forall X. \text{equal}(\text{apply}(i::'a,X),X)) \ \&$
 $(\forall A \ B \ C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(\text{apply}(A::'a,C),\text{apply}(B::'a,C))) \ \&$
 $(\forall D \ F' \ E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(\text{apply}(F'::'a,D),\text{apply}(F'::'a,E))) \ \&$
 $(\forall X. \text{equal}(\text{apply}(\text{apply}(\text{apply}(s::'a,\text{apply}(b::'a,X)),i),\text{apply}(\text{apply}(s::'a,\text{apply}(b::'a,X)),i)),\text{apply}(x::'a,\text{apply}($
 $\&$
 $(\forall Y. \sim \text{equal}(Y::'a,\text{apply}(\text{combinator}::'a,Y))) \longrightarrow \text{False}$
by meson

lemma COL023-1:

$\text{EQU001-0-ax equal} \ \&$
 $(\forall X \ Y \ Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(b::'a,X),Y),Z),\text{apply}(X::'a,\text{apply}(Y::'a,Z))))$
 $\&$
 $(\forall X \ Y \ Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(n::'a,X),Y),Z),\text{apply}(\text{apply}(\text{apply}(X::'a,Z),Y),Z)))$
 $\&$
 $(\forall A \ B \ C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(\text{apply}(A::'a,C),\text{apply}(B::'a,C))) \ \&$
 $(\forall D \ F' \ E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(\text{apply}(F'::'a,D),\text{apply}(F'::'a,E))) \ \&$
 $(\forall Y. \sim \text{equal}(Y::'a,\text{apply}(\text{combinator}::'a,Y))) \longrightarrow \text{False}$
by meson

lemma COL032-1:

$\text{EQU001-0-ax equal} \ \&$
 $(\forall X. \text{equal}(\text{apply}(m::'a,X),\text{apply}(X::'a,X))) \ \&$
 $(\forall Y \ X \ Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(q::'a,X),Y),Z),\text{apply}(Y::'a,\text{apply}(X::'a,Z))))$
 $\&$
 $(\forall A \ B \ C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(\text{apply}(A::'a,C),\text{apply}(B::'a,C))) \ \&$
 $(\forall D \ F' \ E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(\text{apply}(F'::'a,D),\text{apply}(F'::'a,E))) \ \&$
 $(\forall G \ H. \text{equal}(G::'a,H) \longrightarrow \text{equal}(f(G),f(H))) \ \&$
 $(\forall Y. \sim \text{equal}(\text{apply}(Y::'a,f(Y)),\text{apply}(f(Y),\text{apply}(Y::'a,f(Y))))) \longrightarrow \text{False}$
by meson

lemma COL052-2:

$\text{EQU001-0-ax equal} \ \&$
 $(\forall X \ Y \ W. \text{equal}(\text{response}(\text{compos}(X::'a,Y),W),\text{response}(X::'a,\text{response}(Y::'a,W))))$
 $\&$

$(\forall X Y. \text{agreeable}(X) \longrightarrow \text{equal}(\text{response}(X::'a, \text{common-bird}(Y)), \text{response}(Y::'a, \text{common-bird}(Y))))$
 $\&$
 $(\forall Z X. \text{equal}(\text{response}(X::'a, Z), \text{response}(\text{compatible}(X), Z)) \longrightarrow \text{agreeable}(X))$
 $\&$
 $(\forall A B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{common-bird}(A), \text{common-bird}(B))) \&$
 $(\forall C D. \text{equal}(C::'a, D) \longrightarrow \text{equal}(\text{compatible}(C), \text{compatible}(D))) \&$
 $(\forall Q R. \text{equal}(Q::'a, R) \& \text{agreeable}(Q) \longrightarrow \text{agreeable}(R)) \&$
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{compos}(A::'a, C), \text{compos}(B::'a, C))) \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{compos}(F'::'a, D), \text{compos}(F'::'a, E))) \&$
 $(\forall G H I'. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{response}(G::'a, I'), \text{response}(H::'a, I'))) \&$
 $(\forall J L K'. \text{equal}(J::'a, K') \longrightarrow \text{equal}(\text{response}(L::'a, J), \text{response}(L::'a, K'))) \&$
 $(\text{agreeable}(c)) \&$
 $(\sim \text{agreeable}(a)) \&$
 $(\text{equal}(c::'a, \text{compos}(a::'a, b))) \longrightarrow \text{False}$
oops

lemma COL075-2:

$\text{EQU001-0-ax equal} \&$
 $(\forall Y X. \text{equal}(\text{apply}(\text{apply}(k::'a, X), Y), X)) \&$
 $(\forall X Y Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(\text{abstraction}::'a, X), Y), Z), \text{apply}(\text{apply}(X::'a, \text{apply}(k::'a, Z)), \text{apply}(Y::'a, Z)))$
 $\&$
 $(\forall D E F'. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{apply}(D::'a, F'), \text{apply}(E::'a, F'))) \&$
 $(\forall G I' H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{apply}(I'::'a, G), \text{apply}(I'::'a, H))) \&$
 $(\forall A B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(b(A), b(B))) \&$
 $(\forall C D. \text{equal}(C::'a, D) \longrightarrow \text{equal}(c(C), c(D))) \&$
 $(\forall Y. \sim \text{equal}(\text{apply}(\text{apply}(Y::'a, b(Y)), c(Y)), \text{apply}(b(Y), b(Y)))) \longrightarrow \text{False}$
oops

lemma COM001-1:

$(\forall \text{Goal-state Start-state. follows}(\text{Goal-state}::'a, \text{Start-state}) \longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state}))$
 $\&$
 $(\forall \text{Goal-state Intermediate-state Start-state. succeeds}(\text{Goal-state}::'a, \text{Intermediate-state})$
 $\& \text{succeeds}(\text{Intermediate-state}::'a, \text{Start-state}) \longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state}))$
 $\&$
 $(\forall \text{Start-state Label Goal-state. has}(\text{Start-state}::'a, \text{goto}(\text{Label})) \& \text{labels}(\text{Label}::'a, \text{Goal-state})$
 $\longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state})) \&$
 $(\forall \text{Start-state Condition Goal-state. has}(\text{Start-state}::'a, \text{ifthen}(\text{Condition}::'a, \text{Goal-state}))$
 $\longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state})) \&$
 $(\text{labels}(\text{loop}::'a, p3)) \&$
 $(\text{has}(p3::'a, \text{ifthen}(\text{equal}(\text{register-j}::'a, n), p4))) \&$
 $(\text{has}(p4::'a, \text{goto}(\text{out}))) \&$
 $(\text{follows}(p5::'a, p4)) \&$
 $(\text{follows}(p8::'a, p3)) \&$
 $(\text{has}(p8::'a, \text{goto}(\text{loop}))) \&$
 $(\sim \text{succeeds}(p3::'a, p3)) \longrightarrow \text{False}$
by meson

lemma *COM002-1*:

$(\forall \text{Goal-state Start-state. follows}(\text{Goal-state}::'a, \text{Start-state}) \longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state}))$
 $\&$
 $(\forall \text{Goal-state Intermediate-state Start-state. succeeds}(\text{Goal-state}::'a, \text{Intermediate-state})$
 $\& \text{succeeds}(\text{Intermediate-state}::'a, \text{Start-state}) \longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state}))$
 $\&$
 $(\forall \text{Start-state Label Goal-state. has}(\text{Start-state}::'a, \text{goto}(\text{Label})) \& \text{labels}(\text{Label}::'a, \text{Goal-state})$
 $\longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state})) \&$
 $(\forall \text{Start-state Condition Goal-state. has}(\text{Start-state}::'a, \text{ifthen}(\text{Condition}::'a, \text{Goal-state}))$
 $\longrightarrow \text{succeeds}(\text{Goal-state}::'a, \text{Start-state})) \&$
 $(\text{has}(p1::'a, \text{assign}(\text{register-j}::'a, \text{num0}))) \&$
 $(\text{follows}(p2::'a, p1)) \&$
 $(\text{has}(p2::'a, \text{assign}(\text{register-k}::'a, \text{num1}))) \&$
 $(\text{labels}(\text{loop}::'a, p3)) \&$
 $(\text{follows}(p3::'a, p2)) \&$
 $(\text{has}(p3::'a, \text{ifthen}(\text{equal}(\text{register-j}::'a, n), p4))) \&$
 $(\text{has}(p4::'a, \text{goto}(\text{out}))) \&$
 $(\text{follows}(p5::'a, p4)) \&$
 $(\text{follows}(p6::'a, p3)) \&$
 $(\text{has}(p6::'a, \text{assign}(\text{register-k}::'a, \text{mtimes}(\text{num2}::'a, \text{register-k})))) \&$
 $(\text{follows}(p7::'a, p6)) \&$
 $(\text{has}(p7::'a, \text{assign}(\text{register-j}::'a, \text{mplus}(\text{register-j}::'a, \text{num1})))) \&$
 $(\text{follows}(p8::'a, p7)) \&$
 $(\text{has}(p8::'a, \text{goto}(\text{loop}))) \&$
 $(\sim \text{succeeds}(p3::'a, p3)) \longrightarrow \text{False}$
by *meson*

lemma *COM002-2*:

$(\forall \text{Goal-state Start-state. } \sim(\text{fails}(\text{Goal-state}::'a, \text{Start-state}) \& \text{follows}(\text{Goal-state}::'a, \text{Start-state})))$
 $\&$
 $(\forall \text{Goal-state Intermediate-state Start-state. fails}(\text{Goal-state}::'a, \text{Start-state}) \longrightarrow$
 $\text{fails}(\text{Goal-state}::'a, \text{Intermediate-state}) \mid \text{fails}(\text{Intermediate-state}::'a, \text{Start-state})) \&$
 $(\forall \text{Start-state Label Goal-state. } \sim(\text{fails}(\text{Goal-state}::'a, \text{Start-state}) \& \text{has}(\text{Start-state}::'a, \text{goto}(\text{Label}))$
 $\& \text{labels}(\text{Label}::'a, \text{Goal-state}))) \&$
 $(\forall \text{Start-state Condition Goal-state. } \sim(\text{fails}(\text{Goal-state}::'a, \text{Start-state}) \& \text{has}(\text{Start-state}::'a, \text{ifthen}(\text{Condition}::$
 $\&$
 $(\text{has}(p1::'a, \text{assign}(\text{register-j}::'a, \text{num0}))) \&$
 $(\text{follows}(p2::'a, p1)) \&$
 $(\text{has}(p2::'a, \text{assign}(\text{register-k}::'a, \text{num1}))) \&$
 $(\text{labels}(\text{loop}::'a, p3)) \&$
 $(\text{follows}(p3::'a, p2)) \&$
 $(\text{has}(p3::'a, \text{ifthen}(\text{equal}(\text{register-j}::'a, n), p4))) \&$
 $(\text{has}(p4::'a, \text{goto}(\text{out}))) \&$
 $(\text{follows}(p5::'a, p4)) \&$
 $(\text{follows}(p6::'a, p3)) \&$
 $(\text{has}(p6::'a, \text{assign}(\text{register-k}::'a, \text{mtimes}(\text{num2}::'a, \text{register-k})))) \&$
 $(\text{follows}(p7::'a, p6)) \&$

$(has(p7::'a, assign(register-j::'a, mplus(register-j::'a, num1)))) \&$
 $(follows(p8::'a, p7)) \&$
 $(has(p8::'a, goto(loop))) \&$
 $(fails(p3::'a, p3)) \longrightarrow False$
by meson

lemma COM003-2:

$(\forall X\ Y\ Z. program-decides(X) \& program(Y) \longrightarrow decides(X::'a, Y, Z)) \&$
 $(\forall X. program-decides(X) \mid program(f2(X))) \&$
 $(\forall X. decides(X::'a, f2(X), f1(X)) \longrightarrow program-decides(X)) \&$
 $(\forall X. program-program-decides(X) \longrightarrow program(X)) \&$
 $(\forall X. program-program-decides(X) \longrightarrow program-decides(X)) \&$
 $(\forall X. program(X) \& program-decides(X) \longrightarrow program-program-decides(X)) \&$
 $(\forall X. algorithm-program-decides(X) \longrightarrow algorithm(X)) \&$
 $(\forall X. algorithm-program-decides(X) \longrightarrow program-decides(X)) \&$
 $(\forall X. algorithm(X) \& program-decides(X) \longrightarrow algorithm-program-decides(X))$
 $\&$
 $(\forall Y\ X. program-halts2(X::'a, Y) \longrightarrow program(X)) \&$
 $(\forall X\ Y. program-halts2(X::'a, Y) \longrightarrow halts2(X::'a, Y)) \&$
 $(\forall X\ Y. program(X) \& halts2(X::'a, Y) \longrightarrow program-halts2(X::'a, Y)) \&$
 $(\forall W\ X\ Y\ Z. halts3-outputs(X::'a, Y, Z, W) \longrightarrow halts3(X::'a, Y, Z)) \&$
 $(\forall Y\ Z\ X\ W. halts3-outputs(X::'a, Y, Z, W) \longrightarrow outputs(X::'a, W)) \&$
 $(\forall Y\ Z\ X\ W. halts3(X::'a, Y, Z) \& outputs(X::'a, W) \longrightarrow halts3-outputs(X::'a, Y, Z, W))$
 $\&$
 $(\forall Y\ X. program-not-halts2(X::'a, Y) \longrightarrow program(X)) \&$
 $(\forall X\ Y. \sim(program-not-halts2(X::'a, Y) \& halts2(X::'a, Y))) \&$
 $(\forall X\ Y. program(X) \longrightarrow program-not-halts2(X::'a, Y) \mid halts2(X::'a, Y)) \&$
 $(\forall W\ X\ Y. halts2-outputs(X::'a, Y, W) \longrightarrow halts2(X::'a, Y)) \&$
 $(\forall Y\ X\ W. halts2-outputs(X::'a, Y, W) \longrightarrow outputs(X::'a, W)) \&$
 $(\forall Y\ X\ W. halts2(X::'a, Y) \& outputs(X::'a, W) \longrightarrow halts2-outputs(X::'a, Y, W))$
 $\&$
 $(\forall X\ W\ Y\ Z. program-halts2-halts3-outputs(X::'a, Y, Z, W) \longrightarrow program-halts2(Y::'a, Z))$
 $\&$
 $(\forall X\ Y\ Z\ W. program-halts2-halts3-outputs(X::'a, Y, Z, W) \longrightarrow halts3-outputs(X::'a, Y, Z, W))$
 $\&$
 $(\forall X\ Y\ Z\ W. program-halts2(Y::'a, Z) \& halts3-outputs(X::'a, Y, Z, W) \longrightarrow$
 $program-halts2-halts3-outputs(X::'a, Y, Z, W)) \&$
 $(\forall X\ W\ Y\ Z. program-not-halts2-halts3-outputs(X::'a, Y, Z, W) \longrightarrow program-not-halts2(Y::'a, Z))$
 $\&$
 $(\forall X\ Y\ Z\ W. program-not-halts2-halts3-outputs(X::'a, Y, Z, W) \longrightarrow halts3-outputs(X::'a, Y, Z, W))$
 $\&$
 $(\forall X\ Y\ Z\ W. program-not-halts2(Y::'a, Z) \& halts3-outputs(X::'a, Y, Z, W) \longrightarrow$
 $program-not-halts2-halts3-outputs(X::'a, Y, Z, W)) \&$
 $(\forall X\ W\ Y. program-halts2-halts2-outputs(X::'a, Y, W) \longrightarrow program-halts2(Y::'a, Y))$
 $\&$
 $(\forall X\ Y\ W. program-halts2-halts2-outputs(X::'a, Y, W) \longrightarrow halts2-outputs(X::'a, Y, W))$
 $\&$
 $(\forall X\ Y\ W. program-halts2(Y::'a, Y) \& halts2-outputs(X::'a, Y, W) \longrightarrow program-halts2-halts2-outputs(X::'a, Y, W))$

$\&$
 $(\forall X W Y. \text{program-not-halts2-halts2-outputs}(X::'a, Y, W) \longrightarrow \text{program-not-halts2}(Y::'a, Y))$
 $\&$
 $(\forall X Y W. \text{program-not-halts2-halts2-outputs}(X::'a, Y, W) \longrightarrow \text{halts2-outputs}(X::'a, Y, W))$
 $\&$
 $(\forall X Y W. \text{program-not-halts2}(Y::'a, Y) \& \text{halts2-outputs}(X::'a, Y, W) \longrightarrow$
 $\text{program-not-halts2-halts2-outputs}(X::'a, Y, W)) \&$
 $(\forall X. \text{algorithm-program-decides}(X) \longrightarrow \text{program-program-decides}(c1)) \&$
 $(\forall W Y Z. \text{program-program-decides}(W) \longrightarrow \text{program-halts2-halts3-outputs}(W::'a, Y, Z, \text{good}))$
 $\&$
 $(\forall W Y Z. \text{program-program-decides}(W) \longrightarrow \text{program-not-halts2-halts3-outputs}(W::'a, Y, Z, \text{bad}))$
 $\&$
 $(\forall W. \text{program}(W) \& \text{program-halts2-halts3-outputs}(W::'a, f3(W), f3(W), \text{good})$
 $\& \text{program-not-halts2-halts3-outputs}(W::'a, f3(W), f3(W), \text{bad}) \longrightarrow \text{program}(c2))$
 $\&$
 $(\forall W Y. \text{program}(W) \& \text{program-halts2-halts3-outputs}(W::'a, f3(W), f3(W), \text{good})$
 $\& \text{program-not-halts2-halts3-outputs}(W::'a, f3(W), f3(W), \text{bad}) \longrightarrow \text{program-halts2-halts2-outputs}(c2::'a, Y, g$
 $\&$
 $(\forall W Y. \text{program}(W) \& \text{program-halts2-halts3-outputs}(W::'a, f3(W), f3(W), \text{good})$
 $\& \text{program-not-halts2-halts3-outputs}(W::'a, f3(W), f3(W), \text{bad}) \longrightarrow \text{program-not-halts2-halts2-outputs}(c2::'a,$
 $\&$
 $(\forall V. \text{program}(V) \& \text{program-halts2-halts2-outputs}(V::'a, f4(V), \text{good}) \& \text{program-not-halts2-halts2-outputs}(V$
 $\longrightarrow \text{program}(c3)) \&$
 $(\forall V Y. \text{program}(V) \& \text{program-halts2-halts2-outputs}(V::'a, f4(V), \text{good}) \& \text{program-not-halts2-halts2-outputs}$
 $\& \text{program-halts2}(Y::'a, Y) \longrightarrow \text{halts2}(c3::'a, Y)) \&$
 $(\forall V Y. \text{program}(V) \& \text{program-halts2-halts2-outputs}(V::'a, f4(V), \text{good}) \& \text{program-not-halts2-halts2-outputs}$
 $\longrightarrow \text{program-not-halts2-halts2-outputs}(c3::'a, Y, \text{bad})) \&$
 $(\text{algorithm-program-decides}(c4)) \longrightarrow \text{False}$
by meson

lemma COM004-1:

$\text{EQU001-0-ax equal} \&$
 $(\forall C D P Q X Y. \text{failure-node}(X::'a, \text{or}(C::'a, P)) \& \text{failure-node}(Y::'a, \text{or}(D::'a, Q)))$
 $\& \text{contradictory}(P::'a, Q) \& \text{siblings}(X::'a, Y) \longrightarrow \text{failure-node}(\text{parent-of}(X::'a, Y), \text{or}(C::'a, D)))$
 $\&$
 $(\forall X. \text{contradictory}(\text{negate}(X), X)) \&$
 $(\forall X. \text{contradictory}(X::'a, \text{negate}(X))) \&$
 $(\forall X. \text{siblings}(\text{left-child-of}(X), \text{right-child-of}(X))) \&$
 $(\forall D E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{left-child-of}(D), \text{left-child-of}(E))) \&$
 $(\forall F' G. \text{equal}(F'::'a, G) \longrightarrow \text{equal}(\text{negate}(F'), \text{negate}(G))) \&$
 $(\forall H I' J. \text{equal}(H::'a, I') \longrightarrow \text{equal}(\text{or}(H::'a, J), \text{or}(I'::'a, J))) \&$
 $(\forall K' M L. \text{equal}(K'::'a, L) \longrightarrow \text{equal}(\text{or}(M::'a, K'), \text{or}(M::'a, L))) \&$
 $(\forall N O' P. \text{equal}(N::'a, O') \longrightarrow \text{equal}(\text{parent-of}(N::'a, P), \text{parent-of}(O'::'a, P)))$
 $\&$
 $(\forall Q S' R. \text{equal}(Q::'a, R) \longrightarrow \text{equal}(\text{parent-of}(S'::'a, Q), \text{parent-of}(S'::'a, R)))$
 $\&$
 $(\forall T' U. \text{equal}(T'::'a, U) \longrightarrow \text{equal}(\text{right-child-of}(T'), \text{right-child-of}(U))) \&$
 $(\forall V W X. \text{equal}(V::'a, W) \& \text{contradictory}(V::'a, X) \longrightarrow \text{contradictory}(W::'a, X))$

&
 ($\forall Y A1 Z. \text{equal}(Y::'a, Z) \ \& \ \text{contradictory}(A1::'a, Y) \longrightarrow \text{contradictory}(A1::'a, Z)$)
 &
 ($\forall B1 C1 D1. \text{equal}(B1::'a, C1) \ \& \ \text{failure-node}(B1::'a, D1) \longrightarrow \text{failure-node}(C1::'a, D1)$)
 &
 ($\forall E1 G1 F1. \text{equal}(E1::'a, F1) \ \& \ \text{failure-node}(G1::'a, E1) \longrightarrow \text{failure-node}(G1::'a, F1)$)
 &
 ($\forall H1 I1 J1. \text{equal}(H1::'a, I1) \ \& \ \text{siblings}(H1::'a, J1) \longrightarrow \text{siblings}(I1::'a, J1)$) &
 ($\forall K1 M1 L1. \text{equal}(K1::'a, L1) \ \& \ \text{siblings}(M1::'a, K1) \longrightarrow \text{siblings}(M1::'a, L1)$)
 &
 ($\text{failure-node}(n\text{-left}::'a, \text{or}(\text{EMPTY}::'a, \text{atom}))$) &
 ($\text{failure-node}(n\text{-right}::'a, \text{or}(\text{EMPTY}::'a, \text{negate}(\text{atom})))$) &
 ($\text{equal}(n\text{-left}::'a, \text{left-child-of}(n))$) &
 ($\text{equal}(n\text{-right}::'a, \text{right-child-of}(n))$) &
 ($\forall Z. \sim \text{failure-node}(Z::'a, \text{or}(\text{EMPTY}::'a, \text{EMPTY})) \longrightarrow \text{False}$)
oops

abbreviation *GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2*

lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal

between \equiv

($\forall X Y. \text{between}(X::'a, Y, X) \longrightarrow \text{equal}(X::'a, Y)$) &
 ($\forall V X Y Z. \text{between}(X::'a, Y, V) \ \& \ \text{between}(Y::'a, Z, V) \longrightarrow \text{between}(X::'a, Y, Z)$)

&

($\forall Y X V Z. \text{between}(X::'a, Y, Z) \ \& \ \text{between}(X::'a, Y, V) \longrightarrow \text{equal}(X::'a, Y) \mid$
 $\text{between}(X::'a, Z, V) \mid \text{between}(X::'a, V, Z)$) &

($\forall Y X. \text{equidistant}(X::'a, Y, Y, X)$) &

($\forall Z X Y. \text{equidistant}(X::'a, Y, Z, Z) \longrightarrow \text{equal}(X::'a, Y)$) &

($\forall X Y Z V V2 W. \text{equidistant}(X::'a, Y, Z, V) \ \& \ \text{equidistant}(X::'a, Y, V2, W)$
 $\longrightarrow \text{equidistant}(Z::'a, V, V2, W)$) &

($\forall W X Z V Y. \text{between}(X::'a, W, V) \ \& \ \text{between}(Y::'a, V, Z) \longrightarrow \text{between}(X::'a, \text{outer-pasch}(W::'a, X, Y, Z, V),$
 $Y)$) &

&

($\forall W X Y Z V. \text{between}(X::'a, W, V) \ \& \ \text{between}(Y::'a, V, Z) \longrightarrow \text{between}(Z::'a, W, \text{outer-pasch}(W::'a, X, Y, Z,$
 $V))$) &

&

($\forall W X Y Z V. \text{between}(X::'a, V, W) \ \& \ \text{between}(Y::'a, V, Z) \longrightarrow \text{equal}(X::'a, V)$
 $\mid \text{between}(X::'a, Z, \text{euclid1}(W::'a, X, Y, Z, V))$) &

($\forall W X Y Z V. \text{between}(X::'a, V, W) \ \& \ \text{between}(Y::'a, V, Z) \longrightarrow \text{equal}(X::'a, V)$
 $\mid \text{between}(X::'a, Y, \text{euclid2}(W::'a, X, Y, Z, V))$) &

($\forall W X Y Z V. \text{between}(X::'a, V, W) \ \& \ \text{between}(Y::'a, V, Z) \longrightarrow \text{equal}(X::'a, V)$
 $\mid \text{between}(\text{euclid1}(W::'a, X, Y, Z, V), W, \text{euclid2}(W::'a, X, Y, Z, V))$) &

($\forall X1 Y1 X Y Z V Z1 V1. \text{equidistant}(X::'a, Y, X1, Y1) \ \& \ \text{equidistant}(Y::'a, Z, Y1, Z1)$)

& $\text{equidistant}(X::'a, V, X1, V1) \ \& \ \text{equidistant}(Y::'a, V, Y1, V1) \ \& \ \text{between}(X::'a, Y, Z)$

& $\text{between}(X1::'a, Y1, Z1) \longrightarrow \text{equal}(X::'a, Y) \mid \text{equidistant}(Z::'a, V, Z1, V1)$) &

($\forall X Y W V. \text{between}(X::'a, Y, \text{extension}(X::'a, Y, W, V))$) &

($\forall X Y W V. \text{equidistant}(Y::'a, \text{extension}(X::'a, Y, W, V), W, V)$) &

($\sim \text{between}(\text{lower-dimension-point-1}::'a, \text{lower-dimension-point-2}, \text{lower-dimension-point-3})$)

&

($\sim \text{between}(\text{lower-dimension-point-2}::'a, \text{lower-dimension-point-3}, \text{lower-dimension-point-1})$)

&

($\sim \text{between}(\text{lower-dimension-point-3}::'a, \text{lower-dimension-point-1}, \text{lower-dimension-point-2})$)

$\&$
 $(\forall Z X Y W V. \text{equidistant}(X::'a, W, X, V) \& \text{equidistant}(Y::'a, W, Y, V) \& \text{equidistant}(Z::'a, W, Z, V) \longrightarrow \text{between}(X::'a, Y, Z) \mid \text{between}(Y::'a, Z, X) \mid \text{between}(Z::'a, X, Y) \mid \text{equal}(W::'a, V)) \&$
 $(\forall X Y Z X1 Z1 V. \text{equidistant}(V::'a, X, V, X1) \& \text{equidistant}(V::'a, Z, V, Z1) \& \text{between}(V::'a, X, Z) \& \text{between}(X::'a, Y, Z) \longrightarrow \text{equidistant}(V::'a, Y, Z, \text{continuous}(X::'a, Y, Z, X1, Z1, V))) \&$
 $(\forall X Y Z X1 V Z1. \text{equidistant}(V::'a, X, V, X1) \& \text{equidistant}(V::'a, Z, V, Z1) \& \text{between}(V::'a, X, Z) \& \text{between}(X::'a, Y, Z) \longrightarrow \text{between}(X1::'a, \text{continuous}(X::'a, Y, Z, X1, Z1, V), Z1))$

abbreviation *GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant*
 $\text{between equal} \equiv$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{between}(X::'a, W, Z) \longrightarrow \text{between}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{between}(W::'a, X, Z) \longrightarrow \text{between}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{between}(W::'a, Z, X) \longrightarrow \text{between}(W::'a, Z, Y))$
 $\&$
 $(\forall X Y V W Z. \text{equal}(X::'a, Y) \& \text{equidistant}(X::'a, V, W, Z) \longrightarrow \text{equidistant}(Y::'a, V, W, Z)) \&$
 $(\forall X V Y W Z. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, X, W, Z) \longrightarrow \text{equidistant}(V::'a, Y, W, Z)) \&$
 $(\forall X V W Y Z. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, W, X, Z) \longrightarrow \text{equidistant}(V::'a, W, Y, Z)) \&$
 $(\forall X V W Z Y. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, W, Z, X) \longrightarrow \text{equidistant}(V::'a, W, Z, Y)) \&$
 $(\forall X Y V1 V2 V3 V4. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{outer-pasch}(X::'a, V1, V2, V3, V4), \text{outer-pasch}(Y::'a, V1, V2, V3, V4))) \&$
 $(\forall X V1 Y V2 V3 V4. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{outer-pasch}(V1::'a, X, V2, V3, V4), \text{outer-pasch}(V1::'a, Y, V2, V3, V4))) \&$
 $(\forall X V1 V2 Y V3 V4. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{outer-pasch}(V1::'a, V2, X, V3, V4), \text{outer-pasch}(V1::'a, V2, Y, V3, V4))) \&$
 $(\forall X V1 V2 V3 Y V4. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{outer-pasch}(V1::'a, V2, V3, X, V4), \text{outer-pasch}(V1::'a, V2, V3, Y, V4))) \&$
 $(\forall X V1 V2 V3 V4 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{outer-pasch}(V1::'a, V2, V3, V4, X), \text{outer-pasch}(V1::'a, V2, V3, V4, Y))) \&$
 $(\forall A B C D E F'. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{euclid1}(A::'a, C, D, E, F'), \text{euclid1}(B::'a, C, D, E, F')))$
 $\&$
 $(\forall G I' H J K' L. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{euclid1}(I'::'a, G, J, K', L), \text{euclid1}(I'::'a, H, J, K', L)))$
 $\&$
 $(\forall M O' P N Q R. \text{equal}(M::'a, N) \longrightarrow \text{equal}(\text{euclid1}(O'::'a, P, M, Q, R), \text{euclid1}(O'::'a, P, N, Q, R)))$
 $\&$
 $(\forall S' U V W T' X. \text{equal}(S'::'a, T') \longrightarrow \text{equal}(\text{euclid1}(U::'a, V, W, S', X), \text{euclid1}(U::'a, V, W, T', X)))$
 $\&$
 $(\forall Y A1 B1 C1 D1 Z. \text{equal}(Y::'a, Z) \longrightarrow \text{equal}(\text{euclid1}(A1::'a, B1, C1, D1, Y), \text{euclid1}(A1::'a, B1, C1, D1, Z)))$
 $\&$
 $(\forall E1 F1 G1 H1 I1 J1. \text{equal}(E1::'a, F1) \longrightarrow \text{equal}(\text{euclid2}(E1::'a, G1, H1, I1, J1), \text{euclid2}(F1::'a, G1, H1, I1, J1)))$
 $\&$
 $(\forall K1 M1 L1 N1 O1 P1. \text{equal}(K1::'a, L1) \longrightarrow \text{equal}(\text{euclid2}(M1::'a, K1, N1, O1, P1), \text{euclid2}(M1::'a, L1, N1, O1, P1)))$

$\&$
 $(\forall Q1\ S1\ T1\ R1\ U1\ V1. \text{equal}(Q1::'a,R1) \dashrightarrow \text{equal}(\text{euclid2}(S1::'a,T1,Q1,U1,V1),\text{euclid2}(S1::'a,T1,R1,U1,V1)))$
 $\&$
 $(\forall W1\ Y1\ Z1\ A2\ X1\ B2. \text{equal}(W1::'a,X1) \dashrightarrow \text{equal}(\text{euclid2}(Y1::'a,Z1,A2,W1,B2),\text{euclid2}(Y1::'a,Z1,A2,B2,W1)))$
 $\&$
 $(\forall C2\ E2\ F2\ G2\ H2\ D2. \text{equal}(C2::'a,D2) \dashrightarrow \text{equal}(\text{euclid2}(E2::'a,F2,G2,H2,C2),\text{euclid2}(E2::'a,F2,G2,H2,D2)))$
 $\&$
 $(\forall X\ Y\ V1\ V2\ V3. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{extension}(X::'a,V1,V2,V3),\text{extension}(Y::'a,V1,V2,V3)))$
 $\&$
 $(\forall X\ V1\ Y\ V2\ V3. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{extension}(V1::'a,X,V2,V3),\text{extension}(V1::'a,Y,V2,V3)))$
 $\&$
 $(\forall X\ V1\ V2\ Y\ V3. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{extension}(V1::'a,V2,X,V3),\text{extension}(V1::'a,V2,Y,V3)))$
 $\&$
 $(\forall X\ V1\ V2\ V3\ Y. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{extension}(V1::'a,V2,V3,X),\text{extension}(V1::'a,V2,V3,Y)))$
 $\&$
 $(\forall X\ Y\ V1\ V2\ V3\ V4\ V5. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{continuous}(X::'a,V1,V2,V3,V4,V5),\text{continuous}(Y::'a,V1,V2,V3,V4,V5)))$
 $\&$
 $(\forall X\ V1\ Y\ V2\ V3\ V4\ V5. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{continuous}(V1::'a,X,V2,V3,V4,V5),\text{continuous}(V1::'a,Y,V2,V3,V4,V5)))$
 $\&$
 $(\forall X\ V1\ V2\ Y\ V3\ V4\ V5. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{continuous}(V1::'a,V2,X,V3,V4,V5),\text{continuous}(V1::'a,V2,Y,V3,V4,V5)))$
 $\&$
 $(\forall X\ V1\ V2\ V3\ Y\ V4\ V5. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{continuous}(V1::'a,V2,V3,X,V4,V5),\text{continuous}(V1::'a,V2,V3,Y,V4,V5)))$
 $\&$
 $(\forall X\ V1\ V2\ V3\ V4\ Y\ V5. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{continuous}(V1::'a,V2,V3,V4,X,V5),\text{continuous}(V1::'a,V2,V3,V4,Y,V5)))$
 $\&$
 $(\forall X\ V1\ V2\ V3\ V4\ V5\ Y. \text{equal}(X::'a,Y) \dashrightarrow \text{equal}(\text{continuous}(V1::'a,V2,V3,V4,V5,X),\text{continuous}(V1::'a,V2,V3,V4,V5,Y)))$

lemma *GEO003-1:*

EQU001-0-ax equal &
GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2
lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal
between &
GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between
equal &
 $(\sim \text{between}(a::'a,b,b)) \dashrightarrow \text{False}$
by *meson*

abbreviation *GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3*

lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant \equiv
 $(\forall Y\ X. \text{equidistant}(X::'a,Y,Y,X)) \&$
 $(\forall X\ Y\ Z\ V\ V2\ W. \text{equidistant}(X::'a,Y,Z,V) \& \text{equidistant}(X::'a,Y,V2,W) \dashrightarrow \text{equidistant}(Z::'a,V,V2,W)) \&$
 $(\forall Z\ X\ Y. \text{equidistant}(X::'a,Y,Z,Z) \dashrightarrow \text{equal}(X::'a,Y)) \&$
 $(\forall X\ Y\ W\ V. \text{between}(X::'a,Y,\text{extension}(X::'a,Y,W,V))) \&$
 $(\forall X\ Y\ W\ V. \text{equidistant}(Y::'a,\text{extension}(X::'a,Y,W,V),W,V)) \&$
 $(\forall X1\ Y1\ X\ Y\ Z\ V\ Z1\ V1. \text{equidistant}(X::'a,Y,X1,Y1) \& \text{equidistant}(Y::'a,Z,Y1,Z1))$

$\& \text{equidistant}(X::'a, V, X1, V1) \& \text{equidistant}(Y::'a, V, Y1, V1) \& \text{between}(X::'a, Y, Z)$
 $\& \text{between}(X1::'a, Y1, Z1) \dashrightarrow \text{equal}(X::'a, Y) \mid \text{equidistant}(Z::'a, V, Z1, V1)) \&$
 $(\forall X Y. \text{between}(X::'a, Y, X) \dashrightarrow \text{equal}(X::'a, Y)) \&$
 $(\forall U V W X Y. \text{between}(U::'a, V, W) \& \text{between}(Y::'a, X, W) \dashrightarrow \text{between}(V::'a, \text{inner-pasch}(U::'a, V, W, X,$
 $\&$
 $(\forall V W X Y U. \text{between}(U::'a, V, W) \& \text{between}(Y::'a, X, W) \dashrightarrow \text{between}(X::'a, \text{inner-pasch}(U::'a, V, W, X,$
 $\&$
 $(\sim \text{between}(\text{lower-dimension-point-1}::'a, \text{lower-dimension-point-2}, \text{lower-dimension-point-3}))$
 $\&$
 $(\sim \text{between}(\text{lower-dimension-point-2}::'a, \text{lower-dimension-point-3}, \text{lower-dimension-point-1}))$
 $\&$
 $(\sim \text{between}(\text{lower-dimension-point-3}::'a, \text{lower-dimension-point-1}, \text{lower-dimension-point-2}))$
 $\&$
 $(\forall Z X Y W V. \text{equidistant}(X::'a, W, X, V) \& \text{equidistant}(Y::'a, W, Y, V) \& \text{equidis-}$
 $\text{tant}(Z::'a, W, Z, V) \dashrightarrow \text{between}(X::'a, Y, Z) \mid \text{between}(Y::'a, Z, X) \mid \text{between}(Z::'a, X, Y)$
 $\mid \text{equal}(W::'a, V)) \&$
 $(\forall U V W X Y. \text{between}(U::'a, W, Y) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equal}(U::'a, W)$
 $\mid \text{between}(U::'a, V, \text{euclid1}(U::'a, V, W, X, Y))) \&$
 $(\forall U V W X Y. \text{between}(U::'a, W, Y) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equal}(U::'a, W)$
 $\mid \text{between}(U::'a, X, \text{euclid2}(U::'a, V, W, X, Y))) \&$
 $(\forall U V W X Y. \text{between}(U::'a, W, Y) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equal}(U::'a, W)$
 $\mid \text{between}(\text{euclid1}(U::'a, V, W, X, Y), Y, \text{euclid2}(U::'a, V, W, X, Y))) \&$
 $(\forall U V V1 W X X1. \text{equidistant}(U::'a, V, U, V1) \& \text{equidistant}(U::'a, X, U, X1) \&$
 $\text{between}(U::'a, V, X) \& \text{between}(V::'a, W, X) \dashrightarrow \text{between}(V1::'a, \text{continuous}(U::'a, V, V1, W, X, X1), X1))$
 $\&$
 $(\forall U V V1 W X X1. \text{equidistant}(U::'a, V, U, V1) \& \text{equidistant}(U::'a, X, U, X1) \&$
 $\text{between}(U::'a, V, X) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equidistant}(U::'a, W, U, \text{continuous}(U::'a, V, V1, W, X, X1)))$
 $\&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{between}(X::'a, W, Z) \dashrightarrow \text{between}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{between}(W::'a, X, Z) \dashrightarrow \text{between}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{between}(W::'a, Z, X) \dashrightarrow \text{between}(W::'a, Z, Y))$
 $\&$
 $(\forall X Y V W Z. \text{equal}(X::'a, Y) \& \text{equidistant}(X::'a, V, W, Z) \dashrightarrow \text{equidis-}$
 $\text{tant}(Y::'a, V, W, Z)) \&$
 $(\forall X V Y W Z. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, X, W, Z) \dashrightarrow \text{equidis-}$
 $\text{tant}(V::'a, Y, W, Z)) \&$
 $(\forall X V W Y Z. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, W, X, Z) \dashrightarrow \text{equidis-}$
 $\text{tant}(V::'a, W, Y, Z)) \&$
 $(\forall X V W Z Y. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, W, Z, X) \dashrightarrow \text{equidis-}$
 $\text{tant}(V::'a, W, Z, Y)) \&$
 $(\forall X Y V1 V2 V3 V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(X::'a, V1, V2, V3, V4), \text{inner-pasch}(Y::'a, V1, V$
 $\&$
 $(\forall X V1 Y V2 V3 V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(V1::'a, X, V2, V3, V4), \text{inner-pasch}(V1::'a, Y, V$
 $\&$
 $(\forall X V1 V2 Y V3 V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(V1::'a, V2, X, V3, V4), \text{inner-pasch}(V1::'a, V2, Y$
 $\&$
 $(\forall X V1 V2 V3 Y V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(V1::'a, V2, V3, X, V4), \text{inner-pasch}(V1::'a, V2, Y$

$\&$
 $(\forall X V1 V2 V3 V4 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{inner-pasch}(V1::'a, V2, V3, V4, X), \text{inner-pasch}(V1::'a, V2, V3, V4, Y)))$
 $\&$
 $(\forall A B C D E F'. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{euclid1}(A::'a, C, D, E, F'), \text{euclid1}(B::'a, C, D, E, F')))$
 $\&$
 $(\forall G I' H J K' L. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{euclid1}(I'::'a, G, J, K', L), \text{euclid1}(I'::'a, H, J, K', L)))$
 $\&$
 $(\forall M O' P N Q R. \text{equal}(M::'a, N) \longrightarrow \text{equal}(\text{euclid1}(O'::'a, P, M, Q, R), \text{euclid1}(O'::'a, P, N, Q, R)))$
 $\&$
 $(\forall S' U V W T' X. \text{equal}(S'::'a, T') \longrightarrow \text{equal}(\text{euclid1}(U::'a, V, W, S', X), \text{euclid1}(U::'a, V, W, T', X)))$
 $\&$
 $(\forall Y A1 B1 C1 D1 Z. \text{equal}(Y::'a, Z) \longrightarrow \text{equal}(\text{euclid1}(A1::'a, B1, C1, D1, Y), \text{euclid1}(A1::'a, B1, C1, D1, Z)))$
 $\&$
 $(\forall E1 F1 G1 H1 I1 J1. \text{equal}(E1::'a, F1) \longrightarrow \text{equal}(\text{euclid2}(E1::'a, G1, H1, I1, J1), \text{euclid2}(F1::'a, G1, H1, I1, J1)))$
 $\&$
 $(\forall K1 M1 L1 N1 O1 P1. \text{equal}(K1::'a, L1) \longrightarrow \text{equal}(\text{euclid2}(M1::'a, K1, N1, O1, P1), \text{euclid2}(M1::'a, L1, N1, O1, P1)))$
 $\&$
 $(\forall Q1 S1 T1 R1 U1 V1. \text{equal}(Q1::'a, R1) \longrightarrow \text{equal}(\text{euclid2}(S1::'a, T1, Q1, U1, V1), \text{euclid2}(S1::'a, T1, R1, U1, V1)))$
 $\&$
 $(\forall W1 Y1 Z1 A2 X1 B2. \text{equal}(W1::'a, X1) \longrightarrow \text{equal}(\text{euclid2}(Y1::'a, Z1, A2, W1, B2), \text{euclid2}(Y1::'a, Z1, A2, W1, B2)))$
 $\&$
 $(\forall C2 E2 F2 G2 H2 D2. \text{equal}(C2::'a, D2) \longrightarrow \text{equal}(\text{euclid2}(E2::'a, F2, G2, H2, C2), \text{euclid2}(E2::'a, F2, G2, H2, D2)))$
 $\&$
 $(\forall X Y V1 V2 V3. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(X::'a, V1, V2, V3), \text{extension}(Y::'a, V1, V2, V3)))$
 $\&$
 $(\forall X V1 Y V2 V3. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(V1::'a, X, V2, V3), \text{extension}(V1::'a, Y, V2, V3)))$
 $\&$
 $(\forall X V1 V2 Y V3. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(V1::'a, V2, X, V3), \text{extension}(V1::'a, V2, Y, V3)))$
 $\&$
 $(\forall X V1 V2 V3 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(V1::'a, V2, V3, X), \text{extension}(V1::'a, V2, V3, Y)))$
 $\&$
 $(\forall X Y V1 V2 V3 V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(X::'a, V1, V2, V3, V4, V5), \text{continuous}(Y::'a, V1, V2, V3, V4, V5)))$
 $\&$
 $(\forall X V1 Y V2 V3 V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, X, V2, V3, V4, V5), \text{continuous}(V1::'a, Y, V2, V3, V4, V5)))$
 $\&$
 $(\forall X V1 V2 Y V3 V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, X, V3, V4, V5), \text{continuous}(V1::'a, V2, Y, V3, V4, V5)))$
 $\&$
 $(\forall X V1 V2 V3 Y V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, V3, X, V4, V5), \text{continuous}(V1::'a, V2, V3, Y, V4, V5)))$
 $\&$
 $(\forall X V1 V2 V3 V4 Y V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, V3, V4, X, V5), \text{continuous}(V1::'a, V2, V3, V4, Y, V5)))$
 $\&$
 $(\forall X V1 V2 V3 V4 V5 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, V3, V4, V5, X), \text{continuous}(V1::'a, V2, V3, V4, V5, Y)))$

lemma *GEO017-2:*

EQU001-0-ax equal &

GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3

lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension

between equal equidistant &

$(\text{equidistant}(u::'a, v, w, x)) \ \&$
 $(\sim \text{equidistant}(u::'a, v, x, w)) \longrightarrow \text{False}$
oops

lemma *GEO027-3:*

EQU001-0-ax equal &
GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3
lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant &
 $(\forall U \ V. \text{equal}(\text{reflection}(U::'a, V), \text{extension}(U::'a, V, U, V))) \ \&$
 $(\forall X \ Y \ Z. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{reflection}(X::'a, Z), \text{reflection}(Y::'a, Z))) \ \&$
 $(\forall A1 \ C1 \ B1. \text{equal}(A1::'a, B1) \longrightarrow \text{equal}(\text{reflection}(C1::'a, A1), \text{reflection}(C1::'a, B1)))$
 $\&$
 $(\forall U \ V. \text{equidistant}(U::'a, V, U, V)) \ \&$
 $(\forall W \ X \ U \ V. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(W::'a, X, U, V)) \ \&$
 $(\forall V \ U \ W \ X. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(V::'a, U, W, X)) \ \&$
 $(\forall U \ V \ X \ W. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(U::'a, V, X, W)) \ \&$
 $(\forall V \ U \ X \ W. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(V::'a, U, X, W)) \ \&$
 $(\forall W \ X \ V \ U. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(W::'a, X, V, U)) \ \&$
 $(\forall X \ W \ U \ V. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(X::'a, W, U, V)) \ \&$
 $(\forall X \ W \ V \ U. \text{equidistant}(U::'a, V, W, X) \longrightarrow \text{equidistant}(X::'a, W, V, U)) \ \&$
 $(\forall W \ X \ U \ V \ Y \ Z. \text{equidistant}(U::'a, V, W, X) \ \& \ \text{equidistant}(W::'a, X, Y, Z) \longrightarrow$
 $\text{equidistant}(U::'a, V, Y, Z)) \ \&$
 $(\forall U \ V \ W. \text{equal}(V::'a, \text{extension}(U::'a, V, W, W))) \ \&$
 $(\forall W \ X \ U \ V \ Y. \text{equal}(Y::'a, \text{extension}(U::'a, V, W, X)) \longrightarrow \text{between}(U::'a, V, Y))$
 $\&$
 $(\forall U \ V. \text{between}(U::'a, V, \text{reflection}(U::'a, V))) \ \&$
 $(\forall U \ V. \text{equidistant}(V::'a, \text{reflection}(U::'a, V), U, V)) \ \&$
 $(\forall U \ V. \text{equal}(U::'a, V) \longrightarrow \text{equal}(V::'a, \text{reflection}(U::'a, V))) \ \&$
 $(\forall U. \text{equal}(U::'a, \text{reflection}(U::'a, U))) \ \&$
 $(\forall U \ V. \text{equal}(V::'a, \text{reflection}(U::'a, V)) \longrightarrow \text{equal}(U::'a, V)) \ \&$
 $(\forall U \ V. \text{equidistant}(U::'a, U, V, V)) \ \&$
 $(\forall V \ V1 \ U \ W \ U1 \ W1. \text{equidistant}(U::'a, V, U1, V1) \ \& \ \text{equidistant}(V::'a, W, V1, W1)$
 $\& \ \text{between}(U::'a, V, W) \ \& \ \text{between}(U1::'a, V1, W1) \longrightarrow \text{equidistant}(U::'a, W, U1, W1))$
 $\&$
 $(\forall U \ V \ W \ X. \text{between}(U::'a, V, W) \ \& \ \text{between}(U::'a, V, X) \ \& \ \text{equidistant}(V::'a, W, V, X)$
 $\longrightarrow \text{equal}(U::'a, V) \mid \text{equal}(W::'a, X)) \ \&$
 $(\text{between}(u::'a, v, w)) \ \&$
 $(\sim \text{equal}(u::'a, v)) \ \&$
 $(\sim \text{equal}(w::'a, \text{extension}(u::'a, v, v, w))) \longrightarrow \text{False}$
oops

lemma *GEO058-2:*

EQU001-0-ax equal &
GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3
lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
between equal equidistant &

$(\forall U V. \text{equal}(\text{reflection}(U::'a, V), \text{extension}(U::'a, V, U, V))) \ \&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{reflection}(X::'a, Z), \text{reflection}(Y::'a, Z))) \ \&$
 $(\forall A1 C1 B1. \text{equal}(A1::'a, B1) \longrightarrow \text{equal}(\text{reflection}(C1::'a, A1), \text{reflection}(C1::'a, B1)))$
 $\&$
 $(\text{equal}(v::'a, \text{reflection}(u::'a, v))) \ \&$
 $(\sim \text{equal}(u::'a, v) \longrightarrow \text{False})$
oops

lemma *GEO079-1:*

$(\forall U V W X Y Z. \text{right-angle}(U::'a, V, W) \ \& \ \text{right-angle}(X::'a, Y, Z) \longrightarrow \text{eq}(U::'a, V, W, X, Y, Z))$
 $\&$
 $(\forall U V W X Y Z. \text{CONGRUENT}(U::'a, V, W, X, Y, Z) \longrightarrow \text{eq}(U::'a, V, W, X, Y, Z))$
 $\&$
 $(\forall V W U X. \text{trapezoid}(U::'a, V, W, X) \longrightarrow \text{parallel}(V::'a, W, U, X)) \ \&$
 $(\forall U V X Y. \text{parallel}(U::'a, V, X, Y) \longrightarrow \text{eq}(X::'a, V, U, V, X, Y)) \ \&$
 $(\text{trapezoid}(a::'a, b, c, d)) \ \&$
 $(\sim \text{eq}(a::'a, c, b, c, a, d)) \longrightarrow \text{False}$
by *meson*

abbreviation *GRP003-0-ax equal multiply INVERSE identity product* \equiv

$(\forall X. \text{product}(\text{identity}::'a, X, X)) \ \&$
 $(\forall X. \text{product}(X::'a, \text{identity}, X)) \ \&$
 $(\forall X. \text{product}(\text{INVERSE}(X), X, \text{identity})) \ \&$
 $(\forall X. \text{product}(X::'a, \text{INVERSE}(X), \text{identity})) \ \&$
 $(\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \ \&$
 $(\forall X Y Z W. \text{product}(X::'a, Y, Z) \ \& \ \text{product}(X::'a, Y, W) \longrightarrow \text{equal}(Z::'a, W))$
 $\&$
 $(\forall Y U Z X V W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(U::'a, Z, W)$
 $\longrightarrow \text{product}(X::'a, V, W)) \ \&$
 $(\forall Y X V U Z W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(X::'a, V, W)$
 $\longrightarrow \text{product}(U::'a, Z, W))$

abbreviation *GRP003-0-eq product multiply INVERSE equal* \equiv

$(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{INVERSE}(X), \text{INVERSE}(Y))) \ \&$
 $(\forall X Y W. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{multiply}(X::'a, W), \text{multiply}(Y::'a, W)))$
 $\&$
 $(\forall X W Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{multiply}(W::'a, X), \text{multiply}(W::'a, Y)))$
 $\&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \ \& \ \text{product}(X::'a, W, Z) \longrightarrow \text{product}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, X, Z) \longrightarrow \text{product}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, Z, X) \longrightarrow \text{product}(W::'a, Z, Y))$

lemma *GRP001-1:*

EQU001-0-ax equal $\&$
GRP003-0-ax equal multiply INVERSE identity product $\&$

GRP003-0-eq product multiply INVERSE equal &
($\forall X. \text{product}(X::'a, X, \text{identity})$) &
($\text{product}(a::'a, b, c)$) &
($\sim \text{product}(b::'a, a, c)$) \longrightarrow False
oops

lemma *GRP008-1:*

EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
($\forall A B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(h(A), h(B))$) &
($\forall C D. \text{equal}(C::'a, D) \longrightarrow \text{equal}(j(C), j(D))$) &
($\forall A B. \text{equal}(A::'a, B) \& q(A) \longrightarrow q(B)$) &
($\forall B A C. q(A) \& \text{product}(A::'a, B, C) \longrightarrow \text{product}(B::'a, A, C)$) &
($\forall A. \text{product}(j(A), A, h(A)) \mid \text{product}(A::'a, j(A), h(A)) \mid q(A)$) &
($\forall A. \text{product}(j(A), A, h(A)) \& \text{product}(A::'a, j(A), h(A)) \longrightarrow q(A)$) &
($\sim q(\text{identity})$) \longrightarrow False
by meson

lemma *GRP013-1:*

EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
($\forall A. \text{product}(A::'a, A, \text{identity})$) &
($\text{product}(a::'a, b, c)$) &
($\text{product}(\text{INVERSE}(a), \text{INVERSE}(b), d)$) &
($\forall A C B. \text{product}(\text{INVERSE}(A), \text{INVERSE}(B), C) \longrightarrow \text{product}(A::'a, C, B)$) &
($\sim \text{product}(c::'a, d, \text{identity})$) \longrightarrow False
oops

lemma *GRP037-3:*

EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
($\forall A B C. \text{subgroup-member}(A) \& \text{subgroup-member}(B) \& \text{product}(A::'a, \text{INVERSE}(B), C)$)
 $\longrightarrow \text{subgroup-member}(C)$) &
($\forall A B. \text{equal}(A::'a, B) \& \text{subgroup-member}(A) \longrightarrow \text{subgroup-member}(B)$) &
($\forall A. \text{subgroup-member}(A) \longrightarrow \text{product}(G\text{identity}::'a, A, A)$) &
($\forall A. \text{subgroup-member}(A) \longrightarrow \text{product}(A::'a, G\text{identity}, A)$) &
($\forall A. \text{subgroup-member}(A) \longrightarrow \text{product}(A::'a, G\text{inverse}(A), G\text{identity})$) &
($\forall A. \text{subgroup-member}(A) \longrightarrow \text{product}(G\text{inverse}(A), A, G\text{identity})$) &
($\forall A. \text{subgroup-member}(A) \longrightarrow \text{subgroup-member}(G\text{inverse}(A))$) &
($\forall A B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(G\text{inverse}(A), G\text{inverse}(B))$) &
($\forall A C D B. \text{product}(A::'a, B, C) \& \text{product}(A::'a, D, C) \longrightarrow \text{equal}(D::'a, B)$) &
($\forall B C D A. \text{product}(A::'a, B, C) \& \text{product}(D::'a, B, C) \longrightarrow \text{equal}(D::'a, A)$) &
($\text{subgroup-member}(a)$) &

$(\text{subgroup-member}(\text{Gidentity})) \ \&$
 $(\sim \text{equal}(\text{INVERSE}(a), \text{Ginverse}(a))) \longrightarrow \text{False}$
by meson

lemma GRP031-2:

$(\forall X \ Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \ \&$
 $(\forall X \ Y \ Z \ W. \text{product}(X::'a, Y, Z) \ \& \ \text{product}(X::'a, Y, W) \longrightarrow \text{equal}(Z::'a, W))$
 $\&$
 $(\forall Y \ U \ Z \ X \ V \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(U::'a, Z, W)$
 $\longrightarrow \text{product}(X::'a, V, W)) \ \&$
 $(\forall Y \ X \ V \ U \ Z \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(X::'a, V, W)$
 $\longrightarrow \text{product}(U::'a, Z, W)) \ \&$
 $(\forall A. \text{product}(A::'a, \text{INVERSE}(A), \text{identity})) \ \&$
 $(\forall A. \text{product}(A::'a, \text{identity}, A)) \ \&$
 $(\forall A. \sim \text{product}(A::'a, a, \text{identity})) \longrightarrow \text{False}$
by meson

lemma GRP034-4:

$(\forall X \ Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \ \&$
 $(\forall X. \text{product}(\text{identity}::'a, X, X)) \ \&$
 $(\forall X. \text{product}(X::'a, \text{identity}, X)) \ \&$
 $(\forall X. \text{product}(X::'a, \text{INVERSE}(X), \text{identity})) \ \&$
 $(\forall Y \ U \ Z \ X \ V \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(U::'a, Z, W)$
 $\longrightarrow \text{product}(X::'a, V, W)) \ \&$
 $(\forall Y \ X \ V \ U \ Z \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(X::'a, V, W)$
 $\longrightarrow \text{product}(U::'a, Z, W)) \ \&$
 $(\forall B \ A \ C. \text{subgroup-member}(A) \ \& \ \text{subgroup-member}(B) \ \& \ \text{product}(B::'a, \text{INVERSE}(A), C)$
 $\longrightarrow \text{subgroup-member}(C)) \ \&$
 $(\text{subgroup-member}(a)) \ \&$
 $(\sim \text{subgroup-member}(\text{INVERSE}(a))) \longrightarrow \text{False}$
by meson

lemma GRP047-2:

$(\forall X. \text{product}(\text{identity}::'a, X, X)) \ \&$
 $(\forall X. \text{product}(\text{INVERSE}(X), X, \text{identity})) \ \&$
 $(\forall X \ Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \ \&$
 $(\forall X \ Y \ Z \ W. \text{product}(X::'a, Y, Z) \ \& \ \text{product}(X::'a, Y, W) \longrightarrow \text{equal}(Z::'a, W))$
 $\&$
 $(\forall Y \ U \ Z \ X \ V \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(U::'a, Z, W)$
 $\longrightarrow \text{product}(X::'a, V, W)) \ \&$
 $(\forall Y \ X \ V \ U \ Z \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(X::'a, V, W)$
 $\longrightarrow \text{product}(U::'a, Z, W)) \ \&$
 $(\forall X \ W \ Z \ Y. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, Z, X) \longrightarrow \text{product}(W::'a, Z, Y))$
 $\&$
 $(\text{equal}(a::'a, b)) \ \&$
 $(\sim \text{equal}(\text{multiply}(c::'a, a), \text{multiply}(c::'a, b))) \longrightarrow \text{False}$

by meson

lemma *GRP130-1-002:*

$(\text{group-element}(e-1)) \ \&$
 $(\text{group-element}(e-2)) \ \&$
 $(\sim \text{equal}(e-1::'a, e-2)) \ \&$
 $(\sim \text{equal}(e-2::'a, e-1)) \ \&$
 $(\forall X \ Y. \ \text{group-element}(X) \ \& \ \text{group-element}(Y) \ \longrightarrow \ \text{product}(X::'a, Y, e-1) \mid$
 $\text{product}(X::'a, Y, e-2)) \ \&$
 $(\forall X \ Y \ W \ Z. \ \text{product}(X::'a, Y, W) \ \& \ \text{product}(X::'a, Y, Z) \ \longrightarrow \ \text{equal}(W::'a, Z))$
 $\&$
 $(\forall X \ Y \ W \ Z. \ \text{product}(X::'a, W, Y) \ \& \ \text{product}(X::'a, Z, Y) \ \longrightarrow \ \text{equal}(W::'a, Z))$
 $\&$
 $(\forall Y \ X \ W \ Z. \ \text{product}(W::'a, Y, X) \ \& \ \text{product}(Z::'a, Y, X) \ \longrightarrow \ \text{equal}(W::'a, Z))$
 $\&$
 $(\forall Z1 \ Z2 \ Y \ X. \ \text{product}(X::'a, Y, Z1) \ \& \ \text{product}(X::'a, Z1, Z2) \ \longrightarrow \ \text{product}(Z2::'a, Y, X))$
 $\longrightarrow \ \text{False}$
oops

abbreviation *GRP004-0-ax INVERSE identity multiply equal* \equiv

$(\forall X. \ \text{equal}(\text{multiply}(\text{identity}::'a, X), X)) \ \&$
 $(\forall X. \ \text{equal}(\text{multiply}(\text{INVERSE}(X), X), \text{identity})) \ \&$
 $(\forall X \ Y \ Z. \ \text{equal}(\text{multiply}(\text{multiply}(X::'a, Y), Z), \text{multiply}(X::'a, \text{multiply}(Y::'a, Z))))$
 $\&$
 $(\forall A \ B. \ \text{equal}(A::'a, B) \ \longrightarrow \ \text{equal}(\text{INVERSE}(A), \text{INVERSE}(B))) \ \&$
 $(\forall C \ D \ E. \ \text{equal}(C::'a, D) \ \longrightarrow \ \text{equal}(\text{multiply}(C::'a, E), \text{multiply}(D::'a, E))) \ \&$
 $(\forall F' \ H \ G. \ \text{equal}(F'::'a, G) \ \longrightarrow \ \text{equal}(\text{multiply}(H::'a, F'), \text{multiply}(H::'a, G)))$

abbreviation *GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal* \equiv

$(\forall Y \ X. \ \text{equal}(\text{greatest-lower-bound}(X::'a, Y), \text{greatest-lower-bound}(Y::'a, X))) \ \&$
 $(\forall Y \ X. \ \text{equal}(\text{least-upper-bound}(X::'a, Y), \text{least-upper-bound}(Y::'a, X))) \ \&$
 $(\forall X \ Y \ Z. \ \text{equal}(\text{greatest-lower-bound}(X::'a, \text{greatest-lower-bound}(Y::'a, Z)), \text{greatest-lower-bound}(\text{greatest-lower-bound}(X::'a, Y), Z)))$
 $\&$
 $(\forall X \ Y \ Z. \ \text{equal}(\text{least-upper-bound}(X::'a, \text{least-upper-bound}(Y::'a, Z)), \text{least-upper-bound}(\text{least-upper-bound}(X::'a, Y), Z)))$
 $\&$
 $(\forall X. \ \text{equal}(\text{least-upper-bound}(X::'a, X), X)) \ \&$
 $(\forall X. \ \text{equal}(\text{greatest-lower-bound}(X::'a, X), X)) \ \&$
 $(\forall Y \ X. \ \text{equal}(\text{least-upper-bound}(X::'a, \text{greatest-lower-bound}(X::'a, Y)), X)) \ \&$
 $(\forall Y \ X. \ \text{equal}(\text{greatest-lower-bound}(X::'a, \text{least-upper-bound}(X::'a, Y)), X)) \ \&$
 $(\forall Y \ X \ Z. \ \text{equal}(\text{multiply}(X::'a, \text{least-upper-bound}(Y::'a, Z)), \text{least-upper-bound}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall Y \ X \ Z. \ \text{equal}(\text{multiply}(X::'a, \text{greatest-lower-bound}(Y::'a, Z)), \text{greatest-lower-bound}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall Y \ Z \ X. \ \text{equal}(\text{multiply}(\text{least-upper-bound}(Y::'a, Z), X), \text{least-upper-bound}(\text{multiply}(Y::'a, X), \text{multiply}(Z::'a, X))))$
 $\&$
 $(\forall Y \ Z \ X. \ \text{equal}(\text{multiply}(\text{greatest-lower-bound}(Y::'a, Z), X), \text{greatest-lower-bound}(\text{multiply}(Y::'a, X), \text{multiply}(Z::'a, X))))$
 $\&$

$(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{greatest-lower-bound}(A::'a, C), \text{greatest-lower-bound}(B::'a, C)))$
 $\&$
 $(\forall A C B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{greatest-lower-bound}(C::'a, A), \text{greatest-lower-bound}(C::'a, B)))$
 $\&$
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{least-upper-bound}(A::'a, C), \text{least-upper-bound}(B::'a, C)))$
 $\&$
 $(\forall A C B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{least-upper-bound}(C::'a, A), \text{least-upper-bound}(C::'a, B)))$
 $\&$
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{multiply}(A::'a, C), \text{multiply}(B::'a, C))) \&$
 $(\forall A C B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{multiply}(C::'a, A), \text{multiply}(C::'a, B)))$

lemma *GRP156-1:*

EQU001-0-ax equal &
GRP004-0-ax INVERSE identity multiply equal &
GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal &
 $(\text{equal}(\text{least-upper-bound}(a::'a, b), b)) \&$
 $(\sim \text{equal}(\text{greatest-lower-bound}(\text{multiply}(a::'a, c), \text{multiply}(b::'a, c)), \text{multiply}(a::'a, c)))$
 $\longrightarrow \text{False}$
by *meson*

lemma *GRP168-1:*

EQU001-0-ax equal &
GRP004-0-ax INVERSE identity multiply equal &
GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal &
 $(\text{equal}(\text{least-upper-bound}(a::'a, b), b)) \&$
 $(\sim \text{equal}(\text{least-upper-bound}(\text{multiply}(\text{INVERSE}(c), \text{multiply}(a::'a, c)), \text{multiply}(\text{INVERSE}(c), \text{multiply}(b::'a, c))))$
 $\longrightarrow \text{False}$
by *meson*

abbreviation *HEN002-0-ax identity Zero Divide equal mless-equal* \equiv

$(\forall X Y. \text{mless-equal}(X::'a, Y) \longrightarrow \text{equal}(\text{Divide}(X::'a, Y), \text{Zero})) \&$
 $(\forall X Y. \text{equal}(\text{Divide}(X::'a, Y), \text{Zero}) \longrightarrow \text{mless-equal}(X::'a, Y)) \&$
 $(\forall Y X. \text{mless-equal}(\text{Divide}(X::'a, Y), X)) \&$
 $(\forall X Y Z. \text{mless-equal}(\text{Divide}(\text{Divide}(X::'a, Z), \text{Divide}(Y::'a, Z)), \text{Divide}(\text{Divide}(X::'a, Y), Z)))$
 $\&$
 $(\forall X. \text{mless-equal}(\text{Zero}::'a, X)) \&$
 $(\forall X Y. \text{mless-equal}(X::'a, Y) \& \text{mless-equal}(Y::'a, X) \longrightarrow \text{equal}(X::'a, Y)) \&$
 $(\forall X. \text{mless-equal}(X::'a, \text{identity}))$

abbreviation *HEN002-0-eq mless-equal Divide equal* \equiv

$(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{Divide}(A::'a, C), \text{Divide}(B::'a, C))) \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{Divide}(F'::'a, D), \text{Divide}(F'::'a, E))) \&$
 $(\forall G H I'. \text{equal}(G::'a, H) \& \text{mless-equal}(G::'a, I') \longrightarrow \text{mless-equal}(H::'a, I')) \&$
 $(\forall J L K'. \text{equal}(J::'a, K') \& \text{mless-equal}(L::'a, J) \longrightarrow \text{mless-equal}(L::'a, K'))$

lemma *HEN003-3:*

EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
 $(\sim \text{equal}(\text{Divide}(a::'a,a),\text{Zero})) \longrightarrow \text{False}$
oops

lemma *HEN007-2:*

EQU001-0-ax equal &
 $(\forall X\ Y. \text{mless-equal}(X::'a,Y) \longrightarrow \text{quotient}(X::'a,Y,\text{Zero}))$ &
 $(\forall X\ Y. \text{quotient}(X::'a,Y,\text{Zero}) \longrightarrow \text{mless-equal}(X::'a,Y))$ &
 $(\forall Y\ Z\ X. \text{quotient}(X::'a,Y,Z) \longrightarrow \text{mless-equal}(Z::'a,X))$ &
 $(\forall Y\ X\ V3\ V2\ V1\ Z\ V4\ V5. \text{quotient}(X::'a,Y,V1) \ \& \ \text{quotient}(Y::'a,Z,V2) \ \& \ \text{quotient}(X::'a,Z,V3) \ \& \ \text{quotient}(V3::'a,V2,V4) \ \& \ \text{quotient}(V1::'a,Z,V5) \longrightarrow \text{mless-equal}(V4::'a,V5))$ &
 $(\forall X. \text{mless-equal}(\text{Zero}::'a,X))$ &
 $(\forall X\ Y. \text{mless-equal}(X::'a,Y) \ \& \ \text{mless-equal}(Y::'a,X) \longrightarrow \text{equal}(X::'a,Y))$ &
 $(\forall X. \text{mless-equal}(X::'a,\text{identity}))$ &
 $(\forall X\ Y. \text{quotient}(X::'a,Y,\text{Divide}(X::'a,Y)))$ &
 $(\forall X\ Y\ Z\ W. \text{quotient}(X::'a,Y,Z) \ \& \ \text{quotient}(X::'a,Y,W) \longrightarrow \text{equal}(Z::'a,W))$ &
 $(\forall X\ Y\ W\ Z. \text{equal}(X::'a,Y) \ \& \ \text{quotient}(X::'a,W,Z) \longrightarrow \text{quotient}(Y::'a,W,Z))$ &
 $(\forall X\ W\ Y\ Z. \text{equal}(X::'a,Y) \ \& \ \text{quotient}(W::'a,X,Z) \longrightarrow \text{quotient}(W::'a,Y,Z))$ &
 $(\forall X\ W\ Z\ Y. \text{equal}(X::'a,Y) \ \& \ \text{quotient}(W::'a,Z,X) \longrightarrow \text{quotient}(W::'a,Z,Y))$ &
 $(\forall X\ Z\ Y. \text{equal}(X::'a,Y) \ \& \ \text{mless-equal}(Z::'a,X) \longrightarrow \text{mless-equal}(Z::'a,Y))$ &
 $(\forall X\ Y\ Z. \text{equal}(X::'a,Y) \ \& \ \text{mless-equal}(X::'a,Z) \longrightarrow \text{mless-equal}(Y::'a,Z))$ &
 $(\forall X\ Y\ W. \text{equal}(X::'a,Y) \longrightarrow \text{equal}(\text{Divide}(X::'a,W),\text{Divide}(Y::'a,W)))$ &
 $(\forall X\ W\ Y. \text{equal}(X::'a,Y) \longrightarrow \text{equal}(\text{Divide}(W::'a,X),\text{Divide}(W::'a,Y)))$ &
 $(\forall X. \text{quotient}(X::'a,\text{identity},\text{Zero}))$ &
 $(\forall X. \text{quotient}(\text{Zero}::'a,X,\text{Zero}))$ &
 $(\forall X. \text{quotient}(X::'a,X,\text{Zero}))$ &
 $(\forall X. \text{quotient}(X::'a,\text{Zero},X))$ &
 $(\forall Y\ X\ Z. \text{mless-equal}(X::'a,Y) \ \& \ \text{mless-equal}(Y::'a,Z) \longrightarrow \text{mless-equal}(X::'a,Z))$ &
 $(\forall W1\ X\ Z\ W2\ Y. \text{quotient}(X::'a,Y,W1) \ \& \ \text{mless-equal}(W1::'a,Z) \ \& \ \text{quotient}(X::'a,Z,W2) \longrightarrow \text{mless-equal}(W2::'a,Y))$ &
 $(\text{mless-equal}(x::'a,y))$ &
 $(\text{quotient}(z::'a,y,zQy))$ &
 $(\text{quotient}(z::'a,x,zQx))$ &
 $(\sim \text{mless-equal}(zQy::'a,zQx)) \longrightarrow \text{False}$
oops

lemma *HEN008-4:*

EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &

HEN002-0-eq mless-equal Divide equal &
 $(\forall X. \text{equal}(\text{Divide}(X::'a, \text{identity}), \text{Zero})) \ \&$
 $(\forall X. \text{equal}(\text{Divide}(\text{Zero}::'a, X), \text{Zero})) \ \&$
 $(\forall X. \text{equal}(\text{Divide}(X::'a, X), \text{Zero})) \ \&$
 $(\text{equal}(\text{Divide}(a::'a, \text{Zero}), a)) \ \&$
 $(\forall Y X Z. \text{mless-equal}(X::'a, Y) \ \& \ \text{mless-equal}(Y::'a, Z) \longrightarrow \text{mless-equal}(X::'a, Z))$
 $\&$
 $(\forall X Z Y. \text{mless-equal}(\text{Divide}(X::'a, Y), Z) \longrightarrow \text{mless-equal}(\text{Divide}(X::'a, Z), Y))$
 $\&$
 $(\forall Y Z X. \text{mless-equal}(X::'a, Y) \longrightarrow \text{mless-equal}(\text{Divide}(Z::'a, Y), \text{Divide}(Z::'a, X)))$
 $\&$
 $(\text{mless-equal}(a::'a, b)) \ \&$
 $(\sim \text{mless-equal}(\text{Divide}(a::'a, c), \text{Divide}(b::'a, c))) \longrightarrow \text{False}$
oops

lemma *HEN009-5:*

EQU001-0-ax equal &
 $(\forall Y X. \text{equal}(\text{Divide}(\text{Divide}(X::'a, Y), X), \text{Zero})) \ \&$
 $(\forall X Y Z. \text{equal}(\text{Divide}(\text{Divide}(\text{Divide}(X::'a, Z), \text{Divide}(Y::'a, Z)), \text{Divide}(\text{Divide}(X::'a, Y), Z)), \text{Zero}))$
 $\&$
 $(\forall X. \text{equal}(\text{Divide}(\text{Zero}::'a, X), \text{Zero})) \ \&$
 $(\forall X Y. \text{equal}(\text{Divide}(X::'a, Y), \text{Zero}) \ \& \ \text{equal}(\text{Divide}(Y::'a, X), \text{Zero}) \longrightarrow \text{equal}(X::'a, Y))$
 $\&$
 $(\forall X. \text{equal}(\text{Divide}(X::'a, \text{identity}), \text{Zero})) \ \&$
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{Divide}(A::'a, C), \text{Divide}(B::'a, C))) \ \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{Divide}(F'::'a, D), \text{Divide}(F'::'a, E))) \ \&$
 $(\forall Y X Z. \text{equal}(\text{Divide}(X::'a, Y), \text{Zero}) \ \& \ \text{equal}(\text{Divide}(Y::'a, Z), \text{Zero}) \longrightarrow$
 $\text{equal}(\text{Divide}(X::'a, Z), \text{Zero})) \ \&$
 $(\forall X Z Y. \text{equal}(\text{Divide}(\text{Divide}(X::'a, Y), Z), \text{Zero}) \longrightarrow \text{equal}(\text{Divide}(\text{Divide}(X::'a, Z), Y), \text{Zero}))$
 $\&$
 $(\forall Y Z X. \text{equal}(\text{Divide}(X::'a, Y), \text{Zero}) \longrightarrow \text{equal}(\text{Divide}(\text{Divide}(Z::'a, Y), \text{Divide}(Z::'a, X)), \text{Zero}))$
 $\&$
 $(\sim \text{equal}(\text{Divide}(\text{identity}::'a, a), \text{Divide}(\text{identity}::'a, \text{Divide}(\text{identity}::'a, \text{Divide}(\text{identity}::'a, a))))$
 $\&$
 $(\text{equal}(\text{Divide}(\text{identity}::'a, a), b)) \ \&$
 $(\text{equal}(\text{Divide}(\text{identity}::'a, b), c)) \ \&$
 $(\text{equal}(\text{Divide}(\text{identity}::'a, c), d)) \ \&$
 $(\sim \text{equal}(b::'a, d)) \longrightarrow \text{False}$
by meson

lemma *HEN012-3:*

EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
 $(\sim \text{mless-equal}(a::'a, a)) \longrightarrow \text{False}$
oops

lemma *LCL010-1*:

$(\forall X Y. \text{is-a-theorem}(\text{equivalent}(X::'a, Y)) \ \& \ \text{is-a-theorem}(X) \longrightarrow \text{is-a-theorem}(Y))$
 $\&$
 $(\forall X Z Y. \text{is-a-theorem}(\text{equivalent}(\text{equivalent}(X::'a, Y), \text{equivalent}(\text{equivalent}(X::'a, Z), \text{equivalent}(Z::'a, Y))))$
 $\&$
 $(\sim \text{is-a-theorem}(\text{equivalent}(\text{equivalent}(a::'a, b), \text{equivalent}(\text{equivalent}(c::'a, b), \text{equivalent}(a::'a, c))))$
 $\longrightarrow \text{False}$
by *meson*

lemma *LCL077-2*:

$(\forall X Y. \text{is-a-theorem}(\text{implies}(X, Y)) \ \& \ \text{is-a-theorem}(X) \longrightarrow \text{is-a-theorem}(Y))$
 $\&$
 $(\forall Y X. \text{is-a-theorem}(\text{implies}(X, \text{implies}(Y, X)))) \ \&$
 $(\forall Y X Z. \text{is-a-theorem}(\text{implies}(\text{implies}(X, \text{implies}(Y, Z)), \text{implies}(\text{implies}(X, Y), \text{implies}(X, Z)))))$
 $\&$
 $(\forall Y X. \text{is-a-theorem}(\text{implies}(\text{implies}(\text{not}(X), \text{not}(Y)), \text{implies}(Y, X)))) \ \&$
 $(\forall X2 X1 X3. \text{is-a-theorem}(\text{implies}(X1, X2)) \ \& \ \text{is-a-theorem}(\text{implies}(X2, X3)))$
 $\longrightarrow \text{is-a-theorem}(\text{implies}(X1, X3)) \ \&$
 $(\sim \text{is-a-theorem}(\text{implies}(\text{not}(\text{not}(a)), a))) \longrightarrow \text{False}$
by *meson*

lemma *LCL082-1*:

$(\forall X Y. \text{is-a-theorem}(\text{implies}(X::'a, Y)) \ \& \ \text{is-a-theorem}(X) \longrightarrow \text{is-a-theorem}(Y))$
 $\&$
 $(\forall Y Z U X. \text{is-a-theorem}(\text{implies}(\text{implies}(\text{implies}(X::'a, Y), Z), \text{implies}(\text{implies}(Z::'a, X), \text{implies}(U::'a, X)))))$
 $\&$
 $(\sim \text{is-a-theorem}(\text{implies}(a::'a, \text{implies}(b::'a, a)))) \longrightarrow \text{False}$
by *meson*

lemma *LCL111-1*:

$(\forall X Y. \text{is-a-theorem}(\text{implies}(X, Y)) \ \& \ \text{is-a-theorem}(X) \longrightarrow \text{is-a-theorem}(Y))$
 $\&$
 $(\forall Y X. \text{is-a-theorem}(\text{implies}(X, \text{implies}(Y, X)))) \ \&$
 $(\forall Y X Z. \text{is-a-theorem}(\text{implies}(\text{implies}(X, Y), \text{implies}(\text{implies}(Y, Z), \text{implies}(X, Z)))))$
 $\&$
 $(\forall Y X. \text{is-a-theorem}(\text{implies}(\text{implies}(\text{implies}(X, Y), Y), \text{implies}(\text{implies}(Y, X), X))))$
 $\&$
 $(\forall Y X. \text{is-a-theorem}(\text{implies}(\text{implies}(\text{not}(X), \text{not}(Y)), \text{implies}(Y, X)))) \ \&$
 $(\sim \text{is-a-theorem}(\text{implies}(\text{implies}(a, b), \text{implies}(\text{implies}(c, a), \text{implies}(c, b))))) \longrightarrow \text{False}$
by *meson*

lemma *LCL143-1*:

$(\forall X. \text{equal}(X, X)) \ \&$

$(\forall Y X. \text{equal}(X, Y) \longrightarrow \text{equal}(Y, X)) \ \&$
 $(\forall Y X Z. \text{equal}(X, Y) \ \& \ \text{equal}(Y, Z) \longrightarrow \text{equal}(X, Z)) \ \&$
 $(\forall X. \text{equal}(\text{implies}(\text{true}, X), X)) \ \&$
 $(\forall Y X Z. \text{equal}(\text{implies}(\text{implies}(X, Y), \text{implies}(\text{implies}(Y, Z), \text{implies}(X, Z))), \text{true}))$
 $\&$
 $(\forall Y X. \text{equal}(\text{implies}(\text{implies}(X, Y), Y), \text{implies}(\text{implies}(Y, X), X))) \ \&$
 $(\forall Y X. \text{equal}(\text{implies}(\text{implies}(\text{not}(X), \text{not}(Y)), \text{implies}(Y, X)), \text{true})) \ \&$
 $(\forall A B C. \text{equal}(A, B) \longrightarrow \text{equal}(\text{implies}(A, C), \text{implies}(B, C))) \ \&$
 $(\forall D F' E. \text{equal}(D, E) \longrightarrow \text{equal}(\text{implies}(F', D), \text{implies}(F', E))) \ \&$
 $(\forall G H. \text{equal}(G, H) \longrightarrow \text{equal}(\text{not}(G), \text{not}(H))) \ \&$
 $(\forall X Y. \text{equal}(\text{big-V}(X, Y), \text{implies}(\text{implies}(X, Y), Y))) \ \&$
 $(\forall X Y. \text{equal}(\text{big-hat}(X, Y), \text{not}(\text{big-V}(\text{not}(X), \text{not}(Y)))) \ \&$
 $(\forall X Y. \text{ordered}(X, Y) \longrightarrow \text{equal}(\text{implies}(X, Y), \text{true})) \ \&$
 $(\forall X Y. \text{equal}(\text{implies}(X, Y), \text{true}) \longrightarrow \text{ordered}(X, Y)) \ \&$
 $(\forall A B C. \text{equal}(A, B) \longrightarrow \text{equal}(\text{big-V}(A, C), \text{big-V}(B, C))) \ \&$
 $(\forall D F' E. \text{equal}(D, E) \longrightarrow \text{equal}(\text{big-V}(F', D), \text{big-V}(F', E))) \ \&$
 $(\forall G H I'. \text{equal}(G, H) \longrightarrow \text{equal}(\text{big-hat}(G, I'), \text{big-hat}(H, I'))) \ \&$
 $(\forall J L K'. \text{equal}(J, K') \longrightarrow \text{equal}(\text{big-hat}(L, J), \text{big-hat}(L, K'))) \ \&$
 $(\forall M N O'. \text{equal}(M, N) \ \& \ \text{ordered}(M, O') \longrightarrow \text{ordered}(N, O')) \ \&$
 $(\forall P R Q. \text{equal}(P, Q) \ \& \ \text{ordered}(R, P) \longrightarrow \text{ordered}(R, Q)) \ \&$
 $(\text{ordered}(x, y)) \ \&$
 $(\sim \text{ordered}(\text{implies}(z, x), \text{implies}(z, y))) \longrightarrow \text{False}$
by meson

lemma LCL182-1:

$(\forall A. \text{axiom}(\text{or}(\text{not}(\text{or}(A, A)), A))) \ \&$
 $(\forall B A. \text{axiom}(\text{or}(\text{not}(A), \text{or}(B, A)))) \ \&$
 $(\forall B A. \text{axiom}(\text{or}(\text{not}(\text{or}(A, B)), \text{or}(B, A)))) \ \&$
 $(\forall B A C. \text{axiom}(\text{or}(\text{not}(\text{or}(A, \text{or}(B, C))), \text{or}(B, \text{or}(A, C)))) \ \&$
 $(\forall A C B. \text{axiom}(\text{or}(\text{not}(\text{or}(\text{not}(A), B)), \text{or}(\text{not}(\text{or}(C, A)), \text{or}(C, B)))) \ \&$
 $(\forall X. \text{axiom}(X) \longrightarrow \text{theorem}(X)) \ \&$
 $(\forall X Y. \text{axiom}(\text{or}(\text{not}(Y), X)) \ \& \ \text{theorem}(Y) \longrightarrow \text{theorem}(X)) \ \&$
 $(\forall X Y Z. \text{axiom}(\text{or}(\text{not}(X), Y)) \ \& \ \text{theorem}(\text{or}(\text{not}(Y), Z)) \longrightarrow \text{theorem}(\text{or}(\text{not}(X), Z)))$
 $\&$
 $(\sim \text{theorem}(\text{or}(\text{not}(\text{or}(\text{not}(p), q)), \text{or}(\text{not}(\text{not}(q)), \text{not}(p)))) \longrightarrow \text{False}$
by meson

lemma LCL200-1:

$(\forall A. \text{axiom}(\text{or}(\text{not}(\text{or}(A, A)), A))) \ \&$
 $(\forall B A. \text{axiom}(\text{or}(\text{not}(A), \text{or}(B, A)))) \ \&$
 $(\forall B A. \text{axiom}(\text{or}(\text{not}(\text{or}(A, B)), \text{or}(B, A)))) \ \&$
 $(\forall B A C. \text{axiom}(\text{or}(\text{not}(\text{or}(A, \text{or}(B, C))), \text{or}(B, \text{or}(A, C)))) \ \&$
 $(\forall A C B. \text{axiom}(\text{or}(\text{not}(\text{or}(\text{not}(A), B)), \text{or}(\text{not}(\text{or}(C, A)), \text{or}(C, B)))) \ \&$
 $(\forall X. \text{axiom}(X) \longrightarrow \text{theorem}(X)) \ \&$
 $(\forall X Y. \text{axiom}(\text{or}(\text{not}(Y), X)) \ \& \ \text{theorem}(Y) \longrightarrow \text{theorem}(X)) \ \&$
 $(\forall X Y Z. \text{axiom}(\text{or}(\text{not}(X), Y)) \ \& \ \text{theorem}(\text{or}(\text{not}(Y), Z)) \longrightarrow \text{theorem}(\text{or}(\text{not}(X), Z)))$
 $\&$

($\sim \text{theorem}(\text{or}(\text{not}(\text{not}(\text{or}(p,q))),\text{not}(q))) \longrightarrow \text{False}$)
by *meson*

lemma *LCL215-1*:

($\forall A. \text{axiom}(\text{or}(\text{not}(\text{or}(A,A)),A))$) &
($\forall B A. \text{axiom}(\text{or}(\text{not}(A),\text{or}(B,A)))$) &
($\forall B A. \text{axiom}(\text{or}(\text{not}(\text{or}(A,B)),\text{or}(B,A)))$) &
($\forall B A C. \text{axiom}(\text{or}(\text{not}(\text{or}(A,\text{or}(B,C))),\text{or}(B,\text{or}(A,C))))$) &
($\forall A C B. \text{axiom}(\text{or}(\text{not}(\text{or}(\text{not}(A),B)),\text{or}(\text{not}(\text{or}(C,A)),\text{or}(C,B))))$) &
($\forall X. \text{axiom}(X) \longrightarrow \text{theorem}(X)$) &
($\forall X Y. \text{axiom}(\text{or}(\text{not}(Y),X)) \ \& \ \text{theorem}(Y) \longrightarrow \text{theorem}(X)$) &
($\forall X Y Z. \text{axiom}(\text{or}(\text{not}(X),Y)) \ \& \ \text{theorem}(\text{or}(\text{not}(Y),Z)) \longrightarrow \text{theorem}(\text{or}(\text{not}(X),Z))$)
&
($\sim \text{theorem}(\text{or}(\text{not}(\text{or}(\text{not}(p),q)),\text{or}(\text{not}(\text{or}(p,q)),q))) \longrightarrow \text{False}$)
by *meson*

lemma *LCL230-2*:

($q \longrightarrow p \mid r$) &
($\sim p$) &
(q) &
($\sim r$) $\longrightarrow \text{False}$
by *meson*

lemma *LDA003-1*:

EQU001-0-ax equal &
($\forall Y X Z. \text{equal}(f(X::'a,f(Y::'a,Z)),f(f(X::'a,Y),f(X::'a,Z)))$) &
($\forall X Y. \text{left}(X::'a,f(X::'a,Y))$) &
($\forall Y X Z. \text{left}(X::'a,Y) \ \& \ \text{left}(Y::'a,Z) \longrightarrow \text{left}(X::'a,Z)$) &
($\text{equal}(\text{num2}::'a,f(\text{num1}::'a,\text{num1}))$) &
($\text{equal}(\text{num3}::'a,f(\text{num2}::'a,\text{num1}))$) &
($\text{equal}(u::'a,f(\text{num2}::'a,\text{num2}))$) &
($\forall A B C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(f(A::'a,C),f(B::'a,C))$) &
($\forall D F' E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(f(F'::'a,D),f(F'::'a,E))$) &
($\forall G H I'. \text{equal}(G::'a,H) \ \& \ \text{left}(G::'a,I') \longrightarrow \text{left}(H::'a,I')$) &
($\forall J L K'. \text{equal}(J::'a,K') \ \& \ \text{left}(L::'a,J) \longrightarrow \text{left}(L::'a,K')$) &
($\sim \text{left}(\text{num3}::'a,u)$) $\longrightarrow \text{False}$
oops

lemma *MSC002-1*:

($\text{at}(\text{something}::'a,\text{here},\text{now})$) &
($\forall \text{Place Situation}. \text{hand-at}(\text{Place}::'a,\text{Situation}) \longrightarrow \text{hand-at}(\text{Place}::'a,\text{let-go}(\text{Situation}))$)
&
($\forall \text{Place Another-place Situation}. \text{hand-at}(\text{Place}::'a,\text{Situation}) \longrightarrow \text{hand-at}(\text{Another-place}::'a,\text{go}(\text{Another-pl$)
&

$(\forall \text{ Thing Situation. } \sim \text{held}(\text{Thing}::'a, \text{let-go}(\text{Situation}))) \ \&$
 $(\forall \text{ Situation Thing. } \text{at}(\text{Thing}::'a, \text{here}, \text{Situation}) \longrightarrow \text{red}(\text{Thing})) \ \&$
 $(\forall \text{ Thing Place Situation. } \text{at}(\text{Thing}::'a, \text{Place}, \text{Situation}) \longrightarrow \text{at}(\text{Thing}::'a, \text{Place}, \text{let-go}(\text{Situation})))$
 $\&$
 $(\forall \text{ Thing Place Situation. } \text{at}(\text{Thing}::'a, \text{Place}, \text{Situation}) \longrightarrow \text{at}(\text{Thing}::'a, \text{Place}, \text{pick-up}(\text{Situation})))$
 $\&$
 $(\forall \text{ Thing Place Situation. } \text{at}(\text{Thing}::'a, \text{Place}, \text{Situation}) \longrightarrow \text{grabbed}(\text{Thing}::'a, \text{pick-up}(\text{go}(\text{Place}::'a, \text{let-go}(\text{Situation}))))$
 $\&$
 $(\forall \text{ Thing Situation. } \text{red}(\text{Thing}) \ \& \ \text{put}(\text{Thing}::'a, \text{there}, \text{Situation}) \longrightarrow \text{answer}(\text{Situation}))$
 $\&$
 $(\forall \text{ Place Thing Another-place Situation. } \text{at}(\text{Thing}::'a, \text{Place}, \text{Situation}) \ \& \ \text{grabbed}(\text{Thing}::'a, \text{Situation})$
 $\longrightarrow \text{put}(\text{Thing}::'a, \text{Another-place}, \text{go}(\text{Another-place}::'a, \text{Situation}))) \ \&$
 $(\forall \text{ Thing Place Another-place Situation. } \text{at}(\text{Thing}::'a, \text{Place}, \text{Situation}) \longrightarrow \text{held}(\text{Thing}::'a, \text{Situation})$
 $| \text{at}(\text{Thing}::'a, \text{Place}, \text{go}(\text{Another-place}::'a, \text{Situation}))) \ \&$
 $(\forall \text{ One-place Thing Place Situation. } \text{hand-at}(\text{One-place}::'a, \text{Situation}) \ \& \ \text{held}(\text{Thing}::'a, \text{Situation})$
 $\longrightarrow \text{at}(\text{Thing}::'a, \text{Place}, \text{go}(\text{Place}::'a, \text{Situation}))) \ \&$
 $(\forall \text{ Place Thing Situation. } \text{hand-at}(\text{Place}::'a, \text{Situation}) \ \& \ \text{at}(\text{Thing}::'a, \text{Place}, \text{Situation})$
 $\longrightarrow \text{held}(\text{Thing}::'a, \text{pick-up}(\text{Situation}))) \ \&$
 $(\forall \text{ Situation. } \sim \text{answer}(\text{Situation})) \longrightarrow \text{False}$
by meson

lemma MSC003-1:

$(\forall \text{ Number-of-small-parts Small-part Big-part Number-of-mid-parts Mid-part. } \text{has-parts}(\text{Big-part}::'a, \text{Number-of-small-parts})$
 $\longrightarrow \text{in}'(\text{object-in}(\text{Big-part}::'a, \text{Mid-part}, \text{Small-part}, \text{Number-of-mid-parts}, \text{Number-of-small-parts}), \text{Mid-part})$
 $| \text{has-parts}(\text{Big-part}::'a, \text{mtimes}(\text{Number-of-mid-parts}::'a, \text{Number-of-small-parts}), \text{Small-part}))$
 $\&$
 $(\forall \text{ Big-part Mid-part Number-of-mid-parts Number-of-small-parts Small-part. } \text{has-parts}(\text{Big-part}::'a, \text{Number-of-small-parts})$
 $\& \ \text{has-parts}(\text{object-in}(\text{Big-part}::'a, \text{Mid-part}, \text{Small-part}, \text{Number-of-mid-parts}, \text{Number-of-small-parts}), \text{Number-of-mid-parts})$
 $\longrightarrow \text{has-parts}(\text{Big-part}::'a, \text{mtimes}(\text{Number-of-mid-parts}::'a, \text{Number-of-small-parts}), \text{Small-part}))$
 $\&$
 $(\text{in}'(\text{john}::'a, \text{boy})) \ \&$
 $(\forall X. \text{in}'(X::'a, \text{boy}) \longrightarrow \text{in}'(X::'a, \text{human})) \ \&$
 $(\forall X. \text{in}'(X::'a, \text{hand}) \longrightarrow \text{has-parts}(X::'a, \text{num5}, \text{fingers})) \ \&$
 $(\forall X. \text{in}'(X::'a, \text{human}) \longrightarrow \text{has-parts}(X::'a, \text{num2}, \text{arm})) \ \&$
 $(\forall X. \text{in}'(X::'a, \text{arm}) \longrightarrow \text{has-parts}(X::'a, \text{num1}, \text{hand})) \ \&$
 $(\sim \text{has-parts}(\text{john}::'a, \text{mtimes}(\text{num2}::'a, \text{num1}), \text{hand})) \longrightarrow \text{False}$
by meson

lemma MSC004-1:

$(\forall \text{ Number-of-small-parts Small-part Big-part Number-of-mid-parts Mid-part. } \text{has-parts}(\text{Big-part}::'a, \text{Number-of-small-parts})$
 $\longrightarrow \text{in}'(\text{object-in}(\text{Big-part}::'a, \text{Mid-part}, \text{Small-part}, \text{Number-of-mid-parts}, \text{Number-of-small-parts}), \text{Mid-part})$
 $| \text{has-parts}(\text{Big-part}::'a, \text{mtimes}(\text{Number-of-mid-parts}::'a, \text{Number-of-small-parts}), \text{Small-part}))$
 $\&$
 $(\forall \text{ Big-part Mid-part Number-of-mid-parts Number-of-small-parts Small-part. } \text{has-parts}(\text{Big-part}::'a, \text{Number-of-small-parts})$
 $\& \ \text{has-parts}(\text{object-in}(\text{Big-part}::'a, \text{Mid-part}, \text{Small-part}, \text{Number-of-mid-parts}, \text{Number-of-small-parts}), \text{Number-of-mid-parts})$
 $\longrightarrow \text{has-parts}(\text{Big-part}::'a, \text{mtimes}(\text{Number-of-mid-parts}::'a, \text{Number-of-small-parts}), \text{Small-part}))$
 $\&$

$(in'(john::'a,boy)) \ \&$
 $(\forall X. in'(X::'a,boy) \longrightarrow in'(X::'a,human)) \ \&$
 $(\forall X. in'(X::'a,hand) \longrightarrow has-parts(X::'a,num5,fingers)) \ \&$
 $(\forall X. in'(X::'a,human) \longrightarrow has-parts(X::'a,num2,arm)) \ \&$
 $(\forall X. in'(X::'a,arm) \longrightarrow has-parts(X::'a,num1,hand)) \ \&$
 $(\sim has-parts(john::'a,mtimes(mtimes(num2::'a,num1),num5),fingers)) \longrightarrow False$
by meson

lemma MSC005-1:

$(value(truth::'a,truth)) \ \&$
 $(value(falsity::'a,falsity)) \ \&$
 $(\forall X Y. value(X::'a,truth) \ \& \ value(Y::'a,truth) \longrightarrow value(xor(X::'a,Y),falsity))$
 $\ \&$
 $(\forall X Y. value(X::'a,truth) \ \& \ value(Y::'a,falsity) \longrightarrow value(xor(X::'a,Y),truth))$
 $\ \&$
 $(\forall X Y. value(X::'a,falsity) \ \& \ value(Y::'a,truth) \longrightarrow value(xor(X::'a,Y),truth))$
 $\ \&$
 $(\forall X Y. value(X::'a,falsity) \ \& \ value(Y::'a,falsity) \longrightarrow value(xor(X::'a,Y),falsity))$
 $\ \&$
 $(\forall Value. \sim value(xor(xor(xor(xor(truth::'a,falsity),falsity),truth),falsity),Value))$
 $\longrightarrow False$
by meson

lemma MSC006-1:

$(\forall Y X Z. p(X::'a,Y) \ \& \ p(Y::'a,Z) \longrightarrow p(X::'a,Z)) \ \&$
 $(\forall Y X Z. q(X::'a,Y) \ \& \ q(Y::'a,Z) \longrightarrow q(X::'a,Z)) \ \&$
 $(\forall Y X. q(X::'a,Y) \longrightarrow q(Y::'a,X)) \ \&$
 $(\forall X Y. p(X::'a,Y) \mid q(X::'a,Y)) \ \&$
 $(\sim p(a::'a,b)) \ \&$
 $(\sim q(c::'a,d)) \longrightarrow False$
by meson

lemma NUM001-1:

$(\forall A. equal(A::'a,A)) \ \&$
 $(\forall B A C. equal(A::'a,B) \ \& \ equal(B::'a,C) \longrightarrow equal(A::'a,C)) \ \&$
 $(\forall B A. equal(add(A::'a,B),add(B::'a,A))) \ \&$
 $(\forall A B C. equal(add(A::'a,add(B::'a,C)),add(add(A::'a,B),C))) \ \&$
 $(\forall B A. equal(subtract(add(A::'a,B),B),A)) \ \&$
 $(\forall A B. equal(A::'a,subtract(add(A::'a,B),B))) \ \&$
 $(\forall A C B. equal(add(subtract(A::'a,B),C),subtract(add(A::'a,C),B))) \ \&$
 $(\forall A C B. equal(subtract(add(A::'a,B),C),add(subtract(A::'a,C),B))) \ \&$
 $(\forall A C B D. equal(A::'a,B) \ \& \ equal(C::'a,add(A::'a,D)) \longrightarrow equal(C::'a,add(B::'a,D)))$
 $\ \&$
 $(\forall A C D B. equal(A::'a,B) \ \& \ equal(C::'a,add(D::'a,A)) \longrightarrow equal(C::'a,add(D::'a,B)))$
 $\ \&$
 $(\forall A C B D. equal(A::'a,B) \ \& \ equal(C::'a,subtract(A::'a,D)) \longrightarrow equal(C::'a,subtract(B::'a,D)))$

$\&$
 $(\forall A \ C \ D \ B. \text{equal}(A::'a, B) \ \& \ \text{equal}(C::'a, \text{subtract}(D::'a, A)) \dashrightarrow \text{equal}(C::'a, \text{subtract}(D::'a, B)))$
 $\&$
 $(\sim \text{equal}(\text{add}(\text{add}(a::'a, b), c), \text{add}(a::'a, \text{add}(b::'a, c)))) \dashrightarrow \text{False}$
by *meson*

abbreviation *NUM001-0-ax multiply successor num0 add equal* \equiv
 $(\forall A. \text{equal}(\text{add}(A::'a, \text{num0}), A)) \ \&$
 $(\forall A \ B. \text{equal}(\text{add}(A::'a, \text{successor}(B)), \text{successor}(\text{add}(A::'a, B)))) \ \&$
 $(\forall A. \text{equal}(\text{multiply}(A::'a, \text{num0}), \text{num0})) \ \&$
 $(\forall B \ A. \text{equal}(\text{multiply}(A::'a, \text{successor}(B)), \text{add}(\text{multiply}(A::'a, B), A))) \ \&$
 $(\forall A \ B. \text{equal}(\text{successor}(A), \text{successor}(B)) \dashrightarrow \text{equal}(A::'a, B)) \ \&$
 $(\forall A \ B. \text{equal}(A::'a, B) \dashrightarrow \text{equal}(\text{successor}(A), \text{successor}(B)))$

abbreviation *NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless*
 \equiv
 $(\forall A \ C \ B. \text{mless}(A::'a, B) \ \& \ \text{mless}(C::'a, A) \dashrightarrow \text{mless}(C::'a, B)) \ \&$
 $(\forall A \ B \ C. \text{equal}(\text{add}(\text{successor}(A), B), C) \dashrightarrow \text{mless}(B::'a, C)) \ \&$
 $(\forall A \ B. \text{mless}(A::'a, B) \dashrightarrow \text{equal}(\text{add}(\text{successor}(\text{predecessor-of-1st-minus-2nd}(B::'a, A)), A), B))$

abbreviation *NUM001-2-ax equal mless divides* \equiv
 $(\forall A \ B. \text{divides}(A::'a, B) \dashrightarrow \text{mless}(A::'a, B) \mid \text{equal}(A::'a, B)) \ \&$
 $(\forall A \ B. \text{mless}(A::'a, B) \dashrightarrow \text{divides}(A::'a, B)) \ \&$
 $(\forall A \ B. \text{equal}(A::'a, B) \dashrightarrow \text{divides}(A::'a, B))$

lemma *NUM021-1:*

EQU001-0-ax equal $\&$
NUM001-0-ax multiply successor num0 add equal $\&$
NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless $\&$
NUM001-2-ax equal mless divides $\&$
 $(\text{mless}(b::'a, c)) \ \&$
 $(\sim \text{mless}(b::'a, a)) \ \&$
 $(\text{divides}(c::'a, a)) \ \&$
 $(\forall A. \sim \text{equal}(\text{successor}(A), \text{num0})) \dashrightarrow \text{False}$
by *meson*

lemma *NUM024-1:*

EQU001-0-ax equal $\&$
NUM001-0-ax multiply successor num0 add equal $\&$
NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless $\&$
 $(\forall B \ A. \text{equal}(\text{add}(A::'a, B), \text{add}(B::'a, A))) \ \&$
 $(\forall B \ A \ C. \text{equal}(\text{add}(A::'a, B), \text{add}(C::'a, B)) \dashrightarrow \text{equal}(A::'a, C)) \ \&$
 $(\text{mless}(a::'a, a)) \ \&$
 $(\forall A. \sim \text{equal}(\text{successor}(A), \text{num0})) \dashrightarrow \text{False}$
oops

abbreviation *SET004-0-ax not-homomorphism2 not-homomorphism1*

homomorphism compatible operation cantor diagonalise subset-relation
one-to-one choice apply regular function identity-relation
single-valued-class compos powerClass sum-class omega inductive
successor-relation successor image' rng domain range-of INVERSE flip
rot domain-of null-class restrct difference union complement
intersection element-relation second first cross-product ordered-pair
singleton unordered-pair equal universal-class not-subclass-element
member subclass \equiv
 $(\forall X U Y. \text{subclass}(X::'a, Y) \ \& \ \text{member}(U::'a, X) \longrightarrow \text{member}(U::'a, Y)) \ \&$
 $(\forall X Y. \text{member}(\text{not-subclass-element}(X::'a, Y), X) \mid \text{subclass}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{member}(\text{not-subclass-element}(X::'a, Y), Y) \longrightarrow \text{subclass}(X::'a, Y)) \ \&$
 $(\forall X. \text{subclass}(X::'a, \text{universal-class})) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{subclass}(X::'a, Y)) \ \&$
 $(\forall Y X. \text{equal}(X::'a, Y) \longrightarrow \text{subclass}(Y::'a, X)) \ \&$
 $(\forall X Y. \text{subclass}(X::'a, Y) \ \& \ \text{subclass}(Y::'a, X) \longrightarrow \text{equal}(X::'a, Y)) \ \&$
 $(\forall X U Y. \text{member}(U::'a, \text{unordered-pair}(X::'a, Y)) \longrightarrow \text{equal}(U::'a, X) \mid \text{equal}(U::'a, Y))$
 $\ \&$
 $(\forall X Y. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(X::'a, \text{unordered-pair}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y. \text{member}(Y::'a, \text{universal-class}) \longrightarrow \text{member}(Y::'a, \text{unordered-pair}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y. \text{member}(\text{unordered-pair}(X::'a, Y), \text{universal-class})) \ \&$
 $(\forall X. \text{equal}(\text{unordered-pair}(X::'a, X), \text{singleton}(X))) \ \&$
 $(\forall X Y. \text{equal}(\text{unordered-pair}(\text{singleton}(X), \text{unordered-pair}(X::'a, \text{singleton}(Y))), \text{ordered-pair}(X::'a, Y)))$
 $\ \&$
 $(\forall V Y U X. \text{member}(\text{ordered-pair}(U::'a, V), \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(U::'a, X)) \ \&$
 $(\forall U X V Y. \text{member}(\text{ordered-pair}(U::'a, V), \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(V::'a, Y)) \ \&$
 $(\forall U V X Y. \text{member}(U::'a, X) \ \& \ \text{member}(V::'a, Y) \longrightarrow \text{member}(\text{ordered-pair}(U::'a, V), \text{cross-product}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y Z. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{equal}(\text{ordered-pair}(\text{first}(Z), \text{second}(Z)), Z))$
 $\ \&$
 $(\text{subclass}(\text{element-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\ \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{element-relation}) \longrightarrow \text{member}(X::'a, Y))$
 $\ \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))$
 $\ \& \ \text{member}(X::'a, Y) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, Y), \text{element-relation})) \ \&$
 $(\forall Y Z X. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, X)) \ \&$
 $(\forall X Z Y. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, Y)) \ \&$
 $(\forall Z X Y. \text{member}(Z::'a, X) \ \& \ \text{member}(Z::'a, Y) \longrightarrow \text{member}(Z::'a, \text{intersection}(X::'a, Y)))$
 $\ \&$
 $(\forall Z X. \sim(\text{member}(Z::'a, \text{complement}(X)) \ \& \ \text{member}(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{universal-class}) \longrightarrow \text{member}(Z::'a, \text{complement}(X)) \mid$
 $\text{member}(Z::'a, X)) \ \&$
 $(\forall X Y. \text{equal}(\text{complement}(\text{intersection}(\text{complement}(X), \text{complement}(Y))), \text{union}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y. \text{equal}(\text{intersection}(\text{complement}(\text{intersection}(X::'a, Y)), \text{complement}(\text{intersection}(\text{complement}(X), \text{complement}(Y)))))$

$\&$
 $(\forall Xr X Y. \text{equal}(\text{intersection}(Xr::'a, \text{cross-product}(X::'a, Y)), \text{restrct}(Xr::'a, X, Y)))$
 $\&$
 $(\forall Xr X Y. \text{equal}(\text{intersection}(\text{cross-product}(X::'a, Y), Xr), \text{restrct}(Xr::'a, X, Y)))$
 $\&$
 $(\forall Z X. \sim(\text{equal}(\text{restrct}(X::'a, \text{singleton}(Z), \text{universal-class}), \text{null-class}) \& \text{member}(Z::'a, \text{domain-of}(X)))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{universal-class}) \longrightarrow \text{equal}(\text{restrct}(X::'a, \text{singleton}(Z), \text{universal-class}), \text{null-class})$
 $| \text{member}(Z::'a, \text{domain-of}(X))) \&$
 $(\forall X. \text{subclass}(\text{rot}(X), \text{cross-product}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class}), \text{universal-class})))$
 $\&$
 $(\forall V W U X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{rot}(X)) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, W), U), X)) \&$
 $(\forall U V W X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, W), U), X) \& \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{rot}(X))) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{rot}(X))) \&$
 $(\forall X. \text{subclass}(\text{flip}(X), \text{cross-product}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class}), \text{universal-class})))$
 $\&$
 $(\forall V U W X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{flip}(X)) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, U), W), X)) \&$
 $(\forall U V W X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, U), W), X) \& \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{flip}(X))) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{flip}(X))) \&$
 $(\forall Y. \text{equal}(\text{domain-of}(\text{flip}(\text{cross-product}(Y::'a, \text{universal-class}))), \text{INVERSE}(Y)))$
 $\&$
 $(\forall Z. \text{equal}(\text{domain-of}(\text{INVERSE}(Z)), \text{range-of}(Z))) \&$
 $(\forall Z X Y. \text{equal}(\text{first}(\text{not-subclass-element}(\text{restrct}(Z::'a, X, \text{singleton}(Y)), \text{null-class})), \text{domain}(Z::'a, X, Y)))$
 $\&$
 $(\forall Z X Y. \text{equal}(\text{second}(\text{not-subclass-element}(\text{restrct}(Z::'a, \text{singleton}(X), Y), \text{null-class})), \text{rng}(Z::'a, X, Y)))$
 $\&$
 $(\forall Xr X. \text{equal}(\text{range-of}(\text{restrct}(Xr::'a, X, \text{universal-class})), \text{image}'(Xr::'a, X))) \&$
 $(\forall X. \text{equal}(\text{union}(X::'a, \text{singleton}(X)), \text{successor}(X))) \&$
 $(\text{subclass}(\text{successor-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{successor-relation}) \longrightarrow \text{equal}(\text{successor}(X), Y))$
 $\&$
 $(\forall X Y. \text{equal}(\text{successor}(X), Y) \& \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, Y), \text{successor-relation})) \&$
 $(\forall X. \text{inductive}(X) \longrightarrow \text{member}(\text{null-class}::'a, X)) \&$
 $(\forall X. \text{inductive}(X) \longrightarrow \text{subclass}(\text{image}'(\text{successor-relation}::'a, X), X)) \&$
 $(\forall X. \text{member}(\text{null-class}::'a, X) \& \text{subclass}(\text{image}'(\text{successor-relation}::'a, X), X)) \longrightarrow \text{inductive}(X)) \&$
 $(\text{inductive}(\text{omega})) \&$
 $(\forall Y. \text{inductive}(Y) \longrightarrow \text{subclass}(\text{omega}::'a, Y)) \&$
 $(\text{member}(\text{omega}::'a, \text{universal-class})) \&$
 $(\forall X. \text{equal}(\text{domain-of}(\text{restrct}(\text{element-relation}::'a, \text{universal-class}, X)), \text{sum-class}(X)))$
 $\&$
 $(\forall X. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{sum-class}(X), \text{universal-class}))$
 $\&$
 $(\forall X. \text{equal}(\text{complement}(\text{image}'(\text{element-relation}::'a, \text{complement}(X))), \text{powerClass}(X)))$
 $\&$

$(\forall U. \text{member}(U::'a, \text{universal-class}) \longrightarrow \text{member}(\text{powerClass}(U), \text{universal-class}))$
 $\&$
 $(\forall Yr\ Xr. \text{subclass}(\text{compos}(Yr::'a, Xr), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall Z\ Yr\ Xr\ Y. \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compos}(Yr::'a, Xr)) \longrightarrow \text{member}(Z::'a, \text{image}'(Yr::'a, \text{image}'(Xr::'a, \text{singleton}(Y)))) \&$
 $(\forall Y\ Z\ Yr\ Xr. \text{member}(Z::'a, \text{image}'(Yr::'a, \text{image}'(Xr::'a, \text{singleton}(Y)))) \& \text{member}(\text{ordered-pair}(Y::'a, Z), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})) \longrightarrow \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compos}(Yr::'a, Xr))) \&$
 $(\forall X. \text{single-valued-class}(X) \longrightarrow \text{subclass}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation}))$
 $\&$
 $(\forall X. \text{subclass}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation}) \longrightarrow \text{single-valued-class}(X))$
 $\&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{subclass}(Xf::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{subclass}(\text{compos}(Xf::'a, \text{INVERSE}(Xf)), \text{identity-relation}))$
 $\&$
 $(\forall Xf. \text{subclass}(Xf::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})) \& \text{subclass}(\text{compos}(Xf::'a, \text{INVERSE}(Xf)), \text{identity-relation}) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Xf\ X. \text{function}(Xf) \& \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{image}'(Xf::'a, X), \text{universal-class}))$
 $\&$
 $(\forall X. \text{equal}(X::'a, \text{null-class}) \mid \text{member}(\text{regular}(X), X)) \&$
 $(\forall X. \text{equal}(X::'a, \text{null-class}) \mid \text{equal}(\text{intersection}(X::'a, \text{regular}(X)), \text{null-class})) \&$
 $(\forall Xf\ Y. \text{equal}(\text{sum-class}(\text{image}'(Xf::'a, \text{singleton}(Y))), \text{apply}(Xf::'a, Y))) \&$
 $(\text{function}(\text{choice})) \&$
 $(\forall Y. \text{member}(Y::'a, \text{universal-class}) \longrightarrow \text{equal}(Y::'a, \text{null-class}) \mid \text{member}(\text{apply}(\text{choice}::'a, Y), Y))$
 $\&$
 $(\forall Xf. \text{one-to-one}(Xf) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Xf. \text{one-to-one}(Xf) \longrightarrow \text{function}(\text{INVERSE}(Xf))) \&$
 $(\forall Xf. \text{function}(\text{INVERSE}(Xf)) \& \text{function}(Xf) \longrightarrow \text{one-to-one}(Xf)) \&$
 $(\text{equal}(\text{intersection}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class}), \text{intersection}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class})), \text{intersection}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class})), \text{identity-relation}))$
 $\&$
 $(\forall Xr. \text{equal}(\text{complement}(\text{domain-of}(\text{intersection}(Xr::'a, \text{identity-relation}))), \text{diagonalise}(Xr)))$
 $\&$
 $(\forall X. \text{equal}(\text{intersection}(\text{domain-of}(X), \text{diagonalise}(\text{compos}(\text{INVERSE}(\text{element-relation}), X))), \text{cantor}(X)))$
 $\&$
 $(\forall Xf. \text{operation}(Xf) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Xf. \text{operation}(Xf) \longrightarrow \text{equal}(\text{cross-product}(\text{domain-of}(\text{domain-of}(Xf)), \text{domain-of}(\text{domain-of}(Xf))), \text{domain-of}(\text{domain-of}(Xf))))$
 $\&$
 $(\forall Xf. \text{operation}(Xf) \longrightarrow \text{subclass}(\text{range-of}(Xf), \text{domain-of}(\text{domain-of}(Xf))))$
 $\&$
 $(\forall Xf. \text{function}(Xf) \& \text{equal}(\text{cross-product}(\text{domain-of}(\text{domain-of}(Xf)), \text{domain-of}(\text{domain-of}(Xf))), \text{domain-of}(\text{domain-of}(Xf))) \longrightarrow \text{operation}(Xf)) \&$
 $(\forall Xf1\ Xf2\ Xh. \text{compatible}(Xh::'a, Xf1, Xf2) \longrightarrow \text{function}(Xh)) \&$
 $(\forall Xf2\ Xf1\ Xh. \text{compatible}(Xh::'a, Xf1, Xf2) \longrightarrow \text{equal}(\text{domain-of}(\text{domain-of}(Xf1)), \text{domain-of}(Xh)))$
 $\&$
 $(\forall Xf1\ Xh\ Xf2. \text{compatible}(Xh::'a, Xf1, Xf2) \longrightarrow \text{subclass}(\text{range-of}(Xh), \text{domain-of}(\text{domain-of}(Xf2))))$

$\&$
 $(\forall Xh\ Xh1\ Xf1\ Xf2. \text{function}(Xh) \ \& \ \text{equal}(\text{domain-of}(\text{domain-of}(Xf1)), \text{domain-of}(Xh)))$
 $\& \text{subclass}(\text{range-of}(Xh), \text{domain-of}(\text{domain-of}(Xf2))) \longrightarrow \text{compatible}(Xh1::'a, Xf1, Xf2))$
 $\&$
 $(\forall Xh\ Xf2\ Xf1. \text{homomorphism}(Xh::'a, Xf1, Xf2) \longrightarrow \text{operation}(Xf1)) \ \&$
 $(\forall Xh\ Xf1\ Xf2. \text{homomorphism}(Xh::'a, Xf1, Xf2) \longrightarrow \text{operation}(Xf2)) \ \&$
 $(\forall Xh\ Xf1\ Xf2. \text{homomorphism}(Xh::'a, Xf1, Xf2) \longrightarrow \text{compatible}(Xh::'a, Xf1, Xf2))$
 $\&$
 $(\forall Xf2\ Xh\ Xf1\ X\ Y. \text{homomorphism}(Xh::'a, Xf1, Xf2) \ \& \ \text{member}(\text{ordered-pair}(X::'a, Y), \text{domain-of}(Xf1)))$
 $\longrightarrow \text{equal}(\text{apply}(Xf2::'a, \text{ordered-pair}(\text{apply}(Xh::'a, X), \text{apply}(Xh::'a, Y))), \text{apply}(Xh::'a, \text{apply}(Xf1::'a, \text{ordered-pair}(X, Y))))$
 $\&$
 $(\forall Xh\ Xf1\ Xf2. \text{operation}(Xf1) \ \& \ \text{operation}(Xf2) \ \& \ \text{compatible}(Xh::'a, Xf1, Xf2))$
 $\longrightarrow \text{member}(\text{ordered-pair}(\text{not-homomorphism1}(Xh::'a, Xf1, Xf2), \text{not-homomorphism2}(Xh::'a, Xf1, Xf2)), \text{domain-of}(\text{homomorphism}(Xh::'a, Xf1, Xf2)))$
 $(\forall Xh\ Xf1\ Xf2. \text{operation}(Xf1) \ \& \ \text{operation}(Xf2) \ \& \ \text{compatible}(Xh::'a, Xf1, Xf2))$
 $\& \ \text{equal}(\text{apply}(Xf2::'a, \text{ordered-pair}(\text{apply}(Xh::'a, \text{not-homomorphism1}(Xh::'a, Xf1, Xf2)), \text{apply}(Xh::'a, \text{not-homomorphism2}(Xh::'a, Xf1, Xf2))))$
 $\longrightarrow \text{homomorphism}(Xh::'a, Xf1, Xf2))$

abbreviation SET004-0-eq subclass single-valued-class operation

one-to-one member inductive homomorphism function compatible
 unordered-pair union sum-class successor singleton second rot restrict
 regular range-of rng powerClass ordered-pair not-subclass-element
 not-homomorphism2 not-homomorphism1 INVERSE intersection image' flip
 first domain-of domain difference diagonalise cross-product compos
 complement cantor apply equal \equiv
 $(\forall D\ E\ F'. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{apply}(D::'a, F'), \text{apply}(E::'a, F')))) \ \&$
 $(\forall G\ I'\ H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{apply}(I'::'a, G), \text{apply}(I'::'a, H))) \ \&$
 $(\forall J\ K'. \text{equal}(J::'a, K') \longrightarrow \text{equal}(\text{cantor}(J), \text{cantor}(K')))) \ \&$
 $(\forall L\ M. \text{equal}(L::'a, M) \longrightarrow \text{equal}(\text{complement}(L), \text{complement}(M))) \ \&$
 $(\forall N\ O'\ P. \text{equal}(N::'a, O') \longrightarrow \text{equal}(\text{compos}(N::'a, P), \text{compos}(O'::'a, P))) \ \&$
 $(\forall Q\ S'\ R. \text{equal}(Q::'a, R) \longrightarrow \text{equal}(\text{compos}(S'::'a, Q), \text{compos}(S'::'a, R))) \ \&$
 $(\forall T'\ U\ V. \text{equal}(T'::'a, U) \longrightarrow \text{equal}(\text{cross-product}(T'::'a, V), \text{cross-product}(U::'a, V)))$
 $\&$
 $(\forall W\ Y\ X. \text{equal}(W::'a, X) \longrightarrow \text{equal}(\text{cross-product}(Y::'a, W), \text{cross-product}(Y::'a, X)))$
 $\&$
 $(\forall Z\ A1. \text{equal}(Z::'a, A1) \longrightarrow \text{equal}(\text{diagonalise}(Z), \text{diagonalise}(A1))) \ \&$
 $(\forall B1\ C1\ D1. \text{equal}(B1::'a, C1) \longrightarrow \text{equal}(\text{difference}(B1::'a, D1), \text{difference}(C1::'a, D1)))$
 $\&$
 $(\forall E1\ G1\ F1. \text{equal}(E1::'a, F1) \longrightarrow \text{equal}(\text{difference}(G1::'a, E1), \text{difference}(G1::'a, F1)))$
 $\&$
 $(\forall H1\ I1\ J1\ K1. \text{equal}(H1::'a, I1) \longrightarrow \text{equal}(\text{domain}(H1::'a, J1, K1), \text{domain}(I1::'a, J1, K1)))$
 $\&$
 $(\forall L1\ N1\ M1\ O1. \text{equal}(L1::'a, M1) \longrightarrow \text{equal}(\text{domain}(N1::'a, L1, O1), \text{domain}(N1::'a, M1, O1)))$
 $\&$
 $(\forall P1\ R1\ S1\ Q1. \text{equal}(P1::'a, Q1) \longrightarrow \text{equal}(\text{domain}(R1::'a, S1, P1), \text{domain}(R1::'a, S1, Q1)))$
 $\&$
 $(\forall T1\ U1. \text{equal}(T1::'a, U1) \longrightarrow \text{equal}(\text{domain-of}(T1), \text{domain-of}(U1))) \ \&$
 $(\forall V1\ W1. \text{equal}(V1::'a, W1) \longrightarrow \text{equal}(\text{first}(V1), \text{first}(W1))) \ \&$
 $(\forall X1\ Y1. \text{equal}(X1::'a, Y1) \longrightarrow \text{equal}(\text{flip}(X1), \text{flip}(Y1))) \ \&$

$(\forall Z1\ A2\ B2. \text{equal}(Z1::'a, A2) \longrightarrow \text{equal}(\text{image}'(Z1::'a, B2), \text{image}'(A2::'a, B2)))$
 $\&$
 $(\forall C2\ E2\ D2. \text{equal}(C2::'a, D2) \longrightarrow \text{equal}(\text{image}'(E2::'a, C2), \text{image}'(E2::'a, D2)))$
 $\&$
 $(\forall F2\ G2\ H2. \text{equal}(F2::'a, G2) \longrightarrow \text{equal}(\text{intersection}(F2::'a, H2), \text{intersection}(G2::'a, H2)))$
 $\&$
 $(\forall I2\ K2\ J2. \text{equal}(I2::'a, J2) \longrightarrow \text{equal}(\text{intersection}(K2::'a, I2), \text{intersection}(K2::'a, J2)))$
 $\&$
 $(\forall L2\ M2. \text{equal}(L2::'a, M2) \longrightarrow \text{equal}(\text{INVERSE}(L2), \text{INVERSE}(M2))) \&$
 $(\forall N2\ O2\ P2\ Q2. \text{equal}(N2::'a, O2) \longrightarrow \text{equal}(\text{not-homomorphism1}(N2::'a, P2, Q2), \text{not-homomorphism1}(O2::'a, P2, Q2)))$
 $\&$
 $(\forall R2\ T2\ S2\ U2. \text{equal}(R2::'a, S2) \longrightarrow \text{equal}(\text{not-homomorphism1}(T2::'a, R2, U2), \text{not-homomorphism1}(O2::'a, R2, U2)))$
 $\&$
 $(\forall V2\ X2\ Y2\ W2. \text{equal}(V2::'a, W2) \longrightarrow \text{equal}(\text{not-homomorphism1}(X2::'a, Y2, V2), \text{not-homomorphism1}(X2::'a, Y2, W2)))$
 $\&$
 $(\forall Z2\ A3\ B3\ C3. \text{equal}(Z2::'a, A3) \longrightarrow \text{equal}(\text{not-homomorphism2}(Z2::'a, B3, C3), \text{not-homomorphism2}(A3::'a, B3, C3)))$
 $\&$
 $(\forall D3\ F3\ E3\ G3. \text{equal}(D3::'a, E3) \longrightarrow \text{equal}(\text{not-homomorphism2}(F3::'a, D3, G3), \text{not-homomorphism2}(F3::'a, D3, E3)))$
 $\&$
 $(\forall H3\ J3\ K3\ I3. \text{equal}(H3::'a, I3) \longrightarrow \text{equal}(\text{not-homomorphism2}(J3::'a, K3, H3), \text{not-homomorphism2}(J3::'a, K3, I3)))$
 $\&$
 $(\forall L3\ M3\ N3. \text{equal}(L3::'a, M3) \longrightarrow \text{equal}(\text{not-subclass-element}(L3::'a, N3), \text{not-subclass-element}(M3::'a, N3)))$
 $\&$
 $(\forall O3\ Q3\ P3. \text{equal}(O3::'a, P3) \longrightarrow \text{equal}(\text{not-subclass-element}(Q3::'a, O3), \text{not-subclass-element}(Q3::'a, P3)))$
 $\&$
 $(\forall R3\ S3\ T3. \text{equal}(R3::'a, S3) \longrightarrow \text{equal}(\text{ordered-pair}(R3::'a, T3), \text{ordered-pair}(S3::'a, T3)))$
 $\&$
 $(\forall U3\ W3\ V3. \text{equal}(U3::'a, V3) \longrightarrow \text{equal}(\text{ordered-pair}(W3::'a, U3), \text{ordered-pair}(W3::'a, V3)))$
 $\&$
 $(\forall X3\ Y3. \text{equal}(X3::'a, Y3) \longrightarrow \text{equal}(\text{powerClass}(X3), \text{powerClass}(Y3))) \&$
 $(\forall Z3\ A4\ B4\ C4. \text{equal}(Z3::'a, A4) \longrightarrow \text{equal}(\text{rng}(Z3::'a, B4, C4), \text{rng}(A4::'a, B4, C4)))$
 $\&$
 $(\forall D4\ F4\ E4\ G4. \text{equal}(D4::'a, E4) \longrightarrow \text{equal}(\text{rng}(F4::'a, D4, G4), \text{rng}(F4::'a, E4, G4)))$
 $\&$
 $(\forall H4\ J4\ K4\ I4. \text{equal}(H4::'a, I4) \longrightarrow \text{equal}(\text{rng}(J4::'a, K4, H4), \text{rng}(J4::'a, K4, I4)))$
 $\&$
 $(\forall L4\ M4. \text{equal}(L4::'a, M4) \longrightarrow \text{equal}(\text{range-of}(L4), \text{range-of}(M4))) \&$
 $(\forall N4\ O4. \text{equal}(N4::'a, O4) \longrightarrow \text{equal}(\text{regular}(N4), \text{regular}(O4))) \&$
 $(\forall P4\ Q4\ R4\ S4. \text{equal}(P4::'a, Q4) \longrightarrow \text{equal}(\text{restrct}(P4::'a, R4, S4), \text{restrct}(Q4::'a, R4, S4)))$
 $\&$
 $(\forall T4\ V4\ U4\ W4. \text{equal}(T4::'a, U4) \longrightarrow \text{equal}(\text{restrct}(V4::'a, T4, W4), \text{restrct}(V4::'a, U4, W4)))$
 $\&$
 $(\forall X4\ Z4\ A5\ Y4. \text{equal}(X4::'a, Y4) \longrightarrow \text{equal}(\text{restrct}(Z4::'a, A5, X4), \text{restrct}(Z4::'a, A5, Y4)))$
 $\&$
 $(\forall B5\ C5. \text{equal}(B5::'a, C5) \longrightarrow \text{equal}(\text{rot}(B5), \text{rot}(C5))) \&$
 $(\forall D5\ E5. \text{equal}(D5::'a, E5) \longrightarrow \text{equal}(\text{second}(D5), \text{second}(E5))) \&$
 $(\forall F5\ G5. \text{equal}(F5::'a, G5) \longrightarrow \text{equal}(\text{singleton}(F5), \text{singleton}(G5))) \&$
 $(\forall H5\ I5. \text{equal}(H5::'a, I5) \longrightarrow \text{equal}(\text{successor}(H5), \text{successor}(I5))) \&$
 $(\forall J5\ K5. \text{equal}(J5::'a, K5) \longrightarrow \text{equal}(\text{sum-class}(J5), \text{sum-class}(K5))) \&$

$(\forall L5\ M5\ N5. \text{equal}(L5::'a, M5) \longrightarrow \text{equal}(\text{union}(L5::'a, N5), \text{union}(M5::'a, N5)))$
 $\&$
 $(\forall O5\ Q5\ P5. \text{equal}(O5::'a, P5) \longrightarrow \text{equal}(\text{union}(Q5::'a, O5), \text{union}(Q5::'a, P5)))$
 $\&$
 $(\forall R5\ S5\ T5. \text{equal}(R5::'a, S5) \longrightarrow \text{equal}(\text{unordered-pair}(R5::'a, T5), \text{unordered-pair}(S5::'a, T5)))$
 $\&$
 $(\forall U5\ W5\ V5. \text{equal}(U5::'a, V5) \longrightarrow \text{equal}(\text{unordered-pair}(W5::'a, U5), \text{unordered-pair}(W5::'a, V5)))$
 $\&$
 $(\forall X5\ Y5\ Z5\ A6. \text{equal}(X5::'a, Y5) \& \text{compatible}(X5::'a, Z5, A6) \longrightarrow \text{compatible}(Y5::'a, Z5, A6)) \&$
 $(\forall B6\ D6\ C6\ E6. \text{equal}(B6::'a, C6) \& \text{compatible}(D6::'a, B6, E6) \longrightarrow \text{compatible}(D6::'a, C6, E6)) \&$
 $(\forall F6\ H6\ I6\ G6. \text{equal}(F6::'a, G6) \& \text{compatible}(H6::'a, I6, F6) \longrightarrow \text{compatible}(H6::'a, I6, G6)) \&$
 $(\forall J6\ K6. \text{equal}(J6::'a, K6) \& \text{function}(J6) \longrightarrow \text{function}(K6)) \&$
 $(\forall L6\ M6\ N6\ O6. \text{equal}(L6::'a, M6) \& \text{homomorphism}(L6::'a, N6, O6) \longrightarrow \text{homomorphism}(M6::'a, N6, O6)) \&$
 $(\forall P6\ R6\ Q6\ S6. \text{equal}(P6::'a, Q6) \& \text{homomorphism}(R6::'a, P6, S6) \longrightarrow \text{homomorphism}(R6::'a, Q6, S6)) \&$
 $(\forall T6\ V6\ W6\ U6. \text{equal}(T6::'a, U6) \& \text{homomorphism}(V6::'a, W6, T6) \longrightarrow \text{homomorphism}(V6::'a, W6, U6)) \&$
 $(\forall X6\ Y6. \text{equal}(X6::'a, Y6) \& \text{inductive}(X6) \longrightarrow \text{inductive}(Y6)) \&$
 $(\forall Z6\ A7\ B7. \text{equal}(Z6::'a, A7) \& \text{member}(Z6::'a, B7) \longrightarrow \text{member}(A7::'a, B7))$
 $\&$
 $(\forall C7\ E7\ D7. \text{equal}(C7::'a, D7) \& \text{member}(E7::'a, C7) \longrightarrow \text{member}(E7::'a, D7))$
 $\&$
 $(\forall F7\ G7. \text{equal}(F7::'a, G7) \& \text{one-to-one}(F7) \longrightarrow \text{one-to-one}(G7)) \&$
 $(\forall H7\ I7. \text{equal}(H7::'a, I7) \& \text{operation}(H7) \longrightarrow \text{operation}(I7)) \&$
 $(\forall J7\ K7. \text{equal}(J7::'a, K7) \& \text{single-valued-class}(J7) \longrightarrow \text{single-valued-class}(K7))$
 $\&$
 $(\forall L7\ M7\ N7. \text{equal}(L7::'a, M7) \& \text{subclass}(L7::'a, N7) \longrightarrow \text{subclass}(M7::'a, N7))$
 $\&$
 $(\forall O7\ Q7\ P7. \text{equal}(O7::'a, P7) \& \text{subclass}(Q7::'a, O7) \longrightarrow \text{subclass}(Q7::'a, P7))$

abbreviation SET004-1-ax range-of function maps apply

application-function singleton-relation element-relation complement
 intersection single-valued3 singleton image' domain single-valued2
 second single-valued1 identity-relation INVERSE not-subclass-element
 first domain-of domain-relation composition-function compos equal
 ordered-pair member universal-class cross-product compose-class
 subclass \equiv

$(\forall X. \text{subclass}(\text{compose-class}(X), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$

$(\forall X\ Y\ Z. \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compose-class}(X)) \longrightarrow \text{equal}(\text{compos}(X::'a, Y), Z))$
 $\&$

$(\forall Y\ Z\ X. \text{member}(\text{ordered-pair}(Y::'a, Z), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))$
 $\& \text{equal}(\text{compos}(X::'a, Y), Z) \longrightarrow \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compose-class}(X)))$
 $\&$

$(\text{subclass}(\text{composition-function}::'a, \text{cross-product}(\text{universal-class}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))))$

$\&$
 $(\forall X Y Z. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{composition-function}))$
 $\longrightarrow \text{equal}(\text{compos}(X::'a, Y), Z)) \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, \text{compos}(X::'a, Y))), \text{composition-function}))$
 $\&$
 $(\text{subclass}(\text{domain-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))) \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{domain-relation}) \longrightarrow \text{equal}(\text{domain-of}(X), Y))$
 $\&$
 $(\forall X. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{domain-of}(X)), \text{domain-relation}))$
 $\&$
 $(\forall X. \text{equal}(\text{first}(\text{not-subclass-element}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation})), \text{single-valued1}(X)))$
 $\&$
 $(\forall X. \text{equal}(\text{second}(\text{not-subclass-element}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation})), \text{single-valued2}(X)))$
 $\&$
 $(\forall X. \text{equal}(\text{domain}(X::'a, \text{image}'(\text{INVERSE}(X), \text{singleton}(\text{single-valued1}(X))), \text{single-valued2}(X)), \text{single-valued1}(X)))$
 $\&$
 $(\text{equal}(\text{intersection}(\text{complement}(\text{compos}(\text{element-relation}::'a, \text{complement}(\text{identity-relation}))), \text{element-relation}))$
 $\&$
 $(\text{subclass}(\text{application-function}::'a, \text{cross-product}(\text{universal-class}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))))$
 $\&$
 $(\forall Z Y X. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{application-function}))$
 $\longrightarrow \text{member}(Y::'a, \text{domain-of}(X))) \&$
 $(\forall X Y Z. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{application-function}))$
 $\longrightarrow \text{equal}(\text{apply}(X::'a, Y), Z)) \&$
 $(\forall Z X Y. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{cross-product}(\text{universal-class}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))))$
 $\& \text{member}(Y::'a, \text{domain-of}(X)) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, \text{apply}(X::'a, Y))), \text{application-function}))$
 $\&$
 $(\forall X Y Xf. \text{maps}(Xf::'a, X, Y) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Y Xf X. \text{maps}(Xf::'a, X, Y) \longrightarrow \text{equal}(\text{domain-of}(Xf), X)) \&$
 $(\forall X Xf Y. \text{maps}(Xf::'a, X, Y) \longrightarrow \text{subclass}(\text{range-of}(Xf), Y)) \&$
 $(\forall Xf Y. \text{function}(Xf) \& \text{subclass}(\text{range-of}(Xf), Y) \longrightarrow \text{maps}(Xf::'a, \text{domain-of}(Xf), Y))$

abbreviation SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class

$\text{equal} \equiv$
 $(\forall L M. \text{equal}(L::'a, M) \longrightarrow \text{equal}(\text{compose-class}(L), \text{compose-class}(M))) \&$
 $(\forall N2 O2. \text{equal}(N2::'a, O2) \longrightarrow \text{equal}(\text{single-valued1}(N2), \text{single-valued1}(O2)))$
 $\&$
 $(\forall P2 Q2. \text{equal}(P2::'a, Q2) \longrightarrow \text{equal}(\text{single-valued2}(P2), \text{single-valued2}(Q2)))$
 $\&$
 $(\forall R2 S2. \text{equal}(R2::'a, S2) \longrightarrow \text{equal}(\text{single-valued3}(R2), \text{single-valued3}(S2)))$
 $\&$
 $(\forall X2 Y2 Z2 A3. \text{equal}(X2::'a, Y2) \& \text{maps}(X2::'a, Z2, A3) \longrightarrow \text{maps}(Y2::'a, Z2, A3))$
 $\&$
 $(\forall B3 D3 C3 E3. \text{equal}(B3::'a, C3) \& \text{maps}(D3::'a, B3, E3) \longrightarrow \text{maps}(D3::'a, C3, E3))$
 $\&$
 $(\forall F3 H3 I3 G3. \text{equal}(F3::'a, G3) \& \text{maps}(H3::'a, I3, F3) \longrightarrow \text{maps}(H3::'a, I3, G3))$

abbreviation NUM004-0-ax integer-of omega ordinal-multiply

add-relation ordinal-add recursion apply range-of union-of-range-map
 function recursion-equation-functions rest-relation rest-of
 limit-ordinals kind-1-ordinals successor-relation image'
 universal-class sum-class element-relation ordinal-numbers section
 not-well-ordering ordered-pair least member well-ordering singleton
 domain-of segment null-class intersection asymmetric compos transitive
 cross-product connected identity-relation complement restrict subclass
 irreflexive symmetrization-of INVERSE union equal \equiv
 $(\forall X. \text{equal}(\text{union}(X::'a, \text{INVERSE}(X)), \text{symmetrization-of}(X))) \ \&$
 $(\forall X \ Y. \text{irreflexive}(X::'a, Y) \longrightarrow \text{subclass}(\text{restrict}(X::'a, Y, Y), \text{complement}(\text{identity-relation})))$
 $\&$
 $(\forall X \ Y. \text{subclass}(\text{restrict}(X::'a, Y, Y), \text{complement}(\text{identity-relation})) \longrightarrow \text{irreflexive}(X::'a, Y)) \ \&$
 $(\forall Y \ X. \text{connected}(X::'a, Y) \longrightarrow \text{subclass}(\text{cross-product}(Y::'a, Y), \text{union}(\text{identity-relation}::'a, \text{symmetrization-of}(X))))$
 $\&$
 $(\forall X \ Y. \text{subclass}(\text{cross-product}(Y::'a, Y), \text{union}(\text{identity-relation}::'a, \text{symmetrization-of}(X))))$
 $\longrightarrow \text{connected}(X::'a, Y)) \ \&$
 $(\forall Xr \ Y. \text{transitive}(Xr::'a, Y) \longrightarrow \text{subclass}(\text{compos}(\text{restrict}(Xr::'a, Y, Y), \text{restrict}(Xr::'a, Y, Y)), \text{restrict}(Xr::'a, Y, Y)))$
 $\&$
 $(\forall Xr \ Y. \text{subclass}(\text{compos}(\text{restrict}(Xr::'a, Y, Y), \text{restrict}(Xr::'a, Y, Y)), \text{restrict}(Xr::'a, Y, Y)))$
 $\longrightarrow \text{transitive}(Xr::'a, Y)) \ \&$
 $(\forall Xr \ Y. \text{asymmetric}(Xr::'a, Y) \longrightarrow \text{equal}(\text{restrict}(\text{intersection}(Xr::'a, \text{INVERSE}(Xr)), Y, Y), \text{null-class}))$
 $\&$
 $(\forall Xr \ Y. \text{equal}(\text{restrict}(\text{intersection}(Xr::'a, \text{INVERSE}(Xr)), Y, Y), \text{null-class}) \longrightarrow \text{asymmetric}(Xr::'a, Y)) \ \&$
 $(\forall Xr \ Y \ Z. \text{equal}(\text{segment}(Xr::'a, Y, Z), \text{domain-of}(\text{restrict}(Xr::'a, Y, \text{singleton}(Z))))))$
 $\&$
 $(\forall X \ Y. \text{well-ordering}(X::'a, Y) \longrightarrow \text{connected}(X::'a, Y)) \ \&$
 $(\forall Y \ Xr \ U. \text{well-ordering}(Xr::'a, Y) \ \& \ \text{subclass}(U::'a, Y) \longrightarrow \text{equal}(U::'a, \text{null-class}))$
 $| \text{member}(\text{least}(Xr::'a, U), U)) \ \&$
 $(\forall Y \ V \ Xr \ U. \text{well-ordering}(Xr::'a, Y) \ \& \ \text{subclass}(U::'a, Y) \ \& \ \text{member}(V::'a, U))$
 $\longrightarrow \text{member}(\text{least}(Xr::'a, U), U)) \ \&$
 $(\forall Y \ Xr \ U. \text{well-ordering}(Xr::'a, Y) \ \& \ \text{subclass}(U::'a, Y) \longrightarrow \text{equal}(\text{segment}(Xr::'a, U, \text{least}(Xr::'a, U)), \text{null-class}))$
 $\&$
 $(\forall Y \ V \ U \ Xr. \sim(\text{well-ordering}(Xr::'a, Y) \ \& \ \text{subclass}(U::'a, Y) \ \& \ \text{member}(V::'a, U))$
 $\& \ \text{member}(\text{ordered-pair}(V::'a, \text{least}(Xr::'a, U)), Xr))) \ \&$
 $(\forall Xr \ Y. \text{connected}(Xr::'a, Y) \ \& \ \text{equal}(\text{not-well-ordering}(Xr::'a, Y), \text{null-class}))$
 $\longrightarrow \text{well-ordering}(Xr::'a, Y)) \ \&$
 $(\forall Xr \ Y. \text{connected}(Xr::'a, Y) \longrightarrow \text{subclass}(\text{not-well-ordering}(Xr::'a, Y), Y) \ |$
 $\text{well-ordering}(Xr::'a, Y)) \ \&$
 $(\forall V \ Xr \ Y. \text{member}(V::'a, \text{not-well-ordering}(Xr::'a, Y)) \ \& \ \text{equal}(\text{segment}(Xr::'a, \text{not-well-ordering}(Xr::'a, Y), \text{least}(Xr::'a, Y)), \text{null-class}))$
 $\& \ \text{connected}(Xr::'a, Y) \longrightarrow \text{well-ordering}(Xr::'a, Y)) \ \&$
 $(\forall Xr \ Y \ Z. \text{section}(Xr::'a, Y, Z) \longrightarrow \text{subclass}(Y::'a, Z)) \ \&$
 $(\forall Xr \ Z \ Y. \text{section}(Xr::'a, Y, Z) \longrightarrow \text{subclass}(\text{domain-of}(\text{restrict}(Xr::'a, Z, Y)), Y))$
 $\&$
 $(\forall Xr \ Y \ Z. \text{subclass}(Y::'a, Z) \ \& \ \text{subclass}(\text{domain-of}(\text{restrict}(Xr::'a, Z, Y)), Y) \longrightarrow \text{section}(Xr::'a, Y, Z)) \ \&$
 $(\forall X. \text{member}(X::'a, \text{ordinal-numbers}) \longrightarrow \text{well-ordering}(\text{element-relation}::'a, X))$
 $\&$

$(\forall X. \text{member}(X::'a, \text{ordinal-numbers}) \longrightarrow \text{subclass}(\text{sum-class}(X), X)) \ \&$
 $(\forall X. \text{well-ordering}(\text{element-relation}::'a, X) \ \& \ \text{subclass}(\text{sum-class}(X), X) \ \& \ \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(X::'a, \text{ordinal-numbers})) \ \&$
 $(\forall X. \text{well-ordering}(\text{element-relation}::'a, X) \ \& \ \text{subclass}(\text{sum-class}(X), X) \longrightarrow \text{member}(X::'a, \text{ordinal-numbers}) \mid \text{equal}(X::'a, \text{ordinal-numbers})) \ \&$
 $(\text{equal}(\text{union}(\text{singleton}(\text{null-class}), \text{image}'(\text{successor-relation}::'a, \text{ordinal-numbers})), \text{kind-1-ordinals}))$
 $\&$
 $(\text{equal}(\text{intersection}(\text{complement}(\text{kind-1-ordinals}), \text{ordinal-numbers}), \text{limit-ordinals}))$
 $\&$
 $(\forall X. \text{subclass}(\text{rest-of}(X), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))) \ \&$
 $(\forall V \ U \ X. \text{member}(\text{ordered-pair}(U::'a, V), \text{rest-of}(X)) \longrightarrow \text{member}(U::'a, \text{domain-of}(X)))$
 $\&$
 $(\forall X \ U \ V. \text{member}(\text{ordered-pair}(U::'a, V), \text{rest-of}(X)) \longrightarrow \text{equal}(\text{restrct}(X::'a, U, \text{universal-class}), V))$
 $\&$
 $(\forall U \ V \ X. \text{member}(U::'a, \text{domain-of}(X)) \ \& \ \text{equal}(\text{restrct}(X::'a, U, \text{universal-class}), V) \longrightarrow \text{member}(\text{ordered-pair}(U::'a, V), \text{rest-of}(X))) \ \&$
 $(\text{subclass}(\text{rest-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))) \ \&$
 $(\forall X \ Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{rest-relation}) \longrightarrow \text{equal}(\text{rest-of}(X), Y))$
 $\&$
 $(\forall X. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{rest-of}(X)), \text{rest-relation}))$
 $\&$
 $(\forall X \ Z. \text{member}(X::'a, \text{recursion-equation-functions}(Z)) \longrightarrow \text{function}(Z)) \ \&$
 $(\forall Z \ X. \text{member}(X::'a, \text{recursion-equation-functions}(Z)) \longrightarrow \text{function}(X)) \ \&$
 $(\forall Z \ X. \text{member}(X::'a, \text{recursion-equation-functions}(Z)) \longrightarrow \text{member}(\text{domain-of}(X), \text{ordinal-numbers}))$
 $\&$
 $(\forall Z \ X. \text{member}(X::'a, \text{recursion-equation-functions}(Z)) \longrightarrow \text{equal}(\text{compos}(Z::'a, \text{rest-of}(X)), X))$
 $\&$
 $(\forall X \ Z. \text{function}(Z) \ \& \ \text{function}(X) \ \& \ \text{member}(\text{domain-of}(X), \text{ordinal-numbers}) \ \& \ \text{equal}(\text{compos}(Z::'a, \text{rest-of}(X)), X) \longrightarrow \text{member}(X::'a, \text{recursion-equation-functions}(Z)))$
 $\&$
 $(\text{subclass}(\text{union-of-range-map}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall X \ Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{union-of-range-map}) \longrightarrow \text{equal}(\text{sum-class}(\text{range-of}(X)), Y))$
 $\&$
 $(\forall X \ Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))$
 $\& \ \text{equal}(\text{sum-class}(\text{range-of}(X)), Y) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, Y), \text{union-of-range-map}))$
 $\&$
 $(\forall X \ Y. \text{equal}(\text{apply}(\text{recursion}(X::'a, \text{successor-relation}, \text{union-of-range-map}), Y), \text{ordinal-add}(X::'a, Y)))$
 $\&$
 $(\forall X \ Y. \text{equal}(\text{recursion}(\text{null-class}::'a, \text{apply}(\text{add-relation}::'a, X), \text{union-of-range-map}), \text{ordinal-multiply}(X::'a, Y)))$
 $\&$
 $(\forall X. \text{member}(X::'a, \text{omega}) \longrightarrow \text{equal}(\text{integer-of}(X), X)) \ \&$
 $(\forall X. \text{member}(X::'a, \text{omega}) \mid \text{equal}(\text{integer-of}(X), \text{null-class}))$

abbreviation NUM004-0-eq well-ordering transitive section irreflexive
connected asymmetric symmetrization-of segment rest-of
recursion-equation-functions recursion ordinal-multiply ordinal-add
not-well-ordering least integer-of equal \equiv
 $(\forall D \ E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{integer-of}(D), \text{integer-of}(E))) \ \&$

$(\forall F' G H. \text{equal}(F'::'a, G) \longrightarrow \text{equal}(\text{least}(F'::'a, H), \text{least}(G::'a, H))) \ \&$
 $(\forall I' K' J. \text{equal}(I'::'a, J) \longrightarrow \text{equal}(\text{least}(K'::'a, I'), \text{least}(K'::'a, J))) \ \&$
 $(\forall L M N. \text{equal}(L::'a, M) \longrightarrow \text{equal}(\text{not-well-ordering}(L::'a, N), \text{not-well-ordering}(M::'a, N)))$
 $\&$
 $(\forall O' Q P. \text{equal}(O'::'a, P) \longrightarrow \text{equal}(\text{not-well-ordering}(Q::'a, O'), \text{not-well-ordering}(Q::'a, P)))$
 $\&$
 $(\forall R S' T'. \text{equal}(R::'a, S') \longrightarrow \text{equal}(\text{ordinal-add}(R::'a, T'), \text{ordinal-add}(S'::'a, T')))$
 $\&$
 $(\forall U W V. \text{equal}(U::'a, V) \longrightarrow \text{equal}(\text{ordinal-add}(W::'a, U), \text{ordinal-add}(W::'a, V)))$
 $\&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{ordinal-multiply}(X::'a, Z), \text{ordinal-multiply}(Y::'a, Z)))$
 $\&$
 $(\forall A1 C1 B1. \text{equal}(A1::'a, B1) \longrightarrow \text{equal}(\text{ordinal-multiply}(C1::'a, A1), \text{ordinal-multiply}(C1::'a, B1)))$
 $\&$
 $(\forall F1 G1 H1 I1. \text{equal}(F1::'a, G1) \longrightarrow \text{equal}(\text{recursion}(F1::'a, H1, I1), \text{recursion}(G1::'a, H1, I1)))$
 $\&$
 $(\forall J1 L1 K1 M1. \text{equal}(J1::'a, K1) \longrightarrow \text{equal}(\text{recursion}(L1::'a, J1, M1), \text{recursion}(L1::'a, K1, M1)))$
 $\&$
 $(\forall N1 P1 Q1 O1. \text{equal}(N1::'a, O1) \longrightarrow \text{equal}(\text{recursion}(P1::'a, Q1, N1), \text{recursion}(P1::'a, Q1, O1)))$
 $\&$
 $(\forall R1 S1. \text{equal}(R1::'a, S1) \longrightarrow \text{equal}(\text{recursion-equation-functions}(R1), \text{recursion-equation-functions}(S1)))$
 $\&$
 $(\forall T1 U1. \text{equal}(T1::'a, U1) \longrightarrow \text{equal}(\text{rest-of}(T1), \text{rest-of}(U1))) \ \&$
 $(\forall V1 W1 X1 Y1. \text{equal}(V1::'a, W1) \longrightarrow \text{equal}(\text{segment}(V1::'a, X1, Y1), \text{segment}(W1::'a, X1, Y1)))$
 $\&$
 $(\forall Z1 B2 A2 C2. \text{equal}(Z1::'a, A2) \longrightarrow \text{equal}(\text{segment}(B2::'a, Z1, C2), \text{segment}(B2::'a, A2, C2)))$
 $\&$
 $(\forall D2 F2 G2 E2. \text{equal}(D2::'a, E2) \longrightarrow \text{equal}(\text{segment}(F2::'a, G2, D2), \text{segment}(F2::'a, G2, E2)))$
 $\&$
 $(\forall H2 I2. \text{equal}(H2::'a, I2) \longrightarrow \text{equal}(\text{symmetrization-of}(H2), \text{symmetrization-of}(I2)))$
 $\&$
 $(\forall J2 K2 L2. \text{equal}(J2::'a, K2) \ \& \ \text{asymmetric}(J2::'a, L2) \longrightarrow \text{asymmetric}(K2::'a, L2))$
 $\&$
 $(\forall M2 O2 N2. \text{equal}(M2::'a, N2) \ \& \ \text{asymmetric}(O2::'a, M2) \longrightarrow \text{asymmetric}(O2::'a, N2))$
 $\&$
 $(\forall P2 Q2 R2. \text{equal}(P2::'a, Q2) \ \& \ \text{connected}(P2::'a, R2) \longrightarrow \text{connected}(Q2::'a, R2))$
 $\&$
 $(\forall S2 U2 T2. \text{equal}(S2::'a, T2) \ \& \ \text{connected}(U2::'a, S2) \longrightarrow \text{connected}(U2::'a, T2))$
 $\&$
 $(\forall V2 W2 X2. \text{equal}(V2::'a, W2) \ \& \ \text{irreflexive}(V2::'a, X2) \longrightarrow \text{irreflexive}(W2::'a, X2))$
 $\&$
 $(\forall Y2 A3 Z2. \text{equal}(Y2::'a, Z2) \ \& \ \text{irreflexive}(A3::'a, Y2) \longrightarrow \text{irreflexive}(A3::'a, Z2))$
 $\&$
 $(\forall B3 C3 D3 E3. \text{equal}(B3::'a, C3) \ \& \ \text{section}(B3::'a, D3, E3) \longrightarrow \text{section}(C3::'a, D3, E3))$
 $\&$
 $(\forall F3 H3 G3 I3. \text{equal}(F3::'a, G3) \ \& \ \text{section}(H3::'a, F3, I3) \longrightarrow \text{section}(H3::'a, G3, I3))$
 $\&$
 $(\forall J3 L3 M3 K3. \text{equal}(J3::'a, K3) \ \& \ \text{section}(L3::'a, M3, J3) \longrightarrow \text{section}(L3::'a, M3, K3))$
 $\&$

$(\forall N3\ O3\ P3. \text{equal}(N3::'a, O3) \ \& \ \text{transitive}(N3::'a, P3) \longrightarrow \text{transitive}(O3::'a, P3))$
 $\&$
 $(\forall Q3\ S3\ R3. \text{equal}(Q3::'a, R3) \ \& \ \text{transitive}(S3::'a, Q3) \longrightarrow \text{transitive}(S3::'a, R3))$
 $\&$
 $(\forall T3\ U3\ V3. \text{equal}(T3::'a, U3) \ \& \ \text{well-ordering}(T3::'a, V3) \longrightarrow \text{well-ordering}(U3::'a, V3))$
 $\&$
 $(\forall W3\ Y3\ X3. \text{equal}(W3::'a, X3) \ \& \ \text{well-ordering}(Y3::'a, W3) \longrightarrow \text{well-ordering}(Y3::'a, X3))$

lemma NUM180-1:

EQU001-0-ax equal &
 SET004-0-ax not-homomorphism2 not-homomorphism1
 homomorphism compatible operation cantor diagonalise subset-relation
 one-to-one choice apply regular function identity-relation
 single-valued-class compos powerClass sum-class omega inductive
 successor-relation successor image' rng domain range-of INVERSE flip
 rot domain-of null-class restrct difference union complement
 intersection element-relation second first cross-product ordered-pair
 singleton unordered-pair equal universal-class not-subclass-element
 member subclass &
 SET004-0-eq subclass single-valued-class operation
 one-to-one member inductive homomorphism function compatible
 unordered-pair union sum-class successor singleton second rot restrct
 regular range-of rng powerClass ordered-pair not-subclass-element
 not-homomorphism2 not-homomorphism1 INVERSE intersection image' flip
 first domain-of domain difference diagonalise cross-product compos
 complement cantor apply equal &
 SET004-1-ax range-of function maps apply
 application-function singleton-relation element-relation complement
 intersection single-valued3 singleton image' domain single-valued2
 second single-valued1 identity-relation INVERSE not-subclass-element
 first domain-of domain-relation composition-function compos equal
 ordered-pair member universal-class cross-product compose-class
 subclass &
 SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal
 &
 NUM004-0-ax integer-of omega ordinal-multiply
 add-relation ordinal-add recursion apply range-of union-of-range-map
 function recursion-equation-functions rest-relation rest-of
 limit-ordinals kind-1-ordinals successor-relation image'
 universal-class sum-class element-relation ordinal-numbers section
 not-well-ordering ordered-pair least member well-ordering singleton
 domain-of segment null-class intersection asymmetric compos transitive
 cross-product connected identity-relation complement restrct subclass
 irreflexive symmetrization-of INVERSE union equal &
 NUM004-0-eq well-ordering transitive section irreflexive
 connected asymmetric symmetrization-of segment rest-of
 recursion-equation-functions recursion ordinal-multiply ordinal-add
 not-well-ordering least integer-of equal &

(\sim subclass(limit-ordinals::'a, ordinal-numbers)) \longrightarrow False
by meson

lemma NUM228-1:

EQU001-0-ax equal &
SET004-0-ax not-homomorphism2 not-homomorphism1
homomorphism compatible operation cantor diagonalise subset-relation
one-to-one choice apply regular function identity-relation
single-valued-class compos powerClass sum-class omega inductive
successor-relation successor image' rng domain range-of INVERSE flip
rot domain-of null-class restrict difference union complement
intersection element-relation second first cross-product ordered-pair
singleton unordered-pair equal universal-class not-subclass-element
member subclass &
SET004-0-eq subclass single-valued-class operation
one-to-one member inductive homomorphism function compatible
unordered-pair union sum-class successor singleton second rot restrict
regular range-of rng powerClass ordered-pair not-subclass-element
not-homomorphism2 not-homomorphism1 INVERSE intersection image' flip
first domain-of domain difference diagonalise cross-product compos
complement cantor apply equal &
SET004-1-ax range-of function maps apply
application-function singleton-relation element-relation complement
intersection single-valued3 singleton image' domain single-valued2
second single-valued1 identity-relation INVERSE not-subclass-element
first domain-of domain-relation composition-function compos equal
ordered-pair member universal-class cross-product compose-class
subclass &
SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal
&
NUM004-0-ax integer-of omega ordinal-multiply
add-relation ordinal-add recursion apply range-of union-of-range-map
function recursion-equation-functions rest-relation rest-of
limit-ordinals kind-1-ordinals successor-relation image'
universal-class sum-class element-relation ordinal-numbers section
not-well-ordering ordered-pair least member well-ordering singleton
domain-of segment null-class intersection asymmetric compos transitive
cross-product connected identity-relation complement restrict subclass
irreflexive symmetrization-of INVERSE union equal &
NUM004-0-eq well-ordering transitive section irreflexive
connected asymmetric symmetrization-of segment rest-of
recursion-equation-functions recursion ordinal-multiply ordinal-add
not-well-ordering least integer-of equal &
(\sim function(z)) &
(\sim equal(recursion-equation-functions(z), null-class)) \longrightarrow False
by meson

lemma *PLA002-1*:

$(\forall \textit{Situation1} \textit{Situation2}. \textit{warm}(\textit{Situation1}) \mid \textit{cold}(\textit{Situation2})) \ \&$
 $(\forall \textit{Situation}. \textit{at}(a::'a, \textit{Situation}) \longrightarrow \textit{at}(b::'a, \textit{walk}(b::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(a::'a, \textit{Situation}) \longrightarrow \textit{at}(b::'a, \textit{drive}(b::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(b::'a, \textit{Situation}) \longrightarrow \textit{at}(a::'a, \textit{walk}(a::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(b::'a, \textit{Situation}) \longrightarrow \textit{at}(a::'a, \textit{drive}(a::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{cold}(\textit{Situation}) \ \& \ \textit{at}(b::'a, \textit{Situation}) \longrightarrow \textit{at}(c::'a, \textit{skate}(c::'a, \textit{Situation})))$
 $\ \&$
 $(\forall \textit{Situation}. \textit{cold}(\textit{Situation}) \ \& \ \textit{at}(c::'a, \textit{Situation}) \longrightarrow \textit{at}(b::'a, \textit{skate}(b::'a, \textit{Situation})))$
 $\ \&$
 $(\forall \textit{Situation}. \textit{warm}(\textit{Situation}) \ \& \ \textit{at}(b::'a, \textit{Situation}) \longrightarrow \textit{at}(d::'a, \textit{climb}(d::'a, \textit{Situation})))$
 $\ \&$
 $(\forall \textit{Situation}. \textit{warm}(\textit{Situation}) \ \& \ \textit{at}(d::'a, \textit{Situation}) \longrightarrow \textit{at}(b::'a, \textit{climb}(b::'a, \textit{Situation})))$
 $\ \&$
 $(\forall \textit{Situation}. \textit{at}(c::'a, \textit{Situation}) \longrightarrow \textit{at}(d::'a, \textit{go}(d::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(d::'a, \textit{Situation}) \longrightarrow \textit{at}(c::'a, \textit{go}(c::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(c::'a, \textit{Situation}) \longrightarrow \textit{at}(e::'a, \textit{go}(e::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(e::'a, \textit{Situation}) \longrightarrow \textit{at}(c::'a, \textit{go}(c::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(d::'a, \textit{Situation}) \longrightarrow \textit{at}(f::'a, \textit{go}(f::'a, \textit{Situation}))) \ \&$
 $(\forall \textit{Situation}. \textit{at}(f::'a, \textit{Situation}) \longrightarrow \textit{at}(d::'a, \textit{go}(d::'a, \textit{Situation}))) \ \&$
 $(\textit{at}(f::'a, s0)) \ \&$
 $(\forall S'. \sim \textit{at}(a::'a, S')) \longrightarrow \textit{False}$
by *meson*

abbreviation *PLA001-0-ax putdown on pickup do holding table differ clear EMPTY*

and' holds \equiv

$(\forall X \ Y \ \textit{State}. \textit{holds}(X::'a, \textit{State}) \ \& \ \textit{holds}(Y::'a, \textit{State}) \longrightarrow \textit{holds}(\textit{and}'(X::'a, Y), \textit{State}))$
 $\ \&$
 $(\forall \textit{State} \ X. \textit{holds}(\textit{EMPTY}::'a, \textit{State}) \ \& \ \textit{holds}(\textit{clear}(X), \textit{State}) \ \& \ \textit{differ}(X::'a, \textit{table})$
 $\longrightarrow \textit{holds}(\textit{holding}(X), \textit{do}(\textit{pickup}(X), \textit{State}))) \ \&$
 $(\forall Y \ X \ \textit{State}. \textit{holds}(\textit{on}(X::'a, Y), \textit{State}) \ \& \ \textit{holds}(\textit{clear}(X), \textit{State}) \ \& \ \textit{holds}(\textit{EMPTY}::'a, \textit{State})$
 $\longrightarrow \textit{holds}(\textit{clear}(Y), \textit{do}(\textit{pickup}(X), \textit{State}))) \ \&$
 $(\forall Y \ \textit{State} \ X \ Z. \textit{holds}(\textit{on}(X::'a, Y), \textit{State}) \ \& \ \textit{differ}(X::'a, Z) \longrightarrow \textit{holds}(\textit{on}(X::'a, Y), \textit{do}(\textit{pickup}(Z), \textit{State})))$
 $\ \&$
 $(\forall \textit{State} \ X \ Z. \textit{holds}(\textit{clear}(X), \textit{State}) \ \& \ \textit{differ}(X::'a, Z) \longrightarrow \textit{holds}(\textit{clear}(X), \textit{do}(\textit{pickup}(Z), \textit{State})))$
 $\ \&$
 $(\forall X \ Y \ \textit{State}. \textit{holds}(\textit{holding}(X), \textit{State}) \ \& \ \textit{holds}(\textit{clear}(Y), \textit{State}) \longrightarrow \textit{holds}(\textit{EMPTY}::'a, \textit{do}(\textit{putdown}(X::'a, Y), \textit{State})))$
 $\ \&$
 $(\forall X \ Y \ \textit{State}. \textit{holds}(\textit{holding}(X), \textit{State}) \ \& \ \textit{holds}(\textit{clear}(Y), \textit{State}) \longrightarrow \textit{holds}(\textit{on}(X::'a, Y), \textit{do}(\textit{putdown}(X::'a, Y), \textit{State})))$
 $\ \&$
 $(\forall X \ Y \ \textit{State}. \textit{holds}(\textit{holding}(X), \textit{State}) \ \& \ \textit{holds}(\textit{clear}(Y), \textit{State}) \longrightarrow \textit{holds}(\textit{clear}(X), \textit{do}(\textit{putdown}(X::'a, Y), \textit{State})))$
 $\ \&$
 $(\forall Z \ W \ X \ Y \ \textit{State}. \textit{holds}(\textit{on}(X::'a, Y), \textit{State}) \longrightarrow \textit{holds}(\textit{on}(X::'a, Y), \textit{do}(\textit{putdown}(Z::'a, W), \textit{State})))$
 $\ \&$
 $(\forall X \ \textit{State} \ Z \ Y. \textit{holds}(\textit{clear}(Z), \textit{State}) \ \& \ \textit{differ}(Z::'a, Y) \longrightarrow \textit{holds}(\textit{clear}(Z), \textit{do}(\textit{putdown}(X::'a, Y), \textit{State})))$

abbreviation *PLA001-1-ax EMPTY clear s0 on holds table d c b a differ* \equiv

$(\forall Y X. \text{differ}(Y::'a,X) \longrightarrow \text{differ}(X::'a,Y)) \ \&$
 $(\text{differ}(a::'a,b)) \ \&$
 $(\text{differ}(a::'a,c)) \ \&$
 $(\text{differ}(a::'a,d)) \ \&$
 $(\text{differ}(a::'a,\text{table})) \ \&$
 $(\text{differ}(b::'a,c)) \ \&$
 $(\text{differ}(b::'a,d)) \ \&$
 $(\text{differ}(b::'a,\text{table})) \ \&$
 $(\text{differ}(c::'a,d)) \ \&$
 $(\text{differ}(c::'a,\text{table})) \ \&$
 $(\text{differ}(d::'a,\text{table})) \ \&$
 $(\text{holds}(\text{on}(a::'a,\text{table}),s0)) \ \&$
 $(\text{holds}(\text{on}(b::'a,\text{table}),s0)) \ \&$
 $(\text{holds}(\text{on}(c::'a,d),s0)) \ \&$
 $(\text{holds}(\text{on}(d::'a,\text{table}),s0)) \ \&$
 $(\text{holds}(\text{clear}(a),s0)) \ \&$
 $(\text{holds}(\text{clear}(b),s0)) \ \&$
 $(\text{holds}(\text{clear}(c),s0)) \ \&$
 $(\text{holds}(\text{EMPTY}::'a,s0)) \ \&$
 $(\forall \text{State}. \text{holds}(\text{clear}(\text{table}),\text{State}))$

lemma *PLA006-1:*

PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
 $\&$
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
 $(\forall \text{State}. \sim \text{holds}(\text{on}(c::'a,\text{table}),\text{State})) \longrightarrow \text{False}$
by *meson*

lemma *PLA017-1:*

PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
 $\&$
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
 $(\forall \text{State}. \sim \text{holds}(\text{on}(a::'a,c),\text{State})) \longrightarrow \text{False}$
by *meson*

lemma *PLA022-1:*

PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
 $\&$
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
 $(\forall \text{State}. \sim \text{holds}(\text{and}'(\text{on}(c::'a,d),\text{on}(a::'a,c)),\text{State})) \longrightarrow \text{False}$
by *meson*

lemma *PLA022-2:*

PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
 $\&$

PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
 $(\forall \text{State}. \sim \text{holds}(\text{and}'(\text{on}(a::'a,c), \text{on}(c::'a,d)), \text{State})) \longrightarrow \text{False}$
 by meson

lemma PRV001-1:

$(\forall X Y Z. q1(X::'a, Y, Z) \ \& \ \text{mless-or-equal}(X::'a, Y) \longrightarrow q2(X::'a, Y, Z)) \ \&$
 $(\forall X Y Z. q1(X::'a, Y, Z) \longrightarrow \text{mless-or-equal}(X::'a, Y) \mid q3(X::'a, Y, Z)) \ \&$
 $(\forall Z X Y. q2(X::'a, Y, Z) \longrightarrow q4(X::'a, Y, Y)) \ \&$
 $(\forall Z Y X. q3(X::'a, Y, Z) \longrightarrow q4(X::'a, Y, X)) \ \&$
 $(\forall X. \text{mless-or-equal}(X::'a, X)) \ \&$
 $(\forall X Y. \text{mless-or-equal}(X::'a, Y) \ \& \ \text{mless-or-equal}(Y::'a, X) \longrightarrow \text{equal}(X::'a, Y))$
 $\&$
 $(\forall Y X Z. \text{mless-or-equal}(X::'a, Y) \ \& \ \text{mless-or-equal}(Y::'a, Z) \longrightarrow \text{mless-or-equal}(X::'a, Z))$
 $\&$
 $(\forall Y X. \text{mless-or-equal}(X::'a, Y) \mid \text{mless-or-equal}(Y::'a, X)) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{mless-or-equal}(X::'a, Y)) \ \&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \ \& \ \text{mless-or-equal}(X::'a, Z) \longrightarrow \text{mless-or-equal}(Y::'a, Z))$
 $\&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \ \& \ \text{mless-or-equal}(Z::'a, X) \longrightarrow \text{mless-or-equal}(Z::'a, Y))$
 $\&$
 $(q1(a::'a, b, c)) \ \&$
 $(\forall W. \sim(q4(a::'a, b, W) \ \& \ \text{mless-or-equal}(a::'a, W) \ \& \ \text{mless-or-equal}(b::'a, W) \ \&$
 $\text{mless-or-equal}(W::'a, a))) \ \&$
 $(\forall W. \sim(q4(a::'a, b, W) \ \& \ \text{mless-or-equal}(a::'a, W) \ \& \ \text{mless-or-equal}(b::'a, W) \ \&$
 $\text{mless-or-equal}(W::'a, b))) \longrightarrow \text{False}$
 by meson

abbreviation SWV001-1-ax mless-THAN successor predecessor equal \equiv

$(\forall X. \text{equal}(\text{predecessor}(\text{successor}(X)), X)) \ \&$
 $(\forall X. \text{equal}(\text{successor}(\text{predecessor}(X)), X)) \ \&$
 $(\forall X Y. \text{equal}(\text{predecessor}(X), \text{predecessor}(Y)) \longrightarrow \text{equal}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{equal}(\text{successor}(X), \text{successor}(Y)) \longrightarrow \text{equal}(X::'a, Y)) \ \&$
 $(\forall X. \text{mless-THAN}(\text{predecessor}(X), X)) \ \&$
 $(\forall X. \text{mless-THAN}(X::'a, \text{successor}(X))) \ \&$
 $(\forall X Y Z. \text{mless-THAN}(X::'a, Y) \ \& \ \text{mless-THAN}(Y::'a, Z) \longrightarrow \text{mless-THAN}(X::'a, Z))$
 $\&$
 $(\forall X Y. \text{mless-THAN}(X::'a, Y) \mid \text{mless-THAN}(Y::'a, X) \mid \text{equal}(X::'a, Y)) \ \&$
 $(\forall X. \sim \text{mless-THAN}(X::'a, X)) \ \&$
 $(\forall Y X. \sim(\text{mless-THAN}(X::'a, Y) \ \& \ \text{mless-THAN}(Y::'a, X))) \ \&$
 $(\forall Y X Z. \text{equal}(X::'a, Y) \ \& \ \text{mless-THAN}(X::'a, Z) \longrightarrow \text{mless-THAN}(Y::'a, Z))$
 $\&$
 $(\forall Y Z X. \text{equal}(X::'a, Y) \ \& \ \text{mless-THAN}(Z::'a, X) \longrightarrow \text{mless-THAN}(Z::'a, Y))$

abbreviation SWV001-0-eq a successor predecessor equal \equiv

$(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{predecessor}(X), \text{predecessor}(Y))) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{successor}(X), \text{successor}(Y))) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(a(X), a(Y)))$

lemma *PRV003-1*:

EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
 $(\sim \text{mless-THAN}(n::'a,j))$ &
 $(\text{mless-THAN}(k::'a,j))$ &
 $(\sim \text{mless-THAN}(k::'a,i))$ &
 $(\text{mless-THAN}(i::'a,n))$ &
 $(\text{mless-THAN}(a(j),a(k)))$ &
 $(\forall X. \text{mless-THAN}(X::'a,j) \ \& \ \text{mless-THAN}(a(X),a(k)) \longrightarrow \text{mless-THAN}(X::'a,i))$
&
 $(\forall X. \text{mless-THAN}(\text{One}::'a,i) \ \& \ \text{mless-THAN}(a(X),a(\text{predecessor}(i))) \longrightarrow \text{mless-THAN}(X::'a,i)$
 $| \text{mless-THAN}(n::'a,X))$ &
 $(\forall X. \sim (\text{mless-THAN}(\text{One}::'a,X) \ \& \ \text{mless-THAN}(X::'a,i) \ \& \ \text{mless-THAN}(a(X),a(\text{predecessor}(X))))))$
&
 $(\text{mless-THAN}(j::'a,i)) \longrightarrow \text{False}$
by *meson*

lemma *PRV005-1*:

EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
 $(\sim \text{mless-THAN}(n::'a,k))$ &
 $(\sim \text{mless-THAN}(k::'a,l))$ &
 $(\sim \text{mless-THAN}(k::'a,i))$ &
 $(\text{mless-THAN}(l::'a,n))$ &
 $(\text{mless-THAN}(\text{One}::'a,l))$ &
 $(\text{mless-THAN}(a(k),a(\text{predecessor}(l))))$ &
 $(\forall X. \text{mless-THAN}(X::'a,\text{successor}(n)) \ \& \ \text{mless-THAN}(a(X),a(k)) \longrightarrow \text{mless-THAN}(X::'a,l))$
&
 $(\forall X. \text{mless-THAN}(\text{One}::'a,l) \ \& \ \text{mless-THAN}(a(X),a(\text{predecessor}(l))) \longrightarrow \text{mless-THAN}(X::'a,l)$
 $| \text{mless-THAN}(n::'a,X))$ &
 $(\forall X. \sim (\text{mless-THAN}(\text{One}::'a,X) \ \& \ \text{mless-THAN}(X::'a,l) \ \& \ \text{mless-THAN}(a(X),a(\text{predecessor}(X))))))$
 $\longrightarrow \text{False}$
by *meson*

lemma *PRV006-1*:

EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
 $(\sim \text{mless-THAN}(n::'a,m))$ &
 $(\text{mless-THAN}(i::'a,m))$ &
 $(\text{mless-THAN}(i::'a,n))$ &
 $(\sim \text{mless-THAN}(i::'a,\text{One}))$ &
 $(\text{mless-THAN}(a(i),a(m)))$ &

$(\forall X. \text{mless-THAN}(X::'a, \text{successor}(n)) \ \& \ \text{mless-THAN}(a(X), a(m)) \longrightarrow \text{mless-THAN}(X::'a, i))$
 $\&$
 $(\forall X. \text{mless-THAN}(\text{One}::'a, i) \ \& \ \text{mless-THAN}(a(X), a(\text{predecessor}(i))) \longrightarrow \text{mless-THAN}(X::'a, i)$
 $| \text{mless-THAN}(n::'a, X)) \ \&$
 $(\forall X. \sim(\text{mless-THAN}(\text{One}::'a, X) \ \& \ \text{mless-THAN}(X::'a, i) \ \& \ \text{mless-THAN}(a(X), a(\text{predecessor}(X)))))$
 $\longrightarrow \text{False}$
by meson

lemma PRV009-1:

$(\forall Y X. \text{mless-or-equal}(X::'a, Y) \ | \ \text{mless}(Y::'a, X)) \ \&$
 $(\text{mless}(j::'a, i)) \ \&$
 $(\text{mless-or-equal}(m::'a, p)) \ \&$
 $(\text{mless-or-equal}(p::'a, q)) \ \&$
 $(\text{mless-or-equal}(q::'a, n)) \ \&$
 $(\forall X Y. \text{mless-or-equal}(m::'a, X) \ \& \ \text{mless}(X::'a, i) \ \& \ \text{mless}(j::'a, Y) \ \& \ \text{mless-or-equal}(Y::'a, n)$
 $\longrightarrow \text{mless-or-equal}(a(X), a(Y))) \ \&$
 $(\forall X Y. \text{mless-or-equal}(m::'a, X) \ \& \ \text{mless-or-equal}(X::'a, Y) \ \& \ \text{mless-or-equal}(Y::'a, j)$
 $\longrightarrow \text{mless-or-equal}(a(X), a(Y))) \ \&$
 $(\forall X Y. \text{mless-or-equal}(i::'a, X) \ \& \ \text{mless-or-equal}(X::'a, Y) \ \& \ \text{mless-or-equal}(Y::'a, n)$
 $\longrightarrow \text{mless-or-equal}(a(X), a(Y))) \ \&$
 $(\sim \text{mless-or-equal}(a(p), a(q))) \longrightarrow \text{False}$
by meson

lemma PUZ012-1:

$(\forall X. \text{equal-fruits}(X::'a, X)) \ \&$
 $(\forall X. \text{equal-boxes}(X::'a, X)) \ \&$
 $(\forall X Y. \sim(\text{label}(X::'a, Y) \ \& \ \text{contains}(X::'a, Y))) \ \&$
 $(\forall X. \text{contains}(\text{boxa}::'a, X) \ | \ \text{contains}(\text{boxb}::'a, X) \ | \ \text{contains}(\text{boxc}::'a, X)) \ \&$
 $(\forall X. \text{contains}(X::'a, \text{apples}) \ | \ \text{contains}(X::'a, \text{bananas}) \ | \ \text{contains}(X::'a, \text{oranges}))$
 $\&$
 $(\forall X Y Z. \text{contains}(X::'a, Y) \ \& \ \text{contains}(X::'a, Z) \longrightarrow \text{equal-fruits}(Y::'a, Z)) \ \&$
 $(\forall Y X Z. \text{contains}(X::'a, Y) \ \& \ \text{contains}(Z::'a, Y) \longrightarrow \text{equal-boxes}(X::'a, Z)) \ \&$
 $(\sim \text{equal-boxes}(\text{boxa}::'a, \text{boxb})) \ \&$
 $(\sim \text{equal-boxes}(\text{boxb}::'a, \text{boxc})) \ \&$
 $(\sim \text{equal-boxes}(\text{boxa}::'a, \text{boxc})) \ \&$
 $(\sim \text{equal-fruits}(\text{apples}::'a, \text{bananas})) \ \&$
 $(\sim \text{equal-fruits}(\text{bananas}::'a, \text{oranges})) \ \&$
 $(\sim \text{equal-fruits}(\text{apples}::'a, \text{oranges})) \ \&$
 $(\text{label}(\text{boxa}::'a, \text{apples})) \ \&$
 $(\text{label}(\text{boxb}::'a, \text{oranges})) \ \&$
 $(\text{label}(\text{boxc}::'a, \text{bananas})) \ \&$
 $(\text{contains}(\text{boxb}::'a, \text{apples})) \ \&$
 $(\sim(\text{contains}(\text{boxa}::'a, \text{bananas}) \ \& \ \text{contains}(\text{boxc}::'a, \text{oranges}))) \longrightarrow \text{False}$
by meson

lemma PUZ020-1:

EQU001-0-ax equal &
 $(\forall A B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{statement-by}(A), \text{statement-by}(B))) \&$
 $(\forall X. \text{person}(X) \longrightarrow \text{knight}(X) \mid \text{knave}(X)) \&$
 $(\forall X. \sim(\text{person}(X) \& \text{knight}(X) \& \text{knave}(X))) \&$
 $(\forall X Y. \text{says}(X::'a, Y) \& \text{a-truth}(Y) \longrightarrow \text{a-truth}(Y)) \&$
 $(\forall X Y. \sim(\text{says}(X::'a, Y) \& \text{equal}(X::'a, Y))) \&$
 $(\forall Y X. \text{says}(X::'a, Y) \longrightarrow \text{equal}(Y::'a, \text{statement-by}(X))) \&$
 $(\forall X Y. \sim(\text{person}(X) \& \text{equal}(X::'a, \text{statement-by}(Y)))) \&$
 $(\forall X. \text{person}(X) \& \text{a-truth}(\text{statement-by}(X)) \longrightarrow \text{knight}(X)) \&$
 $(\forall X. \text{person}(X) \longrightarrow \text{a-truth}(\text{statement-by}(X)) \mid \text{knave}(X)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{knight}(X) \longrightarrow \text{knight}(Y)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{knave}(X) \longrightarrow \text{knave}(Y)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{person}(X) \longrightarrow \text{person}(Y)) \&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \& \text{says}(X::'a, Z) \longrightarrow \text{says}(Y::'a, Z)) \&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \& \text{says}(Z::'a, X) \longrightarrow \text{says}(Z::'a, Y)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{a-truth}(X) \longrightarrow \text{a-truth}(Y)) \&$
 $(\forall X Y. \text{knight}(X) \& \text{says}(X::'a, Y) \longrightarrow \text{a-truth}(Y)) \&$
 $(\forall X Y. \sim(\text{knave}(X) \& \text{says}(X::'a, Y) \& \text{a-truth}(Y))) \&$
 $(\text{person}(\text{husband})) \&$
 $(\text{person}(\text{wife})) \&$
 $(\sim \text{equal}(\text{husband}::'a, \text{wife})) \&$
 $(\text{says}(\text{husband}::'a, \text{statement-by}(\text{husband}))) \&$
 $(\text{a-truth}(\text{statement-by}(\text{husband})) \& \text{knight}(\text{husband}) \longrightarrow \text{knight}(\text{wife})) \&$
 $(\text{knight}(\text{husband}) \longrightarrow \text{a-truth}(\text{statement-by}(\text{husband}))) \&$
 $(\text{a-truth}(\text{statement-by}(\text{husband})) \mid \text{knight}(\text{wife})) \&$
 $(\text{knight}(\text{wife}) \longrightarrow \text{a-truth}(\text{statement-by}(\text{husband}))) \&$
 $(\sim \text{knight}(\text{husband})) \longrightarrow \text{False}$
by meson

lemma PUZ025-1:

$(\forall X. \text{a-truth}(\text{truthteller}(X)) \mid \text{a-truth}(\text{liar}(X))) \&$
 $(\forall X. \sim(\text{a-truth}(\text{truthteller}(X)) \& \text{a-truth}(\text{liar}(X)))) \&$
 $(\forall \text{Truthteller Statement. a-truth}(\text{truthteller}(\text{Truthteller})) \& \text{a-truth}(\text{says}(\text{Truthteller}::'a, \text{Statement})))$
 $\longrightarrow \text{a-truth}(\text{Statement})) \&$
 $(\forall \text{Liar Statement. } \sim(\text{a-truth}(\text{liar}(\text{Liar})) \& \text{a-truth}(\text{says}(\text{Liar}::'a, \text{Statement})) \&$
 $\text{a-truth}(\text{Statement}))) \&$
 $(\forall \text{Statement Truthteller. a-truth}(\text{Statement}) \& \text{a-truth}(\text{says}(\text{Truthteller}::'a, \text{Statement})))$
 $\longrightarrow \text{a-truth}(\text{truthteller}(\text{Truthteller}))) \&$
 $(\forall \text{Statement Liar. a-truth}(\text{says}(\text{Liar}::'a, \text{Statement})) \longrightarrow \text{a-truth}(\text{Statement}) \mid$
 $\text{a-truth}(\text{liar}(\text{Liar}))) \&$
 $(\forall Z X Y. \text{people}(X::'a, Y, Z) \& \text{a-truth}(\text{liar}(X)) \& \text{a-truth}(\text{liar}(Y)) \longrightarrow \text{a-truth}(\text{equal-type}(X::'a, Y)))$
 $\&$
 $(\forall Z X Y. \text{people}(X::'a, Y, Z) \& \text{a-truth}(\text{truthteller}(X)) \& \text{a-truth}(\text{truthteller}(Y)))$
 $\longrightarrow \text{a-truth}(\text{equal-type}(X::'a, Y))) \&$
 $(\forall X Y. \text{a-truth}(\text{equal-type}(X::'a, Y)) \& \text{a-truth}(\text{truthteller}(X)) \longrightarrow \text{a-truth}(\text{truthteller}(Y)))$
 $\&$
 $(\forall X Y. \text{a-truth}(\text{equal-type}(X::'a, Y)) \& \text{a-truth}(\text{liar}(X)) \longrightarrow \text{a-truth}(\text{liar}(Y)))$
 $\&$

$(\forall X Y. a\text{-truth}(\text{truthteller}(X)) \dashv\vdash a\text{-truth}(\text{equal-type}(X::'a, Y)) \mid a\text{-truth}(\text{liar}(Y)))$
 $\&$
 $(\forall X Y. a\text{-truth}(\text{liar}(X)) \dashv\vdash a\text{-truth}(\text{equal-type}(X::'a, Y)) \mid a\text{-truth}(\text{truthteller}(Y)))$
 $\&$
 $(\forall Y X. a\text{-truth}(\text{equal-type}(X::'a, Y)) \dashv\vdash a\text{-truth}(\text{equal-type}(Y::'a, X))) \&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{truthteller}(X)) \& a\text{-truth}(Y) \dashv\vdash \text{answer}(\text{yes})) \&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{truthteller}(X)) \dashv\vdash a\text{-truth}(Y) \mid \text{answer}(\text{no})) \&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{liar}(X)) \& a\text{-truth}(Y) \dashv\vdash \text{answer}(\text{no}))$
 $\&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{liar}(X)) \dashv\vdash a\text{-truth}(Y) \mid \text{answer}(\text{yes}))$
 $\&$
 $(\text{people}(b::'a, c, a)) \&$
 $(\text{people}(a::'a, b, a)) \&$
 $(\text{people}(a::'a, c, b)) \&$
 $(\text{people}(c::'a, b, a)) \&$
 $(a\text{-truth}(\text{says}(a::'a, \text{equal-type}(b::'a, c)))) \&$
 $(\text{ask-1-if-2}(c::'a, \text{equal-type}(a::'a, b))) \&$
 $(\forall \text{Answer}. \sim \text{answer}(\text{Answer})) \dashv\vdash \text{False}$
oops

lemma *PUZ029-1*:

$(\forall X. \text{dances-on-tightropes}(X) \mid \text{eats-pennybuns}(X) \mid \text{old}(X)) \&$
 $(\forall X. \text{pig}(X) \& \text{liable-to-giddiness}(X) \dashv\vdash \text{treated-with-respect}(X)) \&$
 $(\forall X. \text{wise}(X) \& \text{balloonist}(X) \dashv\vdash \text{has-umbrella}(X)) \&$
 $(\forall X. \sim(\text{looks-ridiculous}(X) \& \text{eats-pennybuns}(X) \& \text{eats-lunch-in-public}(X))) \&$
 $(\forall X. \text{balloonist}(X) \& \text{young}(X) \dashv\vdash \text{liable-to-giddiness}(X)) \&$
 $(\forall X. \text{fat}(X) \& \text{looks-ridiculous}(X) \dashv\vdash \text{dances-on-tightropes}(X) \mid \text{eats-lunch-in-public}(X))$
 $\&$
 $(\forall X. \sim(\text{liable-to-giddiness}(X) \& \text{wise}(X) \& \text{dances-on-tightropes}(X))) \&$
 $(\forall X. \text{pig}(X) \& \text{has-umbrella}(X) \dashv\vdash \text{looks-ridiculous}(X)) \&$
 $(\forall X. \text{treated-with-respect}(X) \dashv\vdash \text{dances-on-tightropes}(X) \mid \text{fat}(X)) \&$
 $(\forall X. \text{young}(X) \mid \text{old}(X)) \&$
 $(\forall X. \sim(\text{young}(X) \& \text{old}(X))) \&$
 $(\text{wise}(\text{piggy})) \&$
 $(\text{young}(\text{piggy})) \&$
 $(\text{pig}(\text{piggy})) \&$
 $(\text{balloonist}(\text{piggy})) \dashv\vdash \text{False}$
by *meson*

abbreviation *RNG001-0-ax* equal additive-inverse add multiply product additive-identity

$\text{sum} \equiv$
 $(\forall X. \text{sum}(\text{additive-identity}::'a, X, X)) \&$
 $(\forall X. \text{sum}(X::'a, \text{additive-identity}, X)) \&$
 $(\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \&$
 $(\forall X Y. \text{sum}(X::'a, Y, \text{add}(X::'a, Y))) \&$

$(\forall X. \text{sum}(\text{additive-inverse}(X), X, \text{additive-identity})) \ \&$
 $(\forall X. \text{sum}(X::'a, \text{additive-inverse}(X), \text{additive-identity})) \ \&$
 $(\forall Y \ U \ Z \ X \ V \ W. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(Y::'a, Z, V) \ \& \ \text{sum}(U::'a, Z, W) \ \longrightarrow$
 $\text{sum}(X::'a, V, W)) \ \&$
 $(\forall Y \ X \ V \ U \ Z \ W. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(Y::'a, Z, V) \ \& \ \text{sum}(X::'a, V, W) \ \longrightarrow$
 $\text{sum}(U::'a, Z, W)) \ \&$
 $(\forall Y \ X \ Z. \text{sum}(X::'a, Y, Z) \ \longrightarrow \ \text{sum}(Y::'a, X, Z)) \ \&$
 $(\forall Y \ U \ Z \ X \ V \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(U::'a, Z, W)$
 $\longrightarrow \ \text{product}(X::'a, V, W)) \ \&$
 $(\forall Y \ X \ V \ U \ Z \ W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(X::'a, V, W)$
 $\longrightarrow \ \text{product}(U::'a, Z, W)) \ \&$
 $(\forall Y \ Z \ X \ V3 \ V1 \ V2 \ V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(X::'a, V3, V4) \ \longrightarrow \ \text{sum}(V1::'a, V2, V4)) \ \&$
 $(\forall Y \ Z \ V1 \ V2 \ X \ V3 \ V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(V1::'a, V2, V4) \ \longrightarrow \ \text{product}(X::'a, V3, V4)) \ \&$
 $(\forall Y \ Z \ V3 \ X \ V1 \ V2 \ V4. \text{product}(Y::'a, X, V1) \ \& \ \text{product}(Z::'a, X, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(V3::'a, X, V4) \ \longrightarrow \ \text{sum}(V1::'a, V2, V4)) \ \&$
 $(\forall Y \ Z \ V1 \ V2 \ V3 \ X \ V4. \text{product}(Y::'a, X, V1) \ \& \ \text{product}(Z::'a, X, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(V1::'a, V2, V4) \ \longrightarrow \ \text{product}(V3::'a, X, V4)) \ \&$
 $(\forall X \ Y \ U \ V. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(X::'a, Y, V) \ \longrightarrow \ \text{equal}(U::'a, V)) \ \&$
 $(\forall X \ Y \ U \ V. \text{product}(X::'a, Y, U) \ \& \ \text{product}(X::'a, Y, V) \ \longrightarrow \ \text{equal}(U::'a, V))$

abbreviation *RNG001-0-eq product multiply sum add additive-inverse equal* \equiv
 $(\forall X \ Y. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{additive-inverse}(X), \text{additive-inverse}(Y))) \ \&$
 $(\forall X \ Y \ W. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{add}(X::'a, W), \text{add}(Y::'a, W))) \ \&$
 $(\forall X \ W \ Y. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{add}(W::'a, X), \text{add}(W::'a, Y))) \ \&$
 $(\forall X \ Y \ W \ Z. \text{equal}(X::'a, Y) \ \& \ \text{sum}(X::'a, W, Z) \ \longrightarrow \ \text{sum}(Y::'a, W, Z)) \ \&$
 $(\forall X \ W \ Y \ Z. \text{equal}(X::'a, Y) \ \& \ \text{sum}(W::'a, X, Z) \ \longrightarrow \ \text{sum}(W::'a, Y, Z)) \ \&$
 $(\forall X \ W \ Z \ Y. \text{equal}(X::'a, Y) \ \& \ \text{sum}(W::'a, Z, X) \ \longrightarrow \ \text{sum}(W::'a, Z, Y)) \ \&$
 $(\forall X \ Y \ W. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{multiply}(X::'a, W), \text{multiply}(Y::'a, W)))$
 $\ \&$
 $(\forall X \ W \ Y. \text{equal}(X::'a, Y) \ \longrightarrow \ \text{equal}(\text{multiply}(W::'a, X), \text{multiply}(W::'a, Y)))$
 $\ \&$
 $(\forall X \ Y \ W \ Z. \text{equal}(X::'a, Y) \ \& \ \text{product}(X::'a, W, Z) \ \longrightarrow \ \text{product}(Y::'a, W, Z))$
 $\ \&$
 $(\forall X \ W \ Y \ Z. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, X, Z) \ \longrightarrow \ \text{product}(W::'a, Y, Z))$
 $\ \&$
 $(\forall X \ W \ Z \ Y. \text{equal}(X::'a, Y) \ \& \ \text{product}(W::'a, Z, X) \ \longrightarrow \ \text{product}(W::'a, Z, Y))$

lemma *RNG001-3:*

$(\forall X. \text{sum}(\text{additive-identity}::'a, X, X)) \ \&$
 $(\forall X. \text{sum}(\text{additive-inverse}(X), X, \text{additive-identity})) \ \&$
 $(\forall Y \ U \ Z \ X \ V \ W. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(Y::'a, Z, V) \ \& \ \text{sum}(U::'a, Z, W) \ \longrightarrow$
 $\text{sum}(X::'a, V, W)) \ \&$
 $(\forall Y \ X \ V \ U \ Z \ W. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(Y::'a, Z, V) \ \& \ \text{sum}(X::'a, V, W) \ \longrightarrow$
 $\text{sum}(U::'a, Z, W)) \ \&$
 $(\forall X \ Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \ \&$
 $(\forall Y \ Z \ X \ V3 \ V1 \ V2 \ V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$

$\& \text{product}(X::'a, V3, V4) \longrightarrow \text{sum}(V1::'a, V2, V4)) \&$
 $(\forall Y Z V1 V2 X V3 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{sum}(V1::'a, V2, V4) \longrightarrow \text{product}(X::'a, V3, V4)) \&$
 $(\sim \text{product}(a::'a, \text{additive-identity}, \text{additive-identity})) \longrightarrow \text{False}$
oops

abbreviation *RNG-other-ax multiply add equal product additive-identity additive-inverse*

$\text{sum} \equiv$
 $(\forall X. \text{sum}(X::'a, \text{additive-inverse}(X), \text{additive-identity})) \&$
 $(\forall Y U Z X V W. \text{sum}(X::'a, Y, U) \& \text{sum}(Y::'a, Z, V) \& \text{sum}(U::'a, Z, W) \longrightarrow$
 $\text{sum}(X::'a, V, W)) \&$
 $(\forall Y X V U Z W. \text{sum}(X::'a, Y, U) \& \text{sum}(Y::'a, Z, V) \& \text{sum}(X::'a, V, W) \longrightarrow$
 $\text{sum}(U::'a, Z, W)) \&$
 $(\forall Y X Z. \text{sum}(X::'a, Y, Z) \longrightarrow \text{sum}(Y::'a, X, Z)) \&$
 $(\forall Y U Z X V W. \text{product}(X::'a, Y, U) \& \text{product}(Y::'a, Z, V) \& \text{product}(U::'a, Z, W)$
 $\longrightarrow \text{product}(X::'a, V, W)) \&$
 $(\forall Y X V U Z W. \text{product}(X::'a, Y, U) \& \text{product}(Y::'a, Z, V) \& \text{product}(X::'a, V, W)$
 $\longrightarrow \text{product}(U::'a, Z, W)) \&$
 $(\forall Y Z X V3 V1 V2 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{product}(X::'a, V3, V4) \longrightarrow \text{sum}(V1::'a, V2, V4)) \&$
 $(\forall Y Z V1 V2 X V3 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{sum}(V1::'a, V2, V4) \longrightarrow \text{product}(X::'a, V3, V4)) \&$
 $(\forall Y Z V3 X V1 V2 V4. \text{product}(Y::'a, X, V1) \& \text{product}(Z::'a, X, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{product}(V3::'a, X, V4) \longrightarrow \text{sum}(V1::'a, V2, V4)) \&$
 $(\forall Y Z V1 V2 V3 X V4. \text{product}(Y::'a, X, V1) \& \text{product}(Z::'a, X, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{sum}(V1::'a, V2, V4) \longrightarrow \text{product}(V3::'a, X, V4)) \&$
 $(\forall X Y U V. \text{sum}(X::'a, Y, U) \& \text{sum}(X::'a, Y, V) \longrightarrow \text{equal}(U::'a, V)) \&$
 $(\forall X Y U V. \text{product}(X::'a, Y, U) \& \text{product}(X::'a, Y, V) \longrightarrow \text{equal}(U::'a, V))$
 $\&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{additive-inverse}(X), \text{additive-inverse}(Y))) \&$
 $(\forall X Y W. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{add}(X::'a, W), \text{add}(Y::'a, W))) \&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{sum}(X::'a, W, Z) \longrightarrow \text{sum}(Y::'a, W, Z)) \&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{sum}(W::'a, X, Z) \longrightarrow \text{sum}(W::'a, Y, Z)) \&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{sum}(W::'a, Z, X) \longrightarrow \text{sum}(W::'a, Z, Y)) \&$
 $(\forall X Y W. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{multiply}(X::'a, W), \text{multiply}(Y::'a, W)))$
 $\&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{product}(X::'a, W, Z) \longrightarrow \text{product}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{product}(W::'a, X, Z) \longrightarrow \text{product}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{product}(W::'a, Z, X) \longrightarrow \text{product}(W::'a, Z, Y))$

lemma *RNG001-5:*

$\text{EQU001-0-ax equal} \&$
 $(\forall X. \text{sum}(\text{additive-identity}::'a, X, X)) \&$
 $(\forall X. \text{sum}(X::'a, \text{additive-identity}, X)) \&$
 $(\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \&$

$(\forall X Y. \text{sum}(X::'a, Y, \text{add}(X::'a, Y))) \ \&$
 $(\forall X. \text{sum}(\text{additive-inverse}(X), X, \text{additive-identity})) \ \&$
 $\text{RNG-other-ax multiply add equal product additive-identity additive-inverse sum}$
 $\&$
 $(\sim \text{product}(a::'a, \text{additive-identity}, \text{additive-identity})) \ \longrightarrow \text{False}$
oops

lemma RNG011-5:

$\text{EQU001-0-ax equal} \ \&$
 $(\forall A B C. \text{equal}(A::'a, B) \ \longrightarrow \ \text{equal}(\text{add}(A::'a, C), \text{add}(B::'a, C))) \ \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \ \longrightarrow \ \text{equal}(\text{add}(F'::'a, D), \text{add}(F'::'a, E))) \ \&$
 $(\forall G H. \text{equal}(G::'a, H) \ \longrightarrow \ \text{equal}(\text{additive-inverse}(G), \text{additive-inverse}(H))) \ \&$
 $(\forall I' J K'. \text{equal}(I'::'a, J) \ \longrightarrow \ \text{equal}(\text{multiply}(I'::'a, K'), \text{multiply}(J::'a, K'))) \ \&$
 $(\forall L N M. \text{equal}(L::'a, M) \ \longrightarrow \ \text{equal}(\text{multiply}(N::'a, L), \text{multiply}(N::'a, M))) \ \&$
 $(\forall A B C D. \text{equal}(A::'a, B) \ \longrightarrow \ \text{equal}(\text{associator}(A::'a, C, D), \text{associator}(B::'a, C, D)))$
 $\&$
 $(\forall E G F' H. \text{equal}(E::'a, F') \ \longrightarrow \ \text{equal}(\text{associator}(G::'a, E, H), \text{associator}(G::'a, F', H)))$
 $\&$
 $(\forall I' K' L J. \text{equal}(I'::'a, J) \ \longrightarrow \ \text{equal}(\text{associator}(K'::'a, L, I'), \text{associator}(K'::'a, L, J)))$
 $\&$
 $(\forall M N O'. \text{equal}(M::'a, N) \ \longrightarrow \ \text{equal}(\text{commutator}(M::'a, O'), \text{commutator}(N::'a, O')))$
 $\&$
 $(\forall P R Q. \text{equal}(P::'a, Q) \ \longrightarrow \ \text{equal}(\text{commutator}(R::'a, P), \text{commutator}(R::'a, Q)))$
 $\&$
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X))) \ \&$
 $(\forall X Y Z. \text{equal}(\text{add}(\text{add}(X::'a, Y), Z), \text{add}(X::'a, \text{add}(Y::'a, Z)))) \ \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-identity}), X)) \ \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-identity}::'a, X), X)) \ \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-inverse}(X)), \text{additive-identity})) \ \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-inverse}(X), X), \text{additive-identity})) \ \&$
 $(\text{equal}(\text{additive-inverse}(\text{additive-identity}), \text{additive-identity})) \ \&$
 $(\forall X Y. \text{equal}(\text{add}(X::'a, \text{add}(\text{additive-inverse}(X), Y)), Y)) \ \&$
 $(\forall X Y. \text{equal}(\text{additive-inverse}(\text{add}(X::'a, Y)), \text{add}(\text{additive-inverse}(X), \text{additive-inverse}(Y))))$
 $\&$
 $(\forall X. \text{equal}(\text{additive-inverse}(\text{additive-inverse}(X)), X)) \ \&$
 $(\forall X. \text{equal}(\text{multiply}(X::'a, \text{additive-identity}), \text{additive-identity})) \ \&$
 $(\forall X. \text{equal}(\text{multiply}(\text{additive-identity}::'a, X), \text{additive-identity})) \ \&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{additive-inverse}(X), \text{additive-inverse}(Y)), \text{multiply}(X::'a, Y)))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(X::'a, \text{additive-inverse}(Y)), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{additive-inverse}(X), Y), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall Y X Z. \text{equal}(\text{multiply}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{multiply}(\text{add}(X::'a, Y), Z), \text{add}(\text{multiply}(X::'a, Z), \text{multiply}(Y::'a, Z))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, Y), Y), \text{multiply}(X::'a, \text{multiply}(Y::'a, Y))))$

$\&$
 $(\forall X Y Z. \text{equal}(\text{associator}(X::'a, Y, Z), \text{add}(\text{multiply}(\text{multiply}(X::'a, Y), Z), \text{additive-inverse}(\text{multiply}(X::'a, Y, Z))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{commutator}(X::'a, Y), \text{add}(\text{multiply}(Y::'a, X), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(\text{associator}(X::'a, X, Y), X), \text{associator}(X::'a, X, Y)), \text{additive-identity}))$
 $\&$
 $(\sim \text{equal}(\text{multiply}(\text{multiply}(\text{associator}(a::'a, a, b), a), \text{associator}(a::'a, a, b)), \text{additive-identity}))$
 $\longrightarrow \text{False}$
by *meson*

lemma *RNG023-6*:

$\text{EQU001-0-ax equal} \&$
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X))) \&$
 $(\forall X Y Z. \text{equal}(\text{add}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{add}(X::'a, Y), Z))) \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-identity}::'a, X), X)) \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-identity}), X)) \&$
 $(\forall X. \text{equal}(\text{multiply}(\text{additive-identity}::'a, X), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{multiply}(X::'a, \text{additive-identity}), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-inverse}(X), X), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-inverse}(X)), \text{additive-identity})) \&$
 $(\forall Y X Z. \text{equal}(\text{multiply}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{multiply}(\text{add}(X::'a, Y), Z), \text{add}(\text{multiply}(X::'a, Z), \text{multiply}(Y::'a, Z))))$
 $\&$
 $(\forall X. \text{equal}(\text{additive-inverse}(\text{additive-inverse}(X)), X)) \&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, Y), Y), \text{multiply}(X::'a, \text{multiply}(Y::'a, Y))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, X), Y), \text{multiply}(X::'a, \text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{associator}(X::'a, Y, Z), \text{add}(\text{multiply}(\text{multiply}(X::'a, Y), Z), \text{additive-inverse}(\text{multiply}(X::'a, Y, Z))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{commutator}(X::'a, Y), \text{add}(\text{multiply}(Y::'a, X), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall D E F'. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{add}(D::'a, F'), \text{add}(E::'a, F'))) \&$
 $(\forall G I' H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{add}(I'::'a, G), \text{add}(I'::'a, H))) \&$
 $(\forall J K'. \text{equal}(J::'a, K') \longrightarrow \text{equal}(\text{additive-inverse}(J), \text{additive-inverse}(K'))) \&$
 $(\forall L M N O'. \text{equal}(L::'a, M) \longrightarrow \text{equal}(\text{associator}(L::'a, N, O'), \text{associator}(M::'a, N, O')))$
 $\&$
 $(\forall P R Q S'. \text{equal}(P::'a, Q) \longrightarrow \text{equal}(\text{associator}(R::'a, P, S'), \text{associator}(R::'a, Q, S')))$
 $\&$
 $(\forall T' V W U. \text{equal}(T'::'a, U) \longrightarrow \text{equal}(\text{associator}(V::'a, W, T'), \text{associator}(V::'a, W, U)))$
 $\&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{commutator}(X::'a, Z), \text{commutator}(Y::'a, Z)))$
 $\&$
 $(\forall A1 C1 B1. \text{equal}(A1::'a, B1) \longrightarrow \text{equal}(\text{commutator}(C1::'a, A1), \text{commutator}(C1::'a, B1)))$
 $\&$
 $(\forall D1 E1 F1. \text{equal}(D1::'a, E1) \longrightarrow \text{equal}(\text{multiply}(D1::'a, F1), \text{multiply}(E1::'a, F1)))$

$\&$
 $(\forall G1\ I1\ H1. \text{equal}(G1::'a, H1) \longrightarrow \text{equal}(\text{multiply}(I1::'a, G1), \text{multiply}(I1::'a, H1)))$
 $\&$
 $(\sim \text{equal}(\text{associator}(x::'a, x, y), \text{additive-identity})) \longrightarrow \text{False}$
by meson

lemma RNG028-2:

$EQU001-0\text{-ax equal \&}$
 $(\forall X. \text{equal}(\text{add}(\text{additive-identity}::'a, X), X)) \&$
 $(\forall X. \text{equal}(\text{multiply}(\text{additive-identity}::'a, X), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{multiply}(X::'a, \text{additive-identity}), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-inverse}(X), X), \text{additive-identity})) \&$
 $(\forall X\ Y. \text{equal}(\text{additive-inverse}(\text{add}(X::'a, Y)), \text{add}(\text{additive-inverse}(X), \text{additive-inverse}(Y))))$
 $\&$
 $(\forall X. \text{equal}(\text{additive-inverse}(\text{additive-inverse}(X)), X)) \&$
 $(\forall Y\ X\ Z. \text{equal}(\text{multiply}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall X\ Y\ Z. \text{equal}(\text{multiply}(\text{add}(X::'a, Y), Z), \text{add}(\text{multiply}(X::'a, Z), \text{multiply}(Y::'a, Z))))$
 $\&$
 $(\forall X\ Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, Y), Y), \text{multiply}(X::'a, \text{multiply}(Y::'a, Y))))$
 $\&$
 $(\forall X\ Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, X), Y), \text{multiply}(X::'a, \text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X\ Y. \text{equal}(\text{multiply}(\text{additive-inverse}(X), Y), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X\ Y. \text{equal}(\text{multiply}(X::'a, \text{additive-inverse}(Y)), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\text{equal}(\text{additive-inverse}(\text{additive-identity}), \text{additive-identity})) \&$
 $(\forall Y\ X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X))) \&$
 $(\forall X\ Y\ Z. \text{equal}(\text{add}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{add}(X::'a, Y), Z))) \&$
 $(\forall Z\ X\ Y. \text{equal}(\text{add}(X::'a, Z), \text{add}(Y::'a, Z)) \longrightarrow \text{equal}(X::'a, Y)) \&$
 $(\forall Z\ X\ Y. \text{equal}(\text{add}(Z::'a, X), \text{add}(Z::'a, Y)) \longrightarrow \text{equal}(X::'a, Y)) \&$
 $(\forall D\ E\ F'. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{add}(D::'a, F'), \text{add}(E::'a, F'))) \&$
 $(\forall G\ I'\ H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{add}(I::'a, G), \text{add}(I::'a, H))) \&$
 $(\forall J\ K'. \text{equal}(J::'a, K') \longrightarrow \text{equal}(\text{additive-inverse}(J), \text{additive-inverse}(K'))) \&$
 $(\forall D1\ E1\ F1. \text{equal}(D1::'a, E1) \longrightarrow \text{equal}(\text{multiply}(D1::'a, F1), \text{multiply}(E1::'a, F1)))$
 $\&$
 $(\forall G1\ I1\ H1. \text{equal}(G1::'a, H1) \longrightarrow \text{equal}(\text{multiply}(I1::'a, G1), \text{multiply}(I1::'a, H1)))$
 $\&$
 $(\forall X\ Y\ Z. \text{equal}(\text{associator}(X::'a, Y, Z), \text{add}(\text{multiply}(\text{multiply}(X::'a, Y), Z), \text{additive-inverse}(\text{multiply}(X::'a, m$
 $\&$
 $(\forall L\ M\ N\ O'. \text{equal}(L::'a, M) \longrightarrow \text{equal}(\text{associator}(L::'a, N, O'), \text{associator}(M::'a, N, O')))$
 $\&$
 $(\forall P\ R\ Q\ S'. \text{equal}(P::'a, Q) \longrightarrow \text{equal}(\text{associator}(R::'a, P, S'), \text{associator}(R::'a, Q, S')))$
 $\&$
 $(\forall T'\ V\ W\ U. \text{equal}(T::'a, U) \longrightarrow \text{equal}(\text{associator}(V::'a, W, T'), \text{associator}(V::'a, W, U)))$
 $\&$
 $(\forall X\ Y. \sim \text{equal}(\text{multiply}(\text{multiply}(Y::'a, X), Y), \text{multiply}(Y::'a, \text{multiply}(X::'a, Y))))$

&
 ($\forall X Y Z. \sim \text{equal}(\text{associator}(Y::'a, X, Z), \text{additive-inverse}(\text{associator}(X::'a, Y, Z)))$)
 &
 ($\forall X Y Z. \sim \text{equal}(\text{associator}(Z::'a, Y, X), \text{additive-inverse}(\text{associator}(X::'a, Y, Z)))$)
 &
 ($\sim \text{equal}(\text{multiply}(\text{multiply}(cx::'a, \text{multiply}(cy::'a, cx)), cz), \text{multiply}(cx::'a, \text{multiply}(cy::'a, \text{multiply}(cx::'a, cz))))$)
 $\longrightarrow \text{False}$
 by meson

lemma *RNG038-2:*

($\forall X. \text{sum}(X::'a, \text{additive-identity}, X)$) &
 ($\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))$) &
 ($\forall X Y. \text{sum}(X::'a, Y, \text{add}(X::'a, Y))$) &
RNG-other-ax multiply add equal product additive-identity additive-inverse sum
 &
 ($\forall X. \text{product}(\text{additive-identity}::'a, X, \text{additive-identity})$) &
 ($\forall X. \text{product}(X::'a, \text{additive-identity}, \text{additive-identity})$) &
 ($\forall X Y. \text{equal}(X::'a, \text{additive-identity}) \longrightarrow \text{product}(X::'a, h(X::'a, Y), Y)$) &
 ($\text{product}(a::'a, b, \text{additive-identity})$) &
 ($\sim \text{equal}(a::'a, \text{additive-identity})$) &
 ($\sim \text{equal}(b::'a, \text{additive-identity})$) $\longrightarrow \text{False}$
 by meson

lemma *RNG040-2:*

EQU001-0-ax equal &
RNG001-0-eq product multiply sum add additive-inverse equal &
 ($\forall X. \text{sum}(\text{additive-identity}::'a, X, X)$) &
 ($\forall X. \text{sum}(X::'a, \text{additive-identity}, X)$) &
 ($\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))$) &
 ($\forall X Y. \text{sum}(X::'a, Y, \text{add}(X::'a, Y))$) &
 ($\forall X. \text{sum}(\text{additive-inverse}(X), X, \text{additive-identity})$) &
 ($\forall X. \text{sum}(X::'a, \text{additive-inverse}(X), \text{additive-identity})$) &
 ($\forall Y U Z X V W. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(Y::'a, Z, V) \ \& \ \text{sum}(U::'a, Z, W) \longrightarrow$
 $\text{sum}(X::'a, V, W))$ &
 ($\forall Y X V U Z W. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(Y::'a, Z, V) \ \& \ \text{sum}(X::'a, V, W) \longrightarrow$
 $\text{sum}(U::'a, Z, W))$ &
 ($\forall Y X Z. \text{sum}(X::'a, Y, Z) \longrightarrow \text{sum}(Y::'a, X, Z)$) &
 ($\forall Y U Z X V W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(U::'a, Z, W)$
 $\longrightarrow \text{product}(X::'a, V, W))$ &
 ($\forall Y X V U Z W. \text{product}(X::'a, Y, U) \ \& \ \text{product}(Y::'a, Z, V) \ \& \ \text{product}(X::'a, V, W)$
 $\longrightarrow \text{product}(U::'a, Z, W))$ &
 ($\forall Y Z X V3 V1 V2 V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{product}(X::'a, V3, V4) \longrightarrow \text{sum}(V1::'a, V2, V4))$ &
 ($\forall Y Z V1 V2 X V3 V4. \text{product}(X::'a, Y, V1) \ \& \ \text{product}(X::'a, Z, V2) \ \& \ \text{sum}(Y::'a, Z, V3)$
 $\ \& \ \text{sum}(V1::'a, V2, V4) \longrightarrow \text{product}(X::'a, V3, V4))$ &
 ($\forall X Y U V. \text{sum}(X::'a, Y, U) \ \& \ \text{sum}(X::'a, Y, V) \longrightarrow \text{equal}(U::'a, V)$) &
 ($\forall X Y U V. \text{product}(X::'a, Y, U) \ \& \ \text{product}(X::'a, Y, V) \longrightarrow \text{equal}(U::'a, V)$)

&
 (∀ A. product(A::*a*,multiplicative-identity,A)) &
 (∀ A. product(multiplicative-identity::*a*,A,A)) &
 (∀ A. product(A::*a*,h(A),multiplicative-identity) | equal(A::*a*,additive-identity))
 &
 (∀ A. product(h(A),A,multiplicative-identity) | equal(A::*a*,additive-identity)) &
 (∀ B A C. product(A::*a*,B,C) ==> product(B::*a*,A,C)) &
 (∀ A B. equal(A::*a*,B) ==> equal(h(A),h(B))) &
 (sum(b::*a*,c,d)) &
 (product(d::*a*,a,additive-identity)) &
 (product(b::*a*,a,l)) &
 (product(c::*a*,a,n)) &
 (~sum(l::*a*,n,additive-identity)) ==> False
 by meson

lemma *RNG041-1*:

EQU001-0-ax equal &
 RNG001-0-ax equal additive-inverse add multiply product additive-identity sum &
 RNG001-0-eq product multiply sum add additive-inverse equal &
 (∀ A B. equal(A::*a*,B) ==> equal(h(A),h(B))) &
 (∀ A. product(additive-identity::*a*,A,additive-identity)) &
 (∀ A. product(A::*a*,additive-identity,additive-identity)) &
 (∀ A. product(A::*a*,multiplicative-identity,A)) &
 (∀ A. product(multiplicative-identity::*a*,A,A)) &
 (∀ A. product(A::*a*,h(A),multiplicative-identity) | equal(A::*a*,additive-identity))
 &
 (∀ A. product(h(A),A,multiplicative-identity) | equal(A::*a*,additive-identity)) &
 (product(a::*a*,b,additive-identity)) &
 (~equal(a::*a*,additive-identity)) &
 (~equal(b::*a*,additive-identity)) ==> False
 oops

lemma *ROB010-1*:

EQU001-0-ax equal &
 (∀ Y X. equal(add(X::*a*,Y),add(Y::*a*,X))) &
 (∀ X Y Z. equal(add(add(X::*a*,Y),Z),add(X::*a*,add(Y::*a*,Z)))) &
 (∀ Y X. equal(negate(add(negate(add(X::*a*,Y)),negate(add(X::*a*,negate(Y))))),X))
 &
 (∀ A B C. equal(A::*a*,B) ==> equal(add(A::*a*,C),add(B::*a*,C))) &
 (∀ D F' E. equal(D::*a*,E) ==> equal(add(F'::*a*,D),add(F'::*a*,E))) &
 (∀ G H. equal(G::*a*,H) ==> equal(negate(G),negate(H))) &
 (equal(negate(add(a::*a*,negate(b))),c)) &
 (~equal(negate(add(c::*a*,negate(add(b::*a*,a))))),a)) ==> False
 oops

lemma ROB013-1:

EQU001-0-ax equal &
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X)))$ &
 $(\forall X Y Z. \text{equal}(\text{add}(\text{add}(X::'a, Y), Z), \text{add}(X::'a, \text{add}(Y::'a, Z))))$ &
 $(\forall Y X. \text{equal}(\text{negate}(\text{add}(\text{negate}(\text{add}(X::'a, Y)), \text{negate}(\text{add}(X::'a, \text{negate}(Y)))))), X))$
&
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{add}(A::'a, C), \text{add}(B::'a, C)))$ &
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{add}(F'::'a, D), \text{add}(F'::'a, E)))$ &
 $(\forall G H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{negate}(G), \text{negate}(H)))$ &
 $(\text{equal}(\text{negate}(\text{add}(a::'a, b)), c))$ &
 $(\sim \text{equal}(\text{negate}(\text{add}(c::'a, \text{negate}(\text{add}(\text{negate}(b), a)))), a)) \longrightarrow \text{False}$
by meson

lemma ROB016-1:

EQU001-0-ax equal &
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X)))$ &
 $(\forall X Y Z. \text{equal}(\text{add}(\text{add}(X::'a, Y), Z), \text{add}(X::'a, \text{add}(Y::'a, Z))))$ &
 $(\forall Y X. \text{equal}(\text{negate}(\text{add}(\text{negate}(\text{add}(X::'a, Y)), \text{negate}(\text{add}(X::'a, \text{negate}(Y)))))), X))$
&
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{add}(A::'a, C), \text{add}(B::'a, C)))$ &
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{add}(F'::'a, D), \text{add}(F'::'a, E)))$ &
 $(\forall G H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{negate}(G), \text{negate}(H)))$ &
 $(\forall J K' L. \text{equal}(J::'a, K') \longrightarrow \text{equal}(\text{multiply}(J::'a, L), \text{multiply}(K'::'a, L)))$ &
 $(\forall M O' N. \text{equal}(M::'a, N) \longrightarrow \text{equal}(\text{multiply}(O'::'a, M), \text{multiply}(O'::'a, N)))$
&
 $(\forall P Q. \text{equal}(P::'a, Q) \longrightarrow \text{equal}(\text{successor}(P), \text{successor}(Q)))$ &
 $(\forall R S'. \text{equal}(R::'a, S') \& \text{positive-integer}(R) \longrightarrow \text{positive-integer}(S'))$ &
 $(\forall X. \text{equal}(\text{multiply}(\text{One}::'a, X), X))$ &
 $(\forall V X. \text{positive-integer}(X) \longrightarrow \text{equal}(\text{multiply}(\text{successor}(V), X), \text{add}(X::'a, \text{multiply}(V::'a, X))))$
&
 $(\text{positive-integer}(\text{One}))$ &
 $(\forall X. \text{positive-integer}(X) \longrightarrow \text{positive-integer}(\text{successor}(X)))$ &
 $(\text{equal}(\text{negate}(\text{add}(d::'a, e)), \text{negate}(e)))$ &
 $(\text{positive-integer}(k))$ &
 $(\forall V k X Y. \text{equal}(\text{negate}(\text{add}(\text{negate}(Y), \text{negate}(\text{add}(X::'a, \text{negate}(Y)))))), X)$ &
 $\text{positive-integer}(V k) \longrightarrow \text{equal}(\text{negate}(\text{add}(Y::'a, \text{multiply}(V k::'a, \text{add}(X::'a, \text{negate}(\text{add}(X::'a, \text{negate}(Y)))))),$
&
 $(\sim \text{equal}(\text{negate}(\text{add}(e::'a, \text{multiply}(k::'a, \text{add}(d::'a, \text{negate}(\text{add}(d::'a, \text{negate}(e)))))), \text{negate}(e)))$
 $\longrightarrow \text{False}$
oops

lemma ROB021-1:

EQU001-0-ax equal &
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X)))$ &
 $(\forall X Y Z. \text{equal}(\text{add}(\text{add}(X::'a, Y), Z), \text{add}(X::'a, \text{add}(Y::'a, Z))))$ &
 $(\forall Y X. \text{equal}(\text{negate}(\text{add}(\text{negate}(\text{add}(X::'a, Y)), \text{negate}(\text{add}(X::'a, \text{negate}(Y)))))), X))$
&

```

(∀ A B C. equal(A::'a,B) --> equal(add(A::'a,C),add(B::'a,C))) &
(∀ D F' E. equal(D::'a,E) --> equal(add(F'::'a,D),add(F'::'a,E))) &
(∀ G H. equal(G::'a,H) --> equal(negate(G),negate(H))) &
(∀ X Y. equal(negate(X),negate(Y)) --> equal(X::'a,Y)) &
(∼ equal(add(negate(add(a::'a,negate(b))),negate(add(negate(a),negate(b)))),b))
--> False
oops

```

lemma SET005-1:

```

(∀ Subset Element Superset. member(Element::'a,Subset) & subset(Subset::'a,Superset)
--> member(Element::'a,Superset)) &
(∀ Superset Subset. subset(Subset::'a,Superset) | member(member-of-1-not-of-2(Subset::'a,Superset),Subset))
&
(∀ Subset Superset. member(member-of-1-not-of-2(Subset::'a,Superset),Superset)
--> subset(Subset::'a,Superset)) &
(∀ Subset Superset. equal-sets(Subset::'a,Superset) --> subset(Subset::'a,Superset))
&
(∀ Subset Superset. equal-sets(Superset::'a,Subset) --> subset(Subset::'a,Superset))
&
(∀ Set2 Set1. subset(Set1::'a,Set2) & subset(Set2::'a,Set1) --> equal-sets(Set2::'a,Set1))
&
(∀ Set2 Intersection Element Set1. intersection(Set1::'a,Set2,Intersection) & mem-
ber(Element::'a,Intersection) --> member(Element::'a,Set1)) &
(∀ Set1 Intersection Element Set2. intersection(Set1::'a,Set2,Intersection) & mem-
ber(Element::'a,Intersection) --> member(Element::'a,Set2)) &
(∀ Set2 Set1 Element Intersection. intersection(Set1::'a,Set2,Intersection) & mem-
ber(Element::'a,Set2) & member(Element::'a,Set1) --> member(Element::'a,Intersection))
&
(∀ Set2 Intersection Set1. member(h(Set1::'a,Set2,Intersection),Intersection) |
intersection(Set1::'a,Set2,Intersection) | member(h(Set1::'a,Set2,Intersection),Set1))
&
(∀ Set1 Intersection Set2. member(h(Set1::'a,Set2,Intersection),Intersection) |
intersection(Set1::'a,Set2,Intersection) | member(h(Set1::'a,Set2,Intersection),Set2))
&
(∀ Set1 Set2 Intersection. member(h(Set1::'a,Set2,Intersection),Intersection) &
member(h(Set1::'a,Set2,Intersection),Set2) & member(h(Set1::'a,Set2,Intersection),Set1)
--> intersection(Set1::'a,Set2,Intersection)) &
(intersection(a::'a,b,aIb)) &
(intersection(b::'a,c,bIc)) &
(intersection(a::'a,bIc,aIbIc)) &
(∼ intersection(aIb::'a,c,aIbIc)) --> False
oops

```

lemma SET009-1:

```

(∀ Subset Element Superset. member(Element::'a,Subset) & ssubset(Subset::'a,Superset)
--> member(Element::'a,Superset)) &

```

$(\forall \text{ Superset Subset. } \text{ssubset}(\text{Subset}::'a, \text{Superset}) \mid \text{member}(\text{member-of-1-not-of-2}(\text{Subset}::'a, \text{Superset}), \text{Subset}))$
 $\&$
 $(\forall \text{ Subset Superset. } \text{member}(\text{member-of-1-not-of-2}(\text{Subset}::'a, \text{Superset}), \text{Superset})$
 $\longrightarrow \text{ssubset}(\text{Subset}::'a, \text{Superset})) \&$
 $(\forall \text{ Subset Superset. } \text{equal-sets}(\text{Subset}::'a, \text{Superset}) \longrightarrow \text{ssubset}(\text{Subset}::'a, \text{Superset}))$
 $\&$
 $(\forall \text{ Subset Superset. } \text{equal-sets}(\text{Superset}::'a, \text{Subset}) \longrightarrow \text{ssubset}(\text{Subset}::'a, \text{Superset}))$
 $\&$
 $(\forall \text{ Set2 Set1. } \text{ssubset}(\text{Set1}::'a, \text{Set2}) \& \text{ssubset}(\text{Set2}::'a, \text{Set1}) \longrightarrow \text{equal-sets}(\text{Set2}::'a, \text{Set1}))$
 $\&$
 $(\forall \text{ Set2 Difference Element Set1. } \text{difference}(\text{Set1}::'a, \text{Set2}, \text{Difference}) \& \text{member}(\text{Element}::'a, \text{Difference})$
 $\longrightarrow \text{member}(\text{Element}::'a, \text{Set1})) \&$
 $(\forall \text{ Element A-set Set1 Set2. } \sim(\text{member}(\text{Element}::'a, \text{Set1}) \& \text{member}(\text{Element}::'a, \text{Set2})$
 $\& \text{difference}(\text{A-set}::'a, \text{Set1}, \text{Set2}))) \&$
 $(\forall \text{ Set1 Difference Element Set2. } \text{member}(\text{Element}::'a, \text{Set1}) \& \text{difference}(\text{Set1}::'a, \text{Set2}, \text{Difference})$
 $\longrightarrow \text{member}(\text{Element}::'a, \text{Difference}) \mid \text{member}(\text{Element}::'a, \text{Set2})) \&$
 $(\forall \text{ Set1 Set2 Difference. } \text{difference}(\text{Set1}::'a, \text{Set2}, \text{Difference}) \mid \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Set1})$
 $\mid \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Difference})) \&$
 $(\forall \text{ Set1 Set2 Difference. } \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Set2}) \longrightarrow \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Difference})$
 $\mid \text{difference}(\text{Set1}::'a, \text{Set2}, \text{Difference})) \&$
 $(\forall \text{ Set1 Set2 Difference. } \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Difference}) \& \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Set1})$
 $\longrightarrow \text{member}(k(\text{Set1}::'a, \text{Set2}, \text{Difference}), \text{Set2})$
 $\mid \text{difference}(\text{Set1}::'a, \text{Set2}, \text{Difference})) \&$
 $(\text{ssubset}(d::'a, a)) \&$
 $(\text{difference}(b::'a, a, bDa)) \&$
 $(\text{difference}(b::'a, d, bDd)) \&$
 $(\sim \text{ssubset}(bDa::'a, bDd)) \longrightarrow \text{False}$
by meson

lemma SET025-4:

$\text{EQU001-0-ax equal} \&$
 $(\forall Y X. \text{member}(X::'a, Y) \longrightarrow \text{little-set}(X)) \&$
 $(\forall X Y. \text{little-set}(f1(X::'a, Y)) \mid \text{equal}(X::'a, Y)) \&$
 $(\forall X Y. \text{member}(f1(X::'a, Y), X) \mid \text{member}(f1(X::'a, Y), Y) \mid \text{equal}(X::'a, Y)) \&$
 $(\forall X Y. \text{member}(f1(X::'a, Y), X) \& \text{member}(f1(X::'a, Y), Y) \longrightarrow \text{equal}(X::'a, Y))$
 $\&$
 $(\forall X U Y. \text{member}(U::'a, \text{non-ordered-pair}(X::'a, Y)) \longrightarrow \text{equal}(U::'a, X) \mid \text{equal}(U::'a, Y))$
 $\&$
 $(\forall Y U X. \text{little-set}(U) \& \text{equal}(U::'a, X) \longrightarrow \text{member}(U::'a, \text{non-ordered-pair}(X::'a, Y)))$
 $\&$
 $(\forall X U Y. \text{little-set}(U) \& \text{equal}(U::'a, Y) \longrightarrow \text{member}(U::'a, \text{non-ordered-pair}(X::'a, Y)))$
 $\&$
 $(\forall X Y. \text{little-set}(\text{non-ordered-pair}(X::'a, Y))) \&$
 $(\forall X. \text{equal}(\text{singleton-set}(X), \text{non-ordered-pair}(X::'a, X))) \&$
 $(\forall X Y. \text{equal}(\text{ordered-pair}(X::'a, Y), \text{non-ordered-pair}(\text{singleton-set}(X), \text{non-ordered-pair}(X::'a, Y))))$
 $\&$
 $(\forall X. \text{ordered-pair-predicate}(X) \longrightarrow \text{little-set}(f2(X))) \&$
 $(\forall X. \text{ordered-pair-predicate}(X) \longrightarrow \text{little-set}(f3(X))) \&$

$(\forall X. \text{ordered-pair-predicate}(X) \longrightarrow \text{equal}(X::'a, \text{ordered-pair}(f2(X), f3(X)))) \&$
 $(\forall X Y Z. \text{little-set}(Y) \& \text{little-set}(Z) \& \text{equal}(X::'a, \text{ordered-pair}(Y::'a, Z)) \longrightarrow$
 $\text{ordered-pair-predicate}(X)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{little-set}(f4(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{little-set}(f5(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{equal}(X::'a, \text{ordered-pair}(f4(Z::'a, X), f5(Z::'a, X))))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{member}(Z::'a, f4(Z::'a, X))) \&$
 $(\forall X V Z U. \text{little-set}(U) \& \text{little-set}(V) \& \text{equal}(X::'a, \text{ordered-pair}(U::'a, V))$
 $\& \text{member}(Z::'a, U) \longrightarrow \text{member}(Z::'a, \text{first}(X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{little-set}(f6(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{little-set}(f7(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{equal}(X::'a, \text{ordered-pair}(f6(Z::'a, X), f7(Z::'a, X))))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{member}(Z::'a, f7(Z::'a, X))) \&$
 $(\forall X U Z V. \text{little-set}(U) \& \text{little-set}(V) \& \text{equal}(X::'a, \text{ordered-pair}(U::'a, V))$
 $\& \text{member}(Z::'a, V) \longrightarrow \text{member}(Z::'a, \text{second}(X))) \&$
 $(\forall Z. \text{member}(Z::'a, \text{estin}) \longrightarrow \text{ordered-pair-predicate}(Z)) \&$
 $(\forall Z. \text{member}(Z::'a, \text{estin}) \longrightarrow \text{member}(\text{first}(Z), \text{second}(Z))) \&$
 $(\forall Z. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Z) \& \text{member}(\text{first}(Z), \text{second}(Z))$
 $\longrightarrow \text{member}(Z::'a, \text{estin})) \&$
 $(\forall Y Z X. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, X)) \&$
 $(\forall X Z Y. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, Y)) \&$
 $(\forall X Z Y. \text{member}(Z::'a, X) \& \text{member}(Z::'a, Y) \longrightarrow \text{member}(Z::'a, \text{intersection}(X::'a, Y)))$
 $\&$
 $(\forall Z X. \sim(\text{member}(Z::'a, \text{complement}(X)) \& \text{member}(Z::'a, X))) \&$
 $(\forall Z X. \text{little-set}(Z) \longrightarrow \text{member}(Z::'a, \text{complement}(X)) \mid \text{member}(Z::'a, X)) \&$
 $(\forall X Y. \text{equal}(\text{union}(X::'a, Y), \text{complement}(\text{intersection}(\text{complement}(X), \text{complement}(Y))))))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{domain-of}(X)) \longrightarrow \text{ordered-pair-predicate}(f8(Z::'a, X)))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{domain-of}(X)) \longrightarrow \text{member}(f8(Z::'a, X), X)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{domain-of}(X)) \longrightarrow \text{equal}(Z::'a, \text{first}(f8(Z::'a, X)))) \&$
 $(\forall X Z Xp. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Xp) \& \text{member}(Xp::'a, X) \&$
 $\text{equal}(Z::'a, \text{first}(Xp)) \longrightarrow \text{member}(Z::'a, \text{domain-of}(X))) \&$
 $(\forall X Y Z. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{ordered-pair-predicate}(Z))$
 $\&$
 $(\forall Y Z X. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(\text{first}(Z), X)) \&$
 $(\forall X Z Y. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(\text{second}(Z), Y))$
 $\&$
 $(\forall X Z Y. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Z) \& \text{member}(\text{first}(Z), X) \&$
 $\text{member}(\text{second}(Z), Y) \longrightarrow \text{member}(Z::'a, \text{cross-product}(X::'a, Y))) \&$
 $(\forall X Z. \text{member}(Z::'a, \text{inv1 } X) \longrightarrow \text{ordered-pair-predicate}(Z)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{inv1 } X) \longrightarrow \text{member}(\text{ordered-pair}(\text{second}(Z), \text{first}(Z)), X))$
 $\&$
 $(\forall Z X. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Z) \& \text{member}(\text{ordered-pair}(\text{second}(Z), \text{first}(Z)), X)$
 $\longrightarrow \text{member}(Z::'a, \text{inv1 } X)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{little-set}(f9(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{little-set}(f10(Z::'a, X))) \&$

$(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{little-set}(f11(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{equal}(Z::'a, \text{ordered-pair}(f9(Z::'a, X), \text{ordered-pair}(f10(Z::'a, X), f11(Z::'a, X)))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{member}(\text{ordered-pair}(f10(Z::'a, X), \text{ordered-pair}(f11(Z::'a, X), f9(Z::'a, X)))) \ \&$
 $(\forall Z V W U X. \text{little-set}(Z) \ \& \ \text{little-set}(U) \ \& \ \text{little-set}(V) \ \& \ \text{little-set}(W) \ \& \ \text{equal}(Z::'a, \text{ordered-pair}(U::'a, \text{ordered-pair}(V::'a, W))) \ \& \ \text{member}(\text{ordered-pair}(V::'a, \text{ordered-pair}(W::'a, U))) \longrightarrow \text{member}(Z::'a, \text{rot-right}(X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{little-set}(f12(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{little-set}(f13(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{little-set}(f14(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{equal}(Z::'a, \text{ordered-pair}(f12(Z::'a, X), \text{ordered-pair}(f13(Z::'a, X), f14(Z::'a, X)))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{member}(\text{ordered-pair}(f12(Z::'a, X), \text{ordered-pair}(f14(Z::'a, X), f13(Z::'a, X)))) \ \&$
 $(\forall Z U W V X. \text{little-set}(Z) \ \& \ \text{little-set}(U) \ \& \ \text{little-set}(V) \ \& \ \text{little-set}(W) \ \& \ \text{equal}(Z::'a, \text{ordered-pair}(U::'a, \text{ordered-pair}(V::'a, W))) \ \& \ \text{member}(\text{ordered-pair}(U::'a, \text{ordered-pair}(W::'a, V))) \longrightarrow \text{member}(Z::'a, \text{flip-range-of}(X))) \ \&$
 $(\forall X. \text{equal}(\text{successor}(X), \text{union}(X::'a, \text{singleton-set}(X)))) \ \&$
 $(\forall Z. \sim \text{member}(Z::'a, \text{empty-set})) \ \&$
 $(\forall Z. \text{little-set}(Z) \longrightarrow \text{member}(Z::'a, \text{universal-set})) \ \&$
 $(\text{little-set}(\text{infinity})) \ \&$
 $(\text{member}(\text{empty-set}::'a, \text{infinity})) \ \&$
 $(\forall X. \text{member}(X::'a, \text{infinity}) \longrightarrow \text{member}(\text{successor}(X), \text{infinity})) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{sigma}(X)) \longrightarrow \text{member}(f16(Z::'a, X), X)) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{sigma}(X)) \longrightarrow \text{member}(Z::'a, f16(Z::'a, X))) \ \&$
 $(\forall X Z Y. \text{member}(Y::'a, X) \ \& \ \text{member}(Z::'a, Y) \longrightarrow \text{member}(Z::'a, \text{sigma}(X))) \ \&$
 $(\forall U. \text{little-set}(U) \longrightarrow \text{little-set}(\text{sigma}(U))) \ \&$
 $(\forall X U Y. \text{ssubset}(X::'a, Y) \ \& \ \text{member}(U::'a, X) \longrightarrow \text{member}(U::'a, Y)) \ \&$
 $(\forall Y X. \text{ssubset}(X::'a, Y) \mid \text{member}(f17(X::'a, Y), X)) \ \&$
 $(\forall X Y. \text{member}(f17(X::'a, Y), Y) \longrightarrow \text{ssubset}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{proper-subset}(X::'a, Y) \longrightarrow \text{ssubset}(X::'a, Y)) \ \&$
 $(\forall X Y. \sim(\text{proper-subset}(X::'a, Y) \ \& \ \text{equal}(X::'a, Y))) \ \&$
 $(\forall X Y. \text{ssubset}(X::'a, Y) \longrightarrow \text{proper-subset}(X::'a, Y) \mid \text{equal}(X::'a, Y)) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{powerset}(X)) \longrightarrow \text{ssubset}(Z::'a, X)) \ \&$
 $(\forall Z X. \text{little-set}(Z) \ \& \ \text{ssubset}(Z::'a, X) \longrightarrow \text{member}(Z::'a, \text{powerset}(X))) \ \&$
 $(\forall U. \text{little-set}(U) \longrightarrow \text{little-set}(\text{powerset}(U))) \ \&$
 $(\forall Z X. \text{relation}(Z) \ \& \ \text{member}(X::'a, Z) \longrightarrow \text{ordered-pair-predicate}(X)) \ \&$
 $(\forall Z. \text{relation}(Z) \mid \text{member}(f18(Z), Z)) \ \&$
 $(\forall Z. \text{ordered-pair-predicate}(f18(Z)) \longrightarrow \text{relation}(Z)) \ \&$
 $(\forall U X V W. \text{single-valued-set}(X) \ \& \ \text{little-set}(U) \ \& \ \text{little-set}(V) \ \& \ \text{little-set}(W) \ \& \ \text{member}(\text{ordered-pair}(U::'a, V), X) \ \& \ \text{member}(\text{ordered-pair}(U::'a, W), X) \longrightarrow \text{equal}(V::'a, W)) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{little-set}(f19(X))) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{little-set}(f20(X))) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{little-set}(f21(X))) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{member}(\text{ordered-pair}(f19(X), f20(X)), X)) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{member}(\text{ordered-pair}(f19(X), f21(X)), X)) \ \&$

$(\forall X. \text{equal}(f20(X), f21(X)) \longrightarrow \text{single-valued-set}(X)) \ \&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{relation}(Xf)) \ \&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{single-valued-set}(Xf)) \ \&$
 $(\forall Xf. \text{relation}(Xf) \ \& \ \text{single-valued-set}(Xf) \longrightarrow \text{function}(Xf)) \ \&$
 $(\forall Z X Xf. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{ordered-pair-predicate}(f22(Z::'a, X, Xf)))$
 $\&$
 $(\forall Z X Xf. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{member}(f22(Z::'a, X, Xf), Xf))$
 $\&$
 $(\forall Z Xf X. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{member}(\text{first}(f22(Z::'a, X, Xf)), X))$
 $\&$
 $(\forall X Xf Z. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{equal}(\text{second}(f22(Z::'a, X, Xf)), Z))$
 $\&$
 $(\forall Xf X Y Z. \text{little-set}(Z) \ \& \ \text{ordered-pair-predicate}(Y) \ \& \ \text{member}(Y::'a, Xf) \ \&$
 $\text{member}(\text{first}(Y), X) \ \& \ \text{equal}(\text{second}(Y), Z) \longrightarrow \text{member}(Z::'a, \text{image}'(X::'a, Xf)))$
 $\&$
 $(\forall X Xf. \text{little-set}(X) \ \& \ \text{function}(Xf) \longrightarrow \text{little-set}(\text{image}'(X::'a, Xf))) \ \&$
 $(\forall X U Y. \sim(\text{disjoint}(X::'a, Y) \ \& \ \text{member}(U::'a, X) \ \& \ \text{member}(U::'a, Y))) \ \&$
 $(\forall Y X. \text{disjoint}(X::'a, Y) \mid \text{member}(f23(X::'a, Y), X)) \ \&$
 $(\forall X Y. \text{disjoint}(X::'a, Y) \mid \text{member}(f23(X::'a, Y), Y)) \ \&$
 $(\forall X. \text{equal}(X::'a, \text{empty-set}) \mid \text{member}(f24(X), X)) \ \&$
 $(\forall X. \text{equal}(X::'a, \text{empty-set}) \mid \text{disjoint}(f24(X), X)) \ \&$
 $(\text{function}(f25)) \ \&$
 $(\forall X. \text{little-set}(X) \longrightarrow \text{equal}(X::'a, \text{empty-set}) \mid \text{member}(f26(X), X)) \ \&$
 $(\forall X. \text{little-set}(X) \longrightarrow \text{equal}(X::'a, \text{empty-set}) \mid \text{member}(\text{ordered-pair}(X::'a, f26(X)), f25))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{range-of}(X)) \longrightarrow \text{ordered-pair-predicate}(f27(Z::'a, X)))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{range-of}(X)) \longrightarrow \text{member}(f27(Z::'a, X), X)) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{range-of}(X)) \longrightarrow \text{equal}(Z::'a, \text{second}(f27(Z::'a, X)))) \ \&$
 $(\forall X Z Xp. \text{little-set}(Z) \ \& \ \text{ordered-pair-predicate}(Xp) \ \& \ \text{member}(Xp::'a, X) \ \&$
 $\text{equal}(Z::'a, \text{second}(Xp)) \longrightarrow \text{member}(Z::'a, \text{range-of}(X))) \ \&$
 $(\forall Z. \text{member}(Z::'a, \text{identity-relation}) \longrightarrow \text{ordered-pair-predicate}(Z)) \ \&$
 $(\forall Z. \text{member}(Z::'a, \text{identity-relation}) \longrightarrow \text{equal}(\text{first}(Z), \text{second}(Z))) \ \&$
 $(\forall Z. \text{little-set}(Z) \ \& \ \text{ordered-pair-predicate}(Z) \ \& \ \text{equal}(\text{first}(Z), \text{second}(Z)) \longrightarrow$
 $\text{member}(Z::'a, \text{identity-relation})) \ \&$
 $(\forall X Y. \text{equal}(\text{restrct}(X::'a, Y), \text{intersection}(X::'a, \text{cross-product}(Y::'a, \text{universal-set}))))$
 $\&$
 $(\forall Xf. \text{one-to-one-function}(Xf) \longrightarrow \text{function}(Xf)) \ \&$
 $(\forall Xf. \text{one-to-one-function}(Xf) \longrightarrow \text{function}(\text{inv1 } Xf)) \ \&$
 $(\forall Xf. \text{function}(Xf) \ \& \ \text{function}(\text{inv1 } Xf) \longrightarrow \text{one-to-one-function}(Xf)) \ \&$
 $(\forall Z Xf Y. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{ordered-pair-predicate}(f28(Z::'a, Xf, Y)))$
 $\&$
 $(\forall Z Y Xf. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{member}(f28(Z::'a, Xf, Y), Xf))$
 $\&$
 $(\forall Z Xf Y. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{equal}(\text{first}(f28(Z::'a, Xf, Y)), Y))$
 $\&$
 $(\forall Z Xf Y. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{member}(Z::'a, \text{second}(f28(Z::'a, Xf, Y))))$
 $\&$
 $(\forall Xf Y Z W. \text{ordered-pair-predicate}(W) \ \& \ \text{member}(W::'a, Xf) \ \& \ \text{equal}(\text{first}(W), Y))$

$\& \text{ member}(Z::'a, \text{second}(W)) \longrightarrow \text{ member}(Z::'a, \text{apply}(Xf::'a, Y)) \&$
 $(\forall Xf X Y. \text{ equal}(\text{apply-to-two-arguments}(Xf::'a, X, Y), \text{apply}(Xf::'a, \text{ordered-pair}(X::'a, Y))))$
 $\&$
 $(\forall X Y Xf. \text{ maps}(Xf::'a, X, Y) \longrightarrow \text{ function}(Xf)) \&$
 $(\forall Y Xf X. \text{ maps}(Xf::'a, X, Y) \longrightarrow \text{ equal}(\text{domain-of}(Xf), X)) \&$
 $(\forall X Xf Y. \text{ maps}(Xf::'a, X, Y) \longrightarrow \text{ ssubset}(\text{range-of}(Xf), Y)) \&$
 $(\forall X Xf Y. \text{ function}(Xf) \& \text{ equal}(\text{domain-of}(Xf), X) \& \text{ ssubset}(\text{range-of}(Xf), Y))$
 $\longrightarrow \text{ maps}(Xf::'a, X, Y)) \&$
 $(\forall Xf Xs. \text{ closed}(Xs::'a, Xf) \longrightarrow \text{ little-set}(Xs)) \&$
 $(\forall Xs Xf. \text{ closed}(Xs::'a, Xf) \longrightarrow \text{ little-set}(Xf)) \&$
 $(\forall Xf Xs. \text{ closed}(Xs::'a, Xf) \longrightarrow \text{ maps}(Xf::'a, \text{cross-product}(Xs::'a, Xs), Xs)) \&$
 $(\forall Xf Xs. \text{ little-set}(Xs) \& \text{ little-set}(Xf) \& \text{ maps}(Xf::'a, \text{cross-product}(Xs::'a, Xs), Xs))$
 $\longrightarrow \text{ closed}(Xs::'a, Xf)) \&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ little-set}(f29(Z::'a, Xf, Xg)))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ little-set}(f30(Z::'a, Xf, Xg)))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ little-set}(f31(Z::'a, Xf, Xg)))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ equal}(Z::'a, \text{ordered-pair}(f29(Z::'a, Xf, Xg), f30(Z::'a, Xf, Xg))))$
 $\&$
 $(\forall Z Xg Xf. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ member}(\text{ordered-pair}(f29(Z::'a, Xf, Xg), f31(Z::'a, Xf, Xg))))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ member}(\text{ordered-pair}(f31(Z::'a, Xf, Xg), f30(Z::'a, Xf, Xg))))$
 $\&$
 $(\forall Z X Xf W Y Xg. \text{ little-set}(Z) \& \text{ little-set}(X) \& \text{ little-set}(Y) \& \text{ little-set}(W) \&$
 $\text{ equal}(Z::'a, \text{ordered-pair}(X::'a, Y)) \& \text{ member}(\text{ordered-pair}(X::'a, W), Xf) \& \text{ mem-}$
 $\text{ ber}(\text{ordered-pair}(W::'a, Y), Xg) \longrightarrow \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg))) \&$
 $(\forall Xh Xs2 Xf2 Xs1 Xf1. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \longrightarrow \text{ closed}(Xs1::'a, Xf1))$
 $\&$
 $(\forall Xh Xs1 Xf1 Xs2 Xf2. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \longrightarrow \text{ closed}(Xs2::'a, Xf2))$
 $\&$
 $(\forall Xf1 Xf2 Xh Xs1 Xs2. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \longrightarrow \text{ maps}(Xh::'a, Xs1, Xs2))$
 $\&$
 $(\forall Xs2 Xs1 Xf1 Xf2 X Xh Y. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \& \text{ mem-}$
 $\text{ ber}(X::'a, Xs1) \& \text{ member}(Y::'a, Xs1) \longrightarrow \text{ equal}(\text{apply}(Xh::'a, \text{apply-to-two-arguments}(Xf1::'a, X, Y)), \text{apply-to-}$
 $\text{ two-arguments}(Xf1::'a, X, Y))) \&$
 $(\forall Xh Xf1 Xs2 Xf2 Xs1. \text{ closed}(Xs1::'a, Xf1) \& \text{ closed}(Xs2::'a, Xf2) \& \text{ maps}(Xh::'a, Xs1, Xs2)$
 $\longrightarrow \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \mid \text{ member}(f32(Xh::'a, Xs1, Xf1, Xs2, Xf2), Xs1))$
 $\&$
 $(\forall Xh Xf1 Xs2 Xf2 Xs1. \text{ closed}(Xs1::'a, Xf1) \& \text{ closed}(Xs2::'a, Xf2) \& \text{ maps}(Xh::'a, Xs1, Xs2)$
 $\longrightarrow \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \mid \text{ member}(f33(Xh::'a, Xs1, Xf1, Xs2, Xf2), Xs1))$
 $\&$
 $(\forall Xh Xs1 Xf1 Xs2 Xf2. \text{ closed}(Xs1::'a, Xf1) \& \text{ closed}(Xs2::'a, Xf2) \& \text{ maps}(Xh::'a, Xs1, Xs2)$
 $\& \text{ equal}(\text{apply}(Xh::'a, \text{apply-to-two-arguments}(Xf1::'a, f32(Xh::'a, Xs1, Xf1, Xs2, Xf2)), f33(Xh::'a, Xs1, Xf1, Xs2, Xf2)))$
 $\longrightarrow \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2)) \&$
 $(\forall A B C. \text{ equal}(A::'a, B) \longrightarrow \text{ equal}(f1(A::'a, C), f1(B::'a, C))) \&$
 $(\forall D F' E. \text{ equal}(D::'a, E) \longrightarrow \text{ equal}(f1(F'::'a, D), f1(F'::'a, E))) \&$
 $(\forall A2 B2. \text{ equal}(A2::'a, B2) \longrightarrow \text{ equal}(f2(A2), f2(B2))) \&$

$(\forall G4\ H4. \text{equal}(G4::'a,H4) \longrightarrow \text{equal}(f3(G4),f3(H4))) \ \&$
 $(\forall O7\ P7\ Q7. \text{equal}(O7::'a,P7) \longrightarrow \text{equal}(f4(O7::'a,Q7),f4(P7::'a,Q7))) \ \&$
 $(\forall R7\ T7\ S7. \text{equal}(R7::'a,S7) \longrightarrow \text{equal}(f4(T7::'a,R7),f4(T7::'a,S7))) \ \&$
 $(\forall U7\ V7\ W7. \text{equal}(U7::'a,V7) \longrightarrow \text{equal}(f5(U7::'a,W7),f5(V7::'a,W7))) \ \&$
 $(\forall X7\ Z7\ Y7. \text{equal}(X7::'a,Y7) \longrightarrow \text{equal}(f5(Z7::'a,X7),f5(Z7::'a,Y7))) \ \&$
 $(\forall A8\ B8\ C8. \text{equal}(A8::'a,B8) \longrightarrow \text{equal}(f6(A8::'a,C8),f6(B8::'a,C8))) \ \&$
 $(\forall D8\ F8\ E8. \text{equal}(D8::'a,E8) \longrightarrow \text{equal}(f6(F8::'a,D8),f6(F8::'a,E8))) \ \&$
 $(\forall G8\ H8\ I8. \text{equal}(G8::'a,H8) \longrightarrow \text{equal}(f7(G8::'a,I8),f7(H8::'a,I8))) \ \&$
 $(\forall J8\ L8\ K8. \text{equal}(J8::'a,K8) \longrightarrow \text{equal}(f7(L8::'a,J8),f7(L8::'a,K8))) \ \&$
 $(\forall M8\ N8\ O8. \text{equal}(M8::'a,N8) \longrightarrow \text{equal}(f8(M8::'a,O8),f8(N8::'a,O8))) \ \&$
 $(\forall P8\ R8\ Q8. \text{equal}(P8::'a,Q8) \longrightarrow \text{equal}(f8(R8::'a,P8),f8(R8::'a,Q8))) \ \&$
 $(\forall S8\ T8\ U8. \text{equal}(S8::'a,T8) \longrightarrow \text{equal}(f9(S8::'a,U8),f9(T8::'a,U8))) \ \&$
 $(\forall V8\ X8\ W8. \text{equal}(V8::'a,W8) \longrightarrow \text{equal}(f9(X8::'a,V8),f9(X8::'a,W8))) \ \&$
 $(\forall G\ H\ I'. \text{equal}(G::'a,H) \longrightarrow \text{equal}(f10(G::'a,I'),f10(H::'a,I'))) \ \&$
 $(\forall J\ L\ K'. \text{equal}(J::'a,K') \longrightarrow \text{equal}(f10(L::'a,J),f10(L::'a,K'))) \ \&$
 $(\forall M\ N\ O'. \text{equal}(M::'a,N) \longrightarrow \text{equal}(f11(M::'a,O'),f11(N::'a,O'))) \ \&$
 $(\forall P\ R\ Q. \text{equal}(P::'a,Q) \longrightarrow \text{equal}(f11(R::'a,P),f11(R::'a,Q))) \ \&$
 $(\forall S' \ T' \ U. \text{equal}(S'::'a,T') \longrightarrow \text{equal}(f12(S'::'a,U),f12(T'::'a,U))) \ \&$
 $(\forall V\ X\ W. \text{equal}(V::'a,W) \longrightarrow \text{equal}(f12(X::'a,V),f12(X::'a,W))) \ \&$
 $(\forall Y\ Z\ A1. \text{equal}(Y::'a,Z) \longrightarrow \text{equal}(f13(Y::'a,A1),f13(Z::'a,A1))) \ \&$
 $(\forall B1\ D1\ C1. \text{equal}(B1::'a,C1) \longrightarrow \text{equal}(f13(D1::'a,B1),f13(D1::'a,C1))) \ \&$
 $(\forall E1\ F1\ G1. \text{equal}(E1::'a,F1) \longrightarrow \text{equal}(f14(E1::'a,G1),f14(F1::'a,G1))) \ \&$
 $(\forall H1\ J1\ I1. \text{equal}(H1::'a,I1) \longrightarrow \text{equal}(f14(J1::'a,H1),f14(J1::'a,I1))) \ \&$
 $(\forall K1\ L1\ M1. \text{equal}(K1::'a,L1) \longrightarrow \text{equal}(f16(K1::'a,M1),f16(L1::'a,M1))) \ \&$
 $(\forall N1\ P1\ O1. \text{equal}(N1::'a,O1) \longrightarrow \text{equal}(f16(P1::'a,N1),f16(P1::'a,O1))) \ \&$
 $(\forall Q1\ R1\ S1. \text{equal}(Q1::'a,R1) \longrightarrow \text{equal}(f17(Q1::'a,S1),f17(R1::'a,S1))) \ \&$
 $(\forall T1\ V1\ U1. \text{equal}(T1::'a,U1) \longrightarrow \text{equal}(f17(V1::'a,T1),f17(V1::'a,U1))) \ \&$
 $(\forall W1\ X1. \text{equal}(W1::'a,X1) \longrightarrow \text{equal}(f18(W1),f18(X1))) \ \&$
 $(\forall Y1\ Z1. \text{equal}(Y1::'a,Z1) \longrightarrow \text{equal}(f19(Y1),f19(Z1))) \ \&$
 $(\forall C2\ D2. \text{equal}(C2::'a,D2) \longrightarrow \text{equal}(f20(C2),f20(D2))) \ \&$
 $(\forall E2\ F2. \text{equal}(E2::'a,F2) \longrightarrow \text{equal}(f21(E2),f21(F2))) \ \&$
 $(\forall G2\ H2\ I2\ J2. \text{equal}(G2::'a,H2) \longrightarrow \text{equal}(f22(G2::'a,I2,J2),f22(H2::'a,I2,J2)))$
 $\&$
 $(\forall K2\ M2\ L2\ N2. \text{equal}(K2::'a,L2) \longrightarrow \text{equal}(f22(M2::'a,K2,N2),f22(M2::'a,L2,N2)))$
 $\&$
 $(\forall O2\ Q2\ R2\ P2. \text{equal}(O2::'a,P2) \longrightarrow \text{equal}(f22(Q2::'a,R2,O2),f22(Q2::'a,R2,P2)))$
 $\&$
 $(\forall S2\ T2\ U2. \text{equal}(S2::'a,T2) \longrightarrow \text{equal}(f23(S2::'a,U2),f23(T2::'a,U2))) \ \&$
 $(\forall V2\ X2\ W2. \text{equal}(V2::'a,W2) \longrightarrow \text{equal}(f23(X2::'a,V2),f23(X2::'a,W2)))$
 $\&$
 $(\forall Y2\ Z2. \text{equal}(Y2::'a,Z2) \longrightarrow \text{equal}(f24(Y2),f24(Z2))) \ \&$
 $(\forall A3\ B3. \text{equal}(A3::'a,B3) \longrightarrow \text{equal}(f26(A3),f26(B3))) \ \&$
 $(\forall C3\ D3\ E3. \text{equal}(C3::'a,D3) \longrightarrow \text{equal}(f27(C3::'a,E3),f27(D3::'a,E3))) \ \&$
 $(\forall F3\ H3\ G3. \text{equal}(F3::'a,G3) \longrightarrow \text{equal}(f27(H3::'a,F3),f27(H3::'a,G3))) \ \&$
 $(\forall I3\ J3\ K3\ L3. \text{equal}(I3::'a,J3) \longrightarrow \text{equal}(f28(I3::'a,K3,L3),f28(J3::'a,K3,L3)))$
 $\&$
 $(\forall M3\ O3\ N3\ P3. \text{equal}(M3::'a,N3) \longrightarrow \text{equal}(f28(O3::'a,M3,P3),f28(O3::'a,N3,P3)))$
 $\&$
 $(\forall Q3\ S3\ T3\ R3. \text{equal}(Q3::'a,R3) \longrightarrow \text{equal}(f28(S3::'a,T3,Q3),f28(S3::'a,T3,R3)))$

$\&$
 $(\forall U3\ V3\ W3\ X3. \text{equal}(U3::'a, V3) \longrightarrow \text{equal}(f29(U3::'a, W3, X3), f29(V3::'a, W3, X3)))$
 $\&$
 $(\forall Y3\ A4\ Z3\ B4. \text{equal}(Y3::'a, Z3) \longrightarrow \text{equal}(f29(A4::'a, Y3, B4), f29(A4::'a, Z3, B4)))$
 $\&$
 $(\forall C4\ E4\ F4\ D4. \text{equal}(C4::'a, D4) \longrightarrow \text{equal}(f29(E4::'a, F4, C4), f29(E4::'a, F4, D4)))$
 $\&$
 $(\forall I4\ J4\ K4\ L4. \text{equal}(I4::'a, J4) \longrightarrow \text{equal}(f30(I4::'a, K4, L4), f30(J4::'a, K4, L4)))$
 $\&$
 $(\forall M4\ O4\ N4\ P4. \text{equal}(M4::'a, N4) \longrightarrow \text{equal}(f30(O4::'a, M4, P4), f30(O4::'a, N4, P4)))$
 $\&$
 $(\forall Q4\ S4\ T4\ R4. \text{equal}(Q4::'a, R4) \longrightarrow \text{equal}(f30(S4::'a, T4, Q4), f30(S4::'a, T4, R4)))$
 $\&$
 $(\forall U4\ V4\ W4\ X4. \text{equal}(U4::'a, V4) \longrightarrow \text{equal}(f31(U4::'a, W4, X4), f31(V4::'a, W4, X4)))$
 $\&$
 $(\forall Y4\ A5\ Z4\ B5. \text{equal}(Y4::'a, Z4) \longrightarrow \text{equal}(f31(A5::'a, Y4, B5), f31(A5::'a, Z4, B5)))$
 $\&$
 $(\forall C5\ E5\ F5\ D5. \text{equal}(C5::'a, D5) \longrightarrow \text{equal}(f31(E5::'a, F5, C5), f31(E5::'a, F5, D5)))$
 $\&$
 $(\forall G5\ H5\ I5\ J5\ K5\ L5. \text{equal}(G5::'a, H5) \longrightarrow \text{equal}(f32(G5::'a, I5, J5, K5, L5), f32(H5::'a, I5, J5, K5, L5)))$
 $\&$
 $(\forall M5\ O5\ N5\ P5\ Q5\ R5. \text{equal}(M5::'a, N5) \longrightarrow \text{equal}(f32(O5::'a, M5, P5, Q5, R5), f32(O5::'a, N5, P5, Q5, R5)))$
 $\&$
 $(\forall S5\ U5\ V5\ T5\ W5\ X5. \text{equal}(S5::'a, T5) \longrightarrow \text{equal}(f32(U5::'a, V5, S5, W5, X5), f32(U5::'a, V5, T5, W5, X5)))$
 $\&$
 $(\forall Y5\ A6\ B6\ C6\ Z5\ D6. \text{equal}(Y5::'a, Z5) \longrightarrow \text{equal}(f32(A6::'a, B6, C6, Y5, D6), f32(A6::'a, B6, C6, Z5, D6)))$
 $\&$
 $(\forall E6\ G6\ H6\ I6\ J6\ F6. \text{equal}(E6::'a, F6) \longrightarrow \text{equal}(f32(G6::'a, H6, I6, J6, E6), f32(G6::'a, H6, I6, J6, F6)))$
 $\&$
 $(\forall K6\ L6\ M6\ N6\ O6\ P6. \text{equal}(K6::'a, L6) \longrightarrow \text{equal}(f33(K6::'a, M6, N6, O6, P6), f33(L6::'a, M6, N6, O6, P6)))$
 $\&$
 $(\forall Q6\ S6\ R6\ T6\ U6\ V6. \text{equal}(Q6::'a, R6) \longrightarrow \text{equal}(f33(S6::'a, Q6, T6, U6, V6), f33(S6::'a, R6, T6, U6, V6)))$
 $\&$
 $(\forall W6\ Y6\ Z6\ X6\ A7\ B7. \text{equal}(W6::'a, X6) \longrightarrow \text{equal}(f33(Y6::'a, Z6, W6, A7, B7), f33(Y6::'a, Z6, X6, A7, B7)))$
 $\&$
 $(\forall C7\ E7\ F7\ G7\ D7\ H7. \text{equal}(C7::'a, D7) \longrightarrow \text{equal}(f33(E7::'a, F7, G7, C7, H7), f33(E7::'a, F7, G7, D7, H7)))$
 $\&$
 $(\forall I7\ K7\ L7\ M7\ N7\ J7. \text{equal}(I7::'a, J7) \longrightarrow \text{equal}(f33(K7::'a, L7, M7, N7, I7), f33(K7::'a, L7, M7, N7, J7)))$
 $\&$
 $(\forall A\ B\ C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{apply}(A::'a, C), \text{apply}(B::'a, C))) \ \&$
 $(\forall D\ F'\ E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{apply}(F'::'a, D), \text{apply}(F'::'a, E))) \ \&$
 $(\forall G\ H\ I'\ J. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{apply-to-two-arguments}(G::'a, I', J), \text{apply-to-two-arguments}(H::'a, I', J)))$
 $\&$
 $(\forall K'\ M\ L\ N. \text{equal}(K'::'a, L) \longrightarrow \text{equal}(\text{apply-to-two-arguments}(M::'a, K', N), \text{apply-to-two-arguments}(M::'a, L, N)))$
 $\&$
 $(\forall O'\ Q\ R\ P. \text{equal}(O'::'a, P) \longrightarrow \text{equal}(\text{apply-to-two-arguments}(Q::'a, R, O'), \text{apply-to-two-arguments}(Q::'a, R, P)))$
 $\&$
 $(\forall S'\ T'. \text{equal}(S'::'a, T') \longrightarrow \text{equal}(\text{complement}(S'), \text{complement}(T'))) \ \&$
 $(\forall U\ V\ W. \text{equal}(U::'a, V) \longrightarrow \text{equal}(\text{composition}(U::'a, W), \text{composition}(V::'a, W)))$

$\&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{composition}(Z::'a, X), \text{composition}(Z::'a, Y)))$
 $\&$
 $(\forall A1 B1. \text{equal}(A1::'a, B1) \longrightarrow \text{equal}(\text{inv1 } A1, \text{inv1 } B1)) \&$
 $(\forall C1 D1 E1. \text{equal}(C1::'a, D1) \longrightarrow \text{equal}(\text{cross-product}(C1::'a, E1), \text{cross-product}(D1::'a, E1)))$
 $\&$
 $(\forall F1 H1 G1. \text{equal}(F1::'a, G1) \longrightarrow \text{equal}(\text{cross-product}(H1::'a, F1), \text{cross-product}(H1::'a, G1)))$
 $\&$
 $(\forall I1 J1. \text{equal}(I1::'a, J1) \longrightarrow \text{equal}(\text{domain-of}(I1), \text{domain-of}(J1))) \&$
 $(\forall I10 J10. \text{equal}(I10::'a, J10) \longrightarrow \text{equal}(\text{first}(I10), \text{first}(J10))) \&$
 $(\forall Q10 R10. \text{equal}(Q10::'a, R10) \longrightarrow \text{equal}(\text{flip-range-of}(Q10), \text{flip-range-of}(R10)))$
 $\&$
 $(\forall S10 T10 U10. \text{equal}(S10::'a, T10) \longrightarrow \text{equal}(\text{image}'(S10::'a, U10), \text{image}'(T10::'a, U10)))$
 $\&$
 $(\forall V10 X10 W10. \text{equal}(V10::'a, W10) \longrightarrow \text{equal}(\text{image}'(X10::'a, V10), \text{image}'(X10::'a, W10)))$
 $\&$
 $(\forall Y10 Z10 A11. \text{equal}(Y10::'a, Z10) \longrightarrow \text{equal}(\text{intersection}(Y10::'a, A11), \text{intersection}(Z10::'a, A11)))$
 $\&$
 $(\forall B11 D11 C11. \text{equal}(B11::'a, C11) \longrightarrow \text{equal}(\text{intersection}(D11::'a, B11), \text{intersection}(D11::'a, C11)))$
 $\&$
 $(\forall E11 F11 G11. \text{equal}(E11::'a, F11) \longrightarrow \text{equal}(\text{non-ordered-pair}(E11::'a, G11), \text{non-ordered-pair}(F11::'a, G11)))$
 $\&$
 $(\forall H11 J11 I11. \text{equal}(H11::'a, I11) \longrightarrow \text{equal}(\text{non-ordered-pair}(J11::'a, H11), \text{non-ordered-pair}(J11::'a, I11)))$
 $\&$
 $(\forall K11 L11 M11. \text{equal}(K11::'a, L11) \longrightarrow \text{equal}(\text{ordered-pair}(K11::'a, M11), \text{ordered-pair}(L11::'a, M11)))$
 $\&$
 $(\forall N11 P11 O11. \text{equal}(N11::'a, O11) \longrightarrow \text{equal}(\text{ordered-pair}(P11::'a, N11), \text{ordered-pair}(P11::'a, O11)))$
 $\&$
 $(\forall Q11 R11. \text{equal}(Q11::'a, R11) \longrightarrow \text{equal}(\text{powerset}(Q11), \text{powerset}(R11))) \&$
 $(\forall S11 T11. \text{equal}(S11::'a, T11) \longrightarrow \text{equal}(\text{range-of}(S11), \text{range-of}(T11))) \&$
 $(\forall U11 V11 W11. \text{equal}(U11::'a, V11) \longrightarrow \text{equal}(\text{restrct}(U11::'a, W11), \text{restrct}(V11::'a, W11)))$
 $\&$
 $(\forall X11 Z11 Y11. \text{equal}(X11::'a, Y11) \longrightarrow \text{equal}(\text{restrct}(Z11::'a, X11), \text{restrct}(Z11::'a, Y11)))$
 $\&$
 $(\forall A12 B12. \text{equal}(A12::'a, B12) \longrightarrow \text{equal}(\text{rot-right}(A12), \text{rot-right}(B12))) \&$
 $(\forall C12 D12. \text{equal}(C12::'a, D12) \longrightarrow \text{equal}(\text{second}(C12), \text{second}(D12))) \&$
 $(\forall K12 L12. \text{equal}(K12::'a, L12) \longrightarrow \text{equal}(\text{sigma}(K12), \text{sigma}(L12))) \&$
 $(\forall M12 N12. \text{equal}(M12::'a, N12) \longrightarrow \text{equal}(\text{singleton-set}(M12), \text{singleton-set}(N12)))$
 $\&$
 $(\forall O12 P12. \text{equal}(O12::'a, P12) \longrightarrow \text{equal}(\text{successor}(O12), \text{successor}(P12))) \&$
 $(\forall Q12 R12 S12. \text{equal}(Q12::'a, R12) \longrightarrow \text{equal}(\text{union}(Q12::'a, S12), \text{union}(R12::'a, S12)))$
 $\&$
 $(\forall T12 V12 U12. \text{equal}(T12::'a, U12) \longrightarrow \text{equal}(\text{union}(V12::'a, T12), \text{union}(V12::'a, U12)))$
 $\&$
 $(\forall W12 X12 Y12. \text{equal}(W12::'a, X12) \& \text{closed}(W12::'a, Y12) \longrightarrow \text{closed}(X12::'a, Y12))$
 $\&$
 $(\forall Z12 B13 A13. \text{equal}(Z12::'a, A13) \& \text{closed}(B13::'a, Z12) \longrightarrow \text{closed}(B13::'a, A13))$
 $\&$
 $(\forall C13 D13 E13. \text{equal}(C13::'a, D13) \& \text{disjoint}(C13::'a, E13) \longrightarrow \text{disjoint}(D13::'a, E13))$

&
 ($\forall F13\ H13\ G13. \text{equal}(F13::'a, G13) \ \& \ \text{disjoint}(H13::'a, F13) \longrightarrow \text{disjoint}(H13::'a, G13)$)
 &
 ($\forall I13\ J13. \text{equal}(I13::'a, J13) \ \& \ \text{function}(I13) \longrightarrow \text{function}(J13)$) &
 ($\forall K13\ L13\ M13\ N13\ O13\ P13. \text{equal}(K13::'a, L13) \ \& \ \text{homomorphism}(K13::'a, M13, N13, O13, P13) \longrightarrow \text{homomorphism}(L13::'a, M13, N13, O13, P13)$) &
 ($\forall Q13\ S13\ R13\ T13\ U13\ V13. \text{equal}(Q13::'a, R13) \ \& \ \text{homomorphism}(S13::'a, Q13, T13, U13, V13) \longrightarrow \text{homomorphism}(S13::'a, R13, T13, U13, V13)$) &
 ($\forall W13\ Y13\ Z13\ X13\ A14\ B14. \text{equal}(W13::'a, X13) \ \& \ \text{homomorphism}(Y13::'a, Z13, W13, A14, B14) \longrightarrow \text{homomorphism}(Y13::'a, Z13, X13, A14, B14)$) &
 ($\forall C14\ E14\ F14\ G14\ D14\ H14. \text{equal}(C14::'a, D14) \ \& \ \text{homomorphism}(E14::'a, F14, G14, C14, H14) \longrightarrow \text{homomorphism}(E14::'a, F14, G14, D14, H14)$) &
 ($\forall I14\ K14\ L14\ M14\ N14\ J14. \text{equal}(I14::'a, J14) \ \& \ \text{homomorphism}(K14::'a, L14, M14, N14, I14) \longrightarrow \text{homomorphism}(K14::'a, L14, M14, N14, J14)$) &
 ($\forall O14\ P14. \text{equal}(O14::'a, P14) \ \& \ \text{little-set}(O14) \longrightarrow \text{little-set}(P14)$) &
 ($\forall Q14\ R14\ S14\ T14. \text{equal}(Q14::'a, R14) \ \& \ \text{maps}(Q14::'a, S14, T14) \longrightarrow \text{maps}(R14::'a, S14, T14)$)
 &
 ($\forall U14\ W14\ V14\ X14. \text{equal}(U14::'a, V14) \ \& \ \text{maps}(W14::'a, U14, X14) \longrightarrow \text{maps}(W14::'a, V14, X14)$) &
 ($\forall Y14\ A15\ B15\ Z14. \text{equal}(Y14::'a, Z14) \ \& \ \text{maps}(A15::'a, B15, Y14) \longrightarrow \text{maps}(A15::'a, B15, Z14)$)
 &
 ($\forall C15\ D15\ E15. \text{equal}(C15::'a, D15) \ \& \ \text{member}(C15::'a, E15) \longrightarrow \text{member}(D15::'a, E15)$)
 &
 ($\forall F15\ H15\ G15. \text{equal}(F15::'a, G15) \ \& \ \text{member}(H15::'a, F15) \longrightarrow \text{member}(H15::'a, G15)$)
 &
 ($\forall I15\ J15. \text{equal}(I15::'a, J15) \ \& \ \text{one-to-one-function}(I15) \longrightarrow \text{one-to-one-function}(J15)$)
 &
 ($\forall K15\ L15. \text{equal}(K15::'a, L15) \ \& \ \text{ordered-pair-predicate}(K15) \longrightarrow \text{ordered-pair-predicate}(L15)$)
 &
 ($\forall M15\ N15\ O15. \text{equal}(M15::'a, N15) \ \& \ \text{proper-subset}(M15::'a, O15) \longrightarrow \text{proper-subset}(N15::'a, O15)$)
 &
 ($\forall P15\ R15\ Q15. \text{equal}(P15::'a, Q15) \ \& \ \text{proper-subset}(R15::'a, P15) \longrightarrow \text{proper-subset}(R15::'a, Q15)$)
 &
 ($\forall S15\ T15. \text{equal}(S15::'a, T15) \ \& \ \text{relation}(S15) \longrightarrow \text{relation}(T15)$) &
 ($\forall U15\ V15. \text{equal}(U15::'a, V15) \ \& \ \text{single-valued-set}(U15) \longrightarrow \text{single-valued-set}(V15)$)
 &
 ($\forall W15\ X15\ Y15. \text{equal}(W15::'a, X15) \ \& \ \text{ssubset}(W15::'a, Y15) \longrightarrow \text{ssubset}(X15::'a, Y15)$)
 &
 ($\forall Z15\ B16\ A16. \text{equal}(Z15::'a, A16) \ \& \ \text{ssubset}(B16::'a, Z15) \longrightarrow \text{ssubset}(B16::'a, A16)$)
 &
 ($\sim \text{little-set}(\text{ordered-pair}(a::'a, b))) \longrightarrow \text{False}$
oops

lemma SET046-5:

($\forall Y\ X. \sim (\text{element}(X::'a, a) \ \& \ \text{element}(X::'a, Y) \ \& \ \text{element}(Y::'a, X))$) &
 ($\forall X. \text{element}(X::'a, f(X)) \mid \text{element}(X::'a, a)$) &
 ($\forall X. \text{element}(f(X), X) \mid \text{element}(X::'a, a)$) $\longrightarrow \text{False}$

by *meson*

lemma *SET047-5*:

$(\forall X Z Y. \text{set-equal}(X::'a, Y) \ \& \ \text{element}(Z::'a, X) \longrightarrow \text{element}(Z::'a, Y)) \ \&$
 $(\forall Y Z X. \text{set-equal}(X::'a, Y) \ \& \ \text{element}(Z::'a, Y) \longrightarrow \text{element}(Z::'a, X)) \ \&$
 $(\forall X Y. \text{element}(f(X::'a, Y), X) \mid \text{element}(f(X::'a, Y), Y) \mid \text{set-equal}(X::'a, Y))$
 $\&$
 $(\forall X Y. \text{element}(f(X::'a, Y), Y) \ \& \ \text{element}(f(X::'a, Y), X) \longrightarrow \text{set-equal}(X::'a, Y))$
 $\&$
 $(\text{set-equal}(a::'a, b) \mid \text{set-equal}(b::'a, a)) \ \&$
 $(\sim(\text{set-equal}(b::'a, a) \ \& \ \text{set-equal}(a::'a, b))) \longrightarrow \text{False}$
by *meson*

lemma *SYN034-1*:

$(\forall A. p(A::'a, a) \mid p(A::'a, f(A))) \ \&$
 $(\forall A. p(A::'a, a) \mid p(f(A), A)) \ \&$
 $(\forall A B. \sim(p(A::'a, B) \ \& \ p(B::'a, A) \ \& \ p(B::'a, a))) \longrightarrow \text{False}$
by *meson*

lemma *SYN071-1*:

$\text{EQU001-0-ax equal} \ \&$
 $(\text{equal}(a::'a, b) \mid \text{equal}(c::'a, d)) \ \&$
 $(\text{equal}(a::'a, c) \mid \text{equal}(b::'a, d)) \ \&$
 $(\sim \text{equal}(a::'a, d)) \ \&$
 $(\sim \text{equal}(b::'a, c)) \longrightarrow \text{False}$
by *meson*

lemma *SYN349-1*:

$(\forall X Y. f(w(X), g(X::'a, Y)) \longrightarrow f(X::'a, g(X::'a, Y))) \ \&$
 $(\forall X Y. f(X::'a, g(X::'a, Y)) \longrightarrow f(w(X), g(X::'a, Y))) \ \&$
 $(\forall Y X. f(X::'a, g(X::'a, Y)) \ \& \ f(Y::'a, g(X::'a, Y)) \longrightarrow f(g(X::'a, Y), Y) \mid$
 $f(g(X::'a, Y), w(X))) \ \&$
 $(\forall Y X. f(g(X::'a, Y), Y) \ \& \ f(Y::'a, g(X::'a, Y)) \longrightarrow f(X::'a, g(X::'a, Y)) \mid$
 $f(g(X::'a, Y), w(X))) \ \&$
 $(\forall Y X. f(X::'a, g(X::'a, Y)) \mid f(g(X::'a, Y), Y) \mid f(Y::'a, g(X::'a, Y)) \mid f(g(X::'a, Y), w(X)))$
 $\&$
 $(\forall Y X. f(X::'a, g(X::'a, Y)) \ \& \ f(g(X::'a, Y), Y) \longrightarrow f(Y::'a, g(X::'a, Y)) \mid$
 $f(g(X::'a, Y), w(X))) \ \&$
 $(\forall Y X. f(X::'a, g(X::'a, Y)) \ \& \ f(g(X::'a, Y), w(X)) \longrightarrow f(g(X::'a, Y), Y) \mid$
 $f(Y::'a, g(X::'a, Y))) \ \&$
 $(\forall Y X. f(g(X::'a, Y), Y) \ \& \ f(g(X::'a, Y), w(X)) \longrightarrow f(X::'a, g(X::'a, Y)) \mid$
 $f(Y::'a, g(X::'a, Y))) \ \&$
 $(\forall Y X. f(Y::'a, g(X::'a, Y)) \ \& \ f(g(X::'a, Y), w(X)) \longrightarrow f(X::'a, g(X::'a, Y)) \mid$
 $f(g(X::'a, Y), Y)) \ \&$
 $(\forall Y X. \sim(f(X::'a, g(X::'a, Y)) \ \& \ f(g(X::'a, Y), Y) \ \& \ f(Y::'a, g(X::'a, Y)) \ \&$

$f(g(X::'a, Y), w(X))) \dashrightarrow \text{False}$
oops

lemma SYN352-1:

$(f(a::'a, b)) \ \&$
 $(\forall X \ Y. f(X::'a, Y) \dashrightarrow f(b::'a, z(X::'a, Y)) \mid f(Y::'a, z(X::'a, Y))) \ \&$
 $(\forall X \ Y. f(X::'a, Y) \mid f(z(X::'a, Y), z(X::'a, Y))) \ \&$
 $(\forall X \ Y. f(b::'a, z(X::'a, Y)) \mid f(X::'a, z(X::'a, Y)) \mid f(z(X::'a, Y), z(X::'a, Y))) \ \&$
 $(\forall X \ Y. f(b::'a, z(X::'a, Y)) \ \& f(X::'a, z(X::'a, Y)) \dashrightarrow f(z(X::'a, Y), z(X::'a, Y)))$
 $\ \&$
 $(\forall X \ Y. \sim(f(X::'a, Y) \ \& f(X::'a, z(X::'a, Y)) \ \& f(Y::'a, z(X::'a, Y)))) \ \&$
 $(\forall X \ Y. f(X::'a, Y) \dashrightarrow f(X::'a, z(X::'a, Y)) \mid f(Y::'a, z(X::'a, Y))) \dashrightarrow \text{False}$
by meson

lemma TOP001-2:

$(\forall Vf \ U. \text{element-of-set}(U::'a, \text{union-of-members}(Vf)) \dashrightarrow \text{element-of-set}(U::'a, f1(Vf::'a, U)))$
 $\ \&$
 $(\forall U \ Vf. \text{element-of-set}(U::'a, \text{union-of-members}(Vf)) \dashrightarrow \text{element-of-collection}(f1(Vf::'a, U), Vf))$
 $\ \&$
 $(\forall U \ Uu1 \ Vf. \text{element-of-set}(U::'a, Uu1) \ \& \text{element-of-collection}(Uu1::'a, Vf)$
 $\dashrightarrow \text{element-of-set}(U::'a, \text{union-of-members}(Vf))) \ \&$
 $(\forall Vf \ X. \text{basis}(X::'a, Vf) \dashrightarrow \text{equal-sets}(\text{union-of-members}(Vf), X)) \ \&$
 $(\forall Vf \ U \ X. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \text{element-of-set}(X::'a, U)$
 $\dashrightarrow \text{element-of-set}(X::'a, f10(Vf::'a, U, X))) \ \&$
 $(\forall U \ X \ Vf. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \text{element-of-set}(X::'a, U)$
 $\dashrightarrow \text{element-of-collection}(f10(Vf::'a, U, X), Vf)) \ \&$
 $(\forall X. \text{subset-sets}(X::'a, X)) \ \&$
 $(\forall X \ U \ Y. \text{subset-sets}(X::'a, Y) \ \& \text{element-of-set}(U::'a, X) \dashrightarrow \text{element-of-set}(U::'a, Y))$
 $\ \&$
 $(\forall X \ Y. \text{equal-sets}(X::'a, Y) \dashrightarrow \text{subset-sets}(X::'a, Y)) \ \&$
 $(\forall Y \ X. \text{subset-sets}(X::'a, Y) \mid \text{element-of-set}(\text{in-1st-set}(X::'a, Y), X)) \ \&$
 $(\forall X \ Y. \text{element-of-set}(\text{in-1st-set}(X::'a, Y), Y) \dashrightarrow \text{subset-sets}(X::'a, Y)) \ \&$
 $(\text{basis}(cx::'a, f)) \ \&$
 $(\sim \text{subset-sets}(\text{union-of-members}(\text{top-of-basis}(f)), cx)) \dashrightarrow \text{False}$
by meson

lemma TOP002-2:

$(\forall Vf \ U. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \mid \text{element-of-set}(f11(Vf::'a, U), U))$
 $\ \&$
 $(\forall X. \sim \text{element-of-set}(X::'a, \text{empty-set})) \ \&$
 $(\sim \text{element-of-collection}(\text{empty-set}::'a, \text{top-of-basis}(f))) \dashrightarrow \text{False}$
by meson

lemma TOP004-1:

$(\forall Vf \ U. \text{element-of-set}(U::'a, \text{union-of-members}(Vf)) \dashrightarrow \text{element-of-set}(U::'a, f1(Vf::'a, U)))$

$\&$
 $(\forall U Vf. \text{element-of-set}(U::'a, \text{union-of-members}(Vf)) \longrightarrow \text{element-of-collection}(f1(Vf::'a, U), Vf))$
 $\&$
 $(\forall U Uu1 Vf. \text{element-of-set}(U::'a, Uu1) \ \& \ \text{element-of-collection}(Uu1::'a, Vf) \longrightarrow \text{element-of-set}(U::'a, \text{union-of-members}(Vf))) \ \&$
 $(\forall Vf U Va. \text{element-of-set}(U::'a, \text{intersection-of-members}(Vf)) \ \& \ \text{element-of-collection}(Va::'a, Vf) \longrightarrow \text{element-of-set}(U::'a, Va)) \ \&$
 $(\forall U Vf. \text{element-of-set}(U::'a, \text{intersection-of-members}(Vf)) \mid \text{element-of-collection}(f2(Vf::'a, U), Vf))$
 $\&$
 $(\forall Vf U. \text{element-of-set}(U::'a, f2(Vf::'a, U)) \longrightarrow \text{element-of-set}(U::'a, \text{intersection-of-members}(Vf)))$
 $\&$
 $(\forall Vt X. \text{topological-space}(X::'a, Vt) \longrightarrow \text{equal-sets}(\text{union-of-members}(Vt), X))$
 $\&$
 $(\forall X Vt. \text{topological-space}(X::'a, Vt) \longrightarrow \text{element-of-collection}(\text{empty-set}::'a, Vt))$
 $\&$
 $(\forall X Vt. \text{topological-space}(X::'a, Vt) \longrightarrow \text{element-of-collection}(X::'a, Vt)) \ \&$
 $(\forall X Y Z Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{element-of-collection}(Y::'a, Vt) \ \& \ \text{element-of-collection}(Z::'a, Vt) \longrightarrow \text{element-of-collection}(\text{intersection-of-sets}(Y::'a, Z), Vt))$
 $\&$
 $(\forall X Vf Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-collections}(Vf::'a, Vt) \longrightarrow \text{element-of-collection}(\text{union-of-members}(Vf), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f3(X::'a, Vt), Vt) \mid \text{subset-collections}(f5(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{union-of-members}(f5(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f3(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f4(X::'a, Vt), Vt) \mid \text{subset-collections}(f5(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{union-of-members}(f5(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f4(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{intersection-of-sets}(f3(X::'a, Vt), f4(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{subset-collections}(f5(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{intersection-of-sets}(f3(X::'a, Vt), f4(X::'a, Vt)), Vt) \ \& \ \text{element-of-collection}(\text{union-of-members}(f5(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt))$
 $\&$
 $(\forall U X Vt. \text{open}(U::'a, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall X U Vt. \text{open}(U::'a, X, Vt) \longrightarrow \text{element-of-collection}(U::'a, Vt)) \ \&$
 $(\forall X U Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{element-of-collection}(U::'a, Vt) \longrightarrow \text{open}(U::'a, X, Vt)) \ \&$
 $(\forall U X Vt. \text{closed}(U::'a, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall U X Vt. \text{closed}(U::'a, X, Vt) \longrightarrow \text{open}(\text{relative-complement-sets}(U::'a, X), X, Vt))$
 $\&$
 $(\forall U X Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{open}(\text{relative-complement-sets}(U::'a, X), X, Vt) \longrightarrow \text{closed}(U::'a, X, Vt)) \ \&$

$(\forall Vs\ X\ Vt.\ finer(Vt::'a, Vs, X) \longrightarrow topological-space(X::'a, Vt)) \ \&$
 $(\forall Vt\ X\ Vs.\ finer(Vt::'a, Vs, X) \longrightarrow topological-space(X::'a, Vs)) \ \&$
 $(\forall X\ Vs\ Vt.\ finer(Vt::'a, Vs, X) \longrightarrow subset-collections(Vs::'a, Vt)) \ \&$
 $(\forall X\ Vs\ Vt.\ topological-space(X::'a, Vt) \ \& \ topological-space(X::'a, Vs) \ \& \ subset-collections(Vs::'a, Vt) \longrightarrow finer(Vt::'a, Vs, X)) \ \&$
 $(\forall Vf\ X.\ basis(X::'a, Vf) \longrightarrow equal-sets(union-of-members(Vf), X)) \ \&$
 $(\forall X\ Vf\ Y\ Vb1\ Vb2.\ basis(X::'a, Vf) \ \& \ element-of-set(Y::'a, X) \ \& \ element-of-collection(Vb1::'a, Vf) \ \& \ element-of-collection(Vb2::'a, Vf) \ \& \ element-of-set(Y::'a, intersection-of-sets(Vb1::'a, Vb2)) \longrightarrow element-of-set(Y::'a, f6(X::'a, Vf, Y, Vb1, Vb2))) \ \&$
 $(\forall X\ Y\ Vb1\ Vb2\ Vf.\ basis(X::'a, Vf) \ \& \ element-of-set(Y::'a, X) \ \& \ element-of-collection(Vb1::'a, Vf) \ \& \ element-of-collection(Vb2::'a, Vf) \ \& \ element-of-set(Y::'a, intersection-of-sets(Vb1::'a, Vb2)) \longrightarrow element-of-collection(f6(X::'a, Vf, Y, Vb1, Vb2), Vf)) \ \&$
 $(\forall X\ Vf\ Y\ Vb1\ Vb2.\ basis(X::'a, Vf) \ \& \ element-of-set(Y::'a, X) \ \& \ element-of-collection(Vb1::'a, Vf) \ \& \ element-of-collection(Vb2::'a, Vf) \ \& \ element-of-set(Y::'a, intersection-of-sets(Vb1::'a, Vb2)) \longrightarrow subset-sets(f6(X::'a, Vf, Y, Vb1, Vb2), intersection-of-sets(Vb1::'a, Vb2))) \ \&$
 $(\forall Vf\ X.\ equal-sets(union-of-members(Vf), X) \longrightarrow basis(X::'a, Vf) \mid element-of-set(f7(X::'a, Vf), X)) \ \&$
 $(\forall X\ Vf.\ equal-sets(union-of-members(Vf), X) \longrightarrow basis(X::'a, Vf) \mid element-of-collection(f8(X::'a, Vf), Vf)) \ \&$
 $(\forall X\ Vf.\ equal-sets(union-of-members(Vf), X) \longrightarrow basis(X::'a, Vf) \mid element-of-collection(f9(X::'a, Vf), Vf)) \ \&$
 $(\forall X\ Vf.\ equal-sets(union-of-members(Vf), X) \longrightarrow basis(X::'a, Vf) \mid element-of-set(f7(X::'a, Vf), intersection-of-sets(f8(X::'a, Vf), f9(X::'a, Vf)))) \ \&$
 $(\forall Uu9\ X\ Vf.\ equal-sets(union-of-members(Vf), X) \ \& \ element-of-set(f7(X::'a, Vf), Uu9) \ \& \ element-of-collection(Uu9::'a, Vf) \ \& \ subset-sets(Uu9::'a, intersection-of-sets(f8(X::'a, Vf), f9(X::'a, Vf))) \longrightarrow basis(X::'a, Vf)) \ \&$
 $(\forall Vf\ U\ X.\ element-of-collection(U::'a, top-of-basis(Vf)) \ \& \ element-of-set(X::'a, U) \longrightarrow element-of-set(X::'a, f10(Vf::'a, U, X))) \ \&$
 $(\forall U\ X\ Vf.\ element-of-collection(U::'a, top-of-basis(Vf)) \ \& \ element-of-set(X::'a, U) \longrightarrow element-of-collection(f10(Vf::'a, U, X), Vf)) \ \&$
 $(\forall Vf\ X\ U.\ element-of-collection(U::'a, top-of-basis(Vf)) \ \& \ element-of-set(X::'a, U) \longrightarrow subset-sets(f10(Vf::'a, U, X), U)) \ \&$
 $(\forall Vf\ U.\ element-of-collection(U::'a, top-of-basis(Vf)) \mid element-of-set(f11(Vf::'a, U), U)) \ \&$
 $(\forall Vf\ Uu11\ U.\ element-of-set(f11(Vf::'a, U), Uu11) \ \& \ element-of-collection(Uu11::'a, Vf) \ \& \ subset-sets(Uu11::'a, U) \longrightarrow element-of-collection(U::'a, top-of-basis(Vf))) \ \&$
 $(\forall U\ Y\ X\ Vt.\ element-of-collection(U::'a, subspace-topology(X::'a, Vt, Y)) \longrightarrow topological-space(X::'a, Vt)) \ \&$
 $(\forall U\ Vt\ Y\ X.\ element-of-collection(U::'a, subspace-topology(X::'a, Vt, Y)) \longrightarrow subset-sets(Y::'a, X)) \ \&$
 $(\forall X\ Y\ U\ Vt.\ element-of-collection(U::'a, subspace-topology(X::'a, Vt, Y)) \longrightarrow element-of-collection(f12(X::'a, Vt, Y, U), Vt)) \ \&$
 $(\forall X\ Vt\ Y\ U.\ element-of-collection(U::'a, subspace-topology(X::'a, Vt, Y)) \longrightarrow equal-sets(U::'a, intersection-of-sets(Y::'a, f12(X::'a, Vt, Y, U)))) \ \&$
 $(\forall X\ Vt\ U\ Y\ Uu12.\ topological-space(X::'a, Vt) \ \& \ subset-sets(Y::'a, X) \ \& \ element-of-collection(Uu12::'a, Vt) \ \& \ equal-sets(U::'a, intersection-of-sets(Y::'a, Uu12)) \longrightarrow element-of-collection(U::'a, subspace-topology(X::'a, Vt, Y))) \ \&$
 $(\forall U\ Y\ X\ Vt.\ element-of-set(U::'a, interior(Y::'a, X, Vt)) \longrightarrow topological-space(X::'a, Vt)) \ \&$

$(\forall U \ Vt \ Y \ X. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{subset-sets}(Y::'a, X))$
 $\&$
 $(\forall Y \ X \ Vt \ U. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{element-of-set}(U::'a, f13(Y::'a, X, Vt, U)))$
 $\&$
 $(\forall X \ Vt \ U \ Y. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{subset-sets}(f13(Y::'a, X, Vt, U), Y))$
 $\&$
 $(\forall Y \ U \ X \ Vt. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{open}(f13(Y::'a, X, Vt, U), X, Vt))$
 $\&$
 $(\forall U \ Y \ Uu13 \ X \ Vt. \text{topological-space}(X::'a, Vt) \& \text{subset-sets}(Y::'a, X) \& \text{element-of-set}(U::'a, Uu13)$
 $\& \text{subset-sets}(Uu13::'a, Y) \& \text{open}(Uu13::'a, X, Vt) \longrightarrow \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)))$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)) \longrightarrow \text{topological-space}(X::'a, Vt))$
 $\&$
 $(\forall U \ Vt \ Y \ X. \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)) \longrightarrow \text{subset-sets}(Y::'a, X))$
 $\&$
 $(\forall Y \ X \ Vt \ U \ V. \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)) \& \text{subset-sets}(Y::'a, V)$
 $\& \text{closed}(V::'a, X, Vt) \longrightarrow \text{element-of-set}(U::'a, V)) \&$
 $(\forall Y \ X \ Vt \ U. \text{topological-space}(X::'a, Vt) \& \text{subset-sets}(Y::'a, X) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X,$
 $| \text{subset-sets}(Y::'a, f14(Y::'a, X, Vt, U))) \&$
 $(\forall Y \ U \ X \ Vt. \text{topological-space}(X::'a, Vt) \& \text{subset-sets}(Y::'a, X) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X,$
 $| \text{closed}(f14(Y::'a, X, Vt, U), X, Vt)) \&$
 $(\forall Y \ X \ Vt \ U. \text{topological-space}(X::'a, Vt) \& \text{subset-sets}(Y::'a, X) \& \text{element-of-set}(U::'a, f14(Y::'a, X, Vt, U))$
 $\longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt))) \&$
 $(\forall U \ Y \ X \ Vt. \text{neighborhood}(U::'a, Y, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \&$
 $(\forall Y \ U \ X \ Vt. \text{neighborhood}(U::'a, Y, X, Vt) \longrightarrow \text{open}(U::'a, X, Vt)) \&$
 $(\forall X \ Vt \ Y \ U. \text{neighborhood}(U::'a, Y, X, Vt) \longrightarrow \text{element-of-set}(Y::'a, U)) \&$
 $(\forall X \ Vt \ Y \ U. \text{topological-space}(X::'a, Vt) \& \text{open}(U::'a, X, Vt) \& \text{element-of-set}(Y::'a, U)$
 $\longrightarrow \text{neighborhood}(U::'a, Y, X, Vt)) \&$
 $(\forall Z \ Y \ X \ Vt. \text{limit-point}(Z::'a, Y, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \&$
 $(\forall Z \ Vt \ Y \ X. \text{limit-point}(Z::'a, Y, X, Vt) \longrightarrow \text{subset-sets}(Y::'a, X)) \&$
 $(\forall Z \ X \ Vt \ U \ Y. \text{limit-point}(Z::'a, Y, X, Vt) \& \text{neighborhood}(U::'a, Z, X, Vt) \longrightarrow$
 $\text{element-of-set}(f15(Z::'a, Y, X, Vt, U), \text{intersection-of-sets}(U::'a, Y))) \&$
 $(\forall Y \ X \ Vt \ U \ Z. \sim(\text{limit-point}(Z::'a, Y, X, Vt) \& \text{neighborhood}(U::'a, Z, X, Vt) \&$
 $\text{eq-p}(f15(Z::'a, Y, X, Vt, U), Z))) \&$
 $(\forall Y \ Z \ X \ Vt. \text{topological-space}(X::'a, Vt) \& \text{subset-sets}(Y::'a, X) \longrightarrow \text{limit-point}(Z::'a, Y, X, Vt)$
 $| \text{neighborhood}(f16(Z::'a, Y, X, Vt), Z, X, Vt)) \&$
 $(\forall X \ Vt \ Y \ Uu16 \ Z. \text{topological-space}(X::'a, Vt) \& \text{subset-sets}(Y::'a, X) \& \text{element-of-set}(Uu16::'a, \text{intersection}$
 $\longrightarrow \text{limit-point}(Z::'a, Y, X, Vt) | \text{eq-p}(Uu16::'a, Z)) \&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{boundary}(Y::'a, X, Vt)) \longrightarrow \text{topological-space}(X::'a, Vt))$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{boundary}(Y::'a, X, Vt)) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)))$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{boundary}(Y::'a, X, Vt)) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(\text{relative-complemen}$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{topological-space}(X::'a, Vt) \& \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt))$
 $\& \text{element-of-set}(U::'a, \text{closure}(\text{relative-complement-sets}(Y::'a, X), X, Vt)) \longrightarrow \text{element-of-set}(U::'a, \text{boundary}$
 $\&$
 $(\forall X \ Vt. \text{hausdorff}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \&$
 $(\forall X-2 \ X-1 \ X \ Vt. \text{hausdorff}(X::'a, Vt) \& \text{element-of-set}(X-1::'a, X) \& \text{element-of-set}(X-2::'a, X)$

$$\begin{aligned}
& \rightarrow eq-p(X-1::'a, X-2) \mid neighborhood(f17(X::'a, Vt, X-1, X-2), X-1, X, Vt)) \& \\
& (\forall X-1 X-2 X Vt. hausdorff(X::'a, Vt) \& element-of-set(X-1::'a, X) \& element-of-set(X-2::'a, X)) \\
& \rightarrow eq-p(X-1::'a, X-2) \mid neighborhood(f18(X::'a, Vt, X-1, X-2), X-2, X, Vt)) \& \\
& (\forall X Vt X-1 X-2. hausdorff(X::'a, Vt) \& element-of-set(X-1::'a, X) \& element-of-set(X-2::'a, X)) \\
& \rightarrow eq-p(X-1::'a, X-2) \mid disjoint-s(f17(X::'a, Vt, X-1, X-2), f18(X::'a, Vt, X-1, X-2))) \\
& \& \\
& (\forall Vt X. topological-space(X::'a, Vt) \rightarrow hausdorff(X::'a, Vt) \mid element-of-set(f19(X::'a, Vt), X)) \\
& \& \\
& (\forall Vt X. topological-space(X::'a, Vt) \rightarrow hausdorff(X::'a, Vt) \mid element-of-set(f20(X::'a, Vt), X)) \\
& \& \\
& (\forall X Vt. topological-space(X::'a, Vt) \& eq-p(f19(X::'a, Vt), f20(X::'a, Vt)) \rightarrow \\
& hausdorff(X::'a, Vt)) \& \\
& (\forall X Vt Uu19 Uu20. topological-space(X::'a, Vt) \& neighborhood(Uu19::'a, f19(X::'a, Vt), X, Vt) \\
& \& neighborhood(Uu20::'a, f20(X::'a, Vt), X, Vt) \& disjoint-s(Uu19::'a, Uu20) \rightarrow \\
& hausdorff(X::'a, Vt)) \& \\
& (\forall Va1 Va2 X Vt. separation(Va1::'a, Va2, X, Vt) \rightarrow topological-space(X::'a, Vt)) \\
& \& \\
& (\forall Va2 X Vt Va1. \sim(separation(Va1::'a, Va2, X, Vt) \& equal-sets(Va1::'a, empty-set))) \\
& \& \\
& (\forall Va1 X Vt Va2. \sim(separation(Va1::'a, Va2, X, Vt) \& equal-sets(Va2::'a, empty-set))) \\
& \& \\
& (\forall Va2 X Va1 Vt. separation(Va1::'a, Va2, X, Vt) \rightarrow element-of-collection(Va1::'a, Vt)) \\
& \& \\
& (\forall Va1 X Va2 Vt. separation(Va1::'a, Va2, X, Vt) \rightarrow element-of-collection(Va2::'a, Vt)) \\
& \& \\
& (\forall Vt Va1 Va2 X. separation(Va1::'a, Va2, X, Vt) \rightarrow equal-sets(union-of-sets(Va1::'a, Va2), X)) \\
& \& \\
& (\forall X Vt Va1 Va2. separation(Va1::'a, Va2, X, Vt) \rightarrow disjoint-s(Va1::'a, Va2)) \\
& \& \\
& (\forall Vt X Va1 Va2. topological-space(X::'a, Vt) \& element-of-collection(Va1::'a, Vt) \\
& \& element-of-collection(Va2::'a, Vt) \& equal-sets(union-of-sets(Va1::'a, Va2), X) \& \\
& disjoint-s(Va1::'a, Va2) \rightarrow separation(Va1::'a, Va2, X, Vt) \mid equal-sets(Va1::'a, empty-set) \\
& \mid equal-sets(Va2::'a, empty-set)) \& \\
& (\forall X Vt. connected-space(X::'a, Vt) \rightarrow topological-space(X::'a, Vt)) \& \\
& (\forall Va1 Va2 X Vt. \sim(connected-space(X::'a, Vt) \& separation(Va1::'a, Va2, X, Vt))) \\
& \& \\
& (\forall X Vt. topological-space(X::'a, Vt) \rightarrow connected-space(X::'a, Vt) \mid separa- \\
& tion(f21(X::'a, Vt), f22(X::'a, Vt), X, Vt)) \& \\
& (\forall Va X Vt. connected-set(Va::'a, X, Vt) \rightarrow topological-space(X::'a, Vt)) \& \\
& (\forall Vt Va X. connected-set(Va::'a, X, Vt) \rightarrow subset-sets(Va::'a, X)) \& \\
& (\forall X Vt Va. connected-set(Va::'a, X, Vt) \rightarrow connected-space(Va::'a, subspace-topology(X::'a, Vt, Va))) \\
& \& \\
& (\forall X Vt Va. topological-space(X::'a, Vt) \& subset-sets(Va::'a, X) \& connected-space(Va::'a, subspace-topology(X::'a, Vt, Va)) \\
& \rightarrow connected-set(Va::'a, X, Vt)) \& \\
& (\forall Vf X Vt. open-covering(Vf::'a, X, Vt) \rightarrow topological-space(X::'a, Vt)) \& \\
& (\forall X Vf Vt. open-covering(Vf::'a, X, Vt) \rightarrow subset-collections(Vf::'a, Vt)) \& \\
& (\forall Vt Vf X. open-covering(Vf::'a, X, Vt) \rightarrow equal-sets(union-of-members(Vf), X)) \\
& \& \\
& (\forall Vt Vf X. topological-space(X::'a, Vt) \& subset-collections(Vf::'a, Vt) \& equal-sets(union-of-members(Vf), X))
\end{aligned}$$

$$\begin{aligned}
& \longrightarrow \text{open-covering}(Vf::'a, X, Vt)) \ \& \\
& (\forall X \ Vt. \text{compact-space}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \& \\
& (\forall X \ Vt \ Vf1. \text{compact-space}(X::'a, Vt) \ \& \ \text{open-covering}(Vf1::'a, X, Vt) \longrightarrow \text{finite}'(f23(X::'a, Vt, Vf1))) \ \& \\
& (\forall X \ Vt \ Vf1. \text{compact-space}(X::'a, Vt) \ \& \ \text{open-covering}(Vf1::'a, X, Vt) \longrightarrow \text{subset-collections}(f23(X::'a, Vt, Vf1))) \ \& \\
& (\forall Vf1 \ X \ Vt. \text{compact-space}(X::'a, Vt) \ \& \ \text{open-covering}(Vf1::'a, X, Vt) \longrightarrow \text{open-covering}(f23(X::'a, Vt, Vf1))) \ \& \\
& (\forall X \ Vt. \text{topological-space}(X::'a, Vt) \longrightarrow \text{compact-space}(X::'a, Vt) \mid \text{open-covering}(f24(X::'a, Vt), X, Vt)) \ \& \\
& (\forall Uu24 \ X \ Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{finite}'(Uu24) \ \& \ \text{subset-collections}(Uu24::'a, f24(X::'a, Vt))) \ \& \\
& \text{open-covering}(Uu24::'a, X, Vt) \longrightarrow \text{compact-space}(X::'a, Vt)) \ \& \\
& (\forall Va \ X \ Vt. \text{compact-set}(Va::'a, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \& \\
& (\forall Vt \ Va \ X. \text{compact-set}(Va::'a, X, Vt) \longrightarrow \text{subset-sets}(Va::'a, X)) \ \& \\
& (\forall X \ Vt \ Va. \text{compact-set}(Va::'a, X, Vt) \longrightarrow \text{compact-space}(Va::'a, \text{subspace-topology}(X::'a, Vt, Va))) \ \& \\
& (\forall X \ Vt \ Va. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Va::'a, X) \ \& \ \text{compact-space}(Va::'a, \text{subspace-topology}(X::'a, Vt, Va))) \ \& \\
& \longrightarrow \text{compact-set}(Va::'a, X, Vt)) \ \& \\
& (\text{basis}(cx::'a, f)) \ \& \\
& (\forall U. \text{element-of-collection}(U::'a, \text{top-of-basis}(f))) \ \& \\
& (\forall V. \text{element-of-collection}(V::'a, \text{top-of-basis}(f))) \ \& \\
& (\forall U \ V. \sim \text{element-of-collection}(\text{intersection-of-sets}(U::'a, V), \text{top-of-basis}(f))) \longrightarrow \\
& \text{False} \\
& \text{by meson}
\end{aligned}$$

lemma TOP004-2:

$$\begin{aligned}
& (\forall U \ Uu1 \ Vf. \text{element-of-set}(U::'a, Uu1) \ \& \ \text{element-of-collection}(Uu1::'a, Vf) \longrightarrow \\
& \text{element-of-set}(U::'a, \text{union-of-members}(Vf))) \ \& \\
& (\forall Vf \ X. \text{basis}(X::'a, Vf) \longrightarrow \text{equal-sets}(\text{union-of-members}(Vf), X)) \ \& \\
& (\forall X \ Vf \ Y \ Vb1 \ Vb2. \text{basis}(X::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, X) \ \& \ \text{element-of-collection}(Vb1::'a, Vf) \\
& \ \& \ \text{element-of-collection}(Vb2::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, \text{intersection-of-sets}(Vb1::'a, Vb2)) \\
& \longrightarrow \text{element-of-set}(Y::'a, f6(X::'a, Vf, Y, Vb1, Vb2))) \ \& \\
& (\forall X \ Y \ Vb1 \ Vb2 \ Vf. \text{basis}(X::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, X) \ \& \ \text{element-of-collection}(Vb1::'a, Vf) \\
& \ \& \ \text{element-of-collection}(Vb2::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, \text{intersection-of-sets}(Vb1::'a, Vb2)) \\
& \longrightarrow \text{element-of-collection}(f6(X::'a, Vf, Y, Vb1, Vb2), Vf)) \ \& \\
& (\forall X \ Vf \ Y \ Vb1 \ Vb2. \text{basis}(X::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, X) \ \& \ \text{element-of-collection}(Vb1::'a, Vf) \\
& \ \& \ \text{element-of-collection}(Vb2::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, \text{intersection-of-sets}(Vb1::'a, Vb2)) \\
& \longrightarrow \text{subset-sets}(f6(X::'a, Vf, Y, Vb1, Vb2), \text{intersection-of-sets}(Vb1::'a, Vb2))) \ \& \\
& (\forall Vf \ U \ X. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \ \text{element-of-set}(X::'a, U) \\
& \longrightarrow \text{element-of-set}(X::'a, f10(Vf::'a, U, X))) \ \& \\
& (\forall U \ X \ Vf. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \ \text{element-of-set}(X::'a, U) \\
& \longrightarrow \text{element-of-collection}(f10(Vf::'a, U, X), Vf)) \ \& \\
& (\forall Vf \ X \ U. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \ \text{element-of-set}(X::'a, U) \\
& \longrightarrow \text{subset-sets}(f10(Vf::'a, U, X), U)) \ \& \\
& (\forall Vf \ U. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \mid \text{element-of-set}(f11(Vf::'a, U), U)) \ \& \\
& (\forall Vf \ Uu11 \ U. \text{element-of-set}(f11(Vf::'a, U), Uu11) \ \& \ \text{element-of-collection}(Uu11::'a, Vf))
\end{aligned}$$

```

& subset-sets(Uu11::'a,U) --> element-of-collection(U::'a,top-of-basis(Vf))) &
  (∀ Y X Z. subset-sets(X::'a,Y) & subset-sets(Y::'a,Z) --> subset-sets(X::'a,Z))
&
  (∀ Y Z X. element-of-set(Z::'a,intersection-of-sets(X::'a,Y)) --> element-of-set(Z::'a,X))
&
  (∀ X Z Y. element-of-set(Z::'a,intersection-of-sets(X::'a,Y)) --> element-of-set(Z::'a,Y))
&
  (∀ X Z Y. element-of-set(Z::'a,X) & element-of-set(Z::'a,Y) --> element-of-set(Z::'a,intersection-of-sets(X::'a,Y)))
&
  (∀ X U Y V. subset-sets(X::'a,Y) & subset-sets(U::'a,V) --> subset-sets(intersection-of-sets(X::'a,U),intersection-of-sets(Y::'a,V)))
&
  (∀ X Z Y. equal-sets(X::'a,Y) & element-of-set(Z::'a,X) --> element-of-set(Z::'a,Y))
&
  (∀ Y X. equal-sets(intersection-of-sets(X::'a,Y),intersection-of-sets(Y::'a,X))) &
  (basis(cx::'a,f)) &
  (∀ U. element-of-collection(U::'a,top-of-basis(f))) &
  (∀ V. element-of-collection(V::'a,top-of-basis(f))) &
  (∀ U V. ~ element-of-collection(intersection-of-sets(U::'a,V),top-of-basis(f))) -->
False
  by meson

```

lemma TOP005-2:

```

  (∀ Vf U. element-of-set(U::'a,union-of-members(Vf)) --> element-of-set(U::'a,f1(Vf::'a,U)))
&
  (∀ U Vf. element-of-set(U::'a,union-of-members(Vf)) --> element-of-collection(f1(Vf::'a,U),Vf))
&
  (∀ Vf U X. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
--> element-of-set(X::'a,f10(Vf::'a,U,X))) &
  (∀ U X Vf. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
--> element-of-collection(f10(Vf::'a,U,X),Vf)) &
  (∀ Vf X U. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
--> subset-sets(f10(Vf::'a,U,X),U)) &
  (∀ Vf U. element-of-collection(U::'a,top-of-basis(Vf)) | element-of-set(f11(Vf::'a,U),U))
&
  (∀ Vf Uu11 U. element-of-set(f11(Vf::'a,U),Uu11) & element-of-collection(Uu11::'a,Vf))
& subset-sets(Uu11::'a,U) --> element-of-collection(U::'a,top-of-basis(Vf))) &
  (∀ X U Y. element-of-set(U::'a,X) --> subset-sets(X::'a,Y) | element-of-set(U::'a,Y))
&
  (∀ Y X Z. subset-sets(X::'a,Y) & element-of-collection(Y::'a,Z) --> subset-sets(X::'a,union-of-members(Z::'a,Y)))
&
  (∀ X U Y. subset-collections(X::'a,Y) & element-of-collection(U::'a,X) -->
element-of-collection(U::'a,Y)) &
  (subset-collections(g::'a,top-of-basis(f))) &
  (~ element-of-collection(union-of-members(g),top-of-basis(f))) --> False
oops

```

end

36 Examples and regression tests for automated termination proofs

```
theory Termination
imports Main Multiset
begin
```

36.1 Manually giving termination relations using *relation* and *measure*

```
function sum :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
where
  sum i N = (if i > N then 0 else i + sum (Suc i) N)
by pat-completeness auto

termination by (relation measure ( $\lambda(i,N). N + 1 - i$ )) auto

function foo :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat
where
  foo i N = (if i > N
             then (if N = 0 then 0 else foo 0 (N - 1))
             else i + foo (Suc i) N)
by pat-completeness auto

termination by (relation measures [ $\lambda(i, N). N$ ,  $\lambda(i,N). N + 1 - i$ ]) auto
```

36.2 *lexicographic-order*: Trivial examples

The *fun* command uses the method *lexicographic-order* by default, so it is not explicitly invoked.

```
fun identity :: nat  $\Rightarrow$  nat
where
  identity n = n

fun yaSuc :: nat  $\Rightarrow$  nat
where
  yaSuc 0 = 0
| yaSuc (Suc n) = Suc (yaSuc n)
```

36.3 Examples on natural numbers

```
fun bin :: (nat * nat)  $\Rightarrow$  nat
where
  bin (0, 0) = 1
| bin (Suc n, 0) = 0
| bin (0, Suc m) = 0
| bin (Suc n, Suc m) = bin (n, m) + bin (Suc n, m)
```

```

fun t :: (nat * nat) ⇒ nat
where
  t (0,n) = 0
| t (n,0) = 0
| t (Suc n, Suc m) = (if (n mod 2 = 0) then (t (Suc n, m)) else (t (n, Suc m)))

```

```

fun k :: (nat * nat) * (nat * nat) ⇒ nat
where
  k ((0,0),(0,0)) = 0
| k ((Suc z, y), (u,v)) = k((z, y), (u, v))
| k ((0, Suc y), (u,v)) = k((1, y), (u, v))
| k ((0,0), (Suc u, v)) = k((1, 1), (u, v))
| k ((0,0), (0, Suc v)) = k((1,1), (1,v))

```

```

fun gcd2 :: nat ⇒ nat ⇒ nat
where
  gcd2 x 0 = x
| gcd2 0 y = y
| gcd2 (Suc x) (Suc y) = (if x < y then gcd2 (Suc x) (y - x)
                           else gcd2 (x - y) (Suc y))

```

```

fun ack :: (nat * nat) ⇒ nat
where
  ack (0, m) = Suc m
| ack (Suc n, 0) = ack(n, 1)
| ack (Suc n, Suc m) = ack (n, ack (Suc n, m))

```

```

fun greedy :: nat * nat * nat * nat * nat => nat
where
  greedy (Suc a, Suc b, Suc c, Suc d, Suc e) =
    (if (a < 10) then greedy (Suc a, Suc b, c, d + 2, Suc e) else
     (if (a < 20) then greedy (Suc a, b, Suc c, d, Suc e) else
      (if (a < 30) then greedy (Suc a, b, Suc c, d, Suc e) else
       (if (a < 40) then greedy (Suc a, b, Suc c, d, Suc e) else
        (if (a < 50) then greedy (Suc a, b, Suc c, d, Suc e) else
         (if (a < 60) then greedy (a, Suc b, Suc c, d, Suc e) else
          (if (a < 70) then greedy (a, Suc b, Suc c, d, Suc e) else
           (if (a < 80) then greedy (a, Suc b, Suc c, d, Suc e) else
            (if (a < 90) then greedy (Suc a, Suc b, Suc c, d, e) else
             greedy (Suc a, Suc b, Suc c, d, e))))))))))
| greedy (a, b, c, d, e) = 0

```

```

fun blowup :: nat => nat => nat => nat => nat => nat => nat => nat =>
nat => nat

```


where

```

    blowup 0 0 0 0 0 0 0 0 0 = 0
| blowup 0 0 0 0 0 0 0 0 (Suc i) = Suc (blowup i i i i i i i i)
| blowup 0 0 0 0 0 0 0 (Suc h) i = Suc (blowup h h h h h h h h i)
| blowup 0 0 0 0 0 0 (Suc g) h i = Suc (blowup g g g g g g h i)
| blowup 0 0 0 0 0 (Suc f) g h i = Suc (blowup f f f f f f g h i)
| blowup 0 0 0 0 (Suc e) f g h i = Suc (blowup e e e e f g h i)
| blowup 0 0 0 (Suc d) e f g h i = Suc (blowup d d d d e f g h i)
| blowup 0 0 (Suc c) d e f g h i = Suc (blowup c c c d e f g h i)
| blowup 0 (Suc b) c d e f g h i = Suc (blowup b b c d e f g h i)
| blowup (Suc a) b c d e f g h i = Suc (blowup a b c d e f g h i)

```

36.4 Simple examples with other datatypes than nat, e.g. trees and lists

datatype *tree* = *Node* | *Branch tree tree*

fun *g-tree* :: *tree* * *tree* \Rightarrow *tree*

where

```

    g-tree (Node, Node) = Node
| g-tree (Node, Branch a b) = Branch Node (g-tree (a,b))
| g-tree (Branch a b, Node) = Branch (g-tree (a,Node)) b
| g-tree (Branch a b, Branch c d) = Branch (g-tree (a,c)) (g-tree (b,d))

```

fun *acklist* :: 'a list * 'a list \Rightarrow 'a list

where

```

    acklist ([], m) = ((hd m)#m)
| acklist (n#ns, []) = acklist (ns, [n])
| acklist ((n#ns), (m#ms)) = acklist (ns, acklist ((n#ns), ms))

```

36.5 Examples with mutual recursion

fun *evn od* :: nat \Rightarrow bool

where

```

    evn 0 = True
| od 0 = False
| evn (Suc n) = od (Suc n)
| od (Suc n) = evn n

```

fun *sizechange-f* :: 'a list \Rightarrow 'a list \Rightarrow 'a list **and**
sizechange-g :: 'a list \Rightarrow 'a list \Rightarrow 'a list \Rightarrow 'a list

where

```

    sizechange-f i x = (if i=[] then x else sizechange-g (tl i) x i)
| sizechange-g a b c = sizechange-f a (b @ c)

```

fun

```

    pedal :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat

```

```

and
  coast :: nat => nat => nat => nat
where
  pedal 0 m c = c
| pedal n 0 c = c
| pedal n m c =
    (if n < m then coast (n - 1) (m - 1) (c + m)
     else pedal (n - 1) m (c + m))

| coast n m c =
    (if n < m then coast n (m - 1) (c + n)
     else pedal n m (c + n))

```

36.6 Refined analysis: The *size-change* method

Unsolvable for *lexicographic-order*

```

function fun1 :: nat * nat => nat
where
  fun1 (0,0) = 1
| fun1 (0, Suc b) = 0
| fun1 (Suc a, 0) = 0
| fun1 (Suc a, Suc b) = fun1 (b, a)
by pat-completeness auto
termination by size-change

```

lexicographic-order can do the following, but it is much slower.

```

function
  prod :: nat => nat => nat => nat and
  eprod :: nat => nat => nat => nat and
  oprod :: nat => nat => nat => nat
where
  prod x y z = (if y mod 2 = 0 then eprod x y z else oprod x y z)
| oprod x y z = eprod x (y - 1) (z+x)
| eprod x y z = (if y=0 then z else prod (2*x) (y div 2) z)
by pat-completeness auto
termination by size-change

```

Permutations of arguments:

```

function perm :: nat => nat => nat => nat
where
  perm m n r = (if r > 0 then perm m (r - 1) n
                else if n > 0 then perm r (n - 1) m
                else m)
by auto
termination by size-change

```

Artificial examples and regression tests:

```

function

```

```

  fun2 :: nat ⇒ nat ⇒ nat ⇒ nat
where
  fun2 x y z =
    (if x > 1000 ∧ z > 0 then
      fun2 (min x y) y (z - 1)
    else if y > 0 ∧ x > 100 then
      fun2 x (y - 1) (2 * z)
    else if z > 0 then
      fun2 (min y (z - 1)) x x
    else
      0
    )
by pat-completeness auto
termination by size-change — requires Multiset

end

```

37 Coherent Logic Problems

```

theory Coherent
imports Main
begin

```

37.1 Equivalence of two versions of Pappus' Axiom

```

no-notation
  comp (infixl o 55) and
  rel-comp (infixr O 75)

lemma p1p2:
  assumes
    col a b c l ∧ col d e f m
    col b f g n ∧ col c e g o
    col b d h p ∧ col a e h q
    col c d i r ∧ col a f i s
    el n o ⇒ goal
    el p q ⇒ goal
    el s r ⇒ goal
  ∧ A. el A A ⇒ pl g A ⇒ pl h A ⇒ pl i A ⇒ goal
  ∧ A B C D. col A B C D ⇒ pl A D
  ∧ A B C D. col A B C D ⇒ pl B D
  ∧ A B C D. col A B C D ⇒ pl C D
  ∧ A B. pl A B ⇒ ep A A
  ∧ A B. ep A B ⇒ ep B A
  ∧ A B C. ep A B ⇒ ep B C ⇒ ep A C
  ∧ A B. pl A B ⇒ el B B
  ∧ A B. el A B ⇒ el B A
  ∧ A B C. el A B ⇒ el B C ⇒ el A C

```

$\bigwedge A B C. ep A B \implies pl B C \implies pl A C$
 $\bigwedge A B C. pl A B \implies el B C \implies pl A C$
 $\bigwedge A B C D E F G H I J K L M N O P Q.$
 $col A B C D \implies col E F G H \implies col B G I J \implies col C F I K \implies$
 $col B E L M \implies col A F L N \implies col C E O P \implies col A G O Q \implies$
 $(\exists R. col I L O R) \vee pl A H \vee pl B H \vee pl C H \vee pl E D \vee pl F D \vee pl G D$
 $\bigwedge A B C D. pl A B \implies pl A C \implies pl D B \implies pl D C \implies ep A D \vee el B C$
 $\bigwedge A B. ep A A \implies ep B B \implies \exists C. pl A C \wedge pl B C$
shows goal using assms
by coherent

lemma *p2p1*:

assumes
 $col a b c l \wedge col d e f m$
 $col b f g n \wedge col c e g o$
 $col b d h p \wedge col a e h q$
 $col c d i r \wedge col a f i s$
 $pl a m \implies goal$
 $pl b m \implies goal$
 $pl c m \implies goal$
 $pl d l \implies goal$
 $pl e l \implies goal$
 $pl f l \implies goal$
 $\bigwedge A. pl g A \implies pl h A \implies pl i A \implies goal$
 $\bigwedge A B C D. col A B C D \implies pl A D$
 $\bigwedge A B C D. col A B C D \implies pl B D$
 $\bigwedge A B C D. col A B C D \implies pl C D$
 $\bigwedge A B. pl A B \implies ep A A$
 $\bigwedge A B. ep A B \implies ep B A$
 $\bigwedge A B C. ep A B \implies ep B C \implies ep A C$
 $\bigwedge A B. pl A B \implies el B B$
 $\bigwedge A B. el A B \implies el B A$
 $\bigwedge A B C. el A B \implies el B C \implies el A C$
 $\bigwedge A B C. ep A B \implies pl B C \implies pl A C$
 $\bigwedge A B C. pl A B \implies el B C \implies pl A C$
 $\bigwedge A B C D E F G H I J K L M N O P Q.$
 $col A B C J \implies col D E F K \implies col B F G L \implies col C E G M \implies$
 $col B D H N \implies col A E H O \implies col C D I P \implies col A F I Q \implies$
 $(\exists R. col G H I R) \vee el L M \vee el N O \vee el P Q$
 $\bigwedge A B C D. pl C A \implies pl C B \implies pl D A \implies pl D B \implies ep C D \vee el A B$
 $\bigwedge A B C. ep A A \implies ep B B \implies \exists C. pl A C \wedge pl B C$
shows goal using assms
by coherent

37.2 Preservation of the Diamond Property under reflexive closure

lemma *diamond*:

assumes

```

  reflexive-rewrite a b reflexive-rewrite a c
   $\bigwedge A. \text{ reflexive-rewrite } b \ A \implies \text{ reflexive-rewrite } c \ A \implies \text{ goal}$ 
   $\bigwedge A. \text{ equalish } A \ A$ 
   $\bigwedge A \ B. \text{ equalish } A \ B \implies \text{ equalish } B \ A$ 
   $\bigwedge A \ B \ C. \text{ equalish } A \ B \implies \text{ reflexive-rewrite } B \ C \implies \text{ reflexive-rewrite } A \ C$ 
   $\bigwedge A \ B. \text{ equalish } A \ B \implies \text{ reflexive-rewrite } A \ B$ 
   $\bigwedge A \ B. \text{ rewrite } A \ B \implies \text{ reflexive-rewrite } A \ B$ 
   $\bigwedge A \ B. \text{ reflexive-rewrite } A \ B \implies \text{ equalish } A \ B \vee \text{ rewrite } A \ B$ 
   $\bigwedge A \ B \ C. \text{ rewrite } A \ B \implies \text{ rewrite } A \ C \implies \exists D. \text{ rewrite } B \ D \wedge \text{ rewrite } C \ D$ 
  shows goal using assms
  by coherent
end

```

38 Some examples for Presburger Arithmetic

```

theory PresburgerEx
imports Presburger
begin

```

```

lemma  $\bigwedge m \ n \ j \ a \ i \ a. \llbracket \neg m \leq j; \neg (n::nat) \leq i; (e::nat) \neq 0; \text{Suc } j \leq ja \rrbracket \implies \exists m. \forall j \ a \ i \ a. m \leq ja \longrightarrow (\text{if } j = ja \wedge i = ia \text{ then } e \text{ else } 0) = 0$  by presburger

```

```

lemma  $(0::nat) < \text{emBits mod } 8 \implies 8 + \text{emBits div } 8 * 8 - \text{emBits} = 8 - \text{emBits mod } 8$ 

```

```

by presburger

```

```

lemma  $(0::nat) < \text{emBits mod } 8 \implies 8 + \text{emBits div } 8 * 8 - \text{emBits} = 8 - \text{emBits mod } 8$ 

```

```

by presburger

```

```

theorem  $(\forall (y::int). 3 \text{ dvd } y) \implies \forall (x::int). b < x \longrightarrow a \leq x$ 
by presburger

```

```

theorem !!  $(y::int) \ (z::int) \ (n::int). 3 \text{ dvd } z \implies 2 \text{ dvd } (y::int) \implies (\exists (x::int). 2*x = y) \ \& \ (\exists (k::int). 3*k = z)$ 
by presburger

```

```

theorem !!  $(y::int) \ (z::int) \ n. \text{Suc}(n::nat) < 6 \implies 3 \text{ dvd } z \implies 2 \text{ dvd } (y::int) \implies (\exists (x::int). 2*x = y) \ \& \ (\exists (k::int). 3*k = z)$ 
by presburger

```

```

theorem  $\forall (x::nat). \exists (y::nat). (0::nat) \leq 5 \longrightarrow y = 5 + x$ 
by presburger

```

Slow: about 7 seconds on a 1.6GHz machine.

```

theorem  $\forall (x::nat). \exists (y::nat). y = 5 + x \mid x \text{ div } 6 + 1 = 2$ 
by presburger

```

```

theorem  $\exists (x::int). 0 < x$ 

```

by *presburger*

theorem $\forall (x::int) y. x < y \longrightarrow 2 * x + 1 < 2 * y$
by *presburger*

theorem $\forall (x::int) y. 2 * x + 1 \neq 2 * y$
by *presburger*

theorem $\exists (x::int) y. 0 < x \ \& \ 0 \leq y \ \& \ 3 * x - 5 * y = 1$
by *presburger*

theorem $\sim (\exists (x::int) (y::int) (z::int). 4 * x + (-6::int) * y = 1)$
by *presburger*

theorem $\forall (x::int). b < x \longrightarrow a \leq x$
apply (*presburger elim*)
oops

theorem $\sim (\exists (x::int). False)$
by *presburger*

theorem $\forall (x::int). (a::int) < 3 * x \longrightarrow b < 3 * x$
apply (*presburger elim*)
oops

theorem $\forall (x::int). (2 \text{ dvd } x) \longrightarrow (\exists (y::int). x = 2 * y)$
by *presburger*

theorem $\forall (x::int). (2 \text{ dvd } x) \longrightarrow (\exists (y::int). x = 2 * y)$
by *presburger*

theorem $\forall (x::int). (2 \text{ dvd } x) = (\exists (y::int). x = 2 * y)$
by *presburger*

theorem $\forall (x::int). ((2 \text{ dvd } x) = (\forall (y::int). x \neq 2 * y + 1))$
by *presburger*

theorem $\sim (\forall (x::int). ((2 \text{ dvd } x) = (\forall (y::int). x \neq 2 * y + 1) \mid (\exists (q::int) (u::int) i. 3 * i + 2 * q - u < 17) \longrightarrow 0 < x \mid ((\sim 3 \text{ dvd } x) \ \& \ (x + 8 = 0))))$
by *presburger*

theorem $\sim (\forall (i::int). 4 \leq i \longrightarrow (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i))$
by *presburger*

theorem $\forall (i::int). 8 \leq i \longrightarrow (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i)$
by *presburger*

theorem $\exists (j::int). \forall i. j \leq i \longrightarrow (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i)$
by *presburger*

theorem $\sim (\forall j (i::int). j \leq i \longrightarrow (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i))$
by *presburger*

Slow: about 5 seconds on a 1.6GHz machine.

theorem $(\exists m::nat. n = 2 * m) \longrightarrow (n + 1) \text{ div } 2 = n \text{ div } 2$
by *presburger*

This following theorem proves that all solutions to the recurrence relation $x_{i+2} = |x_{i+1}| - x_i$ are periodic with period 9. The example was brought to our attention by John Harrison. It does not require Presburger arithmetic but merely quantifier-free linear arithmetic and holds for the rationals as well.

Warning: it takes (in 2006) over 4.2 minutes!

lemma $\llbracket x3 = \text{abs } x2 - x1; x4 = \text{abs } x3 - x2; x5 = \text{abs } x4 - x3;$
 $x6 = \text{abs } x5 - x4; x7 = \text{abs } x6 - x5; x8 = \text{abs } x7 - x6;$
 $x9 = \text{abs } x8 - x7; x10 = \text{abs } x9 - x8; x11 = \text{abs } x10 - x9 \rrbracket$
 $\implies x1 = x10 \ \& \ x2 = (x11::int)$
by *arith*

end

39 Generic reflection and reification

theory *Reflection*
imports *Main*
uses *reify-data.ML* (*reflection.ML*)
begin

setup $\ll \text{Reify-Data.setup} \gg$

lemma *ext2*: $(\forall x. f x = g x) \implies f = g$
by (*blast intro: ext*)

use *reflection.ML*

method-setup *reify* = \ll
 $\text{Attrib.thms} \text{ --}$
 $\text{Scan.option } (\text{Scan.lift } (\text{Args.} \$ \$ \$ \text{ } ()) \text{ |-- } \text{Args.term} \text{ --| } \text{Scan.lift } (\text{Args.} \$ \$ \$ \text{ } ()))$
 \gg
 $(\text{fn } (eqs, to) \Rightarrow \text{fn } ctxt \Rightarrow \text{SIMPLE-METHOD}' (\text{Reflection.genreify-tac } ctxt$
 $(eqs @ (\text{fst } (\text{Reify-Data.get } ctxt)))) \text{ to}))$
 \gg *partial automatic reification*

```

method-setup reflection = <<
  let
    fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ();
    val onlyN = only;
    val rulesN = rules;
    val any-keyword = keyword onlyN || keyword rulesN;
    val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
    val terms = thms >> map (term-of o Drule.dest-term);
  in
    thms --
    Scan.optional (keyword rulesN |-- thms) [] --
    Scan.option (keyword onlyN |-- Args.term) >>
    (fn ((eqs,ths),to) => fn ctxt =>
      let
        val (ceqs,cths) = Reify-Data.get ctxt
        val corr-thms = ths@cths
        val raw-eqs = eqs@ceqs
      in SIMPLE-METHOD' (Reflection.reflection-tac ctxt corr-thms raw-eqs to) end)
    end
  >> reflection
end

```

40 Examples for generic reflection and reification

```

theory ReflectionEx
imports Reflection
begin

```

This theory presents two methods: reify and reflection

Consider an HOL type 'a, the structure of which is not recognisable on the theory level. This is the case of bool, arithmetical terms such as int, real etc ... In order to implement a simplification on terms of type 'a we often need its structure. Traditionnaly such simplifications are written in ML, proofs are synthesized. An other strategy is to declare an HOL-datatype tau and an HOL function (the interpretation) that maps elements of tau to elements of 'a. The functionality of *reify* is to compute a term *s::tau*, which is the representant of *t*. For this it needs equations for the interpretation.

NB: All the interpretations supported by *reify* must have the type '*b list* \Rightarrow *tau* \Rightarrow 'a. The method *reify* can also be told which subterm of the current subgoal should be reified. The general call for *reify* is: *reify eqs (t)*, where *eqs* are the defining equations of the interpretation and (*t*) is an optional parameter which specifies the subterm to which reification should be applied to. If (*t*) is abscent, *reify* tries to reify the whole subgoal.

The method reflection uses *reify* and has a very similar signature: *reflection*

corr-thm eqs (*t*). Here again *eqs* and (*t*) are as described above and *corr-thm* is a theorem proving $I \text{ vs } (f \ t) = I \text{ vs } t$. We assume that *I* is the interpretation and *f* is some useful and executable simplification of type $\text{tau} \Rightarrow \text{tau}$. The method *reflection* applies reification and hence the theorem $t = I \text{ xs } s$ and hence using *corr-thm* derives $t = I \text{ xs } (f \ s)$. It then uses normalization by evaluation to prove $f \ s = s'$ which almost finishes the proof of $t = t'$ where $I \text{ xs } s' = t'$.

Example 1 : Propositional formulae and NNF.

The type *fm* represents simple propositional formulae:

```
datatype form = TrueF | FalseF | Less nat nat |
               And form form | Or form form | Neg form | ExQ form
```

```
fun interp :: form  $\Rightarrow$  ('a::ord) list  $\Rightarrow$  bool where
  interp TrueF e = True |
  interp FalseF e = False |
  interp (Less i j) e = (e!i < e!j) |
  interp (And f1 f2) e = (interp f1 e & interp f2 e) |
  interp (Or f1 f2) e = (interp f1 e | interp f2 e) |
  interp (Neg f) e = (~ interp f e) |
  interp (ExQ f) e = (EX x. interp f (x#e))
```

```
lemmas interp-reify-eqs = interp.simps
```

```
declare interp-reify-eqs[reify]
```

```
lemma EX x. x < y & x < z
apply (reify )
oops
```

```
datatype fm = And fm fm | Or fm fm | Imp fm fm | Iff fm fm | NOT fm | At
nat
```

```
consts Ifm :: fm  $\Rightarrow$  bool list  $\Rightarrow$  bool
```

```
primrec
```

```
  Ifm (At n) vs = vs!n
  Ifm (And p q) vs = (Ifm p vs  $\wedge$  Ifm q vs)
  Ifm (Or p q) vs = (Ifm p vs  $\vee$  Ifm q vs)
  Ifm (Imp p q) vs = (Ifm p vs  $\longrightarrow$  Ifm q vs)
  Ifm (Iff p q) vs = (Ifm p vs = Ifm q vs)
  Ifm (NOT p) vs = (~ (Ifm p vs))
```

```
lemma Q  $\longrightarrow$  (D & F & ((~ D) & (~ F)))
```

```
apply (reify Ifm.simps)
```

```
oops
```

Method *reify* maps a bool to an fm. For this it needs the semantics of fm, i.e. the rewrite rules in *Ifm.simps*.

```
lemma Q  $\longrightarrow$  (D & F & ((~ D) & (~ F)))
```

apply (*reify Ifm.simps* ((($\sim D$) & ($\sim F$))))
oops

Let's perform NNF. This is a version that tends to generate disjunctions

primrec *fmsize* :: *fm* \Rightarrow *nat* **where**
 $fmsize\ (At\ n) = 1$
 $fmsize\ (NOT\ p) = 1 + fmsize\ p$
 $fmsize\ (And\ p\ q) = 1 + fmsize\ p + fmsize\ q$
 $fmsize\ (Or\ p\ q) = 1 + fmsize\ p + fmsize\ q$
 $fmsize\ (Imp\ p\ q) = 2 + fmsize\ p + fmsize\ q$
 $fmsize\ (Iff\ p\ q) = 2 + 2 * fmsize\ p + 2 * fmsize\ q$

lemma [*measure-function*]: *is-measure fmsize ..*

fun *nnf* :: *fm* \Rightarrow *fm*
where
 $nnf\ (At\ n) = At\ n$
 $nnf\ (And\ p\ q) = And\ (nnf\ p)\ (nnf\ q)$
 $nnf\ (Or\ p\ q) = Or\ (nnf\ p)\ (nnf\ q)$
 $nnf\ (Imp\ p\ q) = Or\ (nnf\ (NOT\ p))\ (nnf\ q)$
 $nnf\ (Iff\ p\ q) = Or\ (And\ (nnf\ p)\ (nnf\ q))\ (And\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q)))$
 $nnf\ (NOT\ (And\ p\ q)) = Or\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q))$
 $nnf\ (NOT\ (Or\ p\ q)) = And\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q))$
 $nnf\ (NOT\ (Imp\ p\ q)) = And\ (nnf\ p)\ (nnf\ (NOT\ q))$
 $nnf\ (NOT\ (Iff\ p\ q)) = Or\ (And\ (nnf\ p)\ (nnf\ (NOT\ q)))\ (And\ (nnf\ (NOT\ p))\ (nnf\ q))$
 $nnf\ (NOT\ (NOT\ p)) = nnf\ p$
 $nnf\ (NOT\ p) = NOT\ p$

The correctness theorem of *nnf*: it preserves the semantics of *fm*

lemma *nnf[reflection]*: *Ifm* (*nnf* *p*) *vs* = *Ifm* *p* *vs*
by (*induct p rule: nnf.induct*) *auto*

Now let's perform NNF using our *nnf* function defined above. First to the whole subgoal.

lemma ($\neg (A = B)) \wedge (B \longrightarrow (A \neq (B \mid C \wedge (B \longrightarrow A \mid D)))) \longrightarrow A \vee B \wedge D$
apply (*reflection Ifm.simps*)
oops

Now we specify on which subterm it should be applied

lemma ($\neg (A = B)) \wedge (B \longrightarrow (A \neq (B \mid C \wedge (B \longrightarrow A \mid D)))) \longrightarrow A \vee B \wedge D$
apply (*reflection Ifm.simps only: (B | C \wedge (B \longrightarrow A | D))*)
oops

The type *num* reflects linear expressions over natural number

datatype *num* = *C nat* | *Add num num* | *Mul nat num* | *Var nat* | *CN nat nat num*

This is just technical to make recursive definitions easier.

primrec *num-size* :: *num* \Rightarrow *nat*

where

num-size (*C* *c*) = 1
| *num-size* (*Var* *n*) = 1
| *num-size* (*Add* *a* *b*) = 1 + *num-size* *a* + *num-size* *b*
| *num-size* (*Mul* *c* *a*) = 1 + *num-size* *a*
| *num-size* (*CN* *n* *c* *a*) = 4 + *num-size* *a*

The semantics of *num*

primrec *Inum*:: *num* \Rightarrow *nat list* \Rightarrow *nat*

where

Inum-C : *Inum* (*C* *i*) *vs* = *i*
| *Inum-Var*: *Inum* (*Var* *n*) *vs* = *vs*!*n*
| *Inum-Add*: *Inum* (*Add* *s* *t*) *vs* = *Inum* *s* *vs* + *Inum* *t* *vs*
| *Inum-Mul*: *Inum* (*Mul* *c* *t*) *vs* = *c* * *Inum* *t* *vs*
| *Inum-CN* : *Inum* (*CN* *n* *c* *t*) *vs* = *c**(*vs*!*n*) + *Inum* *t* *vs*

Let's reify some *nat* expressions ...

lemma 4 * (2*x + (y::nat)) + f a \neq 0

apply (reify *Inum.simps* (4 * (2*x + (y::nat)) + f a))

oops

We're in a bad situation!! *x*, *y* and *f a* have been recongnized as a constants, which is correct but does not correspond to our intuition of the constructor *C*. It should encapsulate constants, i.e. numbers, i.e. numerals.

So let's leave the *Inum-C* equation at the end and see what happens ...

lemma 4 * (2*x + (y::nat)) \neq 0

apply (reify *Inum-Var* *Inum-Add* *Inum-Mul* *Inum-CN* *Inum-C* (4 * (2*x + (y::nat))))

oops

Hmmm let's specialize *Inum-C* with numerals.

lemma *Inum-number*: *Inum* (*C* (*number-of* *t*)) *vs* = *number-of* *t* **by** *simp*

lemmas *Inum-eqs* = *Inum-Var* *Inum-Add* *Inum-Mul* *Inum-CN* *Inum-number*

Second attempt

lemma 1 * (2*x + (y::nat)) \neq 0

apply (reify *Inum-eqs* (1 * (2*x + (y::nat))))

oops

That was fine, so let's try another one ...

lemma 1 * (2* x + (y::nat) + 0 + 1) \neq 0

apply (reify *Inum-eqs* (1 * (2*x + (y::nat) + 0 + 1))))

oops

Oh!! 0 is not a variable ... Oh! 0 is not a *number-of* ... thing. The same for 1. So let's add those equations too

lemma *Inum-01*: *Inum* (*C* 0) *vs* = 0 *Inum* (*C* 1) *vs* = 1 *Inum* (*C*(*Suc* *n*)) *vs* = *Suc* *n*

by *simp*+

lemmas *Inum-egs'* = *Inum-egs* *Inum-01*

Third attempt:

lemma $1 * (2*x + (y::nat) + 0 + 1) \neq 0$

apply (*reify* *Inum-egs'* ($1 * (2*x + (y::nat) + 0 + 1)$))

oops

Okay, let's try reflection. Some simplifications on num follow. You can skim until the main theorem *linum*

fun *lin-add* :: *num* \Rightarrow *num* \Rightarrow *num*

where

lin-add (*CN* *n1* *c1* *r1*) (*CN* *n2* *c2* *r2*) =

(*if* *n1*=*n2* *then*

(*let* *c* = *c1* + *c2*

in (*if* *c*=0 *then* *lin-add* *r1* *r2* *else* *CN* *n1* *c* (*lin-add* *r1* *r2*)))

else if *n1* \leq *n2* *then* (*CN* *n1* *c1* (*lin-add* *r1* (*CN* *n2* *c2* *r2*)))

else (*CN* *n2* *c2* (*lin-add* (*CN* *n1* *c1* *r1*) *r2*)))

| *lin-add* (*CN* *n1* *c1* *r1*) *t* = *CN* *n1* *c1* (*lin-add* *r1* *t*)

| *lin-add* *t* (*CN* *n2* *c2* *r2*) = *CN* *n2* *c2* (*lin-add* *t* *r2*)

| *lin-add* (*C* *b1*) (*C* *b2*) = *C* (*b1*+*b2*)

| *lin-add* *a* *b* = *Add* *a* *b*

lemma *lin-add*: *Inum* (*lin-add* *t* *s*) *bs* = *Inum* (*Add* *t* *s*) *bs*

apply (*induct* *t* *s* *rule*: *lin-add.induct*, *simp-all* *add*: *Let-def*)

apply (*case-tac* *c1*+*c2* = 0, *case-tac* *n1* \leq *n2*, *simp-all*)

by (*case-tac* *n1* = *n2*, *simp-all* *add*: *algebra-simps*)

fun *lin-mul* :: *num* \Rightarrow *nat* \Rightarrow *num*

where

lin-mul (*C* *j*) *i* = *C* (*i***j*)

| *lin-mul* (*CN* *n* *c* *a*) *i* = (*if* *i*=0 *then* (*C* 0) *else* *CN* *n* (*i***c*) (*lin-mul* *a* *i*))

| *lin-mul* *t* *i* = (*Mul* *i* *t*)

lemma *lin-mul*: *Inum* (*lin-mul* *t* *i*) *bs* = *Inum* (*Mul* *i* *t*) *bs*

by (*induct* *t* *i* *rule*: *lin-mul.induct*, *auto* *simp* *add*: *algebra-simps*)

lemma [*measure-function*]: *is-measure* *num-size* ..

fun *linum*:: *num* \Rightarrow *num*

where

linum (*C* *b*) = *C* *b*

| *linum* (*Var* *n*) = *CN* *n* 1 (*C* 0)

| *linum* (*Add* *t* *s*) = *lin-add* (*linum* *t*) (*linum* *s*)

| *linum* (*Mul* *c* *t*) = *lin-mul* (*linum* *t*) *c*

| *linum* (*CN* *n* *c* *t*) = *lin-add* (*linum* (*Mul* *c* (*Var* *n*))) (*linum* *t*)

lemma *linum[reflection] : Inum (linum t) bs = Inum t bs*
by (*induct t rule: linum.induct, simp-all add: lin-mul lin-add*)

Now we can use *linum* to simplify *nat* terms using reflection

lemma *(Suc (Suc 1)) * (x + (Suc 1)*y) = 3*x + 6*y*
apply (*reflection Inum-eqs' only: (Suc (Suc 1)) * (x + (Suc 1)*y)*)
oops

Let's lift this to formulae and see what happens

datatype *aform = Lt num num | Eq num num | Ge num num | NEq num num*
 $|$
Conj aform aform | Disj aform aform | NEG aform | T | F

primrec *linaformsize :: aform \Rightarrow nat*

where

linaformsize T = 1
 $|$ *linaformsize F = 1*
 $|$ *linaformsize (Lt a b) = 1*
 $|$ *linaformsize (Ge a b) = 1*
 $|$ *linaformsize (Eq a b) = 1*
 $|$ *linaformsize (NEq a b) = 1*
 $|$ *linaformsize (NEG p) = 2 + linaformsize p*
 $|$ *linaformsize (Conj p q) = 1 + linaformsize p + linaformsize q*
 $|$ *linaformsize (Disj p q) = 1 + linaformsize p + linaformsize q*

lemma [*measure-function*]: *is-measure linaformsize ..*

primrec *is-aform :: aform \Rightarrow nat list \Rightarrow bool*

where

is-aform T vs = True
 $|$ *is-aform F vs = False*
 $|$ *is-aform (Lt a b) vs = (Inum a vs < Inum b vs)*
 $|$ *is-aform (Eq a b) vs = (Inum a vs = Inum b vs)*
 $|$ *is-aform (Ge a b) vs = (Inum a vs \geq Inum b vs)*
 $|$ *is-aform (NEq a b) vs = (Inum a vs \neq Inum b vs)*
 $|$ *is-aform (NEG p) vs = (\neg (is-aform p vs))*
 $|$ *is-aform (Conj p q) vs = (is-aform p vs \wedge is-aform q vs)*
 $|$ *is-aform (Disj p q) vs = (is-aform p vs \vee is-aform q vs)*

Let's reify and do reflection

lemma *(3::nat)*x + t < 0 \wedge (2 * x + y \neq 17)*
apply (*reify Inum-eqs' is-aform.simps*)
oops

Note that reification handles several interpretations at the same time

lemma *(3::nat)*x + t < 0 & x*x + t*x + 3 + 1 = z*t*4*z | x + x + 1 < 0*
apply (*reflection Inum-eqs' is-aform.simps only:x + x + 1*)
oops

For reflection we now define a simple transformation on `aform`: `NNF` + `linum` on atoms

fun `linaform`:: `aform` \Rightarrow `aform`

where

```

  linaform (Lt s t) = Lt (linum s) (linum t)
| linaform (Eq s t) = Eq (linum s) (linum t)
| linaform (Ge s t) = Ge (linum s) (linum t)
| linaform (NEq s t) = NEq (linum s) (linum t)
| linaform (Conj p q) = Conj (linaform p) (linaform q)
| linaform (Disj p q) = Disj (linaform p) (linaform q)
| linaform (NEG T) = F
| linaform (NEG F) = T
| linaform (NEG (Lt a b)) = Ge a b
| linaform (NEG (Ge a b)) = Lt a b
| linaform (NEG (Eq a b)) = NEq a b
| linaform (NEG (NEq a b)) = Eq a b
| linaform (NEG (NEG p)) = linaform p
| linaform (NEG (Conj p q)) = Disj (linaform (NEG p)) (linaform (NEG q))
| linaform (NEG (Disj p q)) = Conj (linaform (NEG p)) (linaform (NEG q))
| linaform p = p

```

lemma `linaform`: `is-aform (linaform p) vs = is-aform p vs`

by (`induct p rule: linaform.induct`) (`auto simp add: linum`)

lemma $((\text{Suc}(\text{Suc}(\text{Suc } 0))) * ((x::\text{nat}) + (\text{Suc}(\text{Suc } 0)))) + (\text{Suc}(\text{Suc}(\text{Suc } 0)))$
 $* ((\text{Suc}(\text{Suc}(\text{Suc } 0))) * ((x::\text{nat}) + (\text{Suc}(\text{Suc } 0)))) < 0) \wedge (\text{Suc } 0 + \text{Suc } 0 < 0)$

apply (`reflection Inum-eqs' is-aform.simps rules: linaform`)

oops

declare `linaform[reflection]`

lemma $((\text{Suc}(\text{Suc}(\text{Suc } 0))) * ((x::\text{nat}) + (\text{Suc}(\text{Suc } 0)))) + (\text{Suc}(\text{Suc}(\text{Suc } 0)))$
 $* ((\text{Suc}(\text{Suc}(\text{Suc } 0))) * ((x::\text{nat}) + (\text{Suc}(\text{Suc } 0)))) < 0) \wedge (\text{Suc } 0 + \text{Suc } 0 < 0)$

apply (`reflection Inum-eqs' is-aform.simps`)

oops

We now give an example where Interpretations have 0 or more than only one envornement of different types and show that automatic reification also deals with binding

datatype `rb` = `BC bool` | `BAnd rb rb` | `BOr rb rb`

primrec `Irb` :: `rb` \Rightarrow `bool`

where

```

  Irb (BC p) = p
| Irb (BAnd s t) = (Irb s  $\wedge$  Irb t)
| Irb (BOr s t) = (Irb s  $\vee$  Irb t)

```

lemma $A \wedge (B \vee D \wedge B) \wedge A \wedge (B \vee D \wedge B) \vee A \wedge (B \vee D \wedge B) \vee A \wedge (B \vee D \wedge B)$

apply (`reify Irb.simps`)

oops

```

datatype rint = IC int | IVar nat | IAdd rint rint | IMult rint rint | INeg rint |
ISub rint rint
primrec Irint :: rint  $\Rightarrow$  int list  $\Rightarrow$  int
where
  Irint-Var: Irint (IVar n) vs = vs!n
| Irint-Neg: Irint (INeg t) vs = - Irint t vs
| Irint-Add: Irint (IAdd s t) vs = Irint s vs + Irint t vs
| Irint-Sub: Irint (ISub s t) vs = Irint s vs - Irint t vs
| Irint-Mult: Irint (IMult s t) vs = Irint s vs * Irint t vs
| Irint-C: Irint (IC i) vs = i
lemma Irint-C0: Irint (IC 0) vs = 0
  by simp
lemma Irint-C1: Irint (IC 1) vs = 1
  by simp
lemma Irint-Cnumberof: Irint (IC (number-of x)) vs = number-of x
  by simp
lemmas Irint-simps = Irint-Var Irint-Neg Irint-Add Irint-Sub Irint-Mult Irint-C0
Irint-C1 Irint-Cnumberof
lemma (3::int) * x + y*y - 9 + (- z) = 0
  apply (reify Irint-simps ((3::int) * x + y*y - 9 + (- z)))
  oops
datatype rlist = LVar nat | LEmpty | LCons rint rlist | LAppend rlist rlist
primrec Irlist :: rlist  $\Rightarrow$  int list  $\Rightarrow$  (int list) list  $\Rightarrow$  (int list)
where
  Irlist (LEmpty) is vs = []
| Irlist (LVar n) is vs = vs!n
| Irlist (LCons i t) is vs = ((Irint i is) # (Irlist t is vs))
| Irlist (LAppend s t) is vs = (Irlist s is vs) @ (Irlist t is vs)
lemma [(1::int)] = []
  apply (reify Irlist.simps Irint-simps ([1]:: int list))
  oops

lemma (((3::int) * x + y*y - 9 + (- z)) @ []) @ xs = [y*y - z - 9 + (3::int)
* x]
  apply (reify Irlist.simps Irint-simps (((3::int) * x + y*y - 9 + (- z)) @ []) @
xs))
  oops

datatype rnat = NC nat | NVar nat | NSuc rnat | NAdd rnat rnat | NMult rnat
rnat | NNeg rnat | NSub rnat rnat | Nlgh rlist
primrec Irnat :: rnat  $\Rightarrow$  int list  $\Rightarrow$  (int list) list  $\Rightarrow$  nat list  $\Rightarrow$  nat
where
  Irnat-Suc: Irnat (NSuc t) is ls vs = Suc (Irnat t is ls vs)
| Irnat-Var: Irnat (NVar n) is ls vs = vs!n
| Irnat-Neg: Irnat (NNeg t) is ls vs = 0
| Irnat-Add: Irnat (NAdd s t) is ls vs = Irnat s is ls vs + Irnat t is ls vs
| Irnat-Sub: Irnat (NSub s t) is ls vs = Irnat s is ls vs - Irnat t is ls vs

```

| *Irnat-Mult*: *Irnat* (*NMult* *s t*) *is ls vs* = *Irnat s is ls vs* * *Irnat t is ls vs*
 | *Irnat-lgth*: *Irnat* (*Nlgh* *rxs*) *is ls vs* = *length* (*Irlist rxs is ls*)
 | *Irnat-C*: *Irnat* (*NC i*) *is ls vs* = *i*
lemma *Irnat-C0*: *Irnat* (*NC 0*) *is ls vs* = *0*
by *simp*
lemma *Irnat-C1*: *Irnat* (*NC 1*) *is ls vs* = *1*
by *simp*
lemma *Irnat-Cnumberof*: *Irnat* (*NC* (*number-of x*)) *is ls vs* = *number-of x*
by *simp*
lemmas *Irnat-simps* = *Irnat-Suc Irnat-Var Irnat-Neg Irnat-Add Irnat-Sub Irnat-Mult*
Irnat-lgth
Irnat-C0 Irnat-C1 Irnat-Cnumberof
lemma (*Suc n*) * *length* (((*(3::int)*) * *x* + *y*y* - 9 + (- *z*)) @ []) @ *xs*) = *length xs*
apply (*reify Irnat-simps Irlist.simps Irint-simps* ((*Suc n*) * *length* (((*(3::int)*) * *x* + *y*y* - 9 + (- *z*)) @ []) @ *xs*)))
oops
datatype *rifm* = *RT* | *RF* | *RVar nat*
 | *RNLT rnat rnat* | *RNILT rnat rint* | *RNEQ rnat rnat*
 | *RAnd rifm rifm* | *ROr rifm rifm* | *RImp rifm rifm* | *RIff rifm rifm*
 | *RNEX rifm* | *RIEX rifm* | *RLEX rifm* | *RNALL rifm* | *RIALL rifm* | *RLALL rifm*
 | *RBEX rifm* | *RBALL rifm*
primrec *Irifm* :: *rifm* \Rightarrow *bool list* \Rightarrow *int list* \Rightarrow (*int list*) *list* \Rightarrow *nat list* \Rightarrow *bool*
where
Irifm RT ps is ls ns = *True*
 | *Irifm RF ps is ls ns* = *False*
 | *Irifm (RVar n) ps is ls ns* = *ps!n*
 | *Irifm (RNLT s t) ps is ls ns* = (*Irnat s is ls ns* < *Irnat t is ls ns*)
 | *Irifm (RNILT s t) ps is ls ns* = (*int* (*Irnat s is ls ns*) < *Irnt t is*)
 | *Irifm (RNEQ s t) ps is ls ns* = (*Irnat s is ls ns* = *Irnat t is ls ns*)
 | *Irifm (RAnd p q) ps is ls ns* = (*Irifm p ps is ls ns* \wedge *Irifm q ps is ls ns*)
 | *Irifm (ROr p q) ps is ls ns* = (*Irifm p ps is ls ns* \vee *Irifm q ps is ls ns*)
 | *Irifm (RImp p q) ps is ls ns* = (*Irifm p ps is ls ns* \longrightarrow *Irifm q ps is ls ns*)
 | *Irifm (RIff p q) ps is ls ns* = (*Irifm p ps is ls ns* = *Irifm q ps is ls ns*)
 | *Irifm (RNEX p) ps is ls ns* = ($\exists x. Irifm p ps is ls (x\#ns)$)
 | *Irifm (RIEX p) ps is ls ns* = ($\exists x. Irifm p ps (x\#is) ls ns$)
 | *Irifm (RLEX p) ps is ls ns* = ($\exists x. Irifm p ps is (x\#ls) ns$)
 | *Irifm (RBEX p) ps is ls ns* = ($\exists x. Irifm p (x\#ps) is ls ns$)
 | *Irifm (RNALL p) ps is ls ns* = ($\forall x. Irifm p ps is ls (x\#ns)$)
 | *Irifm (RIALL p) ps is ls ns* = ($\forall x. Irifm p ps (x\#is) ls ns$)
 | *Irifm (RLALL p) ps is ls ns* = ($\forall x. Irifm p ps is (x\#ls) ns$)
 | *Irifm (RBALL p) ps is ls ns* = ($\forall x. Irifm p (x\#ps) is ls ns$)
lemma $\forall x. \exists n. ((\text{Suc } n) * \text{length } (((3::int) * x + (f\ t)*y - 9 + (-\ z)) @ []) @ xs) = \text{length } xs) \wedge m < 5*n - \text{length } (xs @ [2,3,4,x*z + 8 - y]) \longrightarrow (\exists p. \forall q. p \wedge q \longrightarrow r)$
apply (*reify Irifm.simps Irnat-simps Irlist.simps Irint-simps*)

oops

```

datatype prod = Zero | One | Var nat | Mul prod prod
  | Pw prod nat | PNM nat nat prod
primrec Iprod :: prod  $\Rightarrow$  ('a::{linordered-idom}) list  $\Rightarrow$  'a
where
  Iprod Zero vs = 0
| Iprod One vs = 1
| Iprod (Var n) vs = vs!n
| Iprod (Mul a b) vs = (Iprod a vs * Iprod b vs)
| Iprod (Pw a n) vs = ((Iprod a vs) ^ n)
| Iprod (PNM n k t) vs = (vs ! n) ^ k * Iprod t vs
consts prodmul:: prod  $\times$  prod  $\Rightarrow$  prod
datatype sgn = Pos prod | Neg prod | ZeroEq prod | NZeroEq prod | Tr | F
  | Or sgn sgn | And sgn sgn

primrec Isgn :: sgn  $\Rightarrow$  ('a::{linordered-idom}) list  $\Rightarrow$  bool
where
  Isgn Tr vs = True
| Isgn F vs = False
| Isgn (ZeroEq t) vs = (Iprod t vs = 0)
| Isgn (NZeroEq t) vs = (Iprod t vs  $\neq$  0)
| Isgn (Pos t) vs = (Iprod t vs > 0)
| Isgn (Neg t) vs = (Iprod t vs < 0)
| Isgn (And p q) vs = (Isign p vs  $\wedge$  Isign q vs)
| Isgn (Or p q) vs = (Isign p vs  $\vee$  Isign q vs)

lemmas eqs = Isgn.simps Iprod.simps

lemma (x::'a::{linordered-idom}) ^4 * y * z * y^2 * z^23 > 0
  apply (reify eqs)
  oops

end

```

41 Square roots of primes are irrational

```

theory Sqrt
imports Complex-Main ~~/src/HOL/Number-Theory/Primes
begin

```

The square root of any prime number (including 2) is irrational.

theorem sqrt-prime-irrational:

```

  assumes prime (p::nat)
  shows sqrt (real p)  $\notin$   $\mathbb{Q}$ 

```

proof

```

from ⟨prime p⟩ have p: 1 < p by (simp add: prime-nat-def)
assume sqrt (real p) ∈ ℚ
then obtain m n :: nat where
  n: n ≠ 0 and sqrt-rat: |sqrt (real p)| = real m / real n
  and gcd: gcd m n = 1 by (rule Rats-abs-nat-div-natE)
have eq: m2 = p * n2
proof -
  from n and sqrt-rat have real m = |sqrt (real p)| * real n by simp
  then have real (m2) = (sqrt (real p))2 * real (n2)
    by (auto simp add: power2-eq-square)
  also have (sqrt (real p))2 = real p by simp
  also have ... * real (n2) = real (p * n2) by simp
  finally show ?thesis ..
qed
have p dvd m ∧ p dvd n
proof
  from eq have p dvd m2 ..
  with ⟨prime p⟩ pos2 show p dvd m by (rule prime-dvd-power-nat)
  then obtain k where m = p * k ..
  with eq have p * n2 = p2 * k2 by (auto simp add: power2-eq-square mult-ac)
  with p have n2 = p * k2 by (simp add: power2-eq-square)
  then have p dvd n2 ..
  with ⟨prime p⟩ pos2 show p dvd n by (rule prime-dvd-power-nat)
qed
then have p dvd gcd m n ..
with gcd have p dvd 1 by simp
then have p ≤ 1 by (simp add: dvd-imp-le)
with p show False by simp
qed

corollary sqrt (real (2::nat)) ∉ ℚ
by (rule sqrt-prime-irrational) (rule two-is-prime-nat)

```

41.1 Variations

Here is an alternative version of the main proof, using mostly linear forward-reasoning. While this results in less top-down structure, it is probably closer to proofs seen in mathematics.

theorem

assumes prime (p::nat)
shows sqrt (real p) ∉ ℚ

proof

```

from ⟨prime p⟩ have p: 1 < p by (simp add: prime-nat-def)
assume sqrt (real p) ∈ ℚ
then obtain m n :: nat where
  n: n ≠ 0 and sqrt-rat: |sqrt (real p)| = real m / real n
  and gcd: gcd m n = 1 by (rule Rats-abs-nat-div-natE)
from n and sqrt-rat have real m = |sqrt (real p)| * real n by simp
then have real (m2) = (sqrt (real p))2 * real (n2)

```

```

    by (auto simp add: power2-eq-square)
  also have  $(\text{sqrt } (\text{real } p))^2 = \text{real } p$  by simp
  also have  $\dots * \text{real } (n^2) = \text{real } (p * n^2)$  by simp
  finally have eq:  $m^2 = p * n^2$  ..
  then have  $p \text{ dvd } m^2$  ..
  with  $\langle \text{prime } p \rangle \text{ pos2}$  have dvd-m:  $p \text{ dvd } m$  by (rule prime-dvd-power-nat)
  then obtain  $k$  where  $m = p * k$  ..
  with eq have  $p * n^2 = p^2 * k^2$  by (auto simp add: power2-eq-square mult-ac)
  with  $p$  have  $n^2 = p * k^2$  by (simp add: power2-eq-square)
  then have  $p \text{ dvd } n^2$  ..
  with  $\langle \text{prime } p \rangle \text{ pos2}$  have  $p \text{ dvd } n$  by (rule prime-dvd-power-nat)
  with dvd-m have  $p \text{ dvd gcd } m \ n$  by (rule gcd-greatest-nat)
  with gcd have  $p \text{ dvd } 1$  by simp
  then have  $p \leq 1$  by (simp add: dvd-imp-le)
  with  $p$  show False by simp
qed

end

```

42 Square roots of primes are irrational (script version)

```

theory Sqrt-Script
imports Complex-Main ~~/src/HOL/Number-Theory/Primes
begin

```

Contrast this linear Isabelle/Isar script with Markus Wenzel's more mathematical version.

42.1 Preliminaries

```

lemma prime-nonzero:  $\text{prime } (p::\text{nat}) \implies p \neq 0$ 
  by (force simp add: prime-nat-def)

lemma prime-dvd-other-side:
   $(n::\text{nat}) * n = p * (k * k) \implies \text{prime } p \implies p \text{ dvd } n$ 
  apply (subgoal-tac  $p \text{ dvd } n * n$ , blast dest: prime-dvd-mult-nat)
  apply auto
  done

lemma reduction:  $\text{prime } (p::\text{nat}) \implies$ 
   $0 < k \implies k * k = p * (j * j) \implies k < p * j \wedge 0 < j$ 
  apply (rule ccontr)
  apply (simp add: linorder-not-less)
  apply (erule disjE)
  apply (frule mult-le-mono, assumption)
  apply auto

```

```

apply (force simp add: prime-nat-def)
done

lemma rearrange: (j::nat) * (p * j) = k * k  $\implies$  k * k = p * (j * j)
by (simp add: mult-ac)

lemma prime-not-square:
  prime (p::nat)  $\implies$  ( $\bigwedge k. 0 < k \implies m * m \neq p * (k * k)$ )
apply (induct m rule: nat-less-induct)
apply clarify
apply (frule prime-dvd-other-side, assumption)
apply (erule dvdE)
apply (simp add: nat-mult-eq-cancel-disj prime-nonzero)
apply (blast dest: rearrange reduction)
done

```

42.2 Main theorem

The square root of any prime number (including 2) is irrational.

```

theorem prime-sqrt-irrational:
  prime (p::nat)  $\implies$  x * x = real p  $\implies$  0  $\leq$  x  $\implies$  x  $\notin$   $\mathbb{Q}$ 
apply (rule notI)
apply (erule Rats-abs-nat-div-natE)
apply (simp del: real-of-nat-mult
    add: abs-if divide-eq-eq prime-not-square real-of-nat-mult [symmetric])
done

```

```

lemmas two-sqrt-irrational =
  prime-sqrt-irrational [OF two-is-prime-nat]

```

end

43 Various examples for transfer procedure

```

theory Transfer-Ex
imports Main
begin

```

```

lemma ex1: (x::nat) + y = y + x
by auto

```

```

lemma (0::int)  $\leq$  (y::int)  $\implies$  (0::int)  $\leq$  (x::int)  $\implies$  x + y = y + x
by (fact ex1 [transferred])

```

```

lemma ex2: (a::nat) div b * b + a mod b = a
by (rule mod-div-equality)

```

```

lemma  $(0::int) \leq (b::int) \implies (0::int) \leq (a::int) \implies a \text{ div } b * b + a \text{ mod } b = a$ 
  by (fact ex2 [transferred])

lemma ex3:  $ALL (x::nat). ALL y. EX z. z \geq x + y$ 
  by auto

lemma  $\forall x \geq 0::int. \forall y \geq 0::int. \exists xa \geq 0::int. x + y \leq xa$ 
  by (fact ex3 [transferred nat-int])

lemma ex4:  $(x::nat) \geq y \implies (x - y) + y = x$ 
  by auto

lemma  $(0::int) \leq (x::int) \implies (0::int) \leq (y::int) \implies y \leq x \implies tsub\ x\ y + y = x$ 
  by (fact ex4 [transferred])

lemma ex5:  $(2::nat) * \sum \{..n\} = n * (n + 1)$ 
  by (induct n rule: nat-induct, auto)

lemma  $(0::int) \leq (n::int) \implies (2::int) * \sum \{0::int..n\} = n * (n + (1::int))$ 
  by (fact ex5 [transferred])

lemma  $(0::nat) \leq (n::nat) \implies (2::nat) * \sum \{0::nat..n\} = n * (n + (1::nat))$ 
  by (fact ex5 [transferred, transferred])

theorem ex6:  $0 \leq (n::int) \implies 2 * \sum \{0..n\} = n * (n + 1)$ 
  by (rule ex5 [transferred])

lemma  $(0::nat) \leq (n::nat) \implies (2::nat) * \sum \{0::nat..n\} = n * (n + (1::nat))$ 
  by (fact ex6 [transferred])

end

```

44 Arithmetic Series for Reals

```

theory Arithmetic-Series-Complex
imports Complex-Main
begin

```

```

lemma arith-series-real:
   $(2::real) * (\sum i \in \{..<n\}. a + of\_nat\ i * d) =$ 
   $of\_nat\ n * (a + (a + of\_nat(n - 1) * d))$ 
proof –
  have
     $((1::real) + 1) * (\sum i \in \{..<n\}. a + of\_nat(i) * d) =$ 
     $of\_nat(n) * (a + (a + of\_nat(n - 1) * d))$ 
    by (rule arith-series-general)
  thus ?thesis by simp
qed

```

end

45 Divergence of the Harmonic Series

theory *HarmonicSeries*
imports *Complex-Main*
begin

45.1 Abstract

The following document presents a proof of the Divergence of Harmonic Series theorem formalised in the Isabelle/Isar theorem proving system.

Theorem: The series $\sum_{n=1}^{\infty} \frac{1}{n}$ does not converge to any number.

Informal Proof: The informal proof is based on the following auxillary lemmas:

- *aux*: $\sum_{n=2^m-1}^{2^m} \frac{1}{n} \geq \frac{1}{2}$
- *aux2*: $\sum_{n=1}^{2^M} \frac{1}{n} = 1 + \sum_{m=1}^M \sum_{n=2^m-1}^{2^m} \frac{1}{n}$

From *aux* and *aux2* we can deduce that $\sum_{n=1}^{2^M} \frac{1}{n} \geq 1 + \frac{M}{2}$ for all M . Now for contradiction, assume that $\sum_{n=1}^{\infty} \frac{1}{n} = s$ for some s . Because $\forall n. \frac{1}{n} > 0$ all the partial sums in the series must be less than s . However with our deduction above we can choose $N > 2 * s - 2$ and thus $\sum_{n=1}^{2^N} \frac{1}{n} > s$. This leads to a contradiction and hence $\sum_{n=1}^{\infty} \frac{1}{n}$ is not summable. QED.

45.2 Formal Proof

lemma *two-pow-sub*:

$0 < m \implies (2::nat)^m - 2^{(m-1)} = 2^{(m-1)}$

by (*induct m*) *auto*

We first prove the following auxillary lemma. This lemma simply states that the finite sums: $\frac{1}{2}, \frac{1}{3} + \frac{1}{4}, \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}$ etc. are all greater than or equal to $\frac{1}{2}$. We do this by observing that each term in the sum is greater than or equal to the last term, e.g. $\frac{1}{3} > \frac{1}{4}$ and thus $\frac{1}{3} + \frac{1}{4} > \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$.

lemma *harmonic-aux*:

$\forall m > 0. (\sum_{n \in \{(2::nat)^{(m-1)+1} .. 2^m\}} 1/\text{real } n) \geq 1/2$

(is $\forall m > 0. (\sum_{n \in (?S\ m)} 1/\text{real } n) \geq 1/2$)

proof

fix *m::nat*

obtain *tm* **where** *tmdef*: $tm = (2::nat)^m$ **by** *simp*

{

assume *mgt0*: $0 < m$

```

have  $\bigwedge x. x \in (?S\ m) \implies 1 / (\text{real } x) \geq 1 / (\text{real } tm)$ 
proof -
  fix  $x :: \text{nat}$ 
  assume  $xs: x \in (?S\ m)$ 
  have  $xgt0: x > 0$ 
  proof -
    from  $xs$  have
       $x \geq 2^{(m - 1)} + 1$  by auto
    moreover with  $mgt0$  have
       $2^{(m - 1)} + 1 \geq (1 :: \text{nat})$  by auto
    ultimately have
       $x \geq 1$  by (rule xtrans)
    thus ?thesis by simp
  qed
  moreover from  $xs$  have  $x \leq 2^m$  by auto
  ultimately have
     $\text{inverse } (\text{real } x) \geq \text{inverse } (\text{real } ((2 :: \text{nat})^m))$ 
    by (simp del: real-of-nat-power)
  moreover
  from  $xgt0$  have  $\text{real } x \neq 0$  by simp
  then have
     $\text{inverse } (\text{real } x) = 1 / (\text{real } x)$ 
    by (rule nonzero-inverse-eq-divide)
  moreover from  $mgt0$  have  $\text{real } tm \neq 0$  by (simp add: tmdef)
  then have
     $\text{inverse } (\text{real } tm) = 1 / (\text{real } tm)$ 
    by (rule nonzero-inverse-eq-divide)
  ultimately show
     $1 / (\text{real } x) \geq 1 / (\text{real } tm)$  by (auto simp add: tmdef)
qed
then have
   $(\sum n \in (?S\ m). 1 / \text{real } n) \geq (\sum n \in (?S\ m). 1 / (\text{real } tm))$ 
  by (rule setsum-mono)
moreover have
   $(\sum n \in (?S\ m). 1 / (\text{real } tm)) = 1/2$ 
proof -
  have
     $(\sum n \in (?S\ m). 1 / (\text{real } tm)) =$ 
     $(1 / (\text{real } tm)) * (\sum n \in (?S\ m). 1)$ 
    by simp
  also have
     $\dots = ((1 / (\text{real } tm)) * \text{real } (\text{card } (?S\ m)))$ 
    by (simp add: real-of-card real-of-nat-def)
  also have
     $\dots = ((1 / (\text{real } tm)) * \text{real } (tm - (2^{(m - 1)})))$ 
    by (simp add: tmdef)
  also from  $mgt0$  have
     $\dots = ((1 / (\text{real } tm)) * \text{real } ((2 :: \text{nat})^{(m - 1)}))$ 
    by (auto simp: tmdef dest: two-pow-sub)

```

```

also have
  ... = (real (2::nat))^(m - 1) / (real (2::nat))^m
  by (simp add: tmdef)
also from mgt0 have
  ... = (real (2::nat))^(m - 1) / (real (2::nat))^((m - 1) + 1)
  by auto
also have ... = 1/2 by simp
finally show ?thesis .
qed
ultimately have
  (∑ n∈(?S m). 1 / real n) ≥ 1/2
  by - (erule subst)
}
thus 0 < m → 1 / 2 ≤ (∑ n∈(?S m). 1 / real n) by simp
qed

```

We then show that the sum of a finite number of terms from the harmonic series can be regrouped in increasing powers of 2. For example: $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} = 1 + (\frac{1}{2}) + (\frac{1}{3} + \frac{1}{4}) + (\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8})$.

```

lemma harmonic-aux2 [rule-format]:
  0 < M ⇒ (∑ n∈{1..(2::nat)^M}. 1/real n) =
    (1 + (∑ m∈{1..M}. ∑ n∈{(2::nat)^(m-1)+1..2^m}. 1/real n))
  (is 0 < M ⇒ ?LHS M = ?RHS M)
proof (induct M)
  case 0 show ?case by simp
next
  case (Suc M)
  have ant: 0 < Suc M by fact
  {
    have suc: ?LHS (Suc M) = ?RHS (Suc M)
    proof cases — show that LHS = c and RHS = c, and thus LHS = RHS
      assume mz: M=0
      {
        then have
          ?LHS (Suc M) = ?LHS 1 by simp
        also have
          ... = (∑ n∈{(1::nat)..2}. 1/real n) by simp
        also have
          ... = ((∑ n∈{Suc 1..2}. 1/real n) + 1/(real (1::nat)))
          by (subst setsum-head)
          (auto simp: atLeastSucAtMost-greaterThanAtMost)
        also have
          ... = ((∑ n∈{2..2::nat}. 1/real n) + 1/(real (1::nat)))
          by (simp add: nat-number)
        also have
          ... = 1/(real (2::nat)) + 1/(real (1::nat)) by simp
        finally have
          ?LHS (Suc M) = 1/2 + 1 by simp
      }
    }
  }

```



```

moreover
{
  from mz have
     $?RHS (Suc M) = ?RHS 1$  by simp
  also have
     $\dots = (\sum n \in \{(2::nat) ^ 0 + 1 .. 2 ^ 1\}. 1 / \text{real } n) + 1$ 
    by simp
  also have
     $\dots = (\sum n \in \{2::nat .. 2\}. 1 / \text{real } n) + 1$ 
    by (auto simp: atLeastAtMost-singleton)
  also have
     $\dots = 1/2 + 1$ 
    by simp
  finally have
     $?RHS (Suc M) = 1/2 + 1$  by simp
}
ultimately show  $?LHS (Suc M) = ?RHS (Suc M)$  by simp
next
assume mnz:  $M \neq 0$ 
then have mgtz:  $M > 0$  by simp
with Suc have suc:
   $(?LHS M) = (?RHS M)$  by blast
have
   $(?LHS (Suc M)) =$ 
   $((?LHS M) + (\sum n \in \{(2::nat) ^ M + 1 .. 2 ^ (Suc M)\}. 1 / \text{real } n))$ 
proof –
  have
     $\{1 .. (2::nat) ^ (Suc M)\} =$ 
     $\{1 .. (2::nat) ^ M\} \cup \{(2::nat) ^ M + 1 .. (2::nat) ^ (Suc M)\}$ 
    by auto
  moreover have
     $\{1 .. (2::nat) ^ M\} \cap \{(2::nat) ^ M + 1 .. (2::nat) ^ (Suc M)\} = \{\}$ 
    by auto
  moreover have
    finite  $\{1 .. (2::nat) ^ M\}$  and finite  $\{(2::nat) ^ M + 1 .. (2::nat) ^ (Suc M)\}$ 
    by auto
  ultimately show ?thesis
    by (auto intro: setsum-Un-disjoint)
qed
moreover
{
  have
     $(?RHS (Suc M)) =$ 
     $(1 + (\sum m \in \{1 .. M\}. \sum n \in \{(2::nat) ^ (m - 1) + 1 .. 2 ^ m\}. 1 / \text{real } n) +$ 
     $(\sum n \in \{(2::nat) ^ (Suc M - 1) + 1 .. 2 ^ (Suc M)\}. 1 / \text{real } n))$  by simp
  also have
     $\dots = (?RHS M) + (\sum n \in \{(2::nat) ^ M + 1 .. 2 ^ (Suc M)\}. 1 / \text{real } n)$ 
    by simp
  also from suc have

```

```

... = (?LHS M) + (∑ n∈{(2::nat) ^M+1..2^(Suc M)}. 1/real n)
  by simp
finally have
  (?RHS (Suc M)) = ... by simp
}
ultimately show ?LHS (Suc M) = ?RHS (Suc M) by simp
qed
}
thus ?case by simp
qed

```

Using *harmonic-aux* and *harmonic-aux2* we now show that each group sum is greater than or equal to $\frac{1}{2}$ and thus the finite sum is bounded below by a value proportional to the number of elements we choose.

```

lemma harmonic-aux3 [rule-format]:
  shows ∀ (M::nat). (∑ n∈{1..(2::nat) ^M}. 1 / real n) ≥ 1 + (real M)/2
  (is ∀ M. ?P M ≥ -)
proof (rule allI, cases)
  fix M::nat
  assume M=0
  then show ?P M ≥ 1 + (real M)/2 by simp
next
  fix M::nat
  assume M≠0
  then have M > 0 by simp
  then have
    (?P M) =
      (1 + (∑ m∈{1..M}. ∑ n∈{(2::nat) ^ (m - 1)+1..2^m}. 1/real n))
    by (rule harmonic-aux2)
  also have
    ... ≥ (1 + (∑ m∈{1..M}. 1/2))
proof -
  let ?f = (λx. 1/2)
  let ?g = (λx. (∑ n∈{(2::nat) ^ (x - 1)+1..2^x}. 1/real n))
  from harmonic-aux have ∧x. x∈{1..M} ⇒ ?f x ≤ ?g x by simp
  then have (∑ m∈{1..M}. ?g m) ≥ (∑ m∈{1..M}. ?f m) by (rule setsum-mono)
  thus ?thesis by simp
qed
finally have (?P M) ≥ (1 + (∑ m∈{1..M}. 1/2)) .
moreover
{
  have
    (∑ m∈{1..M}. (1::real)/2) = 1/2 * (∑ m∈{1..M}. 1)
    by auto
  also have
    ... = 1/2*(real (card {1..M}))
    by (simp only: real-of-card[symmetric])
  also have
    ... = 1/2*(real M) by simp
}

```

also have
 $\dots = (\text{real } M)/2$ **by** *simp*
 finally have $(\sum_{m \in \{1..M\}}. (1::\text{real})/2) = (\text{real } M)/2$.
 }
 ultimately show $(?P \ M) \geq (1 + (\text{real } M)/2)$ **by** *simp*
qed

The final theorem shows that as we take more and more elements (see *harmonic-aux3*) we get an ever increasing sum. By assuming the sum converges, the lemma *series-pos-less* ($\llbracket \text{summable } ?f; \forall m \geq ?n. 0 < ?f \ m \rrbracket \implies \text{setsum } ?f \ \{0..<?n\} < \text{suminf } ?f$) states that each sum is bounded above by the series' limit. This contradicts our first statement and thus we prove that the harmonic series is divergent.

theorem *DivergenceOfHarmonicSeries*:
 shows $\neg \text{summable } (\lambda n. 1/\text{real } (\text{Suc } n))$
 (is $\neg \text{summable } ?f$)

proof — by contradiction

let $?s = \text{suminf } ?f$ — let $?s$ equal the sum of the harmonic series

assume $sf: \text{summable } ?f$

then obtain $n::\text{nat}$ where $ndef: n = \text{nat } \lceil 2 * ?s \rceil$ **by** *simp*

then have $\text{ngt}: 1 + \text{real } n/2 > ?s$

proof —

have $\forall n. 0 \leq ?f \ n$ **by** *simp*

with sf have $?s \geq 0$

by — (rule *suminf-0-le*, *simp-all*)

then have $\text{cgt0}: \lceil 2 * ?s \rceil \geq 0$ **by** *simp*

from $ndef$ have $n = \text{nat } \lceil (2 * ?s) \rceil$.

then have $\text{real } n = \text{real } (\text{nat } \lceil 2 * ?s \rceil)$ **by** *simp*

with cgt0 have $\text{real } n = \text{real } \lceil 2 * ?s \rceil$

by (auto dest: *real-nat-eq-real*)

then have $\text{real } n \geq 2 * (?s)$ **by** *simp*

then have $\text{real } n/2 \geq (?s)$ **by** *simp*

then show $1 + \text{real } n/2 > (?s)$ **by** *simp*

qed

obtain j where $jdef: j = (2::\text{nat})^n$ **by** *simp*

have $\forall m \geq j. 0 < ?f \ m$ **by** *simp*

with sf have $(\sum_{i \in \{0..<j\}}. ?f \ i) < ?s$ **by** (rule *series-pos-less*)

then have $(\sum_{i \in \{1..<\text{Suc } j\}}. 1/(\text{real } i)) < ?s$

apply —

apply (*subst(asm) setsum-shift-bounds-Suc-ivl [symmetric]*)

by *simp*

with $jdef$ have

$(\sum_{i \in \{1..<\text{Suc } ((2::\text{nat})^n)\}}. 1 / (\text{real } i)) < ?s$ **by** *simp*

then have

$(\sum_{i \in \{1..(2::\text{nat})^n\}}. 1 / (\text{real } i)) < ?s$

by (*simp only: atLeastLessThanSuc-atLeastAtMost*)

moreover from *harmonic-aux3* have

```

      ( $\sum_{i \in \{1..(2::nat) \wedge n\}} 1 / (\text{real } i) \geq 1 + \text{real } n/2$  by simp
    moreover from ngt have  $1 + \text{real } n/2 > ?s$  by simp
    ultimately show False by simp
  qed

end

```

46 Examples for the 'refute' command

```

theory Refute-Examples imports Main
begin

refute-params [satsolver=dpll]

lemma  $P \wedge Q$ 
  apply (rule conjI)
  refute 1 — refutes  $P$ 
  refute 2 — refutes  $Q$ 
  refute — equivalent to 'refute 1'
    — here 'refute 3' would cause an exception, since we only have 2 subgoals
  refute [maxsize=5] — we can override parameters ...
  refute [satsolver=dpll] 2 — ... and specify a subgoal at the same time
oops

```

46.1 Examples and Test Cases

46.1.1 Propositional logic

```

lemma True
  refute
  apply auto
done

lemma False
  refute
oops

lemma  $P$ 
  refute
oops

lemma  $\sim P$ 
  refute
oops

lemma  $P \ \& \ Q$ 
  refute
oops

```

```
lemma  $P \mid Q$ 
  refute
oops
```

```
lemma  $P \longrightarrow Q$ 
  refute
oops
```

```
lemma  $(P::\text{bool}) = Q$ 
  refute
oops
```

```
lemma  $(P \mid Q) \longrightarrow (P \ \& \ Q)$ 
  refute
oops
```

46.1.2 Predicate logic

```
lemma  $P \ x \ y \ z$ 
  refute
oops
```

```
lemma  $P \ x \ y \longrightarrow P \ y \ x$ 
  refute
oops
```

```
lemma  $P \ (f \ (f \ x)) \longrightarrow P \ x \longrightarrow P \ (f \ x)$ 
  refute
oops
```

46.1.3 Equality

```
lemma  $P = \text{True}$ 
  refute
oops
```

```
lemma  $P = \text{False}$ 
  refute
oops
```

```
lemma  $x = y$ 
  refute
oops
```

```
lemma  $f \ x = g \ x$ 
  refute
oops
```

```
lemma  $(f::'a \Rightarrow 'b) = g$ 
```

```

  refute
oops

```

```

lemma (f::('d⇒'d)⇒('c⇒'d)) = g
  refute
oops

```

```

lemma distinct [a,b]
  refute
  apply simp
  refute
oops

```

46.1.4 First-Order Logic

```

lemma ∃ x. P x
  refute
oops

```

```

lemma ∀ x. P x
  refute
oops

```

```

lemma EX! x. P x
  refute
oops

```

```

lemma Ex P
  refute
oops

```

```

lemma All P
  refute
oops

```

```

lemma Ex1 P
  refute
oops

```

```

lemma (∃ x. P x) ⟶ (∀ x. P x)
  refute
oops

```

```

lemma (∀ x. ∃ y. P x y) ⟶ (∃ y. ∀ x. P x y)
  refute
oops

```

```

lemma (∃ x. P x) ⟶ (EX! x. P x)
  refute

```

oops

A true statement (also testing names of free and bound variables being identical)

```
lemma  $(\forall x y. P x y \longrightarrow P y x) \longrightarrow (\forall x. P x y) \longrightarrow P y x$   
  refute [maxsize=4]  
  apply fast  
done
```

”A type has at most 4 elements.”

```
lemma  $a=b \mid a=c \mid a=d \mid a=e \mid b=c \mid b=d \mid b=e \mid c=d \mid c=e \mid d=e$   
  refute  
oops
```

```
lemma  $\forall a b c d e. a=b \mid a=c \mid a=d \mid a=e \mid b=c \mid b=d \mid b=e \mid c=d \mid c=e \mid d=e$   
  refute  
oops
```

”Every reflexive and symmetric relation is transitive.”

```
lemma  $[\![ \forall x. P x x; \forall x y. P x y \longrightarrow P y x ]\!] \Longrightarrow P x y \longrightarrow P y z \longrightarrow P x z$   
  refute  
oops
```

The ”Drinker’s theorem” ...

```
lemma  $\exists x. f x = g x \longrightarrow f = g$   
  refute [maxsize=4]  
  apply (auto simp add: ext)  
done
```

... and an incorrect version of it

```
lemma  $(\exists x. f x = g x) \longrightarrow f = g$   
  refute  
oops
```

”Every function has a fixed point.”

```
lemma  $\exists x. f x = x$   
  refute  
oops
```

”Function composition is commutative.”

```
lemma  $f (g x) = g (f x)$   
  refute  
oops
```

”Two functions that are equivalent wrt. the same predicate ‘P’ are equal.”

```
lemma  $((P::('a \Rightarrow 'b) \Rightarrow bool) f = P g) \longrightarrow (f x = g x)$   
  refute  
oops
```

46.1.5 Higher-Order Logic

```
lemma  $\exists P. P$   
  refute  
  apply auto  
done
```

```
lemma  $\forall P. P$   
  refute  
oops
```

```
lemma  $EX! P. P$   
  refute  
  apply auto  
done
```

```
lemma  $EX! P. P\ x$   
  refute  
oops
```

```
lemma  $P\ Q \mid Q\ x$   
  refute  
oops
```

```
lemma  $x \neq All$   
  refute  
oops
```

```
lemma  $x \neq Ex$   
  refute  
oops
```

```
lemma  $x \neq Ex1$   
  refute  
oops
```

”The transitive closure ’T’ of an arbitrary relation ’P’ is non-empty.”

```
definition trans :: ( $'a \Rightarrow 'a \Rightarrow bool$ )  $\Rightarrow bool$  where  
  trans  $P == (ALL\ x\ y\ z. P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z)$ 
```

```
definition subset :: ( $'a \Rightarrow 'a \Rightarrow bool$ )  $\Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$  where  
  subset  $P\ Q == (ALL\ x\ y. P\ x\ y \longrightarrow Q\ x\ y)$ 
```

```
definition trans-closure :: ( $'a \Rightarrow 'a \Rightarrow bool$ )  $\Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$  where  
  trans-closure  $P\ Q == (subset\ Q\ P) \ \&\ (trans\ P) \ \&\ (ALL\ R. subset\ Q\ R \longrightarrow trans\ R \longrightarrow subset\ P\ R)$ 
```

```
lemma trans-closure  $T\ P \longrightarrow (\exists x\ y. T\ x\ y)$   
  refute  
oops
```


”The union of transitive closures is equal to the transitive closure of unions.”

```
lemma (∀ x y. (P x y | R x y) ⟶ T x y) ⟶ trans T ⟶ (∀ Q. (∀ x y. (P x y |
R x y) ⟶ Q x y) ⟶ trans Q ⟶ subset T Q)
  ⟶ trans-closure TP P
  ⟶ trans-closure TR R
  ⟶ (T x y = (TP x y | TR x y))
  refute
oops
```

”Every surjective function is invertible.”

```
lemma (∀ y. ∃ x. y = f x) ⟶ (∃ g. ∀ x. g (f x) = x)
  refute
oops
```

”Every invertible function is surjective.”

```
lemma (∃ g. ∀ x. g (f x) = x) ⟶ (∀ y. ∃ x. y = f x)
  refute
oops
```

Every point is a fixed point of some function.

```
lemma ∃ f. f x = x
  refute [maxsize=4]
  apply (rule-tac x=λx. x in exI)
  apply simp
done
```

Axiom of Choice: first an incorrect version ...

```
lemma (∀ x. ∃ y. P x y) ⟶ (EX!f. ∀ x. P x (f x))
  refute
oops
```

... and now two correct ones

```
lemma (∀ x. ∃ y. P x y) ⟶ (∃ f. ∀ x. P x (f x))
  refute [maxsize=4]
  apply (simp add: choice)
done
```

```
lemma (∀ x. EX!y. P x y) ⟶ (EX!f. ∀ x. P x (f x))
  refute [maxsize=2]
  apply auto
  apply (simp add: ex1-implies-ex choice)
  apply (fast intro: ext)
done
```

46.1.6 Meta-logic

```
lemma !!x. P x
  refute
```

oops

```
lemma  $f\ x == g\ x$ 
  refute
oops
```

```
lemma  $P \implies Q$ 
  refute
oops
```

```
lemma  $\llbracket P; Q; R \rrbracket \implies S$ 
  refute
oops
```

```
lemma  $(x == all) \implies False$ 
  refute
oops
```

```
lemma  $(x == (op ==)) \implies False$ 
  refute
oops
```

```
lemma  $(x == (op \implies)) \implies False$ 
  refute
oops
```

46.1.7 Schematic variables

```
schematic-lemma ?P
  refute
  apply auto
done
```

```
schematic-lemma  $x = ?y$ 
  refute
  apply auto
done
```

46.1.8 Abstractions

```
lemma  $(\lambda x. x) = (\lambda x. y)$ 
  refute
oops
```

```
lemma  $(\lambda f. f\ x) = (\lambda f. True)$ 
  refute
oops
```

```
lemma  $(\lambda x. x) = (\lambda y. y)$ 
  refute
```

```

    apply simp
done

```

46.1.9 Sets

```

lemma P (A::'a set)
  refute
oops

```

```

lemma P (A::'a set set)
  refute
oops

```

```

lemma {x. P x} = {y. P y}
  refute
  apply simp
done

```

```

lemma x : {x. P x}
  refute
oops

```

```

lemma P op:
  refute
oops

```

```

lemma P (op: x)
  refute
oops

```

```

lemma P Collect
  refute
oops

```

```

lemma A Un B = A Int B
  refute
oops

```

```

lemma (A Int B) Un C = (A Un C) Int B
  refute
oops

```

```

lemma Ball A P ⟶ Bex A P
  refute
oops

```

46.1.10 undefined

```

lemma undefined
  refute

```

oops

lemma P *undefined*
 refute
oops

lemma *undefined* x
 refute
oops

lemma *undefined* *undefined*
 refute
oops

46.1.11 The

lemma *The* P
 refute
oops

lemma P *The*
 refute
oops

lemma P (*The* P)
 refute
oops

lemma (*THE* $x. x=y$) = z
 refute
oops

lemma $Ex P \longrightarrow P$ (*The* P)
 refute
oops

46.1.12 Eps

lemma *Eps* P
 refute
oops

lemma P *Eps*
 refute
oops

lemma P (*Eps* P)
 refute
oops

```

lemma (SOME x. x=y) = z
  refute
oops

```

```

lemma Ex P  $\longrightarrow$  P (Eps P)
  refute [maxsize=3]
  apply (auto simp add: someI)
done

```

46.1.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of 'a:

```

typedef 'a myTdef = insert (undefined::'a) (undefined::'a set)
  by auto

```

```

lemma (x::'a myTdef) = y
  refute
oops

```

```

typedecl myTdecl

```

```

typedef 'a T-bij = {(f::'a $\Rightarrow$ 'a).  $\forall y. \exists!x. f\ x = y$ }
  by auto

```

```

lemma P (f::(myTdecl myTdef) T-bij)
  refute
oops

```

46.1.14 Inductive datatypes

With *quick-and-dirty* set, the datatype package does not generate certain axioms for recursion operators. Without these axioms, refute may find spurious countermodels.

```

unit

```

```

lemma P (x::unit)
  refute
oops

```

```

lemma  $\forall x::unit. P\ x$ 
  refute
oops

```

```

lemma P ()
  refute
oops

```

```

lemma unit-rec u x = u

```

```

    refute
    apply simp
done

lemma P (unit-rec u x)
  refute
oops

lemma P (case x of ()  $\Rightarrow$  u)
  refute
oops

option

lemma P (x::'a option)
  refute
oops

lemma  $\forall x::'a$  option. P x
  refute
oops

lemma P None
  refute
oops

lemma P (Some x)
  refute
oops

lemma option-rec n s None = n
  refute
  apply simp
done

lemma option-rec n s (Some x) = s x
  refute [maxsize=4]
  apply simp
done

lemma P (option-rec n s x)
  refute
oops

lemma P (case x of None  $\Rightarrow$  n | Some u  $\Rightarrow$  s u)
  refute
oops

*

lemma P (x::'a*'b)

```

```

    refute
oops

lemma  $\forall x::'a*'b. P\ x$ 
  refute
oops

lemma  $P\ (x, y)$ 
  refute
oops

lemma  $P\ (fst\ x)$ 
  refute
oops

lemma  $P\ (snd\ x)$ 
  refute
oops

lemma  $P\ Pair$ 
  refute
oops

lemma  $prod-rec\ p\ (a, b) = p\ a\ b$ 
  refute [maxsize=2]
  apply simp
oops

lemma  $P\ (prod-rec\ p\ x)$ 
  refute
oops

lemma  $P\ (case\ x\ of\ Pair\ a\ b \Rightarrow p\ a\ b)$ 
  refute
oops

+

lemma  $P\ (x::'a+'b)$ 
  refute
oops

lemma  $\forall x::'a+'b. P\ x$ 
  refute
oops

lemma  $P\ (Inl\ x)$ 
  refute
oops

```

```

lemma P (Inr x)
  refute
oops

lemma P Inl
  refute
oops

lemma sum-rec l r (Inl x) = l x
  refute [maxsize=3]
  apply simp
done

lemma sum-rec l r (Inr x) = r x
  refute [maxsize=3]
  apply simp
done

lemma P (sum-rec l r x)
  refute
oops

lemma P (case x of Inl a  $\Rightarrow$  l a | Inr b  $\Rightarrow$  r b)
  refute
oops

Non-recursive datatypes

datatype T1 = A | B

lemma P (x::T1)
  refute
oops

lemma  $\forall x::T1. P x$ 
  refute
oops

lemma P A
  refute
oops

lemma P B
  refute
oops

lemma T1-rec a b A = a
  refute
  apply simp
done

```



```

lemma T1-rec a b B = b
  refute
  apply simp
done

lemma P (T1-rec a b x)
  refute
oops

lemma P (case x of A  $\Rightarrow$  a | B  $\Rightarrow$  b)
  refute
oops

datatype 'a T2 = C T1 | D 'a

lemma P (x::'a T2)
  refute
oops

lemma  $\forall x::'a T2. P x$ 
  refute
oops

lemma P D
  refute
oops

lemma T2-rec c d (C x) = c x
  refute [maxsize=4]
  apply simp
done

lemma T2-rec c d (D x) = d x
  refute [maxsize=4]
  apply simp
done

lemma P (T2-rec c d x)
  refute
oops

lemma P (case x of C u  $\Rightarrow$  c u | D v  $\Rightarrow$  d v)
  refute
oops

datatype ('a,'b) T3 = E 'a  $\Rightarrow$  'b

lemma P (x::('a,'b) T3)

```

refute
oops

lemma $\forall x::('a,'b) \ T3. \ P \ x$
refute
oops

lemma $P \ E$
refute
oops

lemma $T3\text{-}rec \ e \ (E \ x) = e \ x$
refute $[maxsize=2]$
apply *simp*
done

lemma $P \ (T3\text{-}rec \ e \ x)$
refute
oops

lemma $P \ (\text{case } x \text{ of } E \ f \Rightarrow e \ f)$
refute
oops

Recursive datatypes

nat

lemma $P \ (x::nat)$
refute
oops

lemma $\forall x::nat. \ P \ x$
refute
oops

lemma $P \ (Suc \ 0)$
refute
oops

lemma $P \ Suc$
refute — *Suc* is a partial function (regardless of the size of the model), hence *P Suc* is undefined, hence no model will be found
oops

lemma $nat\text{-}rec \ zero \ suc \ 0 = zero$
refute
apply *simp*
done

lemma $nat\text{-}rec \ zero \ suc \ (Suc \ x) = suc \ x \ (nat\text{-}rec \ zero \ suc \ x)$

```

    refute [maxsize=2]
    apply simp
done

lemma P (nat-rec zero suc x)
  refute
oops

lemma P (case x of 0 ⇒ zero | Suc n ⇒ suc n)
  refute
oops

'a list

lemma P (xs::'a list)
  refute
oops

lemma ∀ xs::'a list. P xs
  refute
oops

lemma P [x, y]
  refute
oops

lemma list-rec nil cons [] = nil
  refute [maxsize=3]
  apply simp
done

lemma list-rec nil cons (x#xs) = cons x xs (list-rec nil cons xs)
  refute [maxsize=2]
  apply simp
done

lemma P (list-rec nil cons xs)
  refute
oops

lemma P (case x of Nil ⇒ nil | Cons a b ⇒ cons a b)
  refute
oops

lemma (xs::'a list) = ys
  refute
oops

lemma a # xs = b # xs
  refute

```

```

oops

datatype BitList = BitListNil | Bit0 BitList | Bit1 BitList

lemma P (x::BitList)
  refute
oops

lemma  $\forall x::\text{BitList}. P\ x$ 
  refute
oops

lemma P (Bit0 (Bit1 BitListNil))
  refute
oops

lemma BitList-rec nil bit0 bit1 BitListNil = nil
  refute [maxsize=4]
  apply simp
done

lemma BitList-rec nil bit0 bit1 (Bit0 xs) = bit0 xs (BitList-rec nil bit0 bit1 xs)
  refute [maxsize=2]
  apply simp
done

lemma BitList-rec nil bit0 bit1 (Bit1 xs) = bit1 xs (BitList-rec nil bit0 bit1 xs)
  refute [maxsize=2]
  apply simp
done

lemma P (BitList-rec nil bit0 bit1 x)
  refute
oops

datatype 'a BinTree = Leaf 'a | Node 'a BinTree 'a BinTree

lemma P (x::'a BinTree)
  refute
oops

lemma  $\forall x::'a\ \text{BinTree}. P\ x$ 
  refute
oops

lemma P (Node (Leaf x) (Leaf y))
  refute
oops

```

```

lemma BinTree-rec l n (Leaf x) = l x
  refute [maxsize=1]
  apply simp
done

```

```

lemma BinTree-rec l n (Node x y) = n x y (BinTree-rec l n x) (BinTree-rec l n y)
  refute [maxsize=1]
  apply simp
done

```

```

lemma P (BinTree-rec l n x)
  refute
oops

```

```

lemma P (case x of Leaf a  $\Rightarrow$  l a | Node a b  $\Rightarrow$  n a b)
  refute
oops

```

Mutually recursive datatypes

```

datatype 'a aexp = Number 'a | ITE 'a bexp 'a aexp 'a aexp
  and 'a bexp = Equal 'a aexp 'a aexp

```

```

lemma P (x::'a aexp)
  refute
oops

```

```

lemma  $\forall x::'a aexp. P x$ 
  refute
oops

```

```

lemma P (ITE (Equal (Number x) (Number y)) (Number x) (Number y))
  refute
oops

```

```

lemma P (x::'a bexp)
  refute
oops

```

```

lemma  $\forall x::'a bexp. P x$ 
  refute
oops

```

```

lemma aexp-bexp-rec-1 number ite equal (Number x) = number x
  refute [maxsize=1]
  apply simp
done

```

```

lemma aexp-bexp-rec-1 number ite equal (ITE x y z) = ite x y z (aexp-bexp-rec-2
number ite equal x) (aexp-bexp-rec-1 number ite equal y) (aexp-bexp-rec-1 number

```

```

ite equal z)
  refute [maxsize=1]
  apply simp
done

```

```

lemma P (aexp-bexp-rec-1 number ite equal x)
  refute
oops

```

```

lemma P (case x of Number a  $\Rightarrow$  number a | ITE b a1 a2  $\Rightarrow$  ite b a1 a2)
  refute
oops

```

```

lemma aexp-bexp-rec-2 number ite equal (Equal x y) = equal x y (aexp-bexp-rec-1
number ite equal x) (aexp-bexp-rec-1 number ite equal y)
  refute [maxsize=1]
  apply simp
done

```

```

lemma P (aexp-bexp-rec-2 number ite equal x)
  refute
oops

```

```

lemma P (case x of Equal a1 a2  $\Rightarrow$  equal a1 a2)
  refute
oops

```

```

datatype X = A | B X | C Y
  and Y = D X | E Y | F

```

```

lemma P (x::X)
  refute
oops

```

```

lemma P (y::Y)
  refute
oops

```

```

lemma P (B (B A))
  refute
oops

```

```

lemma P (B (C F))
  refute
oops

```

```

lemma P (C (D A))
  refute
oops

```

```
lemma P (C (E F))
  refute
oops
```

```
lemma P (D (B A))
  refute
oops
```

```
lemma P (D (C F))
  refute
oops
```

```
lemma P (E (D A))
  refute
oops
```

```
lemma P (E (E F))
  refute
oops
```

```
lemma P (C (D (C F)))
  refute
oops
```

```
lemma X-Y-rec-1 a b c d e f A = a
  refute [maxsize=3]
  apply simp
done
```

```
lemma X-Y-rec-1 a b c d e f (B x) = b x (X-Y-rec-1 a b c d e f x)
  refute [maxsize=1]
  apply simp
done
```

```
lemma X-Y-rec-1 a b c d e f (C y) = c y (X-Y-rec-2 a b c d e f y)
  refute [maxsize=1]
  apply simp
done
```

```
lemma X-Y-rec-2 a b c d e f (D x) = d x (X-Y-rec-1 a b c d e f x)
  refute [maxsize=1]
  apply simp
done
```

```
lemma X-Y-rec-2 a b c d e f (E y) = e y (X-Y-rec-2 a b c d e f y)
  refute [maxsize=1]
  apply simp
done
```

```

lemma X-Y-rec-2 a b c d e f F = f
  refute [maxsize=3]
  apply simp
done

```

```

lemma P (X-Y-rec-1 a b c d e f x)
  refute
oops

```

```

lemma P (X-Y-rec-2 a b c d e f y)
  refute
oops

```

Other datatype examples

Indirect recursion is implemented via mutual recursion.

```

datatype XOpt = CX XOpt option | DX bool  $\Rightarrow$  XOpt option

```

```

lemma P (x::XOpt)
  refute
oops

```

```

lemma P (CX None)
  refute
oops

```

```

lemma P (CX (Some (CX None)))
  refute
oops

```

```

lemma XOpt-rec-1 cx dx n1 s1 n2 s2 (CX x) = cx x (XOpt-rec-2 cx dx n1 s1 n2 s2 x)
  refute [maxsize=1]
  apply simp
done

```

```

lemma XOpt-rec-1 cx dx n1 s1 n2 s2 (DX x) = dx x ( $\lambda b$ . XOpt-rec-3 cx dx n1 s1 n2 s2 (x b))
  refute [maxsize=1]
  apply simp
done

```

```

lemma XOpt-rec-2 cx dx n1 s1 n2 s2 None = n1
  refute [maxsize=2]
  apply simp
done

```

```

lemma XOpt-rec-2 cx dx n1 s1 n2 s2 (Some x) = s1 x (XOpt-rec-1 cx dx n1 s1 n2 s2 x)

```



```

    refute [maxsize=1]
  apply simp
done

```

```

lemma XOpt-rec-3 cx dx n1 s1 n2 s2 None = n2
  refute [maxsize=2]
  apply simp
done

```

```

lemma XOpt-rec-3 cx dx n1 s1 n2 s2 (Some x) = s2 x (XOpt-rec-1 cx dx n1 s1
n2 s2 x)
  refute [maxsize=1]
  apply simp
done

```

```

lemma P (XOpt-rec-1 cx dx n1 s1 n2 s2 x)
  refute
oops

```

```

lemma P (XOpt-rec-2 cx dx n1 s1 n2 s2 x)
  refute
oops

```

```

lemma P (XOpt-rec-3 cx dx n1 s1 n2 s2 x)
  refute
oops

```

```

datatype 'a YOpt = CY ('a ⇒ 'a YOpt) option

```

```

lemma P (x::'a YOpt)
  refute
oops

```

```

lemma P (CY None)
  refute
oops

```

```

lemma P (CY (Some (λa. CY None)))
  refute
oops

```

```

lemma YOpt-rec-1 cy n s (CY x) = cy x (YOpt-rec-2 cy n s x)
  refute [maxsize=1]
  apply simp
done

```

```

lemma YOpt-rec-2 cy n s None = n
  refute [maxsize=2]
  apply simp

```

```

done

lemma YOpt-rec-2 cy n s (Some x) = s x (λa. YOpt-rec-1 cy n s (x a))
  refute [maxsize=1]
  apply simp
done

lemma P (YOpt-rec-1 cy n s x)
  refute
oops

lemma P (YOpt-rec-2 cy n s x)
  refute
oops

datatype Trie = TR Trie list

lemma P (x::Trie)
  refute
oops

lemma ∀ x::Trie. P x
  refute
oops

lemma P (TR [TR []])
  refute
oops

lemma Trie-rec-1 tr nil cons (TR x) = tr x (Trie-rec-2 tr nil cons x)
  refute [maxsize=1]
  apply simp
done

lemma Trie-rec-2 tr nil cons [] = nil
  refute [maxsize=3]
  apply simp
done

lemma Trie-rec-2 tr nil cons (x#xs) = cons x xs (Trie-rec-1 tr nil cons x) (Trie-rec-2
tr nil cons xs)
  refute [maxsize=1]
  apply simp
done

lemma P (Trie-rec-1 tr nil cons x)
  refute
oops

```

```

lemma  $P$  (Trie-rec-2 tr nil cons x)
  refute
oops

datatype InfTree = Leaf | Node nat  $\Rightarrow$  InfTree

lemma  $P$  ( $x::\textit{InfTree}$ )
  refute
oops

lemma  $\forall x::\textit{InfTree}. P\ x$ 
  refute
oops

lemma  $P$  (Node ( $\lambda n. \textit{Leaf}$ ))
  refute
oops

lemma InfTree-rec leaf node Leaf = leaf
  refute [maxsize=2]
  apply simp
done

lemma InfTree-rec leaf node (Node x) = node x ( $\lambda n. \textit{InfTree-rec leaf node (x n)}$ )
  refute [maxsize=1]
  apply simp
done

lemma  $P$  (InfTree-rec leaf node x)
  refute
oops

datatype  $'a\ \textit{lambda}$  = Var 'a | App 'a lambda 'a lambda | Lam 'a  $\Rightarrow$   $'a\ \textit{lambda}$ 

lemma  $P$  ( $x::'a\ \textit{lambda}$ )
  refute
oops

lemma  $\forall x::'a\ \textit{lambda}. P\ x$ 
  refute
oops

lemma  $P$  (Lam ( $\lambda a. \textit{Var a}$ ))
  refute
oops

lemma lambda-rec var app lam (Var x) = var x
  refute [maxsize=1]
  apply simp

```

done

lemma *lambda-rec var app lam* (*App* *x y*) = *app* *x y* (*lambda-rec var app lam* *x*)
(*lambda-rec var app lam* *y*)
 refute [*maxsize=1*]
 apply *simp*
done

lemma *lambda-rec var app lam* (*Lam* *x*) = *lam* *x* ($\lambda a.$ *lambda-rec var app lam* (*x*
a))
 refute [*maxsize=1*]
 apply *simp*
done

lemma *P* (*lambda-rec v a l* *x*)
 refute
oops

Taken from "Inductive datatypes in HOL", p.8:

datatype (*'a*, *'b*) *T* = *C* *'a* \Rightarrow *bool* | *D* *'b* *list*
datatype *'c* *U* = *E* (*'c*, *'c* *U*) *T*

lemma *P* (*x::'c* *U*)
 refute
oops

lemma $\forall x::'c$ *U*. *P* *x*
 refute
oops

lemma *P* (*E* (*C* ($\lambda a.$ *True*)))
 refute
oops

lemma *U-rec-1* *e c d nil cons* (*E* *x*) = *e* *x* (*U-rec-2* *e c d nil cons* *x*)
 refute [*maxsize=1*]
 apply *simp*
done

lemma *U-rec-2* *e c d nil cons* (*C* *x*) = *c* *x*
 refute [*maxsize=1*]
 apply *simp*
done

lemma *U-rec-2* *e c d nil cons* (*D* *x*) = *d* *x* (*U-rec-3* *e c d nil cons* *x*)
 refute [*maxsize=1*]
 apply *simp*
done

```

lemma U-rec-3 e c d nil cons [] = nil
  refute [maxsize=2]
  apply simp
done

```

```

lemma U-rec-3 e c d nil cons (x#xs) = cons x xs (U-rec-1 e c d nil cons x)
  (U-rec-3 e c d nil cons xs)
  refute [maxsize=1]
  apply simp
done

```

```

lemma P (U-rec-1 e c d nil cons x)
  refute
oops

```

```

lemma P (U-rec-2 e c d nil cons x)
  refute
oops

```

```

lemma P (U-rec-3 e c d nil cons x)
  refute
oops

```

46.1.15 Records

```

record ('a, 'b) point =
  xpos :: 'a
  ypos :: 'b

```

```

lemma (x::('a, 'b) point) = y
  refute
oops

```

```

record ('a, 'b, 'c) extpoint = ('a, 'b) point +
  ext :: 'c

```

```

lemma (x::('a, 'b, 'c) extpoint) = y
  refute
oops

```

46.1.16 Inductively defined sets

```

inductive-set arbitrarySet :: 'a set
where
  undefined : arbitrarySet

```

```

lemma x : arbitrarySet
  refute
oops

```

```

inductive-set evenCard :: 'a set set
where
  {} : evenCard
| [| S : evenCard; x ∉ S; y ∉ S; x ≠ y |] ⇒ S ∪ {x, y} : evenCard

lemma S : evenCard
  refute
oops

inductive-set
  even :: nat set
  and odd :: nat set
where
  0 : even
| n : even ⇒ Suc n : odd
| n : odd ⇒ Suc n : even

lemma n : odd

oops

consts f :: 'a ⇒ 'a

inductive-set
  a-even :: 'a set
  and a-odd :: 'a set
where
  undefined : a-even
| x : a-even ⇒ f x : a-odd
| x : a-odd ⇒ f x : a-even

lemma x : a-odd

oops

46.1.17 Examples involving special functions

lemma card x = 0
  refute
oops

lemma finite x
  refute — no finite countermodel exists
oops

lemma (x::nat) + y = 0
  refute
oops

```

```

lemma (x::nat) = x + x
  refute
oops

lemma (x::nat) - y + y = x
  refute
oops

lemma (x::nat) = x * x
  refute
oops

lemma (x::nat) < x + y
  refute
oops

lemma xs @ [] = ys @ []
  refute
oops

lemma xs @ ys = ys @ xs
  refute
oops

lemma f (lfp f) = lfp f
  refute
oops

lemma f (gfp f) = GFP f
  refute
oops

lemma lfp f = GFP f
  refute
oops

```

46.1.18 Type classes and overloading

A type class without axioms:

```

class classA

lemma P (x::'a::classA)
  refute
oops

```

An axiom with a type variable (denoting types which have at least two elements):

```

class classC =

```

assumes *classC-ax*: $\exists x y. x \neq y$

lemma *P* (*x*::'*a*::*classC*)
refute
oops

lemma $\exists x y. (x::'a::\textit{classC}) \neq y$
refute — no countermodel exists
oops

A type class for which a constant is defined:

class *classD* =
fixes *classD-const* :: '*a* \Rightarrow '*a*
assumes *classD-ax*: *classD-const* (*classD-const* *x*) = *classD-const* *x*

lemma *P* (*x*::'*a*::*classD*)
refute
oops

A type class with multiple superclasses:

class *classE* = *classC* + *classD*

lemma *P* (*x*::'*a*::*classE*)
refute
oops

OFCLASS:

lemma *OFCLASS*('a::*type*, *type-class*)
refute — no countermodel exists
apply *intro-classes*
done

lemma *OFCLASS*('a::*classC*, *type-class*)
refute — no countermodel exists
apply *intro-classes*
done

lemma *OFCLASS*('a::*type*, *classC-class*)
refute
oops

Overloading:

consts *inverse* :: '*a* \Rightarrow '*a*

defs (**overloaded**)
inverse-bool: *inverse* (*b*::*bool*) == $\sim b$
inverse-set : *inverse* (*S*::'*a* *set*) == $-S$
inverse-pair: *inverse* *p* == (*inverse* (*fst* *p*), *inverse* (*snd* *p*))


```

lemma inverse b
  refute
oops

lemma P (inverse (S::'a set))
  refute
oops

lemma P (inverse (p::'a × 'b))
  refute
oops

Structured proofs

lemma x = y
proof cases
  assume x = y
  show ?thesis
  refute
  refute [no-assms]
  refute [no-assms = false]
oops

refute-params [satsolver=auto]

end

```

47 Examples for the 'quickcheck' command

```

theory Quickcheck-Examples
imports Main
begin

```

The 'quickcheck' command allows to find counterexamples by evaluating formulae under an assignment of free variables to random values. In contrast to 'refute', it can deal with inductive datatypes, but cannot handle quantifiers.

47.1 Lists

```

theorem map g (map f xs) = map (g o f) xs
  quickcheck
oops

theorem map g (map f xs) = map (f o g) xs
  quickcheck
oops

```

theorem $rev\ (xs\ @\ ys) = rev\ ys\ @\ rev\ xs$
quickcheck
oops

theorem $rev\ (xs\ @\ ys) = rev\ xs\ @\ rev\ ys$
quickcheck
oops

theorem $rev\ (rev\ xs) = xs$
quickcheck
oops

theorem $rev\ xs = xs$
quickcheck
oops

An example involving functions inside other data structures

primrec $app :: ('a \Rightarrow 'a)\ list \Rightarrow 'a \Rightarrow 'a$ **where**
 $app\ []\ x = x$
 $| app\ (f\ \# fs)\ x = app\ fs\ (f\ x)$

lemma $app\ (fs\ @\ gs)\ x = app\ gs\ (app\ fs\ x)$
quickcheck
by $(induct\ fs\ arbitrary:\ x)\ simp\ all$

lemma $app\ (fs\ @\ gs)\ x = app\ fs\ (app\ gs\ x)$
quickcheck
oops

primrec $occurs :: 'a \Rightarrow 'a\ list \Rightarrow nat$ **where**
 $occurs\ a\ [] = 0$
 $| occurs\ a\ (x\ \# xs) = (if\ (x=a)\ then\ Suc\ (occurs\ a\ xs)\ else\ occurs\ a\ xs)$

primrec $del1 :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $del1\ a\ [] = []$
 $| del1\ a\ (x\ \# xs) = (if\ (x=a)\ then\ xs\ else\ (x\ \# del1\ a\ xs))$

A lemma, you'd think to be true from our experience with delAll

lemma $Suc\ (occurs\ a\ (del1\ a\ xs)) = occurs\ a\ xs$
— Wrong. Precondition needed.
quickcheck
oops

lemma $xs\ \sim =\ [] \longrightarrow Suc\ (occurs\ a\ (del1\ a\ xs)) = occurs\ a\ xs$
quickcheck
— Also wrong.
oops

lemma $0 < occurs\ a\ xs \longrightarrow Suc\ (occurs\ a\ (del1\ a\ xs)) = occurs\ a\ xs$

quickcheck
by (*induct xs*) *auto*

primrec *replace* :: 'a \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a list **where**
replace a b [] = []
| *replace* a b (x#xs) = (if (x=a) then (b#(*replace* a b xs))
else (x#(*replace* a b xs)))

lemma *occurs a xs = occurs b (replace a b xs)*
quickcheck
— Wrong. Precondition needed.
oops

lemma *occurs b xs = 0 \vee a=b \longrightarrow occurs a xs = occurs b (replace a b xs)*
quickcheck
by (*induct xs*) *simp-all*

47.2 Trees

datatype 'a tree = *Twig* | *Leaf* 'a | *Branch* 'a tree 'a tree

primrec *leaves* :: 'a tree \Rightarrow 'a list **where**
leaves *Twig* = []
| *leaves* (*Leaf* a) = [a]
| *leaves* (*Branch* l r) = (*leaves* l) @ (*leaves* r)

primrec *plant* :: 'a list \Rightarrow 'a tree **where**
plant [] = *Twig*
| *plant* (x#xs) = *Branch* (*Leaf* x) (*plant* xs)

primrec *mirror* :: 'a tree \Rightarrow 'a tree **where**
mirror (*Twig*) = *Twig*
| *mirror* (*Leaf* a) = *Leaf* a
| *mirror* (*Branch* l r) = *Branch* (*mirror* r) (*mirror* l)

theorem *plant (rev (leaves xt)) = mirror xt*
quickcheck
— Wrong!
oops

theorem *plant((leaves xt) @ (leaves yt)) = Branch xt yt*
quickcheck
— Wrong!
oops

datatype 'a ntree = *Tip* 'a | *Node* 'a 'a ntree 'a ntree

primrec *inOrder* :: 'a ntree \Rightarrow 'a list **where**
inOrder (*Tip* a) = [a]

```

| inOrder (Node f x y) = (inOrder x)@[f]@(inOrder y)

primrec root :: 'a ntree ⇒ 'a where
  root (Tip a) = a
  | root (Node f x y) = f

theorem hd (inOrder xt) = root xt
quickcheck
  — Wrong!
oops

end

```

48 Preorders with explicit equivalence relation

```

theory Preorder
imports Orderings
begin

class preorder-equiv = preorder
begin

definition equiv :: 'a ⇒ 'a ⇒ bool where
  equiv x y ⟷ x ≤ y ∧ y ≤ x

notation
  equiv (op ~~) and
  equiv ((-/ ~~ -) [51, 51] 50)

notation (xsymbols)
  equiv (op ≈) and
  equiv ((-/ ≈ -) [51, 51] 50)

notation (HTML output)
  equiv (op ≈) and
  equiv ((-/ ≈ -) [51, 51] 50)

lemma refl [iff]:
  x ≈ x
  unfolding equiv-def by simp

lemma trans:
  x ≈ y ⟹ y ≈ z ⟹ x ≈ z
  unfolding equiv-def by (auto intro: order-trans)

lemma antisym:
  x ≤ y ⟹ y ≤ x ⟹ x ≈ y
  unfolding equiv-def ..

```

```

lemma less-le:  $x < y \longleftrightarrow x \leq y \wedge \neg x \approx y$ 
  by (auto simp add: equiv-def less-le-not-le)

lemma le-less:  $x \leq y \longleftrightarrow x < y \vee x \approx y$ 
  by (auto simp add: equiv-def less-le)

lemma le-imp-less-or-eq:  $x \leq y \implies x < y \vee x \approx y$ 
  by (simp add: less-le)

lemma less-imp-not-eq:  $x < y \implies x \approx y \longleftrightarrow \text{False}$ 
  by (simp add: less-le)

lemma less-imp-not-eq2:  $x < y \implies y \approx x \longleftrightarrow \text{False}$ 
  by (simp add: equiv-def less-le)

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

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

lemma antisym-conv:  $y \leq x \implies x \leq y \longleftrightarrow x \approx y$ 
  by (simp add: equiv-def)

end

end

```

49 Comparing growth of functions on natural numbers by a preorder relation

```

theory Landau
imports Main Preorder
begin

```

We establish a preorder relation \lesssim on functions from \mathbb{N} to \mathbb{N} such that $f \lesssim g \longleftrightarrow f \in O(g)$.

49.1 Auxiliary

```

lemma Ex-All-bounded:
  fixes  $n :: \text{nat}$ 
  assumes  $\exists n. \forall m \geq n. P\ m$ 
  obtains  $m$  where  $m \geq n$  and  $P\ m$ 
proof –
  from assms obtain  $q$  where  $m \geq q. \forall m \geq q. P\ m ..$ 

```

let $?m = \max q\ n$
 have $?m \geq n$ **by** *auto*
 moreover from $m-q$ have $P\ ?m$ **by** *auto*
 ultimately show *thesis* ..
qed

49.2 The \lesssim relation

definition *less-eq-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (**infix** \lesssim 50) **where**
 $f \lesssim g \longleftrightarrow (\exists c\ n. \forall m \geq n. f\ m \leq Suc\ c * g\ m)$

lemma *less-eq-fun-intro*:
 assumes $\exists c\ n. \forall m \geq n. f\ m \leq Suc\ c * g\ m$
 shows $f \lesssim g$
unfolding *less-eq-fun-def* **by** (*rule assms*)

lemma *less-eq-fun-not-intro*:
 assumes $\bigwedge c\ n. \exists m \geq n. Suc\ c * g\ m < f\ m$
 shows $\neg f \lesssim g$
using *assms* **unfolding** *less-eq-fun-def linorder-not-le* [*symmetric*]
by *blast*

lemma *less-eq-fun-elim*:
 assumes $f \lesssim g$
 obtains $n\ c$ **where** $\bigwedge m. m \geq n \implies f\ m \leq Suc\ c * g\ m$
using *assms* **unfolding** *less-eq-fun-def* **by** *blast*

lemma *less-eq-fun-not-elim*:
 assumes $\neg f \lesssim g$
 obtains m **where** $m \geq n$ **and** $Suc\ c * g\ m < f\ m$
using *assms* **unfolding** *less-eq-fun-def linorder-not-le* [*symmetric*]
by *blast*

lemma *less-eq-fun-refl*:
 $f \lesssim f$
proof (*rule less-eq-fun-intro*)
 have $\exists n. \forall m \geq n. f\ m \leq Suc\ 0 * f\ m$ **by** *auto*
 then show $\exists c\ n. \forall m \geq n. f\ m \leq Suc\ c * f\ m$ **by** *blast*
qed

lemma *less-eq-fun-trans*:
 assumes $f-g: f \lesssim g$ **and** $g-h: g \lesssim h$
 shows $f-h: f \lesssim h$
proof –
 from $f-g$ obtain $n_1\ c_1$
 where $P1: \bigwedge m. m \geq n_1 \implies f\ m \leq Suc\ c_1 * g\ m$
by (*erule less-eq-fun-elim*)
 moreover from $g-h$ obtain $n_2\ c_2$
 where $P2: \bigwedge m. m \geq n_2 \implies g\ m \leq Suc\ c_2 * h\ m$

by (erule less-eq-fun-elim)
 ultimately have $\bigwedge m. m \geq \max n_1 n_2 \implies f m \leq \text{Suc } c_1 * g m \wedge g m \leq \text{Suc } c_2 * h m$
 by auto
 moreover {
 fix k l r :: nat
 assume k-l: $k \leq \text{Suc } c_1 * l$ and l-r: $l \leq \text{Suc } c_2 * r$
 from l-r have $\text{Suc } c_1 * l \leq (\text{Suc } c_1 * \text{Suc } c_2) * r$
 by (auto simp add: mult-le-cancel-left mult-assoc simp del: times-nat.simps mult-Suc-right)
 with k-l have $k \leq (\text{Suc } c_1 * \text{Suc } c_2) * r$ by (rule preorder-class.order-trans)
 }
 ultimately have $\bigwedge m. m \geq \max n_1 n_2 \implies f m \leq (\text{Suc } c_1 * \text{Suc } c_2) * h m$ by auto
 then have $\bigwedge m. m \geq \max n_1 n_2 \implies f m \leq \text{Suc } ((\text{Suc } c_1 * \text{Suc } c_2) - 1) * h m$
 by auto
 then show ?thesis unfolding less-eq-fun-def by blast
 qed

49.3 The \approx relation, the equivalence relation induced by \lesssim

definition equiv-fun :: $(\text{nat} \Rightarrow \text{nat}) \Rightarrow (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{bool}$ (infix \cong 50) where
 $f \cong g \longleftrightarrow (\exists d c n. \forall m \geq n. g m \leq \text{Suc } d * f m \wedge f m \leq \text{Suc } c * g m)$

lemma equiv-fun-intro:
 assumes $\exists d c n. \forall m \geq n. g m \leq \text{Suc } d * f m \wedge f m \leq \text{Suc } c * g m$
 shows $f \cong g$
 unfolding equiv-fun-def by (rule assms)

lemma equiv-fun-not-intro:
 assumes $\bigwedge d c n. \exists m \geq n. \text{Suc } d * f m < g m \vee \text{Suc } c * g m < f m$
 shows $\neg f \cong g$
 unfolding equiv-fun-def
 by (auto simp add: assms linorder-not-le simp del: times-nat.simps mult-Suc-right)

lemma equiv-fun-elim:
 assumes $f \cong g$
 obtains $n d c$
 where $\bigwedge m. m \geq n \implies g m \leq \text{Suc } d * f m \wedge f m \leq \text{Suc } c * g m$
 using assms unfolding equiv-fun-def by blast

lemma equiv-fun-not-elim:
 fixes $n d c$
 assumes $\neg f \cong g$
 obtains m where $m \geq n$
 and $\text{Suc } d * f m < g m \vee \text{Suc } c * g m < f m$
 using assms unfolding equiv-fun-def
 by (auto simp add: linorder-not-le, blast)

lemma *equiv-fun-less-eq-fun*:

$f \cong g \iff f \lesssim g \wedge g \lesssim f$

proof

assume $x-y$: $f \cong g$

then obtain $n\ d\ c$

where $interv$: $\bigwedge m. m \geq n \implies g\ m \leq Suc\ d * f\ m \wedge f\ m \leq Suc\ c * g\ m$

by (*erule equiv-fun-elim*)

from $interv$ **have** $\exists c\ n. \forall m \geq n. f\ m \leq Suc\ c * g\ m$ **by** *auto*

then have $f-g$: $f \lesssim g$ **by** (*rule less-eq-fun-intro*)

from $interv$ **have** $\exists d\ n. \forall m \geq n. g\ m \leq Suc\ d * f\ m$ **by** *auto*

then have $g-f$: $g \lesssim f$ **by** (*rule less-eq-fun-intro*)

from $f-g\ g-f$ **show** $f \lesssim g \wedge g \lesssim f$ **by** *auto*

next

assume $assm$: $f \lesssim g \wedge g \lesssim f$

from $assm$ *less-eq-fun-elim* **obtain** $c\ n_1$ **where**

$bound1$: $\bigwedge m. m \geq n_1 \implies f\ m \leq Suc\ c * g\ m$

by *blast*

from $assm$ *less-eq-fun-elim* **obtain** $d\ n_2$ **where**

$bound2$: $\bigwedge m. m \geq n_2 \implies g\ m \leq Suc\ d * f\ m$

by *blast*

from $bound2$ **have** $\bigwedge m. m \geq \max\ n_1\ n_2 \implies g\ m \leq Suc\ d * f\ m$

by *auto*

with $bound1$

have $\forall m \geq \max\ n_1\ n_2. g\ m \leq Suc\ d * f\ m \wedge f\ m \leq Suc\ c * g\ m$

by *auto*

then

have $\exists d\ c\ n. \forall m \geq n. g\ m \leq Suc\ d * f\ m \wedge f\ m \leq Suc\ c * g\ m$

by *blast*

then show $f \cong g$ **by** (*rule equiv-fun-intro*)

qed

49.4 The \prec relation, the strict part of \lesssim

definition *less-fun* :: $(nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat) \Rightarrow bool$ (*infix* \prec 50) **where**

$f \prec g \iff f \lesssim g \wedge \neg g \lesssim f$

lemma *less-fun-intro*:

assumes $\bigwedge c. \exists n. \forall m \geq n. Suc\ c * f\ m < g\ m$

shows $f \prec g$

proof (*unfold less-fun-def, rule conjI*)

from $assms$ **obtain** n

where $\forall m \geq n. Suc\ 0 * f\ m < g\ m$..

then have $\forall m \geq n. f\ m \leq Suc\ 0 * g\ m$ **by** *auto*

then have $\exists c\ n. \forall m \geq n. f\ m \leq Suc\ c * g\ m$ **by** *blast*

then show $f \lesssim g$ **by** (*rule less-eq-fun-intro*)

next

show $\neg g \lesssim f$

proof (*rule less-eq-fun-not-intro*)


```

fix c n :: nat
from assms have  $\exists n. \forall m \geq n. \text{Suc } c * f m < g m$  by blast
then obtain m where  $m \geq n$  and  $\text{Suc } c * f m < g m$ 
  by (rule Ex-All-bounded)
then show  $\exists m \geq n. \text{Suc } c * f m < g m$  by blast
qed
qed

```

We would like to show (or refute) that $f \prec g \iff f \in o(g)$, i.e. $(f \prec g) = (\forall c. \exists n. \forall m > n. f m < \text{Suc } c * g m)$ but did not manage to do so.

49.5 Assert that \lesssim is indeed a preorder

```

interpretation fun-order: preorder-equiv less-eq-fun less-fun
  where preorder-equiv.less-eq-fun = equiv-fun
proof -
  interpret preorder-equiv less-eq-fun less-fun proof
  qed (simp-all add: less-fun-def less-eq-fun-refl, auto intro: less-eq-fun-trans)
  show class.preorder-equiv less-eq-fun less-fun using preorder-equiv-axioms .
  show preorder-equiv.less-eq-fun = equiv-fun
    by (simp add: expand-fun-eq equiv-def equiv-fun-less-eq-fun)
qed

```

49.6 Simple examples

```

lemma ( $\lambda \cdot. n$ )  $\lesssim$  ( $\lambda n. n$ )
proof (rule less-eq-fun-intro)
  show  $\exists c q. \forall m \geq q. n \leq \text{Suc } c * m$ 
  proof -
    have  $\forall m \geq n. n \leq \text{Suc } 0 * m$  by simp
    then show ?thesis by blast
  qed
qed

```

```

lemma ( $\lambda n. n$ )  $\cong$  ( $\lambda n. \text{Suc } k * n$ )
proof (rule equiv-fun-intro)
  show  $\exists d c n. \forall m \geq n. \text{Suc } k * m \leq \text{Suc } d * m \wedge m \leq \text{Suc } c * (\text{Suc } k * m)$ 
  proof -
    have  $\forall m \geq n. \text{Suc } k * m \leq \text{Suc } k * m \wedge m \leq \text{Suc } c * (\text{Suc } k * m)$  by simp
    then show ?thesis by blast
  qed
qed

```

```

lemma ( $\lambda \cdot. n$ )  $\prec$  ( $\lambda n. n$ )
proof (rule less-fun-intro)
  fix c
  show  $\exists q. \forall m \geq q. \text{Suc } c * n < m$ 
  proof -
    have  $\forall m \geq \text{Suc } c * n + 1. \text{Suc } c * n < m$  by simp
  qed

```

```

    then show ?thesis by blast
qed
qed
end

```

50 A simple cookbook example how to eliminate choice in programs.

```

theory Execute-Choice
imports Main AssocList
begin

```

A trivial example:

definition *valuesum* :: ('a, 'b :: ab-group-add) mapping \Rightarrow 'b **where**
valuesum m = $(\sum k \in \text{Mapping.keys } m. \text{ the } (\text{Mapping.lookup } m \ k))$

Not that instead of defining *valuesum* with choice, we define it directly and derive a description involving choice afterwards:

lemma *valuesum-rec*:

```

  assumes fin: finite (dom (Mapping.lookup m))
  shows valuesum m = (if Mapping.is-empty m then 0 else
    let l = (SOME l. l ∈ Mapping.keys m) in the (Mapping.lookup m l) + valuesum
    (Mapping.delete l m))
proof (cases Mapping.is-empty m)
  case True then show ?thesis by (simp add: is-empty-def keys-def valuesum-def)
next
  case False
  then have l: ∃ l. l ∈ dom (Mapping.lookup m) by (auto simp add: is-empty-def
    expand-fun-eq mem-def keys-def)
  then have (let l = SOME l. l ∈ dom (Mapping.lookup m) in
    the (Mapping.lookup m l) +  $(\sum k \in \text{dom } (\text{Mapping.lookup } m) - \{l\}. \text{ the } (\text{Mapping.lookup } m \ k))) =$ 
     $(\sum k \in \text{dom } (\text{Mapping.lookup } m). \text{ the } (\text{Mapping.lookup } m \ k))$ )
  proof (rule someI2-ex)
  fix l
  note fin
  moreover assume l ∈ dom (Mapping.lookup m)
  moreover obtain A where A = dom (Mapping.lookup m) - {l} by simp
  ultimately have dom (Mapping.lookup m) = insert l A and finite A and l ∉ A
  by auto
  then show (let l = l
    in the (Mapping.lookup m l) +  $(\sum k \in \text{dom } (\text{Mapping.lookup } m) - \{l\}. \text{ the } (\text{Mapping.lookup } m \ k))) =$ 
     $(\sum k \in \text{dom } (\text{Mapping.lookup } m). \text{ the } (\text{Mapping.lookup } m \ k))$ )
  by simp
qed

```

then show *?thesis* **by** (*simp add: keys-def valuesum-def is-empty-def*)
qed

In the context of the else-branch we can show that the exact choice is irrelevant; in practice, finding this point where choice becomes irrelevant is the most difficult thing!

lemma *valuesum-choice*:

finite (Mapping.keys M) $\implies x \in \text{Mapping.keys } M \implies y \in \text{Mapping.keys } M \implies$
the (Mapping.lookup M x) + valuesum (Mapping.delete x M) =
the (Mapping.lookup M y) + valuesum (Mapping.delete y M)
by (*simp add: valuesum-def keys-def setsum-diff*)

Given *valuesum-rec* as initial description, we stepwise refine it to something executable; first, we formally insert the constructor *AssocList.Mapping* and split the one equation into two, where the second one provides the necessary context:

lemma *valuesum-rec-Mapping*:

shows [*code*]: *valuesum (Mapping []) = 0*
and *valuesum (Mapping (x # xs)) = (let l = (SOME l. l \in Mapping.keys*
(Mapping (x # xs))) in
the (Mapping.lookup (Mapping (x # xs)) l) + valuesum (Mapping.delete l
(Mapping (x # xs))))
by (*simp-all add: valuesum-rec finite-dom-map-of is-empty-Mapping*)

As a side effect the precondition disappears (but note this has nothing to do with choice!). The first equation deals with the uncritical empty case and can already be used for code generation.

Using *valuesum-choice*, we are able to prove an executable version of *valuesum*:

lemma *valuesum-rec-exec* [*code*]:

valuesum (Mapping (x # xs)) = (let l = fst (hd (x # xs)) in
the (Mapping.lookup (Mapping (x # xs)) l) + valuesum (Mapping.delete l
(Mapping (x # xs))))

proof –

let *?M = Mapping (x # xs)*
let *?l1 = (SOME l. l \in Mapping.keys ?M)*
let *?l2 = fst (hd (x # xs))*
have *finite (Mapping.keys ?M)* **by** (*simp add: keys-Mapping*)
moreover have *?l1 \in Mapping.keys ?M*
by (*rule someI*) (*auto simp add: keys-Mapping*)
moreover have *?l2 \in Mapping.keys ?M*
by (*simp add: keys-Mapping*)
ultimately have *the (Mapping.lookup ?M ?l1) + valuesum (Mapping.delete ?l1*
?M) =
the (Mapping.lookup ?M ?l2) + valuesum (Mapping.delete ?l2 ?M)
by (*rule valuesum-choice*)
then show *?thesis* **by** (*simp add: valuesum-rec-Mapping*)

qed

See how it works:

```
value valuesum (Mapping [("abc", (42::int)), ("def", 1705)])
```

end

51 Some basic facts about discrete summation

theory *Summation*

imports *Main*

begin

Auxiliary.

lemma *add-setsum-orient*:

```
  setsum f {k..<j} + setsum f {l..<k} = setsum f {l..<k} + setsum f {k..<j}
  by (fact add commute)
```

lemma *add-setsum-int*:

```
  fixes j k l :: int
  shows j < k  $\implies$  k < l  $\implies$  setsum f {j..<k} + setsum f {k..<l} = setsum f
    {j..<l}
  by (simp-all add: setsum-Un-Int [symmetric] ivl-disj-un)
```

The shift operator.

definition $\Delta :: (int \Rightarrow 'a::ab-group-add) \Rightarrow int \Rightarrow 'a$ **where**

```
   $\Delta$  f k = f (k + 1) - f k
```

lemma *Δ -shift*:

```
   $\Delta$  ( $\lambda k. l + f k$ ) =  $\Delta$  f
  by (simp add:  $\Delta$ -def expand-fun-eq)
```

lemma *Δ -same-shift*:

```
  assumes  $\Delta$  f =  $\Delta$  g
  shows  $\exists l. plus\ l \circ f = g$ 
```

proof -

```
  fix k
```

```
  from assms have  $\bigwedge k. \Delta f k = \Delta g k$  by simp
```

```
  then have k-incr:  $\bigwedge k. f (k + 1) - g (k + 1) = f k - g k$ 
```

```
    by (simp add:  $\Delta$ -def algebra-simps)
```

```
  then have  $\bigwedge k. f ((k - 1) + 1) - g ((k - 1) + 1) = f (k - 1) - g (k - 1)$ 
```

```
    by blast
```

```
  then have k-decr:  $\bigwedge k. f (k - 1) - g (k - 1) = f k - g k$ 
```

```
    by simp
```

```
  have  $\bigwedge k. f k - g k = f 0 - g 0$ 
```

```
  proof -
```

```
    fix k
```

```

  show  $f\ k - g\ k = f\ 0 - g\ 0$ 
  by (induct k rule: int-induct) (simp-all add: k-incr k-decr)
qed
then have  $\bigwedge k. (plus\ (g\ 0 - f\ 0) \circ f)\ k = g\ k$ 
  by (simp add: algebra-simps)
then have  $plus\ (g\ 0 - f\ 0) \circ f = g\ ..$ 
then show ?thesis ..
qed

```

The formal sum operator.

definition $\Sigma :: (int \Rightarrow 'a::ab-group-add) \Rightarrow int \Rightarrow int \Rightarrow 'a$ **where**
 $\Sigma\ f\ j\ l = (if\ j < l\ then\ setsum\ f\ \{j..<l\}$
 $\quad\quad\quad else\ if\ j > l\ then\ -\ setsum\ f\ \{l..<j\}$
 $\quad\quad\quad else\ 0)$

lemma Σ -same [simp]:
 $\Sigma\ f\ j\ j = 0$
by (simp add: Σ -def)

lemma Σ -positive:
 $j < l \implies \Sigma\ f\ j\ l = setsum\ f\ \{j..<l\}$
by (simp add: Σ -def)

lemma Σ -negative:
 $j > l \implies \Sigma\ f\ j\ l = -\ \Sigma\ f\ l\ j$
by (simp add: Σ -def)

lemma add- Σ :
 $\Sigma\ f\ j\ k + \Sigma\ f\ k\ l = \Sigma\ f\ j\ l$
by (simp add: Σ -def algebra-simps add-setsum-int)
(simp-all add: add-setsum-orient [of f k j l]
add-setsum-orient [of f j l k]
add-setsum-orient [of f j k l] add-setsum-int)

lemma Σ -incr-upper:
 $\Sigma\ f\ j\ (l + 1) = \Sigma\ f\ j\ l + f\ l$
proof –
have $\{l..<l+1\} = \{l\}$ **by** auto
then have $\Sigma\ f\ l\ (l + 1) = f\ l$ **by** (simp add: Σ -def)
moreover have $\Sigma\ f\ j\ (l + 1) = \Sigma\ f\ j\ l + \Sigma\ f\ l\ (l + 1)$ **by** (simp add: add- Σ)
ultimately show ?thesis **by** simp
qed

Fundamental lemmas: The relation between Δ and Σ .

lemma Δ - Σ :
 $\Delta\ (\Sigma\ f\ j) = f$
proof
fix k
show $\Delta\ (\Sigma\ f\ j)\ k = f\ k$

```

    by (simp add:  $\Delta$ -def  $\Sigma$ -incr-upper)
qed

lemma  $\Sigma$ - $\Delta$ :
   $\Sigma (\Delta f) j l = f l - f j$ 
proof -
  from  $\Delta$ - $\Sigma$  have  $\Delta (\Sigma (\Delta f) j) = \Delta f$  .
  then obtain  $k$  where  $plus\ k \circ \Sigma (\Delta f) j = f$  by (blast dest:  $\Delta$ -same-shift)
  then have  $\bigwedge q. f\ q = k + \Sigma (\Delta f) j\ q$  by (simp add: expand-fun-eq)
  then show ?thesis by simp
qed

end

```

52 Theory of Integration on real intervals

```

theory Gauge-Integration
imports Complex-Main
begin

```

Attention: This theory defines the Integration on real intervals. This is just a example theory for historical / expository interests. A better replacement is found in the Multivariate Analysis library. This defines the gauge integral on real vector spaces and in the Real Integral theory is a specialization to the integral on arbitrary real intervals. The Multivariate Analysis package also provides a better support for analysis on integrals.

We follow John Harrison in formalizing the Gauge integral.

52.1 Gauges

```

definition
  gauge :: [real set, real => real] => bool where
  [code del]: gauge  $E\ g = (\forall x \in E. 0 < g(x))$ 

```

52.2 Gauge-fine divisions

```

inductive
  fine :: [real => real, real  $\times$  real, (real  $\times$  real  $\times$  real) list] => bool
for
   $\delta :: real \Rightarrow real$ 
where
  fine-Nil:
    fine  $\delta\ (a, a)\ []$ 
| fine-Cons:
   $\llbracket fine\ \delta\ (b, c)\ D; a < b; a \leq x; x \leq b; b - a < \delta\ x \rrbracket$ 
     $\implies fine\ \delta\ (a, c)\ ((a, x, b) \# D)$ 

```

lemmas *fine-induct* [*induct set: fine*] =
fine.induct [*of* δ (*a*,*b*) *D split P, unfolded split-conv, standard*]

lemma *fine-single*:
 $\llbracket a < b; a \leq x; x \leq b; b - a < \delta x \rrbracket \implies \text{fine } \delta \text{ } (a, b) \llbracket (a, x, b) \rrbracket$
by (*rule fine-Cons* [*OF fine-Nil*])

lemma *fine-append*:
 $\llbracket \text{fine } \delta \text{ } (a, b) \text{ } D; \text{fine } \delta \text{ } (b, c) \text{ } D' \rrbracket \implies \text{fine } \delta \text{ } (a, c) \text{ } (D @ D')$
by (*induct set: fine, simp, simp add: fine-Cons*)

lemma *fine-imp-le*: $\text{fine } \delta \text{ } (a, b) \text{ } D \implies a \leq b$
by (*induct set: fine, simp-all*)

lemma *nonempty-fine-imp-less*: $\llbracket \text{fine } \delta \text{ } (a, b) \text{ } D; D \neq [] \rrbracket \implies a < b$
apply (*induct set: fine, simp*)
apply (*drule fine-imp-le, simp*)
done

lemma *fine-Nil-iff*: $\text{fine } \delta \text{ } (a, b) [] \longleftrightarrow a = b$
by (*auto elim: fine.cases intro: fine.intros*)

lemma *fine-same-iff*: $\text{fine } \delta \text{ } (a, a) \text{ } D \longleftrightarrow D = []$
proof
assume $\text{fine } \delta \text{ } (a, a) \text{ } D$ **thus** $D = []$
by (*metis nonempty-fine-imp-less less-irrefl*)
next
assume $D = []$ **thus** $\text{fine } \delta \text{ } (a, a) \text{ } D$
by (*simp add: fine-Nil*)
qed

lemma *empty-fine-imp-eq*: $\llbracket \text{fine } \delta \text{ } (a, b) \text{ } D; D = [] \rrbracket \implies a = b$
by (*simp add: fine-Nil-iff*)

lemma *mem-fine*:
 $\llbracket \text{fine } \delta \text{ } (a, b) \text{ } D; (u, x, v) \in \text{set } D \rrbracket \implies u < v \wedge u \leq x \wedge x \leq v$
by (*induct set: fine, simp, force*)

lemma *mem-fine2*: $\llbracket \text{fine } \delta \text{ } (a, b) \text{ } D; (u, z, v) \in \text{set } D \rrbracket \implies a \leq u \wedge v \leq b$
apply (*induct arbitrary: z u v set: fine, auto*)
apply (*simp add: fine-imp-le*)
apply (*erule order-trans* [*OF less-imp-le*], *simp*)
done

lemma *mem-fine3*: $\llbracket \text{fine } \delta \text{ } (a, b) \text{ } D; (u, z, v) \in \text{set } D \rrbracket \implies v - u < \delta z$
by (*induct arbitrary: z u v set: fine*) *auto*

lemma *BOLZANO*:

```

fixes  $P :: \text{real} \Rightarrow \text{real} \Rightarrow \text{bool}$ 
assumes 1:  $a \leq b$ 
assumes 2:  $\bigwedge a \ b \ c. \llbracket P \ a \ b; P \ b \ c; a \leq b; b \leq c \rrbracket \Longrightarrow P \ a \ c$ 
assumes 3:  $\bigwedge x. \exists d > 0. \forall a \ b. a \leq x \ \& \ x \leq b \ \& \ (b-a) < d \longrightarrow P \ a \ b$ 
shows  $P \ a \ b$ 
apply (subgoal-tac split  $P \ (a,b)$ , simp)
apply (rule lemma-BOLZANO [OF - - 1])
apply (clarify, erule (3) 2)
apply (clarify, rule 3)
done

```

We can always find a division that is fine wrt any gauge

lemma *fine-exists*:

```

assumes  $a \leq b$  and gauge  $\{a..b\} \ \delta$  shows  $\exists D. \text{fine } \delta \ (a, b) \ D$ 
proof -
{
  fix  $u \ v :: \text{real}$  assume  $u \leq v$ 
  have  $a \leq u \Longrightarrow v \leq b \Longrightarrow \exists D. \text{fine } \delta \ (u, v) \ D$ 
  apply (induct  $u \ v$  rule: BOLZANO, rule  $\langle u \leq v \rangle$ )
  apply (simp, fast intro: fine-append)
  apply (case-tac  $a \leq x \wedge x \leq b$ )
  apply (rule-tac  $x=\delta \ x$  in exI)
  apply (rule conjI)
  apply (simp add:  $\langle \text{gauge } \{a..b\} \ \delta \rangle$  [unfolded gauge-def])
  apply (clarify, rename-tac  $u \ v$ )
  apply (case-tac  $u = v$ )
  apply (fast intro: fine-Nil)
  apply (subgoal-tac  $u < v$ , fast intro: fine-single, simp)
  apply (rule-tac  $x=1$  in exI, clarsimp)
  done
}
with  $\langle a \leq b \rangle$  show ?thesis by auto
qed

```

lemma *fine-covers-all*:

```

assumes fine  $\delta \ (a, c) \ D$  and  $a < x$  and  $x \leq c$ 
shows  $\exists N < \text{length } D. \forall d \ t \ e. D ! N = (d,t,e) \longrightarrow d < x \wedge x \leq e$ 
using assms
proof (induct set: fine)
case (2  $b \ c \ D \ a \ t$ )
thus ?case
proof (cases  $b < x$ )
case True
with 2 obtain  $N$  where *:  $N < \text{length } D$ 
and **:  $\bigwedge d \ t \ e. D ! N = (d,t,e) \Longrightarrow d < x \wedge x \leq e$  by auto
hence  $\text{Suc } N < \text{length } ((a,t,b)\#D) \wedge$ 
 $(\forall d \ t' \ e. ((a,t,b)\#D) ! \text{Suc } N = (d,t',e) \longrightarrow d < x \wedge x \leq e)$  by auto
thus ?thesis by auto
next

```



```

    case False with 2
    have  $0 < \text{length } ((a,t,b)\#D) \wedge$ 
       $(\forall d\ t'\ e. ((a,t,b)\#D) !\ 0 = (d,t',e) \longrightarrow d < x \wedge x \leq e)$  by auto
    thus ?thesis by auto
  qed
qed auto

```

lemma *fine-append-split*:

```

  assumes fine  $\delta\ (a,b)\ D$  and  $D2 \neq []$  and  $D = D1\ @\ D2$ 
  shows fine  $\delta\ (a,\text{fst } (\text{hd } D2))\ D1$  (is ?fine1)
  and fine  $\delta\ (\text{fst } (\text{hd } D2),\ b)\ D2$  (is ?fine2)
proof -
  from assms
  have ?fine1  $\wedge$  ?fine2
proof (induct arbitrary:  $D1\ D2$ )
  case (2  $b\ c\ D\ a'\ x\ D1\ D2$ )
  note induct = this

  thus ?case
proof (cases  $D1$ )
  case Nil
  hence  $\text{fst } (\text{hd } D2) = a'$  using 2 by auto
  with fine-Cons[OF  $\langle \text{fine } \delta\ (b,c)\ D \rangle$  induct(3,4,5)] Nil induct
  show ?thesis by (auto intro: fine-Nil)
next
  case (Cons  $d1\ D1'$ )
  with induct(2)[OF  $\langle D2 \neq [] \rangle$ , of  $D1'$ ] induct(8)
  have fine  $\delta\ (b,\text{fst } (\text{hd } D2))\ D1'$  and fine  $\delta\ (\text{fst } (\text{hd } D2),\ c)\ D2$  and
     $d1 = (a',\ x,\ b)$  by auto
  with fine-Cons[OF this(1) induct(3,4,5), OF induct(6)] Cons
  show ?thesis by auto
qed
qed auto
thus ?fine1 and ?fine2 by auto
qed

```

lemma *fine- δ -expand*:

```

  assumes fine  $\delta\ (a,b)\ D$ 
  and  $\bigwedge x. a \leq x \Longrightarrow x \leq b \Longrightarrow \delta\ x \leq \delta'\ x$ 
  shows fine  $\delta'\ (a,b)\ D$ 
using assms proof induct
  case 1 show ?case by (rule fine-Nil)
next
  case (2  $b\ c\ D\ a\ x$ )
  show ?case
proof (rule fine-Cons)
  show fine  $\delta'\ (b,c)\ D$  using 2 by auto
  from fine-imp-le[OF 2(1)] 2(6)  $\langle x \leq b \rangle$ 
  show  $b - a < \delta'\ x$ 

```

```

    using 2(7)[OF ⟨a ≤ x⟩] by auto
  qed (auto simp add: 2)
qed

```

```

lemma fine-single-boundaries:
  assumes fine δ (a,b) D and D = [(d, t, e)]
  shows a = d ∧ b = e
using assms proof induct
  case (2 b c D a x)
  hence D = [] and a = d and b = e by auto
  moreover
  from ⟨fine δ (b,c) D⟩ ⟨D = []⟩ have b = c
    by (rule empty-fine-imp-eq)
  ultimately show ?case by simp
qed auto

```

```

lemma fine-listsum-eq-diff:
  fixes f :: real ⇒ real
  shows fine δ (a, b) D ⇒ (∑ (u, x, v) ← D. f v - f u) = f b - f a
by (induct set: fine) simp-all

```

Lemmas about combining gauges

```

lemma gauge-min:
  [| gauge(E) g1; gauge(E) g2 |]
  ==> gauge(E) (%x. min (g1(x)) (g2(x)))
by (simp add: gauge-def)

```

```

lemma fine-min:
  fine (%x. min (g1(x)) (g2(x))) (a,b) D
  ==> fine(g1) (a,b) D & fine(g2) (a,b) D
apply (erule fine.induct)
apply (simp add: fine-Nil)
apply (simp add: fine-Cons)
done

```

52.3 Riemann sum

```

definition
  rsum :: [(real × real × real) list, real ⇒ real] ⇒ real where
  rsum D f = (∑ (u, x, v) ← D. f x * (v - u))

```

```

lemma rsum-Nil [simp]: rsum [] f = 0
unfolding rsum-def by simp

```

```

lemma rsum-Cons [simp]: rsum ((u, x, v) # D) f = f x * (v - u) + rsum D f
unfolding rsum-def by simp

```

```

lemma rsum-zero [simp]: rsum D (λx. 0) = 0
by (induct D, auto)

```

lemma *rsum-left-distrib*: $rsum\ D\ f * c = rsum\ D\ (\lambda x. f\ x * c)$
by (*induct D*, *auto simp add: algebra-simps*)

lemma *rsum-right-distrib*: $c * rsum\ D\ f = rsum\ D\ (\lambda x. c * f\ x)$
by (*induct D*, *auto simp add: algebra-simps*)

lemma *rsum-add*: $rsum\ D\ (\lambda x. f\ x + g\ x) = rsum\ D\ f + rsum\ D\ g$
by (*induct D*, *auto simp add: algebra-simps*)

lemma *rsum-append*: $rsum\ (D1\ @\ D2)\ f = rsum\ D1\ f + rsum\ D2\ f$
unfolding *rsum-def map-append listsum-append ..*

52.4 Gauge integrability (definite)

definition

Integral :: $[(real*real), real \Rightarrow real, real] \Rightarrow bool$ **where**
 $[code\ del]:\ Integral = (\%(a,b)\ f\ k.\ \forall e > 0.\$
 $\quad (\exists \delta.\ gauge\ \{a..b\}\ \delta\ \&$
 $\quad (\forall D.\ fine\ \delta\ (a,b)\ D\ \longrightarrow$
 $\quad |rsum\ D\ f - k| < e)))$

lemma *Integral-eq*:

$Integral\ (a,\ b)\ f\ k \longleftrightarrow$
 $(\forall e > 0.\ \exists \delta.\ gauge\ \{a..b\}\ \delta \wedge (\forall D.\ fine\ \delta\ (a,b)\ D \longrightarrow |rsum\ D\ f - k| < e))$

unfolding *Integral-def by simp*

lemma *IntegralI*:

assumes $\bigwedge e.\ 0 < e \implies$
 $\quad \exists \delta.\ gauge\ \{a..b\}\ \delta \wedge (\forall D.\ fine\ \delta\ (a,\ b)\ D \longrightarrow |rsum\ D\ f - k| < e)$
shows $Integral\ (a,\ b)\ f\ k$

using *assms unfolding Integral-def by auto*

lemma *IntegralE*:

assumes $Integral\ (a,\ b)\ f\ k$ **and** $0 < e$
obtains δ **where** $gauge\ \{a..b\}\ \delta$ **and** $\forall D.\ fine\ \delta\ (a,\ b)\ D \longrightarrow |rsum\ D\ f - k|$
 $< e$

using *assms unfolding Integral-def by auto*

lemma *Integral-def2*:

$Integral = (\%(a,b)\ f\ k.\ \forall e > 0.\ (\exists \delta.\ gauge\ \{a..b\}\ \delta\ \&$
 $\quad (\forall D.\ fine\ \delta\ (a,b)\ D\ \longrightarrow$
 $\quad |rsum\ D\ f - k| \leq e)))$

unfolding *Integral-def*

apply (*safe intro!: ext*)

apply (*fast intro: less-imp-le*)

apply (*drule-tac x=e/2 in spec*)

apply *force*

done

The integral is unique if it exists

lemma *Integral-unique:*

assumes *le*: $a \leq b$

assumes *1*: *Integral* (*a*, *b*) *f* *k1*

assumes *2*: *Integral* (*a*, *b*) *f* *k2*

shows $k1 = k2$

proof (*rule ccontr*)

assume $k1 \neq k2$

hence $e: 0 < |k1 - k2| / 2$ **by** *simp*

obtain *d1* **where** *gauge* {*a..b*} *d1* **and**

d1: $\forall D. \text{fine } d1 \ (a, b) \ D \longrightarrow |rsum \ D \ f - k1| < |k1 - k2| / 2$

using *1 e* **by** (*rule IntegralE*)

obtain *d2* **where** *gauge* {*a..b*} *d2* **and**

d2: $\forall D. \text{fine } d2 \ (a, b) \ D \longrightarrow |rsum \ D \ f - k2| < |k1 - k2| / 2$

using *2 e* **by** (*rule IntegralE*)

have *gauge* {*a..b*} ($\lambda x. \min (d1 \ x) (d2 \ x)$)

using $\langle \text{gauge } \{a..b\} \ d1 \rangle$ **and** $\langle \text{gauge } \{a..b\} \ d2 \rangle$

by (*rule gauge-min*)

then obtain *D* **where** *fine* ($\lambda x. \min (d1 \ x) (d2 \ x)$) (*a*, *b*) *D*

using *fine-exists* [*OF le*] **by** *fast*

hence *fine* *d1* (*a*, *b*) *D* **and** *fine* *d2* (*a*, *b*) *D*

by (*auto dest: fine-min*)

hence $|rsum \ D \ f - k1| < |k1 - k2| / 2$ **and** $|rsum \ D \ f - k2| < |k1 - k2| / 2$

using *d1 d2* **by** *simp-all*

hence $|rsum \ D \ f - k1| + |rsum \ D \ f - k2| < |k1 - k2| / 2 + |k1 - k2| / 2$

by (*rule add-strict-mono*)

thus *False* **by** *auto*

qed

lemma *Integral-zero: Integral(a,a) f 0*

apply (*rule IntegralI*)

apply (*rule-tac* $x = \lambda x. 1$ **in** *exI*)

apply (*simp add: fine-same-iff gauge-def*)

done

lemma *Integral-same-iff* [*simp*]: *Integral* (*a*, *a*) *f* *k* $\longleftrightarrow k = 0$

by (*auto intro: Integral-zero Integral-unique*)

lemma *Integral-zero-fun: Integral (a,b) ($\lambda x. 0$) 0*

apply (*rule IntegralI*)

apply (*rule-tac* $x = \lambda x. 1$ **in** *exI*, *simp add: gauge-def*)

done

lemma *fine-rsum-const: fine* $\delta \ (a,b) \ D \Longrightarrow rsum \ D \ (\lambda x. c) = (c * (b - a))$

unfolding *rsum-def*

by (*induct set: fine, auto simp add: algebra-simps*)

lemma *Integral-mult-const: $a \leq b \Longrightarrow \text{Integral}(a,b) \ (\lambda x. c) (c * (b - a))$*

apply (*cases* $a = b$, *simp*)

```

apply (rule IntegralI)
apply (rule-tac  $x = \lambda x. b - a$  in exI)
apply (rule conjI, simp add: gauge-def)
apply (clarify)
apply (subst fine-rsum-const, assumption, simp)
done

lemma Integral-eq-diff-bounds:  $a \leq b \implies \text{Integral}(a,b) (\lambda x. 1) (b - a)$ 
  using Integral-mult-const [of a b 1] by simp

lemma Integral-mult:
  [|  $a \leq b$ ; Integral(a,b) f k |] ==> Integral(a,b) (%x. c * f x) (c * k)
apply (auto simp add: order-le-less)
apply (cases  $c = 0$ , simp add: Integral-zero-fun)
apply (rule IntegralI)
apply (erule-tac  $e = e / |c|$  in IntegralE, simp add: divide-pos-pos)
apply (rule-tac  $x = \delta$  in exI, clarify)
apply (erule-tac  $x = D$  in spec, clarify)
apply (simp add: pos-less-divide-eq abs-mult [symmetric]
  algebra-simps rsum-right-distrib)
done

lemma Integral-add:
  assumes Integral (a, b) f x1
  assumes Integral (b, c) f x2
  assumes  $a \leq b$  and  $b \leq c$ 
  shows Integral (a, c) f (x1 + x2)
proof (cases  $a < b \wedge b < c$ , rule IntegralI)
  fix  $\varepsilon :: \text{real}$  assume  $0 < \varepsilon$ 
  hence  $0 < \varepsilon / 2$  by auto

  assume  $a < b \wedge b < c$ 
  hence  $a < b$  and  $b < c$  by auto

  obtain  $\delta 1$  where  $\delta 1\text{-gauge}$ : gauge {a..b}  $\delta 1$ 
  and I1:  $\bigwedge D. \text{fine } \delta 1 (a,b) D \implies | \text{rsum } D f - x1 | < (\varepsilon / 2)$ 
  using IntegralE [OF  $\langle \text{Integral } (a, b) f x1 \rangle \langle 0 < \varepsilon/2 \rangle$ ] by auto

  obtain  $\delta 2$  where  $\delta 2\text{-gauge}$ : gauge {b..c}  $\delta 2$ 
  and I2:  $\bigwedge D. \text{fine } \delta 2 (b,c) D \implies | \text{rsum } D f - x2 | < (\varepsilon / 2)$ 
  using IntegralE [OF  $\langle \text{Integral } (b, c) f x2 \rangle \langle 0 < \varepsilon/2 \rangle$ ] by auto

  def  $\delta \equiv \lambda x. \text{if } x < b \text{ then } \min (\delta 1 x) (b - x)$ 
    else if  $x = b \text{ then } \min (\delta 1 b) (\delta 2 b)$ 
    else  $\min (\delta 2 x) (x - b)$ 

  have gauge {a..c}  $\delta$ 
    using  $\delta 1\text{-gauge}$   $\delta 2\text{-gauge}$  unfolding  $\delta\text{-def}$  gauge-def by auto

```

```

moreover {
  fix  $D :: (\text{real} \times \text{real} \times \text{real}) \text{ list}$ 
  assume  $\text{fine}: \text{fine } \delta (a, c) D$ 
  from  $\text{fine-covers-all}[OF \text{ this } \langle a < b \rangle \langle b \leq c \rangle]$ 
  obtain  $N$  where  $N < \text{length } D$ 
    and  $*$ :  $\forall d \ t \ e. D ! N = (d, t, e) \longrightarrow d < b \wedge b \leq e$ 
    by auto
  obtain  $d \ t \ e$  where  $D\text{-eq}: D ! N = (d, t, e)$  by (cases  $D!N$ , auto)
  with  $*$  have  $d < b$  and  $b \leq e$  by auto
  have  $\text{in-}D: (d, t, e) \in \text{set } D$ 
    using  $D\text{-eq}[\text{symmetric}]$  using  $\langle N < \text{length } D \rangle$  by auto

  from  $\text{mem-fine}[OF \text{ fine in-}D]$ 
  have  $d < e$  and  $d \leq t$  and  $t \leq e$  by auto

  have  $t = b$ 
  proof (rule ccontr)
    assume  $t \neq b$ 
    with  $\text{mem-fine3}[OF \text{ fine in-}D] \langle b \leq e \rangle \langle d \leq t \rangle \langle t \leq e \rangle \langle d < b \rangle \delta\text{-def}$ 
    show False by (cases  $t < b$ ) auto
  qed

  let  $?D1 = \text{take } N \ D$ 
  let  $?D2 = \text{drop } N \ D$ 
  def  $D1 \equiv \text{take } N \ D \ @ \ [(d, t, b)]$ 
  def  $D2 \equiv (\text{if } b = e \text{ then } [] \text{ else } [(b, t, e)]) \ @ \ \text{drop } (\text{Suc } N) \ D$ 

  have  $D \neq []$  using  $\langle N < \text{length } D \rangle$  by auto
  from  $\text{hd-drop-conv-nth}[OF \text{ this } \langle N < \text{length } D \rangle]$ 
  have  $\text{fst } (\text{hd } ?D2) = d$  using  $\langle D ! N = (d, t, e) \rangle$  by auto
  with  $\text{fine-append-split}[OF \text{ - append-take-drop-id}[\text{symmetric}]]$ 
  have  $\text{fine1}: \text{fine } \delta (a, d) \ ?D1$  and  $\text{fine2}: \text{fine } \delta (d, c) \ ?D2$ 
    using  $\langle N < \text{length } D \rangle$   $\text{fine}$  by auto

  have  $\text{fine } \delta 1 (a, b) \ D1$  unfolding  $D1\text{-def}$ 
  proof (rule fine-append)
    show  $\text{fine } \delta 1 (a, d) \ ?D1$ 
    proof (rule fine1[THEN fine- $\delta$ -expand])
      fix  $x$  assume  $a \leq x \ x \leq d$ 
      hence  $x \leq b$  using  $\langle d < b \rangle \langle x \leq d \rangle$  by auto
      thus  $\delta \ x \leq \delta 1 \ x$  unfolding  $\delta\text{-def}$  by auto
    qed

    have  $b - d < \delta 1 \ t$ 
      using  $\text{mem-fine3}[OF \text{ fine in-}D] \delta\text{-def } \langle b \leq e \rangle \langle t = b \rangle$  by auto
    from  $\langle d < b \rangle \langle d \leq t \rangle \langle t = b \rangle$  this
    show  $\text{fine } \delta 1 (d, b) \ [(d, t, b)]$  using fine-single by auto
  qed
  note  $rsum1 = I1[OF \text{ this}]$ 

```

```

have drop-split: drop N D = [D ! N] @ drop (Suc N) D
  using nth-drop'[OF <N < length D>] by simp

have fine2: fine  $\delta 2$  (e,c) (drop (Suc N) D)
proof (cases drop (Suc N) D = [])
  case True
  note * = fine2[simplified drop-split True D-eq append-Nil2]
  have e = c using fine-single-boundaries[OF * refl] by auto
  thus ?thesis unfolding True using fine-Nil by auto
next
  case False
  note * = fine-append-split[OF fine2 False drop-split]
  from fine-single-boundaries[OF *(1)]
  have fst (hd (drop (Suc N) D)) = e using D-eq by auto
  with *(2) have fine  $\delta$  (e,c) (drop (Suc N) D) by auto
  thus ?thesis
  proof (rule fine- $\delta$ -expand)
    fix x assume e  $\leq$  x and x  $\leq$  c
    thus  $\delta$  x  $\leq$   $\delta 2$  x using <b  $\leq$  e> unfolding  $\delta$ -def by auto
  qed
qed

have fine  $\delta 2$  (b, c) D2
proof (cases e = b)
  case True thus ?thesis using fine2 by (simp add: D1-def D2-def)
next
  case False
  have e - b <  $\delta 2$  b
    using mem-fine3[OF fine in-D]  $\delta$ -def <d < b> <t = b> by auto
  with False <t = b> <b  $\leq$  e>
  show ?thesis using D2-def
    by (auto intro!: fine-append[OF - fine2] fine-single
        simp del: append-Cons)
qed
note rsum2 = I2[OF this]

have rsum D f = rsum (take N D) f + rsum [D ! N] f + rsum (drop (Suc
N) D) f
  using rsum-append[symmetric] nth-drop'[OF <N < length D>] by auto
also have ... = rsum D1 f + rsum D2 f
  by (cases b = e, auto simp add: D1-def D2-def D-eq rsum-append algebra-simps)
finally have |rsum D f - (x1 + x2)| <  $\varepsilon$ 
  using add-strict-mono[OF rsum1 rsum2] by simp
}
ultimately show  $\exists \delta. \text{gauge } \{a \dots c\} \delta \wedge$ 
  ( $\forall D. \text{fine } \delta$  (a,c) D  $\longrightarrow$  |rsum D f - (x1 + x2)| <  $\varepsilon$ )
  by blast
next

```

```

case False
hence  $a = b \vee b = c$  using  $\langle a \leq b \rangle$  and  $\langle b \leq c \rangle$  by auto
thus ?thesis
proof (rule disjE)
  assume  $a = b$  hence  $x1 = 0$ 
  using  $\langle \text{Integral } (a, b) f x1 \rangle$  by simp
  thus ?thesis using  $\langle a = b \rangle \langle \text{Integral } (b, c) f x2 \rangle$  by simp
next
  assume  $b = c$  hence  $x2 = 0$ 
  using  $\langle \text{Integral } (b, c) f x2 \rangle$  by simp
  thus ?thesis using  $\langle b = c \rangle \langle \text{Integral } (a, b) f x1 \rangle$  by simp
qed
qed

```

Fundamental theorem of calculus (Part I)

”Straddle Lemma” : Swartz and Thompson: AMM 95(7) 1988

```

lemma strad1:
   $\llbracket \forall z::\text{real}. z \neq x \wedge |z - x| < s \longrightarrow$ 
     $|f z - f x| / (z - x) - f' x| < e/2;$ 
     $0 < s; 0 < e; a \leq x; x \leq b \rrbracket$ 
   $\implies \forall z. |z - x| < s \longrightarrow |f z - f x - f' x * (z - x)| \leq e/2 * |z - x|$ 
apply clarify
apply (case-tac  $z = x$ , simp)
apply (drule-tac  $x = z$  in spec)
apply (rule-tac  $z1 = |inverse (z - x)|$ 
  in real-mult-le-cancel-iff2 [THEN iffD1])
apply simp
apply (simp del: abs-inverse add: abs-mult [symmetric]
  mult-assoc [symmetric])
apply (subgoal-tac  $inverse (z - x) * (f z - f x - f' x * (z - x))$ 
   $= (f z - f x) / (z - x) - f' x$ )
  apply (simp add: abs-mult [symmetric] mult-ac diff-minus)
  apply (subst mult-commute)
  apply (simp add: left-distrib diff-minus)
  apply (simp add: mult-assoc divide-inverse)
  apply (simp add: left-distrib)
done

```

```

lemma lemma-straddle:
  assumes  $f': \forall x. a \leq x \ \& \ x \leq b \longrightarrow \text{DERIV } f x :> f'(x)$  and  $0 < e$ 
  shows  $\exists g. \text{gauge } \{a..b\} \ g \ \&$ 
     $(\forall x \ u \ v. a \leq u \ \& \ u \leq x \ \& \ x \leq v \ \& \ v \leq b \ \& \ (v - u) < g(x)$ 
     $\longrightarrow |(f(v) - f(u)) - (f'(x) * (v - u))| \leq e * (v - u))$ 
proof -
  have  $\forall x \in \{a..b\}. \quad$ 
     $(\exists d > 0. \forall u \ v. u \leq x \ \& \ x \leq v \ \& \ (v - u) < d \longrightarrow$ 
     $|f(v) - f(u)) - (f'(x) * (v - u))| \leq e * (v - u))$ 
  proof (clarisimp)
    fix  $x :: \text{real}$  assume  $a \leq x$  and  $x \leq b$ 

```


with f' **have** $DERIV\ f\ x := f'(x)$ **by** *simp*
then have $\forall r > 0. \exists s > 0. \forall z. z \neq x \wedge |z - x| < s \longrightarrow |(f\ z - f\ x) / (z - x) - f' x| < r$
by (*simp add: DERIV-iff2 LIM-eq*)
with $\langle 0 < e \rangle$ **obtain** s
where $\forall z. z \neq x \wedge |z - x| < s \longrightarrow |(f\ z - f\ x) / (z - x) - f' x| < e/2$ **and**
 $0 < s$
by (*drule-tac x=e/2 in spec, auto*)
then have *strad* [*rule-format*]:
 $\forall z. |z - x| < s \longrightarrow |f\ z - f\ x - f' x * (z - x)| \leq e/2 * |z - x|$
using $\langle 0 < e \rangle \langle a \leq x \rangle \langle x \leq b \rangle$ **by** (*rule strad1*)
show $\exists d > 0. \forall u\ v. u \leq x \wedge x \leq v \wedge v - u < d \longrightarrow |f\ v - f\ u - f' x * (v - u)| \leq e * (v - u)$
proof (*safe intro!: exI*)
show $0 < s$ **by** *fact*
next
fix $u\ v :: real$ **assume** $u \leq x$ **and** $x \leq v$ **and** $v - u < s$
have $|f\ v - f\ u - f' x * (v - u)| =$
 $|(f\ v - f\ x - f' x * (v - x)) + (f\ x - f\ u - f' x * (x - u))|$
by (*simp add: right-diff-distrib*)
also have $\dots \leq |f\ v - f\ x - f' x * (v - x)| + |f\ x - f\ u - f' x * (x - u)|$
by (*rule abs-triangle-ineq*)
also have $\dots = |f\ v - f\ x - f' x * (v - x)| + |f\ u - f\ x - f' x * (u - x)|$
by (*simp add: right-diff-distrib*)
also have $\dots \leq (e/2) * |v - x| + (e/2) * |u - x|$
using $\langle u \leq x \rangle \langle x \leq v \rangle \langle v - u < s \rangle$ **by** (*intro add-mono strad, simp-all*)
also have $\dots \leq e * (v - u) / 2 + e * (v - u) / 2$
using $\langle u \leq x \rangle \langle x \leq v \rangle \langle 0 < e \rangle$ **by** (*intro add-mono, simp-all*)
also have $\dots = e * (v - u)$
by *simp*
finally show $|f\ v - f\ u - f' x * (v - u)| \leq e * (v - u)$.
qed
qed
thus *?thesis*
by (*simp add: gauge-def*) (*drule bchoice, auto*)
qed

lemma *fundamental-theorem-of-calculus:*

assumes $a \leq b$
assumes $f': \forall x. a \leq x \wedge x \leq b \longrightarrow DERIV\ f\ x := f'(x)$
shows $Integral\ (a, b)\ f' (f(b) - f(a))$
proof (*cases a = b*)
assume $a = b$ **thus** *?thesis* **by** *simp*

next

assume $a \neq b$ **with** $\langle a \leq b \rangle$ **have** $a < b$ **by** *simp*
show *?thesis*
proof (*simp add: Integral-def2, clarify*)
fix $e :: real$ **assume** $0 < e$
with $\langle a < b \rangle$ **have** $0 < e / (b - a)$ **by** (*simp add: divide-pos-pos*)

```

from lemma-straddle [OF f' this]
obtain  $\delta$  where gauge {a..b}  $\delta$ 
  and  $\delta$ :  $\bigwedge x u v. \llbracket a \leq u; u \leq x; x \leq v; v \leq b; v - u < \delta x \rrbracket \implies$ 
     $|f v - f u - f' x * (v - u)| \leq e * (v - u) / (b - a)$  by auto

have  $\forall D. \text{fine } \delta (a, b) D \longrightarrow |rsum D f' - (f b - f a)| \leq e$ 
proof (clarify)
  fix D assume D: fine  $\delta (a, b) D$ 
  hence  $(\sum (u, x, v) \leftarrow D. f v - f u) = f b - f a$ 
    by (rule fine-listsum-eq-diff)
  hence  $|rsum D f' - (f b - f a)| = |rsum D f' - (\sum (u, x, v) \leftarrow D. f v - f u)|$ 
    by simp
  also have  $\dots = |(\sum (u, x, v) \leftarrow D. f v - f u) - rsum D f'|$ 
    by (rule abs-minus-commute)
  also have  $\dots = |\sum (u, x, v) \leftarrow D. (f v - f u) - f' x * (v - u)|$ 
    by (simp only: rsum-def listsum-subtractf split-def)
  also have  $\dots \leq (\sum (u, x, v) \leftarrow D. |(f v - f u) - f' x * (v - u)|)$ 
    by (rule ord-le-eq-trans [OF listsum-abs], simp add: o-def split-def)
  also have  $\dots \leq (\sum (u, x, v) \leftarrow D. (e / (b - a)) * (v - u))$ 
    apply (rule listsum-mono, clarify, rename-tac u x v)
    using D apply (simp add:  $\delta$  mem-fine mem-fine2 mem-fine3)
    done
  also have  $\dots = e$ 
    using fine-listsum-eq-diff [OF D, where  $f = \lambda x. x$ ]
    unfolding split-def listsum-const-mult
    using  $\langle a < b \rangle$  by simp
  finally show  $|rsum D f' - (f b - f a)| \leq e$  .
qed

with  $\langle \text{gauge } \{a..b\} \delta \rangle$ 
show  $\exists \delta. \text{gauge } \{a..b\} \delta \wedge (\forall D. \text{fine } \delta (a, b) D \longrightarrow |rsum D f' - (f b - f a)|$ 
 $\leq e)$ 
  by auto
qed
qed
end

```

```

theory Dedekind-Real
imports Rat Lubs
begin

```

53 Positive real numbers

Could be generalized and moved to *Groups*

lemma *add-eq-exists*: $\exists x. a+x = (b::rat)$
by (*rule-tac* $x=b-a$ **in** *exI*, *simp*)

definition

cut :: *rat set* => *bool* **where**
~~[code del]:~~ *cut* $A = (\{\} \subset A \ \& \ A < \{r. 0 < r\} \ \& \ (\forall y \in A. ((\forall z. 0 < z \ \& \ z < y \longrightarrow z \in A) \ \& \ (\exists u \in A. y < u))))$

lemma *interval-empty-iff*:

$\{y. (x::'a::dense-linorder) < y \wedge y < z\} = \{\} \longleftrightarrow \neg x < z$
by (*auto dest: dense*)

lemma *cut-of-rat*:

assumes $q: 0 < q$ **shows** *cut* $\{r::rat. 0 < r \ \& \ r < q\}$ (**is** *cut* $?A$)

proof –

from q **have** $pos: ?A < \{r. 0 < r\}$ **by** *force*

have *nonempty*: $\{\} \subset ?A$

proof

show $\{\} \subseteq ?A$ **by** *simp*

show $\{\} \neq ?A$

by (*force simp only: q eq-commute [of {}] interval-empty-iff*)

qed

show *?thesis*

by (*simp add: cut-def pos nonempty,*
blast dest: dense intro: order-less-trans)

qed

typedef *preal* = $\{A. \text{cut } A\}$

by (*blast intro: cut-of-rat [OF zero-less-one]*)

definition

psup :: *preal set* => *preal* **where**
~~[code del]:~~ *psup* $P = \text{Abs-preal } (\bigcup X \in P. \text{Rep-preal } X)$

definition

add-set :: $[rat \ set, rat \ set] \Rightarrow rat \ set$ **where**
add-set $A \ B = \{w. \exists x \in A. \exists y \in B. w = x + y\}$

definition

diff-set :: $[rat \ set, rat \ set] \Rightarrow rat \ set$ **where**
~~[code del]:~~ *diff-set* $A \ B = \{w. \exists x. 0 < w \ \& \ 0 < x \ \& \ x \notin B \ \& \ x + w \in A\}$

definition

mult-set :: $[rat \ set, rat \ set] \Rightarrow rat \ set$ **where**
mult-set $A \ B = \{w. \exists x \in A. \exists y \in B. w = x * y\}$

definition

inverse-set :: *rat set* => *rat set* **where**
[*code del*]: *inverse-set* *A* = {*x*. $\exists y. 0 < x \ \& \ x < y \ \& \ \text{inverse } y \notin A$ }

instantiation *preal* :: {*ord*, *plus*, *minus*, *times*, *inverse*, *one*}
begin

definition

preal-less-def [*code del*]:
 $R < S == \text{Rep-preal } R < \text{Rep-preal } S$

definition

preal-le-def [*code del*]:
 $R \leq S == \text{Rep-preal } R \subseteq \text{Rep-preal } S$

definition

preal-add-def:
 $R + S == \text{Abs-preal } (\text{add-set } (\text{Rep-preal } R) (\text{Rep-preal } S))$

definition

preal-diff-def:
 $R - S == \text{Abs-preal } (\text{diff-set } (\text{Rep-preal } R) (\text{Rep-preal } S))$

definition

preal-mult-def:
 $R * S == \text{Abs-preal } (\text{mult-set } (\text{Rep-preal } R) (\text{Rep-preal } S))$

definition

preal-inverse-def:
 $\text{inverse } R == \text{Abs-preal } (\text{inverse-set } (\text{Rep-preal } R))$

definition $R / S = R * \text{inverse } (S::\text{preal})$

definition

preal-one-def:
 $1 == \text{Abs-preal } \{x. 0 < x \ \& \ x < 1\}$

instance ..

end

Reduces equality on abstractions to equality on representatives

declare *Abs-preal-inject* [*simp*]
declare *Abs-preal-inverse* [*simp*]

lemma *rat-mem-preal*: $0 < q ==> \{r::\text{rat}. 0 < r \ \& \ r < q\} \in \text{preal}$
by (*simp add: preal-def cut-of-rat*)

lemma *preal-nonempty*: $A \in \text{preal} ==> \exists x \in A. 0 < x$

by (*unfold preal-def cut-def, blast*)

lemma *preal-Ex-mem*: $A \in \text{preal} \implies \exists x. x \in A$
by (*drule preal-nonempty, fast*)

lemma *preal-imp-psubset-positives*: $A \in \text{preal} \implies A < \{r. 0 < r\}$
by (*force simp add: preal-def cut-def*)

lemma *preal-exists-bound*: $A \in \text{preal} \implies \exists x. 0 < x \ \& \ x \notin A$
by (*drule preal-imp-psubset-positives, auto*)

lemma *preal-exists-greater*: $[\![A \in \text{preal}; y \in A \!]\!] \implies \exists u \in A. y < u$
by (*unfold preal-def cut-def, blast*)

lemma *preal-downwards-closed*: $[\![A \in \text{preal}; y \in A; 0 < z; z < y \!]\!] \implies z \in A$
by (*unfold preal-def cut-def, blast*)

Relaxing the final premise

lemma *preal-downwards-closed'*:
 $[\![A \in \text{preal}; y \in A; 0 < z; z \leq y \!]\!] \implies z \in A$
apply (*simp add: order-le-less*)
apply (*blast intro: preal-downwards-closed*)
done

A positive fraction not in a positive real is an upper bound. Gleason p. 122
- Remark (1)

lemma *not-in-preal-ub*:
assumes $A: A \in \text{preal}$
and $\text{not}x: x \notin A$
and $y: y \in A$
and $\text{pos}: 0 < x$
shows $y < x$
proof (*cases rule: linorder-cases*)
assume $x < y$
with $\text{not}x$ **show** *?thesis*
by (*simp add: preal-downwards-closed [OF A y] pos*)
next
assume $x = y$
with $\text{not}x$ **and** y **show** *?thesis* **by** *simp*
next
assume $y < x$
thus *?thesis* .
qed

preal lemmas instantiated to *Rep-preal X*

lemma *mem-Rep-preal-Ex*: $\exists x. x \in \text{Rep-preal } X$
by (*rule preal-Ex-mem [OF Rep-preal]*)

lemma *Rep-preal-exists-bound*: $\exists x > 0. x \notin \text{Rep-preal } X$

by (*rule preal-exists-bound [OF Rep-preal]*)

lemmas *not-in-Rep-preal-ub = not-in-preal-ub [OF Rep-preal]*

53.1 Properties of Ordering

instance *preal* :: *order*

proof

fix *w* :: *preal*

show $w \leq w$ **by** (*simp add: preal-le-def*)

next

fix *i j k* :: *preal*

assume $i \leq j$ **and** $j \leq k$

then show $i \leq k$ **by** (*simp add: preal-le-def*)

next

fix *z w* :: *preal*

assume $z \leq w$ **and** $w \leq z$

then show $z = w$ **by** (*simp add: preal-le-def Rep-preal-inject*)

next

fix *z w* :: *preal*

show $z < w \longleftrightarrow z \leq w \wedge \neg w \leq z$

by (*auto simp add: preal-le-def preal-less-def Rep-preal-inject*)

qed

lemma *preal-imp-pos*: $[|A \in \text{preal}; r \in A|] \implies 0 < r$

by (*insert preal-imp-psubset-positives, blast*)

instance *preal* :: *linorder*

proof

fix *x y* :: *preal*

show $x \leq y \mid y \leq x$

apply (*auto simp add: preal-le-def*)

apply (*rule ccontr*)

apply (*blast dest: not-in-Rep-preal-ub intro: preal-imp-pos [OF Rep-preal]*
elim: order-less-asym)

done

qed

instantiation *preal* :: *distrib-lattice*

begin

definition

(*inf* :: *preal* \Rightarrow *preal* \Rightarrow *preal*) = *min*

definition

(*sup* :: *preal* \Rightarrow *preal* \Rightarrow *preal*) = *max*

instance

by *intro-classes*

(*auto simp add: inf-preal-def sup-preal-def min-max.sup-inf-distrib1*)

end

53.2 Properties of Addition

lemma *preal-add-commute*: $(x::\text{preal}) + y = y + x$
apply (*unfold preal-add-def add-set-def*)
apply (*rule-tac f = Abs-preal in arg-cong*)
apply (*force simp add: add-commute*)
done

Lemmas for proving that addition of two positive reals gives a positive real

Part 1 of Dedekind sections definition

lemma *add-set-not-empty*:
 $[[A \in \text{preal}; B \in \text{preal}]] \implies \{\} \subset \text{add-set } A \ B$
apply (*drule preal-nonempty*) +
apply (*auto simp add: add-set-def*)
done

Part 2 of Dedekind sections definition. A structured version of this proof is *preal-not-mem-mult-set-Ex* below.

lemma *preal-not-mem-add-set-Ex*:
 $[[A \in \text{preal}; B \in \text{preal}]] \implies \exists q > 0. q \notin \text{add-set } A \ B$
apply (*insert preal-exists-bound [of A] preal-exists-bound [of B], auto*)
apply (*rule-tac x = x + xa in exI*)
apply (*simp add: add-set-def, clarify*)
apply (*drule (3) not-in-preal-ub*) +
apply (*force dest: add-strict-mono*)
done

lemma *add-set-not-rat-set*:
assumes *A: A ∈ preal*
and *B: B ∈ preal*
shows *add-set A B < {r. 0 < r}*
proof
from *preal-imp-pos [OF A] preal-imp-pos [OF B]*
show *add-set A B ⊆ {r. 0 < r}* **by** (*force simp add: add-set-def*)
next
show *add-set A B ≠ {r. 0 < r}*
by (*insert preal-not-mem-add-set-Ex [OF A B], blast*)
qed

Part 3 of Dedekind sections definition

lemma *add-set-lemma3*:
 $[[A \in \text{preal}; B \in \text{preal}; u \in \text{add-set } A \ B; 0 < z; z < u]]$
 $\implies z \in \text{add-set } A \ B$
proof (*unfold add-set-def, clarify*)

```

fix  $x::rat$  and  $y::rat$ 
assume  $A: A \in preal$ 
  and  $B: B \in preal$ 
  and  $[simp]: 0 < z$ 
  and  $zless: z < x + y$ 
  and  $x: x \in A$ 
  and  $y: y \in B$ 
have  $xpos [simp]: 0 < x$  by (rule preal-imp-pos [OF A x])
have  $ypos [simp]: 0 < y$  by (rule preal-imp-pos [OF B y])
have  $xypos [simp]: 0 < x + y$  by (simp add: pos-add-strict)
let  $?f = z / (x + y)$ 
have  $fless: ?f < 1$  by (simp add: zless pos-divide-less-eq)
show  $\exists x' \in A. \exists y' \in B. z = x' + y'$ 
proof (intro bexI)
  show  $z = x * ?f + y * ?f$ 
    by (simp add: left-distrib [symmetric] divide-inverse mult-ac
      order-less-imp-not-eq2)
next
  show  $y * ?f \in B$ 
proof (rule preal-downwards-closed [OF B y])
  show  $0 < y * ?f$ 
    by (simp add: divide-inverse zero-less-mult-iff)
next
  show  $y * ?f < y$ 
    by (insert mult-strict-left-mono [OF fless ypos], simp)
qed
next
  show  $x * ?f \in A$ 
proof (rule preal-downwards-closed [OF A x])
  show  $0 < x * ?f$ 
    by (simp add: divide-inverse zero-less-mult-iff)
next
  show  $x * ?f < x$ 
    by (insert mult-strict-left-mono [OF fless xpos], simp)
qed
qed
qed

```

Part 4 of Dedekind sections definition

lemma *add-set-lemma4*:

```

   $[|A \in preal; B \in preal; y \in add-set A B|] ==> \exists u \in add-set A B. y < u$ 
apply (auto simp add: add-set-def)
apply (frule preal-exists-greater [of A], auto)
apply (rule-tac  $x = u + y$  in exI)
apply (auto intro: add-strict-left-mono)
done

```

lemma *mem-add-set*:

```

 $[|A \in preal; B \in preal|] ==> add-set A B \in preal$ 

```



```

apply (simp (no-asm-simp) add: preal-def cut-def)
apply (blast intro!: add-set-not-empty add-set-not-rat-set
      add-set-lemma3 add-set-lemma4)
done

lemma preal-add-assoc:  $((x::\text{preal}) + y) + z = x + (y + z)$ 
apply (simp add: preal-add-def mem-add-set Rep-preal)
apply (force simp add: add-set-def add-ac)
done

instance preal :: ab-semigroup-add
proof
  fix a b c :: preal
  show  $(a + b) + c = a + (b + c)$  by (rule preal-add-assoc)
  show  $a + b = b + a$  by (rule preal-add-commute)
qed

```

53.3 Properties of Multiplication

Proofs essentially same as for addition

```

lemma preal-mult-commute:  $(x::\text{preal}) * y = y * x$ 
apply (unfold preal-mult-def mult-set-def)
apply (rule-tac f = Abs-preal in arg-cong)
apply (force simp add: mult-commute)
done

```

Multiplication of two positive reals gives a positive real.

Lemmas for proving positive reals multiplication set in *preal*

Part 1 of Dedekind sections definition

```

lemma mult-set-not-empty:
   $[A \in \text{preal}; B \in \text{preal}] \implies \{\} \subset \text{mult-set } A \ B$ 
apply (insert preal-nonempty [of A] preal-nonempty [of B])
apply (auto simp add: mult-set-def)
done

```

Part 2 of Dedekind sections definition

```

lemma preal-not-mem-mult-set-Ex:
  assumes A:  $A \in \text{preal}$ 
  and B:  $B \in \text{preal}$ 
  shows  $\exists q. 0 < q \ \& \ q \notin \text{mult-set } A \ B$ 
proof –
  from preal-exists-bound [OF A]
  obtain x where [simp]:  $0 < x \ x \notin A$  by blast
  from preal-exists-bound [OF B]
  obtain y where [simp]:  $0 < y \ y \notin B$  by blast
  show ?thesis
  proof (intro exI conjI)

```

```

show  $0 < x*y$  by (simp add: mult-pos-pos)
show  $x * y \notin \text{mult-set } A \ B$ 
proof –
  { fix  $u::\text{rat}$  and  $v::\text{rat}$ 
    assume  $u \in A$  and  $v \in B$  and  $x*y = u*v$ 
    moreover
    with prems have  $u < x$  and  $v < y$  by (blast dest: not-in-preal-ub) +
    moreover
    with prems have  $0 \leq v$ 
      by (blast intro: preal-imp-pos [OF B] order-less-imp-le prems)
    moreover
    from calculation
      have  $u*v < x*y$  by (blast intro: mult-strict-mono prems)
      ultimately have False by force }
  thus ?thesis by (auto simp add: mult-set-def)
qed
qed
qed

```

```

lemma mult-set-not-rat-set:
  assumes  $A: A \in \text{preal}$ 
  and  $B: B \in \text{preal}$ 
  shows  $\text{mult-set } A \ B < \{r. 0 < r\}$ 
proof
  show  $\text{mult-set } A \ B \subseteq \{r. 0 < r\}$ 
    by (force simp add: mult-set-def
      intro: preal-imp-pos [OF A] preal-imp-pos [OF B] mult-pos-pos)
  show  $\text{mult-set } A \ B \neq \{r. 0 < r\}$ 
    using preal-not-mem-mult-set-Ex [OF A B] by blast
qed

```

Part 3 of Dedekind sections definition

```

lemma mult-set-lemma3:
   $[A \in \text{preal}; B \in \text{preal}; u \in \text{mult-set } A \ B; 0 < z; z < u]$ 
   $\implies z \in \text{mult-set } A \ B$ 
proof (unfold mult-set-def, clarify)
  fix  $x::\text{rat}$  and  $y::\text{rat}$ 
  assume  $A: A \in \text{preal}$ 
  and  $B: B \in \text{preal}$ 
  and [simp]:  $0 < z$ 
  and zless:  $z < x * y$ 
  and  $x: x \in A$ 
  and  $y: y \in B$ 
  have [simp]:  $0 < y$  by (rule preal-imp-pos [OF B y])
  show  $\exists x' \in A. \exists y' \in B. z = x' * y'$ 
proof
  show  $\exists y' \in B. z = (z/y) * y'$ 
proof
  show  $z = (z/y) * y$ 

```

```

      by (simp add: divide-inverse mult-commute [of y] mult-assoc
          order-less-imp-not-eq2)
    show  $y \in B$  by fact
  qed
next
  show  $z/y \in A$ 
  proof (rule preal-downwards-closed [OF A x])
    show  $0 < z/y$ 
    by (simp add: zero-less-divide-iff)
    show  $z/y < x$  by (simp add: pos-divide-less-eq zless)
  qed
qed
qed
qed

```

Part 4 of Dedekind sections definition

```

lemma mult-set-lemma4:
   $[A \in \text{preal}; B \in \text{preal}; y \in \text{mult-set } A \ B] \implies \exists u \in \text{mult-set } A \ B. y < u$ 
  apply (auto simp add: mult-set-def)
  apply (frule preal-exists-greater [of A], auto)
  apply (rule-tac  $x = u * y$  in exI)
  apply (auto intro: preal-imp-pos [of A] preal-imp-pos [of B]
          mult-strict-right-mono)
done

```

```

lemma mem-mult-set:
   $[A \in \text{preal}; B \in \text{preal}] \implies \text{mult-set } A \ B \in \text{preal}$ 
  apply (simp (no-asm-simp) add: preal-def cut-def)
  apply (blast intro!: mult-set-not-empty mult-set-not-rat-set
          mult-set-lemma3 mult-set-lemma4)
done

```

```

lemma preal-mult-assoc:  $((x::\text{preal}) * y) * z = x * (y * z)$ 
  apply (simp add: preal-mult-def mem-mult-set Rep-preal)
  apply (force simp add: mult-set-def mult-ac)
done

```

```

instance preal :: ab-semigroup-mult
proof
  fix  $a \ b \ c :: \text{preal}$ 
  show  $(a * b) * c = a * (b * c)$  by (rule preal-mult-assoc)
  show  $a * b = b * a$  by (rule preal-mult-commute)
qed

```

Positive real 1 is the multiplicative identity element

```

lemma preal-mult-1:  $(1::\text{preal}) * z = z$ 
proof (induct z)
  fix  $A :: \text{rat set}$ 
  assume  $A: A \in \text{preal}$ 

```

```

have {w.  $\exists u. 0 < u \wedge u < 1 \ \& \ (\exists v \in A. w = u * v)$ } = A (is ?lhs = A)
proof
  show ?lhs  $\subseteq$  A
  proof clarify
    fix x::rat and u::rat and v::rat
    assume upos:  $0 < u$  and u<1 and v:  $v \in A$ 
    have vpos:  $0 < v$  by (rule preal-imp-pos [OF A v])
    hence  $u * v < 1 * v$  by (simp only: mult-strict-right-mono prems)
    thus  $u * v \in A$ 
    by (force intro: preal-downwards-closed [OF A v] mult-pos-pos
        upos vpos)
  qed
next
show  $A \subseteq ?lhs$ 
proof clarify
  fix x::rat
  assume x:  $x \in A$ 
  have xpos:  $0 < x$  by (rule preal-imp-pos [OF A x])
  from preal-exists-greater [OF A x]
  obtain v where v:  $v \in A$  and xlessv:  $x < v$  ..
  have vpos:  $0 < v$  by (rule preal-imp-pos [OF A v])
  show  $\exists u. 0 < u \wedge u < 1 \wedge (\exists v \in A. x = u * v)$ 
  proof (intro exI conjI)
    show  $0 < x/v$ 
    by (simp add: zero-less-divide-iff xpos vpos)
    show  $x / v < 1$ 
    by (simp add: pos-divide-less-eq vpos xlessv)
    show  $\exists v' \in A. x = (x / v) * v'$ 
    proof
      show  $x = (x/v) * v$ 
      by (simp add: divide-inverse mult-assoc vpos
          order-less-imp-not-eq2)
      show  $v \in A$  by fact
    qed
  qed
qed
qed
qed
thus  $1 * \text{Abs-preal } A = \text{Abs-preal } A$ 
by (simp add: preal-one-def preal-mult-def mult-set-def
    rat-mem-preal A)
qed

instance preal :: comm-monoid-mult
by intro-classes (rule preal-mult-1)

```

53.4 Distribution of Multiplication across Addition

lemma *mem-Rep-preal-add-iff*:

$$(z \in \text{Rep-preal}(R+S)) = (\exists x \in \text{Rep-preal } R. \exists y \in \text{Rep-preal } S. z = x + y)$$

apply (*simp add: preal-add-def mem-add-set Rep-preal*)
apply (*simp add: add-set-def*)
done

lemma *mem-Rep-preal-mult-iff*:
 $(z \in \text{Rep-preal}(R * S)) = (\exists x \in \text{Rep-preal } R. \exists y \in \text{Rep-preal } S. z = x * y)$
apply (*simp add: preal-mult-def mem-mult-set Rep-preal*)
apply (*simp add: mult-set-def*)
done

lemma *distrib-subset1*:
 $\text{Rep-preal } (w * (x + y)) \subseteq \text{Rep-preal } (w * x + w * y)$
apply (*auto simp add: Bex-def mem-Rep-preal-add-iff mem-Rep-preal-mult-iff*)
apply (*force simp add: right-distrib*)
done

lemma *preal-add-mult-distrib-mean*:
assumes *a*: $a \in \text{Rep-preal } w$
and *b*: $b \in \text{Rep-preal } w$
and *d*: $d \in \text{Rep-preal } x$
and *e*: $e \in \text{Rep-preal } y$
shows $\exists c \in \text{Rep-preal } w. a * d + b * e = c * (d + e)$
proof
let $?c = (a*d + b*e)/(d+e)$
have [*simp*]: $0 < a \ 0 < b \ 0 < d \ 0 < e \ 0 < d+e$
by (*blast intro: preal-imp-pos [OF Rep-preal] a b d e pos-add-strict*)
have *cpos*: $0 < ?c$
by (*simp add: zero-less-divide-iff zero-less-mult-iff pos-add-strict*)
show $a * d + b * e = ?c * (d + e)$
by (*simp add: divide-inverse mult-assoc order-less-imp-not-eq2*)
show $?c \in \text{Rep-preal } w$
proof (*cases rule: linorder-le-cases*)
assume $a \leq b$
hence $?c \leq b$
by (*simp add: pos-divide-le-eq right-distrib mult-right-mono order-less-imp-le*)
thus *thesis* **by** (*rule preal-downwards-closed' [OF Rep-preal b cpos]*)
next
assume $b \leq a$
hence $?c \leq a$
by (*simp add: pos-divide-le-eq right-distrib mult-right-mono order-less-imp-le*)
thus *thesis* **by** (*rule preal-downwards-closed' [OF Rep-preal a cpos]*)
qed
qed

lemma *distrib-subset2*:
 $\text{Rep-preal } (w * x + w * y) \subseteq \text{Rep-preal } (w * (x + y))$
apply (*auto simp add: Bex-def mem-Rep-preal-add-iff mem-Rep-preal-mult-iff*)

apply (*drule-tac* $w=w$ **and** $x=x$ **and** $y=y$ **in** *preal-add-mult-distrib-mean*, *auto*)
done

lemma *preal-add-mult-distrib2*: $(w * ((x::preal) + y)) = (w * x) + (w * y)$
apply (*rule* *Rep-preal-inject* [*THEN iffD1*])
apply (*rule equalityI* [*OF distrib-subset1 distrib-subset2*])
done

lemma *preal-add-mult-distrib*: $((x::preal) + y) * w = (x * w) + (y * w)$
by (*simp add: preal-mult-commute preal-add-mult-distrib2*)

instance *preal* :: *comm-semiring*
by *intro-classes* (*rule preal-add-mult-distrib*)

53.5 Existence of Inverse, a Positive Real

lemma *mem-inv-set-ex*:
assumes $A: A \in preal$ **shows** $\exists x y. 0 < x \ \& \ x < y \ \& \ inverse \ y \notin A$
proof –
from *preal-exists-bound* [*OF A*]
obtain x **where** [*simp*]: $0 < x \ x \notin A$ **by** *blast*
show *?thesis*
proof (*intro exI conjI*)
show $0 < inverse \ (x+1)$
by (*simp add: order-less-trans* [*OF - less-add-one*])
show $inverse(x+1) < inverse \ x$
by (*simp add: less-imp-inverse-less less-add-one*)
show $inverse \ (inverse \ x) \notin A$
by (*simp add: order-less-imp-not-eq2*)
qed
qed

Part 1 of Dedekind sections definition

lemma *inverse-set-not-empty*:
 $A \in preal ==> \{ \} \subset inverse-set \ A$
apply (*insert mem-inv-set-ex* [*of A*])
apply (*auto simp add: inverse-set-def*)
done

Part 2 of Dedekind sections definition

lemma *preal-not-mem-inverse-set-Ex*:
assumes $A: A \in preal$ **shows** $\exists q. 0 < q \ \& \ q \notin inverse-set \ A$
proof –
from *preal-nonempty* [*OF A*]
obtain x **where** $x: x \in A$ **and** $xpos$ [*simp*]: $0 < x \ ..$
show *?thesis*
proof (*intro exI conjI*)
show $0 < inverse \ x$ **by** *simp*
show $inverse \ x \notin inverse-set \ A$

```

proof –
  { fix  $y::\text{rat}$ 
    assume  $ygt: \text{inverse } x < y$ 
    have  $[simp]: 0 < y$  by ( $\text{simp add: order-less-trans } [OF - ygt]$ )
    have  $iyless: \text{inverse } y < x$ 
      by ( $\text{simp add: inverse-less-imp-less } [of x] ygt$ )
    have  $\text{inverse } y \in A$ 
      by ( $\text{simp add: preal-downwards-closed } [OF A x] iyless$ )
    thus  $?thesis$  by ( $\text{auto simp add: inverse-set-def}$ )
  }
qed
qed
qed

lemma inverse-set-not-rat-set:
  assumes  $A: A \in \text{preal}$  shows  $\text{inverse-set } A < \{r. 0 < r\}$ 
proof
  show  $\text{inverse-set } A \subseteq \{r. 0 < r\}$  by ( $\text{force simp add: inverse-set-def}$ )
next
  show  $\text{inverse-set } A \neq \{r. 0 < r\}$ 
    by ( $\text{insert preal-not-mem-inverse-set-Ex } [OF A], \text{blast}$ )
qed

```

Part 3 of Dedekind sections definition

```

lemma inverse-set-lemma3:
   $[A \in \text{preal}; u \in \text{inverse-set } A; 0 < z; z < u]$ 
   $\implies z \in \text{inverse-set } A$ 
apply ( $\text{auto simp add: inverse-set-def}$ )
apply ( $\text{auto intro: order-less-trans}$ )
done

```

Part 4 of Dedekind sections definition

```

lemma inverse-set-lemma4:
   $[A \in \text{preal}; y \in \text{inverse-set } A] \implies \exists u \in \text{inverse-set } A. y < u$ 
apply ( $\text{auto simp add: inverse-set-def}$ )
apply ( $\text{drule dense } [of y]$ )
apply ( $\text{blast intro: order-less-trans}$ )
done

```

```

lemma mem-inverse-set:
   $A \in \text{preal} \implies \text{inverse-set } A \in \text{preal}$ 
apply ( $\text{simp (no-asm-simp) add: preal-def cut-def}$ )
apply ( $\text{blast intro!: inverse-set-not-empty inverse-set-not-rat-set}$ 
   $\text{inverse-set-lemma3 inverse-set-lemma4}$ )
done

```

53.6 Gleason's Lemma 9-3.4, page 122

```

lemma Gleason9-34-exists:

```

```

assumes  $A: A \in \text{preal}$ 
and  $\forall x \in A. x + u \in A$ 
and  $0 \leq z$ 
shows  $\exists b \in A. b + (\text{of-int } z) * u \in A$ 
proof (cases z rule: int-cases)
  case (nonneg n)
    show ?thesis
    proof (simp add: prems, induct n)
      case 0
        from preal-nonempty [OF  $A$ ]
        show ?case by force
      case (Suc k)
        from this obtain  $b$  where  $b \in A$   $b + \text{of-nat } k * u \in A$  ..
        hence  $b + \text{of-int } (\text{int } k) * u + u \in A$  by (simp add: prems)
        thus ?case by (force simp add: algebra-simps prems)
    qed
  next
    case (neg n)
    with prems show ?thesis by simp
qed

lemma Gleason9-34-contr:
  assumes  $A: A \in \text{preal}$ 
  shows  $[|\forall x \in A. x + u \in A; 0 < u; 0 < y; y \notin A|] \implies \text{False}$ 
proof (induct u, induct y)
  fix  $a::\text{int}$  and  $b::\text{int}$ 
  fix  $c::\text{int}$  and  $d::\text{int}$ 
  assume  $bpos$  [simp]:  $0 < b$ 
  and  $dpos$  [simp]:  $0 < d$ 
  and  $closed$ :  $\forall x \in A. x + (\text{Fract } c \ d) \in A$ 
  and  $upos$ :  $0 < \text{Fract } c \ d$ 
  and  $ypos$ :  $0 < \text{Fract } a \ b$ 
  and  $notin$ :  $\text{Fract } a \ b \notin A$ 
  have  $cpos$  [simp]:  $0 < c$ 
  by (simp add: zero-less-Fract-iff [OF  $dpos$ , symmetric]  $upos$ )
  have  $apos$  [simp]:  $0 < a$ 
  by (simp add: zero-less-Fract-iff [OF  $bpos$ , symmetric]  $ypos$ )
  let ? $k = a * d$ 
  have  $frle$ :  $\text{Fract } a \ b \leq \text{Fract } ?k \ 1 * (\text{Fract } c \ d)$ 
  proof –
    have ?thesis =  $((a * d * b * d) \leq c * b * (a * d * b * d))$ 
    by (simp add: order-less-imp-not-eq2 mult-ac)
  moreover
    have  $(1 * (a * d * b * d)) \leq c * b * (a * d * b * d)$ 
    by (rule mult-mono,
      simp-all add: int-one-le-iff-zero-less zero-less-mult-iff
      order-less-imp-le)
  ultimately
  show ?thesis by simp

```



```

qed
have k:  $0 \leq ?k$  by (simp add: order-less-imp-le zero-less-mult-iff)
from Gleason9-34-exists [OF A closed k]
obtain z where z:  $z \in A$ 
      and mem:  $z + \text{of-int } ?k * \text{Fract } c \, d \in A$  ..
have less:  $z + \text{of-int } ?k * \text{Fract } c \, d < \text{Fract } a \, b$ 
  by (rule not-in-preal-ub [OF A notin mem ypos])
have  $0 < z$  by (rule preal-imp-pos [OF A z])
with frle and less show False by (simp add: Fract-of-int-eq)
qed

```

```

lemma Gleason9-34:
  assumes A:  $A \in \text{preal}$ 
    and upos:  $0 < u$ 
  shows  $\exists r \in A. r + u \notin A$ 
proof (rule ccontr, simp)
  assume closed:  $\forall r \in A. r + u \in A$ 
  from preal-exists-bound [OF A]
  obtain y where y:  $y \notin A$  and ypos:  $0 < y$  by blast
  show False
    by (rule Gleason9-34-contr [OF A closed upos ypos y])
qed

```

53.7 Gleason's Lemma 9-3.6

```

lemma lemma-gleason9-36:
  assumes A:  $A \in \text{preal}$ 
    and x:  $1 < x$ 
  shows  $\exists r \in A. r * x \notin A$ 
proof -
  from preal-nonempty [OF A]
  obtain y where y:  $y \in A$  and ypos:  $0 < y$  ..
  show ?thesis
proof (rule classical)
  assume  $\sim(\exists r \in A. r * x \notin A)$ 
  with y have ymem:  $y * x \in A$  by blast
  from ypos mult-strict-left-mono [OF x]
  have yless:  $y < y * x$  by simp
  let ?d =  $y * x - y$ 
  from yless have dpos:  $0 < ?d$  and eq:  $y + ?d = y * x$  by auto
  from Gleason9-34 [OF A dpos]
  obtain r where r:  $r \in A$  and notin:  $r + ?d \notin A$  ..
  have rpos:  $0 < r$  by (rule preal-imp-pos [OF A r])
  with dpos have rdpos:  $0 < r + ?d$  by arith
  have  $\sim(r + ?d \leq y + ?d)$ 
proof
  assume le:  $r + ?d \leq y + ?d$ 
  from ymem have yd:  $y + ?d \in A$  by (simp add: eq)

```

```

    have  $r + ?d \in A$  by (rule preal-downwards-closed' [OF A yd rdpos le])
    with notin show False by simp
qed
hence  $y < r$  by simp
with ypos have dless:  $?d < (r * ?d)/y$ 
  by (simp add: pos-less-divide-eq mult-commute [of ?d]
      mult-strict-right-mono dpos)
have  $r + ?d < r*x$ 
proof -
  have  $r + ?d < r + (r * ?d)/y$  by (simp add: dless)
  also with ypos have  $\dots = (r/y) * (y + ?d)$ 
    by (simp only: algebra-simps divide-inverse, simp)
  also have  $\dots = r*x$  using ypos
    by simp
  finally show  $r + ?d < r*x$  .
qed
with r notin rdpos
show  $\exists r \in A. r * x \notin A$  by (blast dest: preal-downwards-closed [OF A])
qed
qed

```

53.8 Existence of Inverse: Part 2

```

lemma mem-Rep-preal-inverse-iff:
  ( $z \in \text{Rep-preal}(\text{inverse } R)$ ) =
    ( $0 < z \wedge (\exists y. z < y \wedge \text{inverse } y \notin \text{Rep-preal } R)$ )
apply (simp add: preal-inverse-def mem-inverse-set Rep-preal)
apply (simp add: inverse-set-def)
done

```

```

lemma Rep-preal-one:
   $\text{Rep-preal } 1 = \{x. 0 < x \wedge x < 1\}$ 
by (simp add: preal-one-def rat-mem-preal)

```

```

lemma subset-inverse-mult-lemma:
  assumes xpos:  $0 < x$  and xless:  $x < 1$ 
  shows  $\exists r \ u \ y. 0 < r \ \& \ r < y \ \& \ \text{inverse } y \notin \text{Rep-preal } R \ \& \$ 
     $u \in \text{Rep-preal } R \ \& \ x = r * u$ 
proof -
  from xpos and xless have  $1 < \text{inverse } x$  by (simp add: one-less-inverse-iff)
  from lemma-gleason9-36 [OF Rep-preal this]
  obtain r where r:  $r \in \text{Rep-preal } R$ 
    and notin:  $r * (\text{inverse } x) \notin \text{Rep-preal } R$  ..
  have rpos:  $0 < r$  by (rule preal-imp-pos [OF Rep-preal r])
  from preal-exists-greater [OF Rep-preal r]
  obtain u where u:  $u \in \text{Rep-preal } R$  and rless:  $r < u$  ..
  have upos:  $0 < u$  by (rule preal-imp-pos [OF Rep-preal u])
  show ?thesis
proof (intro exI conjI)

```

```

    show  $0 < x/u$  using  $xpos\ upos$ 
      by (simp add: zero-less-divide-iff)
    show  $x/u < x/r$  using  $xpos\ upos\ rpos$ 
      by (simp add: divide-inverse mult-less-cancel-left rless)
    show  $inverse\ (x / r) \notin Rep\text{-}preal\ R$  using  $notin$ 
      by (simp add: divide-inverse mult-commute)
    show  $u \in Rep\text{-}preal\ R$  by (rule  $u$ )
    show  $x = x / u * u$  using  $upos$ 
      by (simp add: divide-inverse mult-commute)
  qed
qed

lemma subset-inverse-mult:
   $Rep\text{-}preal\ 1 \subseteq Rep\text{-}preal\ (inverse\ R * R)$ 
  apply (auto simp add: Bex-def Rep-preal-one mem-Rep-preal-inverse-iff
    mem-Rep-preal-mult-iff)
  apply (blast dest: subset-inverse-mult-lemma)
  done

lemma inverse-mult-subset-lemma:
  assumes  $rpos: 0 < r$ 
  and  $rless: r < y$ 
  and  $notin: inverse\ y \notin Rep\text{-}preal\ R$ 
  and  $q: q \in Rep\text{-}preal\ R$ 
  shows  $r*q < 1$ 
  proof -
    have  $q < inverse\ y$  using  $rpos\ rless$ 
      by (simp add: not-in-preal-ub [OF Rep-preal notin]  $q$ )
    hence  $r * q < r/y$  using  $rpos$ 
      by (simp add: divide-inverse mult-less-cancel-left)
    also have  $\dots \leq 1$  using  $rpos\ rless$ 
      by (simp add: pos-divide-le-eq)
    finally show ?thesis .
  qed

lemma inverse-mult-subset:
   $Rep\text{-}preal\ (inverse\ R * R) \subseteq Rep\text{-}preal\ 1$ 
  apply (auto simp add: Bex-def Rep-preal-one mem-Rep-preal-inverse-iff
    mem-Rep-preal-mult-iff)
  apply (simp add: zero-less-mult-iff preal-imp-pos [OF Rep-preal])
  apply (blast intro: inverse-mult-subset-lemma)
  done

lemma preal-mult-inverse:  $inverse\ R * R = (1::preal)$ 
  apply (rule Rep-preal-inject [THEN iffD1])
  apply (rule equalityI [OF inverse-mult-subset subset-inverse-mult])
  done

lemma preal-mult-inverse-right:  $R * inverse\ R = (1::preal)$ 

```

```

apply (rule preal-mult-commute [THEN subst])
apply (rule preal-mult-inverse)
done

```

Theorems needing *Gleason9-34*

```

lemma Rep-preal-self-subset: Rep-preal (R)  $\subseteq$  Rep-preal(R + S)
proof
  fix r
  assume r: r  $\in$  Rep-preal R
  have rpos: 0 < r by (rule preal-imp-pos [OF Rep-preal r])
  from mem-Rep-preal-Ex
  obtain y where y: y  $\in$  Rep-preal S ..
  have ypos: 0 < y by (rule preal-imp-pos [OF Rep-preal y])
  have ry: r + y  $\in$  Rep-preal(R + S) using r y
    by (auto simp add: mem-Rep-preal-add-iff)
  show r  $\in$  Rep-preal(R + S) using r ypos rpos
    by (simp add: preal-downwards-closed [OF Rep-preal ry])
qed

```

```

lemma Rep-preal-sum-not-subset:  $\sim$  Rep-preal (R + S)  $\subseteq$  Rep-preal(R)
proof -
  from mem-Rep-preal-Ex
  obtain y where y: y  $\in$  Rep-preal S ..
  have ypos: 0 < y by (rule preal-imp-pos [OF Rep-preal y])
  from Gleason9-34 [OF Rep-preal ypos]
  obtain r where r: r  $\in$  Rep-preal R and notin: r + y  $\notin$  Rep-preal R ..
  have r + y  $\in$  Rep-preal (R + S) using r y
    by (auto simp add: mem-Rep-preal-add-iff)
  thus ?thesis using notin by blast
qed

```

```

lemma Rep-preal-sum-not-eq: Rep-preal (R + S)  $\neq$  Rep-preal(R)
by (insert Rep-preal-sum-not-subset, blast)

```

at last, Gleason prop. 9-3.5(iii) page 123

```

lemma preal-self-less-add-left: (R::preal) < R + S
apply (unfold preal-less-def less-le)
apply (simp add: Rep-preal-self-subset Rep-preal-sum-not-eq [THEN not-sym])
done

```

53.9 Subtraction for Positive Reals

Gleason prop. 9-3.5(iv), page 123: proving $A < B \implies \exists D. A + D = B$.
We define the claimed D and show that it is a positive real

Part 1 of Dedekind sections definition

```

lemma diff-set-not-empty:
  R < S  $\implies$   $\{\} \subset \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R)$ 

```

```

apply (auto simp add: preal-less-def diff-set-def elim!: equalityE)
apply (frule-tac  $x1 = S$  in Rep-preal [THEN preal-exists-greater])
apply (drule preal-imp-pos [OF Rep-preal], clarify)
apply (cut-tac  $a=x$  and  $b=u$  in add-eq-exists, force)
done

```

Part 2 of Dedekind sections definition

```

lemma diff-set-nonempty:
   $\exists q. 0 < q \ \& \ q \notin \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R)$ 
apply (cut-tac  $X = S$  in Rep-preal-exists-bound)
apply (erule exE)
apply (rule-tac  $x = x$  in exI, auto)
apply (simp add: diff-set-def)
apply (auto dest: Rep-preal [THEN preal-downwards-closed])
done

```

```

lemma diff-set-not-rat-set:
   $\text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R) < \{r. 0 < r\} \text{ (is ?lhs < ?rhs)}$ 
proof
  show  $?lhs \subseteq ?rhs$  by (auto simp add: diff-set-def)
  show  $?lhs \neq ?rhs$  using diff-set-nonempty by blast
qed

```

Part 3 of Dedekind sections definition

```

lemma diff-set-lemma3:
   $[|R < S; u \in \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R); 0 < z; z < u|]$ 
   $\implies z \in \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R)$ 
apply (auto simp add: diff-set-def)
apply (rule-tac  $x=x$  in exI)
apply (drule Rep-preal [THEN preal-downwards-closed], auto)
done

```

Part 4 of Dedekind sections definition

```

lemma diff-set-lemma4:
   $[|R < S; y \in \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R)|]$ 
   $\implies \exists u \in \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R). y < u$ 
apply (auto simp add: diff-set-def)
apply (drule Rep-preal [THEN preal-exists-greater], clarify)
apply (cut-tac  $a=x+y$  and  $b=u$  in add-eq-exists, clarify)
apply (rule-tac  $x=y+xa$  in exI)
apply (auto simp add: add-ac)
done

```

```

lemma mem-diff-set:
   $R < S \implies \text{diff-set } (\text{Rep-preal } S) (\text{Rep-preal } R) \in \text{preal}$ 
apply (unfold preal-def cut-def)
apply (blast intro!: diff-set-not-empty diff-set-not-rat-set
  diff-set-lemma3 diff-set-lemma4)
done

```

```

lemma mem-Rep-preal-diff-iff:
   $R < S \implies$ 
     $(z \in \text{Rep-preal}(S-R)) =$ 
     $(\exists x. 0 < x \ \& \ 0 < z \ \& \ x \notin \text{Rep-preal } R \ \& \ x + z \in \text{Rep-preal } S)$ 
apply (simp add: preal-diff-def mem-diff-set Rep-preal)
apply (force simp add: diff-set-def)
done

```

proving that $R + D \leq S$

```

lemma less-add-left-lemma:
  assumes Rless:  $R < S$ 
  and a:  $a \in \text{Rep-preal } R$ 
  and cb:  $c + b \in \text{Rep-preal } S$ 
  and  $c \notin \text{Rep-preal } R$ 
  and  $0 < b$ 
  and  $0 < c$ 
  shows  $a + b \in \text{Rep-preal } S$ 
proof –
  have  $0 < a$  by (rule preal-imp-pos [OF Rep-preal a])
  moreover
  have  $a < c$  using prems
    by (blast intro: not-in-Rep-preal-ub)
  ultimately show ?thesis using prems
    by (simp add: preal-downwards-closed [OF Rep-preal cb])
qed

```

```

lemma less-add-left-le1:
   $R < (S::\text{preal}) \implies R + (S-R) \leq S$ 
apply (auto simp add: Bex-def preal-le-def mem-Rep-preal-add-iff
  mem-Rep-preal-diff-iff)
apply (blast intro: less-add-left-lemma)
done

```

53.10 proving that $S \leq R + D$ — trickier

```

lemma lemma-sum-mem-Rep-preal-ex:
   $x \in \text{Rep-preal } S \implies \exists e. 0 < e \ \& \ x + e \in \text{Rep-preal } S$ 
apply (drule Rep-preal [THEN preal-exists-greater], clarify)
apply (cut-tac a=x and b=u in add-eq-exists, auto)
done

```

```

lemma less-add-left-lemma2:
  assumes Rless:  $R < S$ 
  and x:  $x \in \text{Rep-preal } S$ 
  and xnot:  $x \notin \text{Rep-preal } R$ 
  shows  $\exists u \ v \ z. 0 < v \ \& \ 0 < z \ \& \ u \in \text{Rep-preal } R \ \& \ z \notin \text{Rep-preal } R \ \&$ 
   $z + v \in \text{Rep-preal } S \ \& \ x = u + v$ 
proof –

```

```

have xpos:  $0 < x$  by (rule preal-imp-pos [OF Rep-preal x])
from lemma-sum-mem-Rep-preal-ex [OF x]
obtain e where epos:  $0 < e$  and xe:  $x + e \in \text{Rep-preal } S$  by blast
from Gleason9-34 [OF Rep-preal epos]
obtain r where r:  $r \in \text{Rep-preal } R$  and notin:  $r + e \notin \text{Rep-preal } R$  ..
with x xnot xpos have rless:  $r < x$  by (blast intro: not-in-Rep-preal-ub)
from add-eq-exists [of r x]
obtain y where eq:  $x = r + y$  by auto
show ?thesis
proof (intro exI conjI)
  show  $r \in \text{Rep-preal } R$  by (rule r)
  show  $r + e \notin \text{Rep-preal } R$  by (rule notin)
  show  $r + e + y \in \text{Rep-preal } S$  using xe eq by (simp add: add-ac)
  show  $x = r + y$  by (simp add: eq)
  show  $0 < r + e$  using epos preal-imp-pos [OF Rep-preal r]
    by simp
  show  $0 < y$  using rless eq by arith
qed
qed

lemma less-add-left-le2:  $R < (S::\text{preal}) \implies S \leq R + (S - R)$ 
apply (auto simp add: preal-le-def)
apply (case-tac  $x \in \text{Rep-preal } R$ )
apply (cut-tac Rep-preal-self-subset [of R], force)
apply (auto simp add: Bex-def mem-Rep-preal-add-iff mem-Rep-preal-diff-iff)
apply (blast dest: less-add-left-lemma2)
done

lemma less-add-left:  $R < (S::\text{preal}) \implies R + (S - R) = S$ 
by (blast intro: antisym [OF less-add-left-le1 less-add-left-le2])

lemma less-add-left-Ex:  $R < (S::\text{preal}) \implies \exists D. R + D = S$ 
by (fast dest: less-add-left)

lemma preal-add-less2-mono1:  $R < (S::\text{preal}) \implies R + T < S + T$ 
apply (auto dest!: less-add-left-Ex simp add: preal-add-assoc)
apply (rule-tac  $y1 = D$  in preal-add-commute [THEN subst])
apply (auto intro: preal-self-less-add-left simp add: preal-add-assoc [symmetric])
done

lemma preal-add-less2-mono2:  $R < (S::\text{preal}) \implies T + R < T + S$ 
by (auto intro: preal-add-less2-mono1 simp add: preal-add-commute [of T])

lemma preal-add-right-less-cancel:  $R + T < S + T \implies R < (S::\text{preal})$ 
apply (insert linorder-less-linear [of R S], auto)
apply (drule-tac  $R = S$  and  $T = T$  in preal-add-less2-mono1)
apply (blast dest: order-less-trans)
done

```

lemma *preal-add-left-less-cancel*: $T + R < T + S \implies R < (S::preal)$
by (*auto elim: preal-add-right-less-cancel simp add: preal-add-commute [of T]*)

lemma *preal-add-less-cancel-left*: $(T + (R::preal) < T + S) = (R < S)$
by (*blast intro: preal-add-less2-mono2 preal-add-left-less-cancel*)

lemma *preal-add-le-cancel-left*: $(T + (R::preal) \leq T + S) = (R \leq S)$
by (*simp add: linorder-not-less [symmetric] preal-add-less-cancel-left*)

lemma *preal-add-right-cancel*: $(R::preal) + T = S + T \implies R = S$
apply (*insert linorder-less-linear [of R S], safe*)
apply (*drule-tac [!] T = T in preal-add-less2-mono1, auto*)
done

lemma *preal-add-left-cancel*: $C + A = C + B \implies A = (B::preal)$
by (*auto intro: preal-add-right-cancel simp add: preal-add-commute*)

instance *preal :: linordered-cancel-ab-semigroup-add*
proof
 fix *a b c :: preal*
 show $a + b = a + c \implies b = c$ **by** (*rule preal-add-left-cancel*)
 show $a \leq b \implies c + a \leq c + b$ **by** (*simp only: preal-add-le-cancel-left*)
qed

53.11 Completeness of type *preal*

Prove that supremum is a cut

Part 1 of Dedekind sections definition

lemma *preal-sup-set-not-empty*:
 $P \neq \{\} \implies \{\} \subset (\bigcup X \in P. \text{Rep-preal}(X))$
apply *auto*
apply (*cut-tac X = x in mem-Rep-preal-Ex, auto*)
done

Part 2 of Dedekind sections definition

lemma *preal-sup-not-exists*:
 $\forall X \in P. X \leq Y \implies \exists q. 0 < q \ \& \ q \notin (\bigcup X \in P. \text{Rep-preal}(X))$
apply (*cut-tac X = Y in Rep-preal-exists-bound*)
apply (*auto simp add: preal-le-def*)
done

lemma *preal-sup-set-not-rat-set*:
 $\forall X \in P. X \leq Y \implies (\bigcup X \in P. \text{Rep-preal}(X)) < \{r. 0 < r\}$
apply (*drule preal-sup-not-exists*)
apply (*blast intro: preal-imp-pos [OF Rep-preal]*)
done

Part 3 of Dedekind sections definition

lemma *preal-sup-set-lemma3*:

$$[[P \neq \{\}; \forall X \in P. X \leq Y; u \in (\bigcup X \in P. \text{Rep-preal}(X)); 0 < z; z < u]]$$

$$\implies z \in (\bigcup X \in P. \text{Rep-preal}(X))$$

by (*auto elim*: *Rep-preal* [*THEN preal-downwards-closed*])

Part 4 of Dedekind sections definition

lemma *preal-sup-set-lemma4*:

$$[[P \neq \{\}; \forall X \in P. X \leq Y; y \in (\bigcup X \in P. \text{Rep-preal}(X))]]$$

$$\implies \exists u \in (\bigcup X \in P. \text{Rep-preal}(X)). y < u$$

by (*blast dest*: *Rep-preal* [*THEN preal-exists-greater*])

lemma *preal-sup*:

$$[[P \neq \{\}; \forall X \in P. X \leq Y]] \implies (\bigcup X \in P. \text{Rep-preal}(X)) \in \text{preal}$$

apply (*unfold preal-def cut-def*)
apply (*blast intro!*: *preal-sup-set-not-empty preal-sup-set-not-rat-set*
preal-sup-set-lemma3 preal-sup-set-lemma4)
done

lemma *preal-psup-le*:

$$[[\forall X \in P. X \leq Y; x \in P]] \implies x \leq \text{psup } P$$

apply (*simp* (*no-asm-simp*) *add*: *preal-le-def*)
apply (*subgoal-tac* $P \neq \{\}$)
apply (*auto simp add*: *psup-def preal-sup*)
done

lemma *psup-le-ub*: $[[P \neq \{\}; \forall X \in P. X \leq Y]] \implies \text{psup } P \leq Y$
apply (*simp* (*no-asm-simp*) *add*: *preal-le-def*)
apply (*simp add*: *psup-def preal-sup*)
apply (*auto simp add*: *preal-le-def*)
done

Supremum property

lemma *preal-complete*:

$$[[P \neq \{\}; \forall X \in P. X \leq Y]] \implies (\exists X \in P. Z < X) = (Z < \text{psup } P)$$

apply (*simp add*: *preal-less-def psup-def preal-sup*)
apply (*auto simp add*: *preal-le-def*)
apply (*rename-tac* *U*)
apply (*cut-tac* $x = U$ **and** $y = Z$ **in** *linorder-less-linear*)
apply (*auto simp add*: *preal-less-def*)
done

54 Defining the Reals from the Positive Reals

definition

realrel :: $((\text{preal} * \text{preal}) * (\text{preal} * \text{preal})) \text{ set where}$

$$[\text{code del}]: \text{realrel} = \{p. \exists x1\ y1\ x2\ y2. p = ((x1, y1), (x2, y2)) \ \& \ x1 + y2 = x2 + y1\}$$

typedef (*Real*) *real* = *UNIV*//*realrel*

by (*auto simp add: quotient-def*)

definition

real-of-preal :: *preal* \Rightarrow *real* **where**
 $[code\ del]: real-of-preal\ m = Abs-Real\ (realrel\ \{(m + 1, 1)\})$

instantiation *real* :: {*zero, one, plus, minus, uminus, times, inverse, ord, abs, sgn*}
begin

definition

real-zero-def $[code\ del]: 0 = Abs-Real(realrel\ \{(1, 1)\})$

definition

real-one-def $[code\ del]: 1 = Abs-Real(realrel\ \{(1 + 1, 1)\})$

definition

real-add-def $[code\ del]: z + w =$
 $contents\ (\bigcup (x,y) \in Rep-Real(z). \bigcup (u,v) \in Rep-Real(w). \\ \{ Abs-Real(realrel\ \{(x+u, y+v)\}) \})$

definition

real-minus-def $[code\ del]: -\ r = contents\ (\bigcup (x,y) \in Rep-Real(r). \{ Abs-Real(realrel\ \{(y,x)\}) \})$

definition

real-diff-def $[code\ del]: r - (s::real) = r + -\ s$

definition

real-mult-def $[code\ del]:$
 $z * w =$
 $contents\ (\bigcup (x,y) \in Rep-Real(z). \bigcup (u,v) \in Rep-Real(w). \\ \{ Abs-Real(realrel\ \{(x*u + y*v, x*v + y*u)\}) \})$

definition

real-inverse-def $[code\ del]: inverse\ (R::real) = (THE\ S. (R = 0 \ \&\ S = 0) \mid S * R = 1)$

definition

real-divide-def $[code\ del]: R / (S::real) = R * inverse\ S$

definition

real-le-def $[code\ del]: z \leq (w::real) \longleftrightarrow$
 $(\exists\ x\ y\ u\ v. x+v \leq u+y \ \&\ (x,y) \in Rep-Real\ z \ \&\ (u,v) \in Rep-Real\ w)$

definition

real-less-def $[code\ del]: x < (y::real) \longleftrightarrow x \leq y \wedge x \neq y$

definition

real-abs-def: $\text{abs } (r::\text{real}) = (\text{if } r < 0 \text{ then } - r \text{ else } r)$

definition

real-sgn-def: $\text{sgn } (x::\text{real}) = (\text{if } x=0 \text{ then } 0 \text{ else if } 0<x \text{ then } 1 \text{ else } - 1)$

instance ..

end

54.1 Equivalence relation over positive reals

lemma *preal-trans-lemma*:

assumes $x + y1 = x1 + y$

and $x + y2 = x2 + y$

shows $x1 + y2 = x2 + (y1::\text{preal})$

proof –

have $(x1 + y2) + x = (x + y2) + x1$ **by** (*simp add: add-ac*)

also have $\dots = (x2 + y) + x1$ **by** (*simp add: prems*)

also have $\dots = x2 + (x1 + y)$ **by** (*simp add: add-ac*)

also have $\dots = x2 + (x + y1)$ **by** (*simp add: prems*)

also have $\dots = (x2 + y1) + x$ **by** (*simp add: add-ac*)

finally have $(x1 + y2) + x = (x2 + y1) + x$.

thus *?thesis* **by** (*rule add-right-imp-eq*)

qed

lemma *realrel-iff* [*simp*]: $((x1,y1),(x2,y2)) \in \text{realrel} = (x1 + y2 = x2 + y1)$

by (*simp add: realrel-def*)

lemma *equiv-realrel*: *equiv UNIV realrel*

apply (*auto simp add: equiv-def refl-on-def sym-def trans-def realrel-def*)

apply (*blast dest: preal-trans-lemma*)

done

Reduces equality of equivalence classes to the *realrel* relation: $(\text{realrel } \{x\} = \text{realrel } \{y\}) = ((x, y) \in \text{realrel})$

lemmas *equiv-realrel-iff* =

eq-equiv-class-iff [*OF equiv-realrel UNIV-I UNIV-I*]

declare *equiv-realrel-iff* [*simp*]

lemma *realrel-in-real* [*simp*]: $\text{realrel } \{(x,y)\} : \text{Real}$

by (*simp add: Real-def realrel-def quotient-def, blast*)

declare *Abs-Real-inject* [*simp*]

declare *Abs-Real-inverse* [*simp*]

Case analysis on the representation of a real number as an equivalence class

of pairs of positive reals.

```

lemma eq-Abs-Real [case-names Abs-Real, cases type: real]:
  (!!x y. z = Abs-Real(realrel“(x,y)”) ==> P) ==> P
apply (rule Rep-Real [of z, unfolded Real-def, THEN quotientE])
apply (drule arg-cong [where f=Abs-Real])
apply (auto simp add: Rep-Real-inverse)
done

```

54.2 Addition and Subtraction

```

lemma real-add-congruent2-lemma:
  [|a + ba = aa + b; ab + bc = ac + bb|]
  ==> a + ab + (ba + bc) = aa + ac + (b + (bb::preal))
apply (simp add: add-assoc)
apply (rule add-left-commute [of ab, THEN ssubst])
apply (simp add: add-assoc [symmetric])
apply (simp add: add-ac)
done

```

```

lemma real-add:
  Abs-Real (realrel“(x,y)”) + Abs-Real (realrel“(u,v)”) =
  Abs-Real (realrel“(x+u, y+v)”)
proof -
  have (λz w. (λ(x,y). (λ(u,v). {Abs-Real (realrel “{(x+u, y+v)”)}) w) z)
    respects2 realrel
  by (simp add: congruent2-def, blast intro: real-add-congruent2-lemma)
  thus ?thesis
  by (simp add: real-add-def UN-UN-split-split-eq
    UN-equiv-class2 [OF equiv-realrel equiv-realrel])
qed

```

```

lemma real-minus: - Abs-Real(realrel“(x,y)”) = Abs-Real(realrel “{(y,x)”)
proof -
  have (λ(x,y). {Abs-Real (realrel“(y,x)”)}) respects realrel
  by (simp add: congruent-def add-commute)
  thus ?thesis
  by (simp add: real-minus-def UN-equiv-class [OF equiv-realrel])
qed

```

```

instance real :: ab-group-add
proof
  fix x y z :: real
  show (x + y) + z = x + (y + z)
  by (cases x, cases y, cases z, simp add: real-add add-assoc)
  show x + y = y + x
  by (cases x, cases y, simp add: real-add add-commute)
  show 0 + x = x
  by (cases x, simp add: real-add real-zero-def add-ac)
  show - x + x = 0

```

```

    by (cases x, simp add: real-minus real-add real-zero-def add-commute)
  show  $x - y = x + - y$ 
    by (simp add: real-diff-def)
qed

```

54.3 Multiplication

```

lemma real-mult-congruent2-lemma:
  !!( $x1::preal$ ). [|  $x1 + y2 = x2 + y1$  |] ==>
     $x * x1 + y * y1 + (x * y2 + y * x2) =$ 
     $x * x2 + y * y2 + (x * y1 + y * x1)$ 
apply (simp add: add-left-commute add-assoc [symmetric])
apply (simp add: add-assoc right-distrib [symmetric])
apply (simp add: add-commute)
done

lemma real-mult-congruent2:
  (% $p1$   $p2$ .
    (%( $x1,y1$ ). (%( $x2,y2$ ).
      { Abs-Real (realrel "{( $x1*x2 + y1*y2, x1*y2+y1*x2$ )}") } }  $p2$ )  $p1$ )
    respects2 realrel)
apply (rule congruent2-commuteI [OF equiv-realrel], clarify)
apply (simp add: mult-commute add-commute)
apply (auto simp add: real-mult-congruent2-lemma)
done

lemma real-mult:
  Abs-Real((realrel "{( $x1,y1$ )}")) * Abs-Real((realrel "{( $x2,y2$ )}")) =
  Abs-Real(realrel "{( $x1*x2+y1*y2,x1*y2+y1*x2$ )}")
by (simp add: real-mult-def UN-UN-split-split-eq
  UN-equiv-class2 [OF equiv-realrel equiv-realrel real-mult-congruent2])

lemma real-mult-commute: ( $z::real$ ) *  $w = w * z$ 
by (cases  $z$ , cases  $w$ , simp add: real-mult add-ac mult-ac)

lemma real-mult-assoc: (( $z1::real$ ) *  $z2$ ) *  $z3 = z1 * (z2 * z3)$ 
apply (cases  $z1$ , cases  $z2$ , cases  $z3$ )
apply (simp add: real-mult algebra-simps)
done

lemma real-mult-1: ( $1::real$ ) *  $z = z$ 
apply (cases  $z$ )
apply (simp add: real-mult real-one-def algebra-simps)
done

lemma real-add-mult-distrib: (( $z1::real$ ) +  $z2$ ) *  $w = (z1 * w) + (z2 * w)$ 
apply (cases  $z1$ , cases  $z2$ , cases  $w$ )
apply (simp add: real-add real-mult algebra-simps)
done

```

one and zero are distinct

```

lemma real-zero-not-eq-one:  $0 \neq (1::\text{real})$ 
proof -
  have  $(1::\text{preal}) < 1 + 1$ 
    by (simp add: preal-self-less-add-left)
  thus ?thesis
    by (simp add: real-zero-def real-one-def)
qed

```

instance *real :: comm-ring-1*

```

proof
  fix  $x\ y\ z :: \text{real}$ 
  show  $(x * y) * z = x * (y * z)$  by (rule real-mult-assoc)
  show  $x * y = y * x$  by (rule real-mult-commute)
  show  $1 * x = x$  by (rule real-mult-1)
  show  $(x + y) * z = x * z + y * z$  by (rule real-add-mult-distrib)
  show  $0 \neq (1::\text{real})$  by (rule real-zero-not-eq-one)
qed

```

54.4 Inverse and Division

```

lemma real-zero-iff: Abs-Real (realrel “  $\{(x, x)\}$  ”) = 0
by (simp add: real-zero-def add-commute)

```

Instead of using an existential quantifier and constructing the inverse within the proof, we could define the inverse explicitly.

```

lemma real-mult-inverse-left-ex:  $x \neq 0 \implies \exists y. y * x = (1::\text{real})$ 
apply (simp add: real-zero-def real-one-def, cases x)
apply (cut-tac x = xa and y = y in linorder-less-linear)
apply (auto dest!: less-add-left-Ex simp add: real-zero-iff)
apply (rule-tac
   $x = \text{Abs-Real } (\text{realrel} “ \{(1, \text{inverse } (D) + 1)\} ”)$ 
  in exI)
apply (rule-tac [2]
   $x = \text{Abs-Real } (\text{realrel} “ \{(\text{inverse } (D) + 1, 1)\} ”)$ 
  in exI)
apply (auto simp add: real-mult preal-mult-inverse-right algebra-simps)
done

```

```

lemma real-mult-inverse-left:  $x \neq 0 \implies \text{inverse}(x) * x = (1::\text{real})$ 
apply (simp add: real-inverse-def)
apply (drule real-mult-inverse-left-ex, safe)
apply (rule theI, assumption, rename-tac z)
apply (subgoal-tac (z * x) * y = z * (x * y))
apply (simp add: mult-commute)
apply (rule mult-assoc)
done

```

54.5 The Real Numbers form a Field

instance *real* :: *field-inverse-zero*

proof

fix *x y z* :: *real*

show $x \neq 0 \implies \text{inverse } x * x = 1$ **by** (*rule real-mult-inverse-left*)

show $x / y = x * \text{inverse } y$ **by** (*simp add: real-divide-def*)

show $\text{inverse } 0 = (0::\text{real})$ **by** (*simp add: real-inverse-def*)

qed

54.6 The \leq Ordering

lemma *real-le-refl*: $w \leq (w::\text{real})$

by (*cases w, force simp add: real-le-def*)

The arithmetic decision procedure is not set up for type *preal*. This lemma is currently unused, but it could simplify the proofs of the following two lemmas.

lemma *preal-eq-le-imp-le*:

assumes *eq*: $a+b = c+d$ **and** *le*: $c \leq a$

shows $b \leq (d::\text{preal})$

proof –

have $c+d \leq a+d$ **by** (*simp add: prems*)

hence $a+b \leq a+d$ **by** (*simp add: prems*)

thus $b \leq d$ **by** *simp*

qed

lemma *real-le-lemma*:

assumes *l*: $u1 + v2 \leq u2 + v1$

and $x1 + v1 = u1 + y1$

and $x2 + v2 = u2 + y2$

shows $x1 + y2 \leq x2 + (y1::\text{preal})$

proof –

have $(x1+v1) + (u2+y2) = (u1+y1) + (x2+v2)$ **by** (*simp add: prems*)

hence $(x1+y2) + (u2+v1) = (x2+y1) + (u1+v2)$ **by** (*simp add: add-ac*)

also have $\dots \leq (x2+y1) + (u2+v1)$ **by** (*simp add: prems*)

finally show *?thesis* **by** *simp*

qed

lemma *real-le*:

$(\text{Abs-Real}(\text{realrel}^{\{\{x1,y1\}\}}) \leq \text{Abs-Real}(\text{realrel}^{\{\{x2,y2\}\}})) =$

$(x1 + y2 \leq x2 + y1)$

apply (*simp add: real-le-def*)

apply (*auto intro: real-le-lemma*)

done

lemma *real-le-antisym*: $[| z \leq w; w \leq z |] \implies z = (w::\text{real})$

by (*cases z, cases w, simp add: real-le*)

lemma *real-trans-lemma*:

```

    assumes  $x + v \leq u + y$ 
      and  $u + v' \leq u' + v$ 
      and  $x2 + v2 = u2 + y2$ 
    shows  $x + v' \leq u' + (y::\text{real})$ 
  proof -
    have  $(x+v') + (u+v) = (x+v) + (u+v')$  by (simp add: add-ac)
    also have  $\dots \leq (u+y) + (u+v')$  by (simp add: prems)
    also have  $\dots \leq (u+y) + (u'+v)$  by (simp add: prems)
    also have  $\dots = (u'+y) + (u+v)$  by (simp add: add-ac)
    finally show ?thesis by simp
  qed

lemma real-le-trans:  $[[i \leq j; j \leq k]] \implies i \leq (k::\text{real})$ 
  apply (cases i, cases j, cases k)
  apply (simp add: real-le)
  apply (blast intro: real-trans-lemma)
  done

instance real :: order
  proof
    fix  $u v :: \text{real}$ 
    show  $u < v \longleftrightarrow u \leq v \wedge \neg v \leq u$ 
      by (auto simp add: real-less-def intro: real-le-antisym)
  qed (assumption | rule real-le-refl real-le-trans real-le-antisym)+

lemma real-le-linear:  $(z::\text{real}) \leq w \mid w \leq z$ 
  apply (cases z, cases w)
  apply (auto simp add: real-le real-zero-def add-ac)
  done

instance real :: linorder
  by (intro-classes, rule real-le-linear)

lemma real-le-eq-diff:  $(x \leq y) = (x - y \leq (0::\text{real}))$ 
  apply (cases x, cases y)
  apply (auto simp add: real-le real-zero-def real-diff-def real-add real-minus
    add-ac)
  apply (simp-all add: add-assoc [symmetric])
  done

lemma real-add-left-mono:
  assumes  $le: x \leq y$  shows  $z + x \leq z + (y::\text{real})$ 
  proof -
    have  $z + x - (z + y) = (z + -z) + (x - y)$ 
      by (simp add: algebra-simps)
    with le show ?thesis
      by (simp add: real-le-eq-diff[of x] real-le-eq-diff[of z+x] diff-minus)
  qed

```


qed

lemma *real-sum-gt-zero-less*: $(0 < S + (-W::real)) \implies (W < S)$
by (*simp add: linorder-not-le [symmetric] real-le-eq-diff [of S] diff-minus*)

lemma *real-less-sum-gt-zero*: $(W < S) \implies (0 < S + (-W::real))$
by (*simp add: linorder-not-le [symmetric] real-le-eq-diff [of S] diff-minus*)

lemma *real-mult-order*: $[| 0 < x; 0 < y |] \implies (0::real) < x * y$
apply (*cases x, cases y*)
apply (*simp add: linorder-not-le [where 'a = real, symmetric]*
linorder-not-le [where 'a = preal]
real-zero-def real-le real-mult)
— Reduce to the (simpler) \leq relation
apply (*auto dest!: less-add-left-Ex*
simp add: algebra-simps preal-self-less-add-left)
done

lemma *real-mult-less-mono2*: $[| (0::real) < z; x < y |] \implies z * x < z * y$
apply (*rule real-sum-gt-zero-less*)
apply (*drule real-less-sum-gt-zero [of x y]*)
apply (*drule real-mult-order, assumption*)
apply (*simp add: right-distrib*)
done

instantiation *real* :: *distrib-lattice*
begin

definition
(inf :: real \Rightarrow real \Rightarrow real) = min

definition
(sup :: real \Rightarrow real \Rightarrow real) = max

instance
by *default (auto simp add: inf-real-def sup-real-def min-max.sup-inf-distrib1)*

end

54.7 The Reals Form an Ordered Field

instance *real* :: *linordered-field-inverse-zero*
proof

fix *x y z :: real*
show $x \leq y \implies z + x \leq z + y$ **by** (*rule real-add-left-mono*)
show $x < y \implies 0 < z \implies z * x < z * y$ **by** (*rule real-mult-less-mono2*)
show $|x| = (if\ x < 0\ then\ -x\ else\ x)$ **by** (*simp only: real-abs-def*)
show $sgn\ x = (if\ x=0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -1)$
by (*simp only: real-sgn-def*)

qed

The function *real-of-preal* requires many proofs, but it seems to be essential for proving completeness of the reals from that of the positive reals.

lemma *real-of-preal-add*:

$real-of-preal ((x::preal) + y) = real-of-preal x + real-of-preal y$

by (*simp add: real-of-preal-def real-add algebra-simps*)

lemma *real-of-preal-mult*:

$real-of-preal ((x::preal) * y) = real-of-preal x * real-of-preal y$

by (*simp add: real-of-preal-def real-mult algebra-simps*)

Gleason prop 9-4.4 p 127

lemma *real-of-preal-trichotomy*:

$\exists m. (x::real) = real-of-preal m \mid x = 0 \mid x = -(real-of-preal m)$

apply (*simp add: real-of-preal-def real-zero-def, cases x*)

apply (*auto simp add: real-minus add-ac*)

apply (*cut-tac x = x and y = y in linorder-less-linear*)

apply (*auto dest!: less-add-left-Ex simp add: add-assoc [symmetric]*)

done

lemma *real-of-preal-leD*:

$real-of-preal m1 \leq real-of-preal m2 ==> m1 \leq m2$

by (*simp add: real-of-preal-def real-le*)

lemma *real-of-preal-lessI*: $m1 < m2 ==> real-of-preal m1 < real-of-preal m2$

by (*auto simp add: real-of-preal-leD linorder-not-le [symmetric]*)

lemma *real-of-preal-lessD*:

$real-of-preal m1 < real-of-preal m2 ==> m1 < m2$

by (*simp add: real-of-preal-def real-le linorder-not-le [symmetric]*)

lemma *real-of-preal-less-iff* [*simp*]:

$(real-of-preal m1 < real-of-preal m2) = (m1 < m2)$

by (*blast intro: real-of-preal-lessI real-of-preal-lessD*)

lemma *real-of-preal-le-iff*:

$(real-of-preal m1 \leq real-of-preal m2) = (m1 \leq m2)$

by (*simp add: linorder-not-less [symmetric]*)

lemma *real-of-preal-zero-less*: $0 < real-of-preal m$

apply (*insert preal-self-less-add-left [of 1 m]*)

apply (*auto simp add: real-zero-def real-of-preal-def*

$real-less-def real-le-def add-ac$)

apply (*rule-tac x=m + 1 in exI, rule-tac x=1 in exI*)

apply (*simp add: add-ac*)

done

lemma *real-of-preal-minus-less-zero*: $- real-of-preal m < 0$

by (simp add: real-of-preal-zero-less)

lemma *real-of-preal-not-minus-gt-zero*: $\sim 0 < - \text{real-of-preal } m$
proof –
 from *real-of-preal-minus-less-zero*
 show ?thesis by (blast dest: order-less-trans)
qed

54.8 Theorems About the Ordering

lemma *real-gt-zero-preal-Ex*: $(0 < x) = (\exists y. x = \text{real-of-preal } y)$
apply (auto simp add: real-of-preal-zero-less)
apply (cut-tac $x = x$ in *real-of-preal-trichotomy*)
apply (blast elim!: *real-of-preal-not-minus-gt-zero* [THEN notE])
done

lemma *real-gt-preal-preal-Ex*:
 $\text{real-of-preal } z < x \implies \exists y. x = \text{real-of-preal } y$
by (blast dest!: *real-of-preal-zero-less* [THEN order-less-trans]
 intro: *real-gt-zero-preal-Ex* [THEN iffD1])

lemma *real-ge-preal-preal-Ex*:
 $\text{real-of-preal } z \leq x \implies \exists y. x = \text{real-of-preal } y$
by (blast dest: order-le-imp-less-or-eq *real-gt-preal-preal-Ex*)

lemma *real-less-all-preal*: $y \leq 0 \implies \forall x. y < \text{real-of-preal } x$
by (auto elim: order-le-imp-less-or-eq [THEN disjE]
 intro: *real-of-preal-zero-less* [THEN [2] order-less-trans]
 simp add: *real-of-preal-zero-less*)

lemma *real-less-all-real2*: $\sim 0 < y \implies \forall x. y < \text{real-of-preal } x$
by (blast intro!: *real-less-all-preal* linorder-not-less [THEN iffD1])

54.9 Numerals and Arithmetic

instantiation *real* :: *number-ring*
begin

definition
real-number-of-def [code del]: $(\text{number-of } w :: \text{real}) = \text{of-int } w$

instance
 by *intro-classes* (simp add: *real-number-of-def*)

end

54.10 Completeness of Positive Reals

Supremum property for the set of positive reals

Let P be a non-empty set of positive reals, with an upper bound y . Then P has a least upper bound (written S).

FIXME: Can the premise be weakened to $\forall x \in P. x \leq y$?

lemma *posreal-complete*:

assumes *positive-P*: $\forall x \in P. (0::\text{real}) < x$
and *not-empty-P*: $\exists x. x \in P$
and *upper-bound-Ex*: $\exists y. \forall x \in P. x < y$
shows $\exists S. \forall y. (\exists x \in P. y < x) = (y < S)$

proof (*rule exI, rule allI*)

fix y

let $?pP = \{w. \text{real-of-preal } w \in P\}$

show $(\exists x \in P. y < x) = (y < \text{real-of-preal } (\text{psup } ?pP))$

proof (*cases* $0 < y$)

assume *neg-y*: $\neg 0 < y$

show *?thesis*

proof

assume $\exists x \in P. y < x$

have $\forall x. y < \text{real-of-preal } x$

using *neg-y* **by** (*rule real-less-all-real2*)

thus $y < \text{real-of-preal } (\text{psup } ?pP) ..$

next

assume $y < \text{real-of-preal } (\text{psup } ?pP)$

obtain x **where** *x-in-P*: $x \in P$ **using** *not-empty-P* ..

hence $0 < x$ **using** *positive-P* **by** *simp*

hence $y < x$ **using** *neg-y* **by** *simp*

thus $\exists x \in P. y < x$ **using** *x-in-P* ..

qed

next

assume *pos-y*: $0 < y$

then obtain py **where** *y-is-py*: $y = \text{real-of-preal } py$

by (*auto simp add: real-gt-zero-preal-Ex*)

obtain a **where** $a \in P$ **using** *not-empty-P* ..

with *positive-P* **have** *a-pos*: $0 < a$..

then obtain pa **where** $a = \text{real-of-preal } pa$

by (*auto simp add: real-gt-zero-preal-Ex*)

hence $pa \in ?pP$ **using** $\langle a \in P \rangle$ **by** *auto*

hence *pP-not-empty*: $?pP \neq \{\}$ **by** *auto*

obtain sup **where** *sup*: $\forall x \in P. x < sup$

using *upper-bound-Ex* ..

from this and $\langle a \in P \rangle$ **have** $a < sup$..

hence $0 < sup$ **using** *a-pos* **by** *arith*

then obtain $possup$ **where** $sup = \text{real-of-preal } possup$

by (*auto simp add: real-gt-zero-preal-Ex*)

hence $\forall X \in ?pP. X \leq possup$

```

    using sup by (auto simp add: real-of-preal-lessI)
  with pP-not-empty have psup:  $\bigwedge Z. (\exists X \in ?pP. Z < X) = (Z < \text{psup } ?pP)$ 
    by (rule preal-complete)

  show ?thesis
proof
  assume  $\exists x \in P. y < x$ 
  then obtain x where x-in-P:  $x \in P$  and y-less-x:  $y < x$  ..
  hence  $0 < x$  using pos-y by arith
  then obtain px where x-is-px:  $x = \text{real-of-preal } px$ 
    by (auto simp add: real-gt-zero-preal-Ex)

  have py-less-X:  $\exists X \in ?pP. py < X$ 
proof
  show  $py < px$  using y-is-py and x-is-px and y-less-x
    by (simp add: real-of-preal-lessI)
  show  $px \in ?pP$  using x-in-P and x-is-px by simp
qed

  have  $(\exists X \in ?pP. py < X) ==> (py < \text{psup } ?pP)$ 
    using psup by simp
  hence  $py < \text{psup } ?pP$  using py-less-X by simp
  thus  $y < \text{real-of-preal } (\text{psup } \{w. \text{real-of-preal } w \in P\})$ 
    using y-is-py and pos-y by (simp add: real-of-preal-lessI)
next
  assume y-less-psup:  $y < \text{real-of-preal } (\text{psup } ?pP)$ 

  hence  $py < \text{psup } ?pP$  using y-is-py
    by (simp add: real-of-preal-lessI)
  then obtain X where py-less-X:  $py < X$  and X-in-pP:  $X \in ?pP$ 
    using psup by auto
  then obtain x where x-is-X:  $x = \text{real-of-preal } X$ 
    by (simp add: real-gt-zero-preal-Ex)
  hence  $y < x$  using py-less-X and y-is-py
    by (simp add: real-of-preal-lessI)

  moreover have  $x \in P$  using x-is-X and X-in-pP by simp

  ultimately show  $\exists x \in P. y < x$  ..
qed
qed
qed

```

Completeness properties using *isUb*, *isLub* etc.

lemma *posreals-complete*:

```

assumes positive-S:  $\forall x \in S. 0 < x$ 
  and not-empty-S:  $\exists x. x \in S$ 
  and upper-bound-Ex:  $\exists u. \text{isUb } (\text{UNIV}::\text{real set}) S u$ 
shows  $\exists t. \text{isLub } (\text{UNIV}::\text{real set}) S t$ 

```

proof

let $?pS = \{w. \text{real-of-preal } w \in S\}$

obtain u **where** $isUb \ UNIV \ S \ u$ **using** $upper-bound-Ex$ **..**

hence $sup: \forall x \in S. x \leq u$ **by** $(simp \ add: \ isUb-def \ settle-def)$

obtain x **where** $x-in-S: x \in S$ **using** $not-empty-S$ **..**

hence $x-gt-zero: 0 < x$ **using** $positive-S$ **by** $simp$

have $x \leq u$ **using** sup **and** $x-in-S$ **..**

hence $0 < u$ **using** $x-gt-zero$ **by** $arith$

then obtain pu **where** $u-is-pu: u = \text{real-of-preal } pu$

by $(auto \ simp \ add: \ real-gt-zero-preal-Ex)$

have $pS-less-pu: \forall pa \in ?pS. pa \leq pu$

proof

fix pa

assume $pa \in ?pS$

then obtain a **where** $a \in S$ **and** $a = \text{real-of-preal } pa$

by $simp$

moreover hence $a \leq u$ **using** sup **by** $simp$

ultimately show $pa \leq pu$

using sup **and** $u-is-pu$ **by** $(simp \ add: \ real-of-preal-le-iff)$

qed

have $\forall y \in S. y \leq \text{real-of-preal } (psup \ ?pS)$

proof

fix y

assume $y-in-S: y \in S$

hence $0 < y$ **using** $positive-S$ **by** $simp$

then obtain py **where** $y-is-py: y = \text{real-of-preal } py$

by $(auto \ simp \ add: \ real-gt-zero-preal-Ex)$

hence $py-in-pS: py \in ?pS$ **using** $y-in-S$ **by** $simp$

with $pS-less-pu$ **have** $py \leq psup \ ?pS$

by $(rule \ preal-psup-le)$

thus $y \leq \text{real-of-preal } (psup \ ?pS)$

using $y-is-py$ **by** $(simp \ add: \ real-of-preal-le-iff)$

qed

moreover {

fix x

assume $x-ub-S: \forall y \in S. y \leq x$

have $\text{real-of-preal } (psup \ ?pS) \leq x$

proof –

obtain s **where** $s-in-S: s \in S$ **using** $not-empty-S$ **..**

hence $s-pos: 0 < s$ **using** $positive-S$ **by** $simp$

hence $\exists \ ps. s = \text{real-of-preal } ps$ **by** $(simp \ add: \ real-gt-zero-preal-Ex)$

then obtain ps **where** $s-is-ps: s = \text{real-of-preal } ps$ **..**

hence $ps\text{-in-}pS$: $ps \in \{w. \text{real-of-preal } w \in S\}$ **using** $s\text{-in-}S$ **by** *simp*
 from $x\text{-ub-}S$ **have** $s \leq x$ **using** $s\text{-in-}S$..
 hence $0 < x$ **using** $s\text{-pos}$ **by** *simp*
 hence $\exists px. x = \text{real-of-preal } px$ **by** (*simp add: real-gt-zero-preal-Ex*)
 then **obtain** px **where** $x\text{-is-}px$: $x = \text{real-of-preal } px$..

 have $\forall pe \in ?pS. pe \leq px$
proof
 fix pe
 assume $pe \in ?pS$
 hence $\text{real-of-preal } pe \in S$ **by** *simp*
 hence $\text{real-of-preal } pe \leq x$ **using** $x\text{-ub-}S$ **by** *simp*
 thus $pe \leq px$ **using** $x\text{-is-}px$ **by** (*simp add: real-of-preal-le-iff*)
qed

 moreover **have** $?pS \neq \{\}$ **using** $ps\text{-in-}pS$ **by** *auto*
 ultimately **have** $(psup ?pS) \leq px$ **by** (*simp add: psup-le-ub*)
 thus $\text{real-of-preal } (psup ?pS) \leq x$ **using** $x\text{-is-}px$ **by** (*simp add: real-of-preal-le-iff*)
qed
 }
 ultimately **show** $\text{isLub } UNIV \ S \ (\text{real-of-preal } (psup ?pS))$
 by (*simp add: isLub-def leastP-def isUb-def settle-def setge-def*)
qed

reals Completeness (again!)

lemma *reals-complete*:
assumes $\text{notempty-}S$: $\exists X. X \in S$
 and exists-Ub : $\exists Y. \text{isUb } (UNIV::\text{real set}) \ S \ Y$
shows $\exists t. \text{isLub } (UNIV::\text{real set}) \ S \ t$
proof –
obtain X **where** $X\text{-in-}S$: $X \in S$ **using** $\text{notempty-}S$..
obtain Y **where** $Y\text{-isUb}$: $\text{isUb } (UNIV::\text{real set}) \ S \ Y$
 using exists-Ub ..
let $?SHIFT = \{z. \exists x \in S. z = x + (-X) + 1\} \cap \{x. 0 < x\}$

 {
 fix x
 assume $\text{isUb } (UNIV::\text{real set}) \ S \ x$
 hence $S\text{-le-}x$: $\forall y \in S. y \leq x$
 by (*simp add: isUb-def settle-def*)
 {
 fix s
 assume $s \in \{z. \exists x \in S. z = x + -X + 1\}$
 hence $\exists x \in S. s = x + -X + 1$..
 then obtain $x1$ **where** $x1 \in S$ **and** $s = x1 + (-X) + 1$..
 moreover hence $x1 \leq x$ **using** $S\text{-le-}x$ **by** *simp*
 ultimately **have** $s \leq x + -X + 1$ **by** *arith*
 }
 }

then have $isUb\ (UNIV::real\ set)\ ?SHIFT\ (x + (-X) + 1)$
 by $(auto\ simp\ add:\ isUb-def\ settle-def)$
 } note $S-Ub-is-SHIFT-Ub = this$

hence $isUb\ UNIV\ ?SHIFT\ (Y + (-X) + 1)$ using $Y-isUb$ by $simp$
 hence $\exists Z. isUb\ UNIV\ ?SHIFT\ Z ..$
 moreover have $\forall y \in ?SHIFT. 0 < y$ by $auto$
 moreover have $shifted-not-empty: \exists u. u \in ?SHIFT$
 using $X-in-S$ and $Y-isUb$ by $auto$
 ultimately obtain t where $t-is-Lub: isLub\ UNIV\ ?SHIFT\ t$
 using $posreals-complete\ [of\ ?SHIFT]$ by $blast$

show $?thesis$

proof

show $isLub\ UNIV\ S\ (t + X + (-1))$
 proof $(rule\ isLubI2)$
 {
 fix x
 assume $isUb\ (UNIV::real\ set)\ S\ x$
 hence $isUb\ (UNIV::real\ set)\ (?SHIFT)\ (x + (-X) + 1)$
 using $S-Ub-is-SHIFT-Ub$ by $simp$
 hence $t \leq (x + (-X) + 1)$
 using $t-is-Lub$ by $(simp\ add:\ isLub-le-isUb)$
 hence $t + X + -1 \leq x$ by $arith$
 }
 then show $(t + X + -1) \leq^* Collect\ (isUb\ UNIV\ S)$
 by $(simp\ add:\ setgeI)$

next

show $isUb\ UNIV\ S\ (t + X + -1)$
 proof -
 {
 fix y
 assume $y-in-S: y \in S$
 have $y \leq t + X + -1$
 proof -
 obtain u where $u-in-shift: u \in ?SHIFT$ using $shifted-not-empty ..$
 hence $\exists x \in S. u = x + -X + 1$ by $simp$
 then obtain x where $x-and-u: u = x + -X + 1 ..$
 have $u-le-t: u \leq t$ using $u-in-shift$ and $t-is-Lub$ by $(simp\ add:\ isLubD2)$

show $?thesis$

proof cases

assume $y \leq x$
 moreover have $x = u + X + -1$ using $x-and-u$ by $arith$
 moreover have $u + X + -1 \leq t + X + -1$ using $u-le-t$ by $arith$
 ultimately show $y \leq t + X + -1$ by $arith$

next

assume $\sim(y \leq x)$
 hence $x-less-y: x < y$ by $arith$


```

      have  $x + (-X) + 1 \in ?SHIFT$  using  $x\text{-and-}u$  and  $u\text{-in-shift}$  by simp
      hence  $0 < x + (-X) + 1$  by simp
      hence  $0 < y + (-X) + 1$  using  $x\text{-less-}y$  by arith
      hence  $y + (-X) + 1 \in ?SHIFT$  using  $y\text{-in-}S$  by simp
      hence  $y + (-X) + 1 \leq t$  using  $t\text{-is-Lub}$  by (simp add: isLubD2)
      thus  $?thesis$  by simp
    qed
  qed
}
then show  $?thesis$  by (simp add: isUb-def settle-def)
qed
qed
qed
qed

```

A version of the same theorem without all those predicates!

```

lemma reals-complete2:
  fixes  $S :: (real\ set)$ 
  assumes  $\exists y. y \in S$  and  $\exists (x :: real). \forall y \in S. y \leq x$ 
  shows  $\exists x. (\forall y \in S. y \leq x) \ \&\$ 
     $(\forall z. ((\forall y \in S. y \leq z) \longrightarrow x \leq z))$ 
proof -
  have  $\exists x. isLub\ UNIV\ S\ x$ 
    by (rule reals-complete)
    (auto simp add: isLub-def isUb-def leastP-def settle-def setge-def prems)
  thus  $?thesis$ 
    by (metis UNIV-I isLub-isUb isLub-le-isUb isUbD isUb-def settleI)
qed

```

54.11 The Archimedean Property of the Reals

```

theorem reals-Archimedean:
  fixes  $x :: real$ 
  assumes  $x\text{-pos}: 0 < x$ 
  shows  $\exists n. inverse\ (of\text{-}nat\ (Suc\ n)) < x$ 
proof (rule ccontr)
  assume contr:  $\neg ?thesis$ 
  have  $\forall n. x * of\text{-}nat\ (Suc\ n) \leq 1$ 
  proof
    fix  $n$ 
    from contr have  $x \leq inverse\ (of\text{-}nat\ (Suc\ n))$ 
      by (simp add: linorder-not-less)
    hence  $x \leq (1 / (of\text{-}nat\ (Suc\ n)))$ 
      by (simp add: inverse-eq-divide)
    moreover have  $(0 :: real) \leq of\text{-}nat\ (Suc\ n)$ 
      by (rule of-nat-0-le-iff)
    ultimately have  $x * of\text{-}nat\ (Suc\ n) \leq (1 / of\text{-}nat\ (Suc\ n)) * of\text{-}nat\ (Suc\ n)$ 
      by (rule mult-right-mono)
  qed

```

thus $x * \text{of-nat } (\text{Suc } n) \leq 1$ **by** (*simp del: of-nat-Suc*)
qed
 hence $\{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} * \leq 1$
by (*simp add: settle-def del: of-nat-Suc, safe, rule spec*)
 hence $\text{isUb } (\text{UNIV}::\text{real set}) \{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} 1$
by (*simp add: isUbI*)
 hence $\exists Y. \text{isUb } (\text{UNIV}::\text{real set}) \{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} Y ..$
moreover have $\exists X. X \in \{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\}$ **by** *auto*
ultimately have $\exists t. \text{isLub UNIV } \{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} t$
by (*simp add: reals-complete*)
then obtain t **where**
 $t\text{-is-Lub: isLub UNIV } \{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} t ..$

have $\forall n::\text{nat}. x * \text{of-nat } n \leq t + -x$
proof
fix n
from $t\text{-is-Lub}$ **have** $x * \text{of-nat } (\text{Suc } n) \leq t$
by (*simp add: isLubD2*)
hence $x * (\text{of-nat } n) + x \leq t$
by (*simp add: right-distrib*)
thus $x * (\text{of-nat } n) \leq t + -x$ **by** *arith*
qed

hence $\forall m. x * \text{of-nat } (\text{Suc } m) \leq t + -x$ **by** (*simp del: of-nat-Suc*)
hence $\{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} * \leq (t + -x)$
by (*auto simp add: settle-def*)
hence $\text{isUb } (\text{UNIV}::\text{real set}) \{z. \exists n. z = x * (\text{of-nat } (\text{Suc } n))\} (t + (-x))$
by (*simp add: isUbI*)
hence $t \leq t + -x$
using $t\text{-is-Lub}$ **by** (*simp add: isLub-le-isUb*)
thus False **using** $x\text{-pos}$ **by** *arith*
qed

There must be other proofs, e.g. *Suc* of the largest integer in the cut representing x .

lemma *reals-Archimedean2*: $\exists n. (x::\text{real}) < \text{of-nat } (n::\text{nat})$
proof *cases*
assume $x \leq 0$
hence $x < \text{of-nat } (1::\text{nat})$ **by** *simp*
thus *?thesis* ..
next
assume $\neg x \leq 0$
hence $x\text{-greater-zero: } 0 < x$ **by** *simp*
hence $0 < \text{inverse } x$ **by** *simp*
then obtain n **where** $\text{inverse } (\text{of-nat } (\text{Suc } n)) < \text{inverse } x$
using *reals-Archimedean* **by** *blast*
hence $\text{inverse } (\text{of-nat } (\text{Suc } n)) * x < \text{inverse } x * x$
using $x\text{-greater-zero}$ **by** (*rule mult-strict-right-mono*)
hence $\text{inverse } (\text{of-nat } (\text{Suc } n)) * x < 1$

```

    using x-greater-zero by simp
  hence of-nat (Suc n) * (inverse (of-nat (Suc n)) * x) < of-nat (Suc n) * 1
    by (rule mult-strict-left-mono) (simp del: of-nat-Suc)
  hence x < of-nat (Suc n)
    by (simp add: algebra-simps del: of-nat-Suc)
  thus  $\exists (n::nat). x < of-nat n ..$ 
qed

instance real :: archimedean-field
proof
  fix r :: real
  obtain n :: nat where r < of-nat n
    using reals-Archimedean2 ..
  then have r ≤ of-int (int n)
    by simp
  then show  $\exists z. r \leq of-int z ..$ 
qed

end

```

55 Hilbert's choice and classical logic

theory *Hilbert-Classical* imports *Main* begin

Derivation of the classical law of tertium-non-datur by means of Hilbert's choice operator (due to M. J. Beeson and J. Harrison).

55.1 Proof text

```

theorem tnd:  $A \vee \neg A$ 
proof -
  let ?P =  $\lambda X. X = False \vee X = True \wedge A$ 
  let ?Q =  $\lambda X. X = False \wedge A \vee X = True$ 

  have a: ?P (Eps ?P)
  proof (rule someI)
    have False = False ..
    thus ?P False ..
  qed
  have b: ?Q (Eps ?Q)
  proof (rule someI)
    have True = True ..
    thus ?Q True ..
  qed

  from a show ?thesis
proof
  assume Eps ?P = True ∧ A

```

```

    hence  $A$  ..
    thus  $?thesis$  ..
next
  assume  $P$ :  $Eps\ ?P = False$ 
  from  $b$  show  $?thesis$ 
  proof
    assume  $Eps\ ?Q = False \wedge A$ 
    hence  $A$  ..
    thus  $?thesis$  ..
  next
    assume  $Q$ :  $Eps\ ?Q = True$ 
    have  $neg$ :  $?P \neq ?Q$ 
    proof
      assume  $?P = ?Q$ 
      hence  $Eps\ ?P = Eps\ ?Q$  by (rule arg-cong)
      also note  $P$ 
      also note  $Q$ 
      finally show  $False$  by (rule False-neg-True)
    qed
    have  $\neg A$ 
    proof
      assume  $a$ :  $A$ 
      have  $?P = ?Q$ 
      proof (rule ext)
        fix  $x$  show  $?P\ x = ?Q\ x$ 
        proof
          assume  $?P\ x$ 
          thus  $?Q\ x$ 
          proof
            assume  $x = False$ 
            from this and  $a$  have  $x = False \wedge A$  ..
            thus  $?Q\ x$  ..
          next
            assume  $x = True \wedge A$ 
            hence  $x = True$  ..
            thus  $?Q\ x$  ..
          qed
        next
          assume  $?Q\ x$ 
          thus  $?P\ x$ 
          proof
            assume  $x = False \wedge A$ 
            hence  $x = False$  ..
            thus  $?P\ x$  ..
          next
            assume  $x = True$ 
            from this and  $a$  have  $x = True \wedge A$  ..
            thus  $?P\ x$  ..
          qed
        qed
      qed
    qed
  qed

```

```

      qed
    qed
  with neq show False by contradiction
  qed
  thus ?thesis ..
  qed
  qed
  qed

```

55.2 Proof term of text

```

disjE . . . . .
(someI · (λX. X = False ∨ X = True ∧ ?A) . . . thm ·
  (disjI1 . . . . . (HOL.refl . . . thm))) ·
(λH: -.
  disjE . . . . .
    (someI · (λX. X = False ∧ ?A ∨ X = True) . . . thm ·
      (disjI2 . . . . . (HOL.refl . . . thm))) ·
    (λH: -. disjI1 . . . . . (conjE . . . . . H · (λ(H: -. H) H))) ·
    (λHa: -.
      disjI2 . . . . .
        (notI . . .
          (λHb: -.
            notE . . . . .
              (notI . . .
                (λHb: -.
                  False-neq-True . . .
                    (order-trans-rules-29 . . . . . thm ·
                      (order-trans-rules-14 ·
                        (λa. a = (SOME X. X = False ∧ ?A ∨ X = True)) ·
                          . .
                          . .
                          thm ·
                            (arg-cong · (λX. X = False ∨ X = True ∧ ?A) ·
                              (λX. X = False ∧ ?A ∨ X = True) ·
                                Eps ·
                                  (? · thm · thm) ·
                                    thm ·
                                      Hb) ·
                                        H) ·
                                          Ha)))) ·
                    (ext . . . . . thm · thm ·
                      (λX. iffI . . . . .
                        (λH: -.
                          disjE . . . . . H ·
                            (λH: -. disjI1 . . . . . (conjI . . . . . H · Hb)) ·
                              (λH: -.
                                disjI2 . . . . .
                                  (conjE . . . . . H · (λ(H: -. Ha: -. H)))))) ·

```

$$\begin{aligned}
& (\lambda H: -. \\
& \quad \text{disjE} \cdot \dots \cdot H \cdot \\
& \quad (\lambda H: -. \\
& \quad \quad \text{disjI1} \cdot \dots \cdot \\
& \quad \quad (\text{conjE} \cdot \dots \cdot H \cdot (\lambda(H: -) Ha: -. H))) \cdot \\
& \quad (\lambda H: -. \\
& \quad \quad \text{disjI2} \cdot \dots \cdot (\text{conjI} \cdot \dots \cdot H \cdot Hb)))))) \cdot \\
& (\lambda H: -. \text{disjI1} \cdot \dots \cdot (\text{conjE} \cdot \dots \cdot H \cdot (\lambda(H: -) H: -. H)))
\end{aligned}$$

55.3 Proof script

```

theorem tnd': A ∨ ¬ A
  apply (subgoal-tac
    (((SOME x. x = False ∨ x = True ∧ A) = False) ∨
     ((SOME x. x = False ∨ x = True ∧ A) = True) ∧ A) ∧
    (((SOME x. x = False ∧ A ∨ x = True) = False) ∧ A ∨
     ((SOME x. x = False ∧ A ∨ x = True) = True)))
  prefer 2
  apply (rule conjI)
  apply (rule someI)
  apply (rule disjI1)
  apply (rule refl)
  apply (rule someI)
  apply (rule disjI2)
  apply (rule refl)
  apply (erule conjE)
  apply (erule disjE)
  apply (erule disjE)
  apply (erule conjE)
  apply (erule disjI1)
  prefer 2
  apply (erule conjE)
  apply (erule disjI1)
  apply (subgoal-tac
    (λx. (x = False) ∨ (x = True) ∧ A) ≠
    (λx. (x = False) ∧ A ∨ (x = True)))
  prefer 2
  apply (rule notI)
  apply (drule-tac f = λy. SOME x. y x in arg-cong)
  apply (drule trans, assumption)
  apply (drule sym)
  apply (drule trans, assumption)
  apply (erule False-neq-True)
  apply (rule disjI2)
  apply (rule notI)
  apply (erule notE)
  apply (rule ext)
  apply (rule iffI)

```

```

apply (erule disjE)
apply (rule disjI1)
apply (erule conjI)
apply assumption
apply (erule conjE)
apply (erule disjI2)
apply (erule disjE)
apply (erule conjE)
apply (erule disjI1)
apply (rule disjI2)
apply (erule conjI)
apply assumption
done

```

55.4 Proof term of script

```

conjE · · · · ·
(conjI · · · · ·
  (someI · (λx. x = False ∨ x = True ∧ ?A) · · · thm ·
    (disjI1 · · · · · (HOL.refl · · · thm))) ·
  (someI · (λx. x = False ∧ ?A ∨ x = True) · · · thm ·
    (disjI2 · · · · · (HOL.refl · · · thm)))) ·
(λ(H: -) Ha: -.
  disjE · · · · · H ·
  (λH: -.
    disjE · · · · · Ha ·
    (λH: -. conjE · · · · · H · (λH: -. disjI1 · · · -)) ·
    (λHa: -.
      disjI2 · · · · ·
      (notI · · ·
        (λHb: -.
          notE · · · · ·
          (notI · · ·
            (λHb: -.
              False-neg-True · · ·
              (HOL.trans · · · · · thm · (HOL.sym · · · · · thm · H) ·
              (HOL.trans · · · · · thm ·
                (arg-cong · (λx. x = False ∨ x = True ∧ ?A) ·
                  (λx. x = False ∧ ?A ∨ x = True) ·
                  Eps ·
                  (? · thm · thm) ·
                  thm ·
                  Hb) ·
                  Ha)))) ·
              (ext · · · · · thm · thm ·
                (λx. iffI · · · · ·
                  (λH: -.
                    disjE · · · · · H ·
                    (λH: -. disjI1 · · · · · (conjI · · · · · H · Hb)) ·

```

```

      (λH: -.
        conjE . . . . . H .
        (λ(H: -) Ha: -. disjI2 . . . . . H))) .
    (λH: -.
      disjE . . . . . H .
      (λH: -.
        conjE . . . . . H .
        (λ(H: -) Ha: -. disjI1 . . . . . H)) .
      (λH: -.
        disjI2 . . . . . (conjI . . . . . H . Hb)))))) .
  (λH: -. conjE . . . . . H . (λH: -. disjI1 . . . . .))

```

end

56 Installing an oracle for SVC (Stanford Validity Checker)

```

theory SVC-Oracle
imports Main
uses svc-funcs.ML
begin

```

```

consts
  iff-keep :: [bool, bool] => bool
  iff-unfold :: [bool, bool] => bool

```

```

hide-const iff-keep iff-unfold

```

```

oracle svc-oracle = Svc.oracle

```

```

ML ⟨⟨
  (*
    Installing the oracle for SVC (Stanford Validity Checker)

```

*The following code merely CALLS the oracle;
the soundness-critical functions are at svc-funcs.ML*

Based upon the work of Søren T. Heilmann
*)

```

(*Generalize an Isabelle formula, replacing by Vars
  all subterms not intelligible to SVC.*)
fun svc-abstract t =
  let
    (*The oracle's result is given to the subgoal using compose-tac because
      its premises are matched against the assumptions rather than used

```



```

to make subgoals. Therefore , abstraction must copy the parameters
precisely and make them available to all generated Vars.*)
val params = Term.strip-all-vars t
and body   = Term.strip-all-body t
val Us = map #2 params
val nPar = length params
val vname = Unsynchronized.ref V-a
val pairs = Unsynchronized.ref ([] : (term*term) list)
fun insert t =
  let val T = fastype-of t
      val v = Logic.combound (Var (!vname,0), Us---->T), 0, nPar)
  in vname := Symbol.bump-string (!vname);
     pairs := (t, v) :: !pairs;
     v
  end;
fun replace t =
  case t of
    Free - => t (*but not existing Vars, lest the names clash*)
  | Bound - => t
  | - => (case AList.lookup Pattern.aecon v (!pairs) t of
        SOME v => v
        | NONE   => insert t)
  (*abstraction of a numeric literal*)
  fun lit t = if can HOLogic.dest-number t then t else replace t;
  (*abstraction of a real/rational expression*)
  fun rat ((c as Const(@{const-name Groups.plus}, -)) $ x $ y) = c $ (rat x) $
(rat y)
  | rat ((c as Const(@{const-name Groups.minus}, -)) $ x $ y) = c $ (rat x) $
(rat y)
  | rat ((c as Const(@{const-name Rings.divide}, -)) $ x $ y) = c $ (rat x) $
(rat y)
  | rat ((c as Const(@{const-name Groups.times}, -)) $ x $ y) = c $ (rat x) $
(rat y)
  | rat ((c as Const(@{const-name Groups.uminus}, -)) $ x) = c $ (rat x)
  | rat t = lit t
  (*abstraction of an integer expression: no div, mod*)
  fun int ((c as Const(@{const-name Groups.plus}, -)) $ x $ y) = c $ (int x) $
(int y)
  | int ((c as Const(@{const-name Groups.minus}, -)) $ x $ y) = c $ (int x) $
(int y)
  | int ((c as Const(@{const-name Groups.times}, -)) $ x $ y) = c $ (int x) $
(int y)
  | int ((c as Const(@{const-name Groups.uminus}, -)) $ x) = c $ (int x)
  | int t = lit t
  (*abstraction of a natural number expression: no minus*)
  fun nat ((c as Const(@{const-name Groups.plus}, -)) $ x $ y) = c $ (nat x) $
(nat y)
  | nat ((c as Const(@{const-name Groups.times}, -)) $ x $ y) = c $ (nat x)
$ (nat y)

```

```

| nat ((c as Const(@{const-name Suc}, -)) $ x) = c $ (nat x)
| nat t = lit t
(*abstraction of a relation: =, <, <=*)
fun rel (T, c $ x $ y) =
  if T = HOLogic.realT then c $ (rat x) $ (rat y)
  else if T = HOLogic.intT then c $ (int x) $ (int y)
  else if T = HOLogic.natT then c $ (nat x) $ (nat y)
  else if T = HOLogic.boolT then c $ (fm x) $ (fm y)
  else replace (c $ x $ y) (*non-numeric comparison*)
(*abstraction of a formula*)
and fm ((c as Const(op &, -)) $ p $ q) = c $ (fm p) $ (fm q)
| fm ((c as Const(op |, -)) $ p $ q) = c $ (fm p) $ (fm q)
| fm ((c as Const(op ==>, -)) $ p $ q) = c $ (fm p) $ (fm q)
| fm ((c as Const(Not, -)) $ p) = c $ (fm p)
| fm ((c as Const(True, -))) = c
| fm ((c as Const(False, -))) = c
| fm (t as Const(op =, Type (fun, [T,-]))) $ - $ - = rel (T, t)
| fm (t as Const(@{const-name Orderings.less}, Type (fun, [T,-]))) $ - $ - =
rel (T, t)
| fm (t as Const(@{const-name Orderings.less-eq}, Type (fun, [T,-]))) $ - $ - =
= rel (T, t)
| fm t = replace t
(*entry point, and abstraction of a meta-formula*)
fun mt ((c as Const(Trueprop, -)) $ p) = c $ (fm p)
| mt ((c as Const(==>, -)) $ p $ q) = c $ (mt p) $ (mt q)
| mt t = fm t (*it might be a formula*)
in (list-all (params, mt body), !pairs) end;

(*Present the entire subgoal to the oracle, assumptions and all, but possibly
abstracted. Use via compose-tac, which performs no lifting but will
instantiate variables.*)

val svc-tac = CSUBGOAL (fn (ct, i) =>
  let
    val thy = Thm.theory-of-cterm ct;
    val (abs-goal, -) = svc-abstract (Thm.term-of ct);
    val th = svc-oracle (Thm.cterm-of thy abs-goal);
  in compose-tac (false, th, 0) i end
  handle TERM - => no-tac);
>>

end

```

References

- [1] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.
- [2] K. McMillan. Lecture notes on verification of digital and hybrid systems. NATO summer school, <http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial/toc.html>.
- [3] K. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.