

Miscellaneous HOL Examples

June 8, 2008

Contents

1	Foundations of HOL	6
1.1	Pure Logic	6
1.1.1	Basic logical connectives	6
1.1.2	Extensional equality	6
1.1.3	Derived connectives	7
1.2	Classical logic	11
2	Abstract Natural Numbers primitive recursion	12
3	Proof by guessing	15
4	Simple and efficient binary numerals	16
4.1	Binary representation of natural numbers	16
4.2	Direct operations – plain normalization	16
4.3	Indirect operations – ML will produce witnesses	17
4.4	Concrete syntax	19
4.5	Examples	20
5	Examples of recdef definitions	22
6	Examples of function definitions	25
6.1	Very basic	25
6.2	Currying	25
6.3	Nested recursion	25
6.4	More general patterns	26
6.4.1	Overlapping patterns	26
6.4.2	Guards	27
6.5	Mutual Recursion	28
6.6	Definitions in local contexts	28
6.7	Regression tests	29
7	Examples of automatically derived induction rules	32
7.1	Some simple induction principles on nat	32

8	Some of the results in Inductive Invariants for Nested Recursion	33
9	Example use if an inductive invariant to solve termination conditions	35
10	Using locales in Isabelle/Isar – outdated version!	37
10.1	Overview	37
10.2	Local contexts as mathematical structures	39
10.3	Explicit structures referenced implicitly	42
10.4	Simple meta-theory of structures	44
11	Test of Locale Interpretation	46
12	Interpretation of Defined Concepts	46
12.1	Lattices	46
12.1.1	Definitions	46
12.1.2	Total order \leq on <i>int</i>	55
12.1.3	Total order \leq on <i>nat</i>	57
12.1.4	Lattice <i>dvd</i> on <i>nat</i>	58
12.2	Group example with defined operations <i>inv</i> and <i>unit</i>	59
12.2.1	Locale declarations and lemmas	59
12.2.2	Interpretation of Functions	63
13	Monoids and Groups as predicates over record schemes	65
14	Binary arithmetic examples	66
14.1	Regression Testing for Cancellation Simprocs	66
14.2	Arithmetic Method Tests	67
14.3	The Integers	68
14.4	The Natural Numbers	71
15	Examples for hexadecimal and binary numerals	73
16	Antiquotations	74
17	Multiple nested quotations and anti-quotations	74
18	Partial equivalence relations	75
18.1	Partial equivalence	76
18.2	Equivalence on function spaces	76
18.3	Total equivalence	77
18.4	Quotient types	78
18.5	Equality on quotients	78
18.6	Picking representing elements	79

19 Summing natural numbers	80
20 Three Divides Theorem	82
20.1 Abstract	82
20.2 Formal proof	82
20.2.1 Miscellaneous summation lemmas	82
20.2.2 Generalised Three Divides	83
20.2.3 Three Divides Natural	84
21 Higher-Order Logic: Intuitionistic predicate calculus problems	86
22 CTL formulae	93
22.1 Basic fixed point properties	94
22.2 The tree induction principle	96
22.3 An application of tree induction	97
23 Arithmetic	98
23.1 Splitting of Operators: <i>max, min, abs, op −, nat, op mod, op div</i>	98
23.2 Meta-Logic	100
23.3 Various Other Examples	101
24 Binary trees	103
25 Sorting: Basic Theory	106
26 Merge Sort	107
27 A question from “Bundeswettbewerb Mathematik”	108
28 A lemma for Lagrange’s theorem	109
29 Groebner Basis Examples	110
29.1 Basic examples	110
29.2 Lemmas for Lagrange’s theorem	111
29.3 Colinearity is invariant by rotation	112
30 Milner-Tofte: Co-induction in Relational Semantics	112
31 Case study: Unification Algorithm	132
31.1 Basic definitions	132
31.2 Basic lemmas	133
31.3 Specification: Most general unifiers	134
31.4 The unification algorithm	134
31.5 Partial correctness	135

31.6	Properties used in termination proof	137
31.7	Termination proof	142
32	Some examples demonstrating the comm-ring method	142
33	Primitive Recursive Functions	143
34	The Full Theorem of Tarski	150
34.1	Partial Order	152
34.2	sublattice	156
34.3	lub	156
34.4	glb	157
34.5	fixed points	158
34.6	lemmas for Tarski, lub	158
34.7	Tarski fixpoint theorem 1, first part	160
34.8	interval	160
34.9	Top and Bottom	163
34.10	fixed points form a partial order	164
35	Implementation of carry chain incrementor and adder	168
35.1	Carry chain incrementor	168
36	Classical Predicate Calculus Problems	170
36.1	Traditional Classical Reasoner	170
36.1.1	Pelletier's examples	170
36.1.2	Classical Logic: examples with quantifiers	172
36.1.3	Problems requiring quantifier duplication	172
36.1.4	Hard examples with quantifiers	173
36.1.5	Problems (mainly) involving equality or functions	176
36.2	Model Elimination Prover	178
36.2.1	Pelletier's examples	179
36.2.2	Classical Logic: examples with quantifiers	180
36.2.3	Hard examples with quantifiers	181
37	Set Theory examples: Cantor's Theorem, Schröder-Bernstein Theorem, etc.	187
37.1	Examples for the <i>blast</i> paper	187
37.2	Cantor's Theorem: There is no surjection from a set to its powerset	188
37.3	The Schröder-Berstein Theorem	188
37.4	A simple party theorem	189
38	Meson test cases	191
38.1	Interactive examples	191

39 Hilbert’s choice and classical logic	267
39.1 Proof text	267
39.2 Proof term of text	268
39.3 Proof script	269
39.4 Proof term of script	270
40 Dense linear order without endpoints and a quantifier elimination procedure in Ferrante and Rackoff style	271
41 The classical QE after Langford for dense linear orders	276
42 Constructive dense linear orders yield QE for linear arithmetic over ordered Fields – see <i>Arith-Tools.thy</i>	278
42.1 Ferrante and Rackoff algorithm over ordered fields	283
43 Examples for Ferrante and Rackoff’s quantifier elimination procedure	291
44 Some examples for Presburger Arithmetic	295
45 Generic reflection and reification	341
46 Installing an oracle for SVC (Stanford Validity Checker)	342
47 Examples for the ‘refute’ command	345
47.1 Examples and Test Cases	345
47.1.1 Propositional logic	345
47.1.2 Predicate logic	346
47.1.3 Equality	346
47.1.4 First-Order Logic	347
47.1.5 Higher-Order Logic	349
47.1.6 Meta-logic	351
47.1.7 Schematic variables	351
47.1.8 Abstractions	351
47.1.9 Sets	352
47.1.10 arbitrary	353
47.1.11 The	353
47.1.12 Eps	353
47.1.13 Subtypes (typedef), typedecl	354
47.1.14 Inductive datatypes	354
47.1.15 Records	370
47.1.16 Inductively defined sets	371
47.1.17 Examples involving special functions	371
47.1.18 Axiomatic type classes and overloading	372

48 Examples for the 'quickcheck' command	374
48.1 Lists	375
48.2 Trees	376

1 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.

1.1 Pure Logic

classes *type*
defaultsort *type*

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

1.1.1 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*]: $\forall x. P\ x \Longrightarrow P\ a$

1.1.2 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*]: $(A \implies B) \implies (B \implies A) \implies A = B$

theorem *sym* [*sym*]: $x = y \implies y = x$

proof –

assume $x = y$

then show $y = x$ **by** (*rule subst*) (*rule refl*)

qed

lemma [*trans*]: $x = y \implies P\ y \implies P\ x$

by (*rule subst*) (*rule sym*)

lemma [*trans*]: $P\ x \implies x = y \implies P\ y$

by (*rule subst*)

theorem *trans* [*trans*]: $x = y \implies y = z \implies x = z$

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*)

1.1.3 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 :: $o \Rightarrow o \rightarrow \neg$ [*40*] **where**

not $\equiv \lambda A. A \longrightarrow \perp$

definition

conj :: $o \Rightarrow o \Rightarrow o \rightarrow \wedge$ [*35*] **where**

conj $\equiv \lambda A\ B. \forall C. (A \longrightarrow B \longrightarrow C) \longrightarrow C$

definition

disj :: $o \Rightarrow o \Rightarrow o \rightarrow \vee$ [*30*] **where**

disj $\equiv \lambda A\ B. \forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$

definition

Ex :: $('a \Rightarrow o) \Rightarrow o \rightarrow \exists$ [*10*] **where**

$\exists x. P\ x \equiv \forall C. (\forall x. P\ x \longrightarrow C) \longrightarrow C$

abbreviation

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

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

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*]: $(A \Longrightarrow \perp) \Longrightarrow \neg A$

proof (*unfold not-def*)

assume $A \Longrightarrow \perp$

then show $A \longrightarrow \perp$..

qed

theorem *notE* [*elim*]: $\neg A \Longrightarrow A \Longrightarrow B$

proof (*unfold not-def*)

assume $A \longrightarrow \perp$

also assume A

finally have \perp ..

then show B ..

qed

lemma *notE'*: $A \Longrightarrow \neg A \Longrightarrow B$

by (*rule notE*)

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

theorem *conjI* [*intro*]: $A \Longrightarrow B \Longrightarrow A \wedge B$

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]:  $A \implies A \vee B$ 
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]:  $B \implies A \vee B$ 
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
 qed
 qed
 qed

theorem *disjE* [*elim*]: $A \vee B \implies (A \implies C) \implies (B \implies C) \implies C$
proof (*unfold disj-def*)
 assume *c*: $\forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$
 assume *r1*: $A \implies C$ and *r2*: $B \implies C$
 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

1.2 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

2 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

$Rec\text{-}zero: Rec\ e\ r\ zero\ e$

| $Rec\text{-}succ: Rec\ e\ r\ m\ n \Longrightarrow Rec\ e\ r\ (succ\ m)\ (r\ m\ n)$

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 $\exists!y. ?R\ zero\ y$

proof

show $?R\ zero\ e\ ..$

fix y **assume** $?R\ zero\ y$

then show $y = e$ **by** *cases simp-all*

qed

next

case (*succ m*)

from $\langle \exists!y. ?R\ m\ y \rangle$

obtain y **where** $y: ?R\ m\ y$

and $yy': \bigwedge y'. ?R\ m\ y' \Longrightarrow y = y'$ **by** *blast*

show $\exists!z. ?R\ (succ\ m)\ z$

proof

from y **show** $?R\ (succ\ m)\ (r\ m\ y)\ ..$

fix z **assume** $?R\ (succ\ m)\ z$

then obtain u **where** $z = r\ m\ u$ **and** $?R\ m\ u$ **by** *cases simp-all*

with yy' **show** $z = r\ m\ y$ **by** (*simp only:*)

qed

qed

qed

The recursion operator – polymorphic!

definition

$rec :: 'a \Rightarrow ('n \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'n \Rightarrow 'a$ **where**

$rec\ e\ r\ x = (THE\ y. Rec\ e\ r\ x\ y)$

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 m (rec e r m)
proof (rule rec-eval)
  let ?R = Rec e r
  have ?R m (rec e r m)
    unfolding rec-def using Rec-functional by (rule theI')
  then show ?R (succ m) (r m (rec e r m)) ..
qed

```

Example: addition (monomorphic)

```

definition
  add :: 'n  $\Rightarrow$  'n  $\Rightarrow$  'n where
    add m n = rec n ( $\lambda$ -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  $\Rightarrow$  'a  $\Rightarrow$  'a list where
    repl n x = rec [] ( $\lambda$ -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:  $\bigwedge n. P\ n \implies P\ (Suc\ n)$ 
  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

```

3 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

```

4 Simple and efficient binary numerals

```
theory Binary
imports Main
begin
```

4.1 Binary representation of natural numbers

definition

```
bit :: nat ⇒ bool ⇒ 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 $\langle\langle$

```
structure Binary =
struct
```

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

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

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

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

```
end
 $\rangle\rangle$ 
```

4.2 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*
*bit m True * n = bit (m * n) False + n*
*bit m False * n = bit (m * n) False*
*n * bit m True = bit (m * n) False + n*
*n * bit m False = bit (m * n) False*
by (*simp-all add: bit-simps*)

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

4.3 Indirect operations – ML will produce witnesses

lemma *binary-less-eq*:

fixes *n :: nat*
shows *n ≡ m + k ⟹ (m ≤ n) ≡ True*
and *m ≡ n + k + 1 ⟹ (m ≤ n) ≡ False*
by *simp-all*

lemma *binary-less*:

fixes *n :: nat*
shows *m ≡ n + k ⟹ (m < n) ≡ False*
and *n ≡ m + k + 1 ⟹ (m < n) ≡ True*
by *simp-all*

lemma *binary-diff*:

fixes *n :: nat*
shows *m ≡ n + k ⟹ m - n ≡ k*
and *n ≡ m + k ⟹ m - n ≡ 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 <<
  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  $m\ n = @\{\text{term times} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}\} \$ m \$ n$ ;

    val binary-ss = HOL-basic-ss addsimps  $@\{\text{thms binary-simps}\}$ ;
    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 (pairself (Binary.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  $k \geq 0$  then
          SOME ( $@\{\text{thm binary-less-eq}(1)\}$  OF [prove ctxt ( $u == \text{plus } t \text{ (Binary.mk-binary } k)\)$ ])])
        else
          SOME ( $@\{\text{thm binary-less-eq}(2)\}$  OF
            [prove ctxt ( $t == \text{plus (plus } u \text{ (Binary.mk-binary } (\sim k - 1)) \text{ (Binary.mk-binary } 1)\)$ ])])
          end);

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

```

```

    [prove ctxt (u == plus (plus t (Binary.mk-binary (~ k - 1))) (Binary.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 (Binary.mk-binary
k))])
    else
      SOME (@{thm binary-diff(2)} OF [prove ctxt (u == plus t (Binary.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 (Binary.mk-binary k))
(Binary.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 = <<
  Method.no-args (Method.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

```

4.4 Concrete syntax

syntax

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

```

parse-translation <<
  let

    val syntax-consts = map-aterms (fn Const (c, T) => Const (Syntax.constN ^ 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 (Binary.mk-binary n) end
      | binary-tr ts = raise TERM (binary-tr, ts);

    in [(-Binary, binary-tr)] end
  >>

```

4.5 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 $11111111112222222222333333333334444444444 - $998877665544332211
  =
  $1111111111222222222233334455668900112233
  by binary-simp

lemma (11111111112222222222333333333334444444444::nat) - 998877665544332211

```

```

=
111111111122222222223334455668900112233
by simp

lemma (1111111111222222222233333333334444444444::int) - 998877665544332211
=
111111111122222222223334455668900112233
by simp

lemma $1111111111222222222233333333334444444444 * $998877665544332211
=
$1109864072938022197293802219729380221972383090160869185684
by binary-simp

lemma $1111111111222222222233333333334444444444 * $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 $1111111111222222222233333333334444444444 < $998877665544332211
= False
by binary-simp

lemma $998877665544332211 < $1111111111222222222233333333334444444444
= True
by binary-simp

lemma $1111111111222222222233333333334444444444 <= $998877665544332211
= False
by binary-simp

lemma $998877665544332211 <= $1111111111222222222233333333334444444444
= True
by binary-simp

lemma $1234 div $23 = $53

```

```

by binary-simp

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

lemma $111111111222222222233333333333333334444444444 div $998877665544332211
=
  $1112359550673033707875
by binary-simp

lemma $111111111222222222233333333333333334444444444 mod $998877665544332211
=
  $42245174317582819
by binary-simp

lemma (111111111222222222233333333333333334444444444::int) div 998877665544332211
=
  1112359550673033707875
by simp — legacy numerals: 30 times slower

lemma (111111111222222222233333333333333334444444444::int) mod 998877665544332211
=
  42245174317582819
by simp — legacy numerals: 30 times slower

end

```

5 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
```

6 Examples of function definitions

```
theory Fundefs
imports Main
begin
```

6.1 Very basic

```
fun fib :: nat  $\Rightarrow$  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
```

6.2 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
```

6.3 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
using trm by induct auto

```

```

termination
proof
  let ?R = measure (%x. 101 - x)
  show wf ?R ..

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

  assume inner-trm: f91-dom (n + 11)
  with f91-estimate have n + 11 < f91 (n + 11) + 11 .
  with (~ 100 < n) show (f91 (n + 11), n) : ?R by simp
qed

```

6.4 More general patterns

6.4.1 Overlapping patterns

Currently, patterns must always be compatible with each other, since no automatic splitting takes place. But the following definition of gcd is ok, although patterns overlap:

```

fun gcd2 :: nat => nat => nat
where
  gcd2 x 0 = x
| gcd2 0 y = y

```

$$\mid \text{gcd2 } (\text{Suc } x) (\text{Suc } y) = (\text{if } x < y \text{ then } \text{gcd2 } (\text{Suc } x) (y - x) \\ \text{else } \text{gcd2 } (x - y) (\text{Suc } y))$$

thm *gcd2.simps*
thm *gcd2.induct*

6.4.2 Guards

We can reformulate the above example using guarded patterns

```
function gcd3 :: nat ⇒ nat ⇒ nat
where
  gcd3 x 0 = x
  gcd3 0 y = y
  gcd3 x y = if x < y then gcd3 (Suc x) (Suc y) = gcd3 (Suc x) (y - x)
  gcd3 x y = if ¬ x < y then 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 ⇒ 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: ∧n. x = 2 * n ⇒ P
  and c2: ∧n. x = 2 * n + 1 ⇒ P
  have divmod: x = 2 * (x div 2) + (x mod 2) by auto
  show P
  proof cases
    assume x mod 2 = 0
    with divmod have x = 2 * (x div 2) by simp
    with c1 show P .
  next
    assume x mod 2 ≠ 0
    hence x mod 2 = 1 by simp
    with divmod have x = 2 * (x 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

```

6.5 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

```

6.6 Definitions in local contexts

```

locale my-monoid =
fixes opr :: 'a  $\Rightarrow$  'a  $\Rightarrow$  '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  $\Rightarrow$  'a
where
  foldR [] = un
| foldR (x#xs) = opr x (foldR xs)

```

```

fun foldL :: 'a list  $\Rightarrow$  '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

```

6.7 Regression tests

The following examples mainly serve as tests for the function package

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

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

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

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

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

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

```
function k :: nat  $\Rightarrow$  nat
where
  k x = (let a = x; b = x in k x)
```

```

by pat-completeness auto

function f2 :: (nat × nat) ⇒ (nat × nat)
where
  f2 p = (let (x,y) = p in f2 (y,x))
  by pat-completeness auto

fun f3 :: 'a set ⇒ bool
where
  f3 x = finite x

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

lemma lem:x ∈ set l ⇒ size x < Suc (list-size size l)
by (induct l, auto)

function treemap :: ('a ⇒ 'a) ⇒ 'a tree ⇒ 'a tree
where
  treemap fn (Leaf n) = (Leaf (fn n))
  | treemap fn (Branch l) = (Branch (map (treemap fn) l))
by pat-completeness auto
termination by (lexicographic-order simp:lem)

declare lem[simp]

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

record point =
  Xcoord :: int
  Ycoord :: int

function swp :: point ⇒ point
where
  swp (| Xcoord = x, Ycoord = y |) = (| Xcoord = y, Ycoord = x |)
proof –
  fix P x

```

```

assume  $\bigwedge xa\ y. x = (Xcoord = xa, Ycoord = y) \implies P$ 
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 =
   $A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M \mid N \mid P$ 
|  $Q \mid R \mid S \mid T \mid U \mid V$ 

```

```

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 ⇒ nat ⇒ bool
where
  f4 0 0 = True
  | f4 - - = False

end

```

7 Examples of automatically derived induction rules

```

theory Induction-Scheme
imports Main
begin

```

7.1 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 induct-scheme (pat-completeness, lexicographic-order)

```

```

lemma nat-induct2:
   $\llbracket P\ 0; P\ (Suc\ 0); \bigwedge k. P\ k \implies P\ (Suc\ (Suc\ k)) \rrbracket$ 
 $\implies P\ n$ 
by induct-scheme (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 induct-scheme (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 induct-scheme (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 induct-scheme (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 induct-scheme (pat-completeness, lexicographic-order)

end

```

8 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) set => ('a => 'b => bool) => (('a => 'b) => ('a => 'b))
=> bool where
  indinv r S F = (∀ f x. (∀ y. (y,x) : r --> S y (f y)) --> 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-on :: ('a * 'a) set => 'a set => ('a => 'b => bool) => (('a => 'b) =>
('a => 'b)) => bool where
  indinv-on r D S F = (∀ f. ∀ x ∈ D. (∀ y ∈ D. (y,x) ∈ r --> S y (f y)) --> 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) ∈ r ==> S y (wfrec r F y)
  then have      !!y. (y,x) ∈ r ==> 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 ($insert\ INV\ D\ indinv\text{-}wfrec\ [OF\ WF, of\ \% \ x\ y.\ x \in D \longrightarrow S\ x\ y]$)
by ($simp\ add: indinv\text{-}on\text{-}def\ indinv\text{-}def$)

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 ($rule\ indinv\text{-}on\text{-}wfrec\ [OF\ WF - D, of\ \% \ a\ b.\ F\ (wfrec\ r\ F)\ a = b \ \&\ wfrec\ r\ F\ a = b \ \&\ S\ a\ b\ F, simplified]$)
show $indinv\text{-}on\ r\ D\ (\% a\ b.\ F\ (wfrec\ r\ F)\ a = b \ \&\ wfrec\ r\ F\ a = b \ \&\ S\ a\ b)\ F$
proof ($unfold\ indinv\text{-}on\text{-}def, clarify$)
fix $f\ x$
assume $A1$: $\forall y \in D. (y, x) \in r \longrightarrow F\ (wfrec\ r\ F)\ y = f\ y \ \&\ wfrec\ r\ F\ y = f\ y \ \&\ S\ y\ (f\ y)$
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** $\forall y \in D. (y, x) \in r \longrightarrow S\ y\ (wfrec\ r\ F\ y)$ **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 : $\forall f\ x. (\forall y. (y, x) \in r \longrightarrow S\ y\ (wfrec\ r\ F\ y) \ \&\ f\ y = wfrec\ r\ F\ y)$
 $\longrightarrow S\ x\ (wfrec\ r\ F\ x) \ \&\ F\ f\ x = wfrec\ r\ F\ x$
shows $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x \ \&\ S\ x\ (wfrec\ r\ F\ x)$
apply ($rule\ ind\text{-}fixpoint\text{-}on\text{-}lemma\ [OF\ WF - UNIV\text{-}I, simplified]$)
by ($rule\ INV$)

theorem *tfl-indinv-wfrec*:
 $[| f == wfrec\ r\ F; wf\ r; indinv\ r\ S\ F |]$
 $\implies S\ x\ (f\ x)$
by ($simp\ add: indinv\text{-}wfrec$)

theorem *tfl-indinv-on-wfrec*:
 $[| f == wfrec\ r\ F; wf\ r; indinv\text{-}on\ r\ D\ S\ F; x \in D |]$

```

==> S x (f x)
by (simp add: indinv-on-wfrec)

end

```

9 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 $g :: \text{nat} \Rightarrow \text{nat}$

recdef (**permissive**) g *less-than*
 $g\ 0 = 0$
 $g\ (\text{Suc } n) = g\ (g\ n)$

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

recdef-tc $g\text{-tc}[simp]: g$
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 $g.\text{simps}$ [*simplified, simp*]

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

consts $g' :: \text{nat} \Rightarrow \text{nat}$
recdef (**permissive**) g' *less-than*
 $g'\ 0 = 0$
 $g'\ (\text{Suc } n) = g'\ n + g'\ (g'\ n)$

thm $g'.\text{simps}$

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

lemma $g'\text{-inv}: g'\ n = 0$

```

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'.sims [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-even n = g-even (g-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: $S \ x \ y == x \downarrow y + 11$

```

lemma ninety-one-inv:  $n < ninety\text{-one } n + 11$ 
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

```

10 Using locales in Isabelle/Isar – outdated version!

```

theory Locales imports Main begin

```

10.1 Overview

Locales provide a mechanism for encapsulating local contexts. The original version due to Florian Kammüller [2] refers directly to Isabelle’s meta-logic [7], which is minimal higher-order logic with connectives \bigwedge (universal quantification), \implies (implication), and \equiv (equality).

From this perspective, a locale is essentially a meta-level predicate, together with some infrastructure to manage the abstracted parameters (\bigwedge), assumptions (\implies), and definitions for (\equiv) in a reasonable way during the proof process. This simple predicate view also provides a solid semantical basis for our specification concepts to be developed later.

The present version of locales for Isabelle/Isar builds on top of the rich infrastructure of proof contexts [9, 11, 10], which in turn is based on the same meta-logic. Thus we achieve a tight integration with Isar proof texts, and a slightly more abstract view of the underlying logical concepts. An Isar proof context encapsulates certain language elements that correspond to $\bigwedge/\implies/\equiv$ at the level of structure proof texts. Moreover, there are extra-logical concepts like term abbreviations or local theorem attributes (declarations of

simplification rules etc.) that are useful in applications (e.g. consider standard simplification rules declared in a group context).

Locales also support concrete syntax, i.e. a localized version of the existing concept of mixfix annotations of Isabelle [8]. Furthermore, there is a separate concept of “implicit structures” that admits to refer to particular locale parameters in a casual manner (basically a simplified version of the idea of “anti-quotations”, or generalized de-Bruijn indexes as demonstrated elsewhere [12, §13–14]).

Implicit structures work particular well together with extensible records in HOL [5] (without the “object-oriented” features discussed there as well). Thus we achieve a specification technique where record type schemes represent polymorphic signatures of mathematical structures, and actual locales describe the corresponding logical properties. Semantically speaking, such abstract mathematical structures are just predicates over record types. Due to type inference of simply-typed records (which subsumes structural subtyping) we arrive at succinct specification texts — “signature morphisms” degenerate to implicit type-instantiations. Additional eye-candy is provided by the separate concept of “indexed concrete syntax” used for record selectors, so we get close to informal mathematical notation.

Operations for building up locale contexts from existing ones include *merge* (disjoint union) and *rename* (of term parameters only, types are inferred automatically). Here we draw from existing traditions of algebraic specification languages. A structured specification corresponds to a directed acyclic graph of potentially renamed nodes (due to distributivity renames may be pushed inside of merges). The result is a “flattened” list of primitive context elements in canonical order (corresponding to left-to-right reading of merges, while suppressing duplicates).

The present version of Isabelle/Isar locales still lacks some important specification concepts.

- Separate language elements for *instantiation* of locales.

Currently users may simulate this to some extent by having primitive Isabelle/Isar operations (*of* for substitution and *OF* for composition, [11]) act on the automatically exported results stemming from different contexts.

- Interpretation of locales (think of “views”, “functors” etc.).

In principle one could directly work with functions over structures (extensible records), and predicates being derived from locale definitions.

Subsequently, we demonstrate some readily available concepts of Isabelle/Isar locales by some simple examples of abstract algebraic reasoning.

10.2 Local contexts as mathematical structures

The following definitions of *group-context* and *abelian-group-context* merely encapsulate local parameters (with private syntax) and assumptions; local definitions of derived concepts could be given, too, but are unused below.

```

locale group-context =
  fixes prod :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a    (infixl  $\cdot$  70)
  and inv :: 'a  $\Rightarrow$  'a    (( $-^1$ ) [1000] 999)
  and one :: 'a    (1)
  assumes assoc: (x  $\cdot$  y)  $\cdot$  z = x  $\cdot$  (y  $\cdot$  z)
  and left-inv: x $^{-1}$   $\cdot$  x = 1
  and left-one: 1  $\cdot$  x = x

```

```

locale abelian-group-context = group-context +
  assumes commute: x  $\cdot$  y = y  $\cdot$  x

```

We may now prove theorems within a local context, just by including a directive “(**in** *name*)” in the goal specification. The final result will be stored within the named locale, still holding the context; a second copy is exported to the enclosing theory context (with qualified name).

```

theorem (in group-context)
  right-inv: x  $\cdot$  x $^{-1}$  = 1
proof -
  have x  $\cdot$  x $^{-1}$  = 1  $\cdot$  (x  $\cdot$  x $^{-1}$ ) by (simp only: left-one)
  also have ... = 1  $\cdot$  x  $\cdot$  x $^{-1}$  by (simp only: assoc)
  also have ... = (x $^{-1}$ ) $^{-1}$   $\cdot$  x $^{-1}$   $\cdot$  x  $\cdot$  x $^{-1}$  by (simp only: left-inv)
  also have ... = (x $^{-1}$ ) $^{-1}$   $\cdot$  (x $^{-1}$   $\cdot$  x)  $\cdot$  x $^{-1}$  by (simp only: assoc)
  also have ... = (x $^{-1}$ ) $^{-1}$   $\cdot$  1  $\cdot$  x $^{-1}$  by (simp only: left-inv)
  also have ... = (x $^{-1}$ ) $^{-1}$   $\cdot$  (1  $\cdot$  x $^{-1}$ ) by (simp only: assoc)
  also have ... = (x $^{-1}$ ) $^{-1}$   $\cdot$  x $^{-1}$  by (simp only: left-one)
  also have ... = 1 by (simp only: left-inv)
  finally show ?thesis .
qed

```

```

theorem (in group-context)
  right-one: x  $\cdot$  1 = x
proof -
  have x  $\cdot$  1 = x  $\cdot$  (x $^{-1}$   $\cdot$  x) by (simp only: left-inv)
  also have ... = x  $\cdot$  x $^{-1}$   $\cdot$  x by (simp only: assoc)
  also have ... = 1  $\cdot$  x by (simp only: right-inv)
  also have ... = x by (simp only: left-one)
  finally show ?thesis .
qed

```

Facts like *right-one* are available *group-context* as stated above. The exported version loses the additional infrastructure of Isar proof contexts (syntax etc.) retaining only the pure logical content: *group-context.right-one*

becomes *group-context* $?prod\ ?inv\ ?one \implies ?prod\ ?x\ ?one = ?x$ (in Isabelle outermost \bigwedge quantification is replaced by schematic variables).

Apart from a named locale we may also refer to further context elements (parameters, assumptions, etc.) in an ad-hoc fashion, just for this particular statement. In the result (local or global), any additional elements are discharged as usual.

```

theorem (in group-context)
  assumes eq:  $e \cdot x = x$ 
  shows one-equality:  $1 = e$ 
proof -
  have  $1 = x \cdot x^{-1}$  by (simp only: right-inv)
  also have  $\dots = (e \cdot x) \cdot x^{-1}$  by (simp only: eq)
  also have  $\dots = e \cdot (x \cdot x^{-1})$  by (simp only: assoc)
  also have  $\dots = e \cdot 1$  by (simp only: right-inv)
  also have  $\dots = e$  by (simp only: right-one)
  finally show ?thesis .
qed

```

```

theorem (in group-context)
  assumes eq:  $x' \cdot x = 1$ 
  shows inv-equality:  $x^{-1} = x'$ 
proof -
  have  $x^{-1} = 1 \cdot x^{-1}$  by (simp only: left-one)
  also have  $\dots = (x' \cdot x) \cdot x^{-1}$  by (simp only: eq)
  also have  $\dots = x' \cdot (x \cdot x^{-1})$  by (simp only: assoc)
  also have  $\dots = x' \cdot 1$  by (simp only: right-inv)
  also have  $\dots = x'$  by (simp only: right-one)
  finally show ?thesis .
qed

```

```

theorem (in group-context)
  inv-prod:  $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$ 
proof (rule inv-equality)
  show  $(y^{-1} \cdot x^{-1}) \cdot (x \cdot y) = 1$ 
  proof -
    have  $(y^{-1} \cdot x^{-1}) \cdot (x \cdot y) = (y^{-1} \cdot (x^{-1} \cdot x)) \cdot y$  by (simp only: assoc)
    also have  $\dots = (y^{-1} \cdot 1) \cdot y$  by (simp only: left-inv)
    also have  $\dots = y^{-1} \cdot y$  by (simp only: right-one)
    also have  $\dots = 1$  by (simp only: left-inv)
    finally show ?thesis .
  qed
qed

```

Established results are automatically propagated through the hierarchy of locales. Below we establish a trivial fact in commutative groups, while referring both to theorems of *group* and the additional assumption of *abelian-group*.

```

theorem (in abelian-group-context)

```


$inv\text{-}prod': (x \cdot y)^{-1} = x^{-1} \cdot y^{-1}$
proof –
 have $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$ **by** (*rule inv-prod*)
 also have $\dots = x^{-1} \cdot y^{-1}$ **by** (*rule commute*)
 finally show *?thesis* .
qed

We see that the initial import of *group* within the definition of *abelian-group* is actually evaluated dynamically. Thus any results in *group* are made available to the derived context of *abelian-group* as well. Note that the alternative context element **includes** would import existing locales in a static fashion, without participating in further facts emerging later on.

Some more properties of inversion in general group theory follow.

theorem (*in group-context*)
 $inv\text{-}inv: (x^{-1})^{-1} = x$
proof (*rule inv-equality*)
 show $x \cdot x^{-1} = 1$ **by** (*simp only: right-inv*)
qed

theorem (*in group-context*)
 assumes $eq: x^{-1} = y^{-1}$
 shows $inv\text{-}inject: x = y$
proof –
 have $x = x \cdot 1$ **by** (*simp only: right-one*)
 also have $\dots = x \cdot (y^{-1} \cdot y)$ **by** (*simp only: left-inv*)
 also have $\dots = x \cdot (x^{-1} \cdot y)$ **by** (*simp only: eq*)
 also have $\dots = (x \cdot x^{-1}) \cdot y$ **by** (*simp only: assoc*)
 also have $\dots = 1 \cdot y$ **by** (*simp only: right-inv*)
 also have $\dots = y$ **by** (*simp only: left-one*)
 finally show *?thesis* .
qed

We see that this representation of structures as local contexts is rather lightweight and convenient to use for abstract reasoning. Here the “components” (the group operations) have been exhibited directly as context parameters; logically this corresponds to a curried predicate definition:

$group\text{-}context \text{ prod inv one} \equiv$
 $(\forall x y z. \text{prod} (\text{prod } x y) z = \text{prod } x (\text{prod } y z)) \wedge$
 $(\forall x. \text{prod} (\text{inv } x) x = one) \wedge (\forall x. \text{prod } one x = x)$

The corresponding introduction rule is as follows:

$(\wedge x y z. \text{prod} (\text{prod } x y) z = \text{prod } x (\text{prod } y z)) \implies$
 $(\wedge x. \text{prod} (\text{inv } x) x = one) \implies$
 $(\wedge x. \text{prod } one x = x) \implies group\text{-}context \text{ prod inv one}$

Occasionally, this “externalized” version of the informal idea of classes of tuple structures may cause some inconveniences, especially in meta-theoretical studies (involving functors from groups to groups, for example).

Another minor drawback of the naive approach above is that concrete syntax will get lost on any kind of operation on the locale itself (such as renaming, copying, or instantiation). Whenever the particular terminology of local parameters is affected the associated syntax would have to be changed as well, which is hard to achieve formally.

10.3 Explicit structures referenced implicitly

We introduce the same hierarchy of basic group structures as above, this time using extensible record types for the signature part, together with concrete syntax for selector functions.

```
record 'a semigroup =
  prod :: 'a ⇒ 'a ⇒ 'a   (infixl · 70)

record 'a group = 'a semigroup +
  inv :: 'a ⇒ 'a   ((-1) [1000] 999)
  one :: 'a   (1)
```

The mixfix annotations above include a special “structure index indicator” $\mathbf{1}$ that makes grammar productions dependent on certain parameters that have been declared as “structure” in a locale context later on. Thus we achieve casual notation as encountered in informal mathematics, e.g. $x \cdot y$ for $\text{prod } G \ x \ y$.

The following locale definitions introduce `operate` on a single parameter declared as “**structure**”. Type inference takes care to fill in the appropriate record type schemes internally.

```
locale semigroup =
  fixes S   (structure)
  assumes assoc: (x · y) · z = x · (y · z)

locale group = semigroup G +
  assumes left-inv: x-1 · x = 1
  and left-one: 1 · x = x

declare semigroup.intro [intro?]
  group.intro [intro?] group-axioms.intro [intro?]
```

Note that we prefer to call the *group* record structure G rather than S inherited from *semigroup*. This does not affect our concrete syntax, which is only dependent on the *positional* arrangements of currently active structures (actually only one above), rather than names. In fact, these parameter

names rarely occur in the term language at all (due to the “indexed syntax” facility of Isabelle). On the other hand, names of locale facts will get qualified accordingly, e.g. $S.assoc$ versus $G.assoc$.

We may now proceed to prove results within *group* just as before for *group*. The subsequent proof texts are exactly the same as despite the more advanced internal arrangement.

```

theorem (in group)
  right-inv:  $x \cdot x^{-1} = 1$ 
proof –
  have  $x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1})$  by (simp only: left-one)
  also have  $\dots = 1 \cdot x \cdot x^{-1}$  by (simp only: assoc)
  also have  $\dots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1}$  by (simp only: left-inv)
  also have  $\dots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1}$  by (simp only: assoc)
  also have  $\dots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1}$  by (simp only: left-inv)
  also have  $\dots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1})$  by (simp only: assoc)
  also have  $\dots = (x^{-1})^{-1} \cdot x^{-1}$  by (simp only: left-one)
  also have  $\dots = 1$  by (simp only: left-inv)
  finally show ?thesis .
qed

```

```

theorem (in group)
  right-one:  $x \cdot 1 = x$ 
proof –
  have  $x \cdot 1 = x \cdot (x^{-1} \cdot x)$  by (simp only: left-inv)
  also have  $\dots = x \cdot x^{-1} \cdot x$  by (simp only: assoc)
  also have  $\dots = 1 \cdot x$  by (simp only: right-inv)
  also have  $\dots = x$  by (simp only: left-one)
  finally show ?thesis .
qed

```

Several implicit structures may be active at the same time. The concrete syntax facility for locales actually maintains indexed structures that may be references implicitly — via mixfix annotations that have been decorated by an “index argument” (i).

The following synthetic example demonstrates how to refer to several structures of type *group* succinctly. We work with two versions of the *group* locale above.

```

lemma
  includes group  $G$ 
  includes group  $H$ 
  shows  $x \cdot y \cdot 1 = \text{prod } G \ (\text{prod } G \ x \ y) \ (\text{one } G)$ 
    and  $x \cdot_2 y \cdot_2 1_2 = \text{prod } H \ (\text{prod } H \ x \ y) \ (\text{one } H)$ 
    and  $x \cdot 1_2 = \text{prod } G \ x \ (\text{one } H)$ 
  by (rule refl)+

```

Note that the trivial statements above need to be given as a simultaneous

goal in order to have type-inference make the implicit typing of structures G and H agree.

10.4 Simple meta-theory of structures

The packaging of the logical specification as a predicate and the syntactic structure as a record type provides a reasonable starting point for simple meta-theoretic studies of mathematical structures. This includes operations on structures (also known as “functors”), and statements about such constructions.

For example, the direct product of semigroups works as follows.

constdefs

```
semigroup-product :: 'a semigroup  $\Rightarrow$  'b semigroup  $\Rightarrow$  ('a  $\times$  'b) semigroup
semigroup-product S T  $\equiv$ 
  ( $\lambda p = \lambda p q. (prod S (fst p) (fst q), prod T (snd p) (snd q))$ )
```

lemma *semigroup-product* [intro]:

```
assumes S: semigroup S
and T: semigroup T
shows semigroup (semigroup-product S T)
```

proof

```
fix p q r :: 'a  $\times$  'b
have prod S (prod S (fst p) (fst q)) (fst r) =
  prod S (fst p) (prod S (fst q) (fst r))
by (rule semigroup.assoc [OF S])
moreover have prod T (prod T (snd p) (snd q)) (snd r) =
  prod T (snd p) (prod T (snd q) (snd r))
by (rule semigroup.assoc [OF T])
ultimately
show prod (semigroup-product S T) (prod (semigroup-product S T) p q) r =
  prod (semigroup-product S T) p (prod (semigroup-product S T) q r)
by (simp add: semigroup-product-def)
```

qed

The above proof is fairly easy, but obscured by the lack of concrete syntax. In fact, we didn’t make use of the infrastructure of locales, apart from the raw predicate definition of *semigroup*.

The alternative version below uses local context expressions to achieve a succinct proof body. The resulting statement is exactly the same as before, even though its specification is a bit more complex.

lemma

```
includes semigroup S + semigroup T
fixes U (structure)
defines U  $\equiv$  semigroup-product S T
shows semigroup U
```

proof

```

fix p q r :: 'a × 'b
have (fst p ·1 fst q) ·1 fst r = fst p ·1 (fst q ·1 fst r)
  by (rule S.assoc)
moreover have (snd p ·2 snd q) ·2 snd r = snd p ·2 (snd q ·2 snd r)
  by (rule T.assoc)
ultimately show (p ·3 q) ·3 r = p ·3 (q ·3 r)
  by (simp add: U-def semigroup-product-def semigroup.defs)
qed

```

Direct products of group structures may be defined in a similar manner, taking two further operations into account. Subsequently, we use high-level record operations to convert between different signature types explicitly; see also [6, §8.3].

```

constdefs
group-product :: 'a group ⇒ 'b group ⇒ ('a × 'b) group
group-product G H ≡
  semigroup.extend
    (semigroup-product (semigroup.truncate G) (semigroup.truncate H))
    (group.fields (λp. (inv G (fst p), inv H (snd p))) (one G, one H))

```

```

lemma group-product-aux:
includes group G + group H
fixes I (structure)
defines I ≡ group-product G H
shows group I

```

```

proof
show semigroup I
proof –
  let ?G' = semigroup.truncate G and ?H' = semigroup.truncate H
  have prod (semigroup-product ?G' ?H') = prod I
    by (simp add: I-def group-product-def group.defs
      semigroup-product-def semigroup.defs)
  moreover
  have semigroup ?G' and semigroup ?H'
    using prems by (simp-all add: semigroup-def semigroup.defs G.assoc H.assoc)
  then have semigroup (semigroup-product ?G' ?H') ..
  ultimately show ?thesis by (simp add: I-def semigroup-def)

```

qed

```

show group-axioms I

```

```

proof
fix p :: 'a × 'b
have (fst p)-11 ·1 fst p = 11
  by (rule G.left-inv)
moreover have (snd p)-12 ·2 snd p = 12
  by (rule H.left-inv)
ultimately show p-13 ·3 p = 13
  by (simp add: I-def group-product-def group.defs
    semigroup-product-def semigroup.defs)
have 11 ·1 fst p = fst p by (rule G.left-one)

```

```

    moreover have  $1_2 \cdot_2 \text{snd } p = \text{snd } p$  by (rule  $H.\text{left-one}$ )
    ultimately show  $1_3 \cdot_3 p = p$ 
      by (simp add:  $I\text{-def group-product-def group.defs}$ 
           $\text{semigroup-product-def semigroup.defs}$ )
    qed
  qed

theorem group-product: group  $G \implies$  group  $H \implies$  group (group-product  $G H$ )
  by (rule group-product-aux) (assumption | rule group.axioms)+

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 anti-sym [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 ⊆ x ∧ i ⊆ y ∧ (∀ z. z ⊆ x ∧ z ⊆ y ⟶ z ⊆ 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 anti-sym)
  from inf' show i ⊆ i'
proof (rule is-inf-greatest)
  from inf show i ⊆ x ..
  from inf show i ⊆ y ..
qed

```

```

    from inf show  $i' \sqsubseteq i$ 
  proof (rule is-inf-greatest)
    from inf' show  $i' \sqsubseteq x$  ..
    from inf' show  $i' \sqsubseteq y$  ..
  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  $(\text{THE } i. \text{is-inf } x \ y \ i) = i$ 
    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  $\text{is-inf } x \ y \ (x \sqcap y)$ 
    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?]:
   $\text{is-sup } x \ y \ s \implies (x \sqsubseteq s \implies y \sqsubseteq s \implies C) \implies C$ 
by (unfold is-sup-def) blast

theorem is-sup-uniq:  $\text{is-sup } x \ y \ s \implies \text{is-sup } x \ y \ s' \implies s = s'$ 
proof –
  assume sup:  $\text{is-sup } x \ y \ s$ 
  assume sup':  $\text{is-sup } x \ y \ s'$ 
  show ?thesis
  proof (rule anti-sym)
    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?]:  $x \sqsubseteq y \implies \text{is-sup } x \ y \ y$ 
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?]:  $\text{is-sup } x \ y \ s \implies x \sqcup y = s$ 
proof (unfold join-def)
  assume  $\text{is-sup } x \ y \ s$ 
  then show  $(\text{THE } s. \text{is-sup } x \ y \ s) = s$ 
  by (rule the-equality) (rule is-sup-uniq [OF -  $\langle \text{is-sup } x \ y \ s \rangle$ ])
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  $\dots \sqsubseteq y$  ..
      finally show ?thesis .
  qed
  show  $x \sqcap (y \sqcap z) \sqsubseteq z$ 
  proof –
    have  $x \sqcap (y \sqcap z) \sqsubseteq y \sqcap z$  ..
    also have  $\dots \sqsubseteq z$  ..
    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  $\dots \sqsubseteq x$  ..

```

```

    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  $\dots \sqsubseteq y$  ..
      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  $\dots \sqsubseteq x \sqcup (y \sqcup z)$  ..
      finally show ?thesis .
    qed
  qed
qed
show  $z \sqsubseteq x \sqcup (y \sqcup z)$ 
proof -
  have  $z \sqsubseteq y \sqcup z$  ..
  also have  $\dots \sqsubseteq x \sqcup (y \sqcup z)$  ..
  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  $\dots \sqsubseteq w$  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 $L1 = \text{join-commute meet-commute}$

lemmas $L2 = \text{join-assoc meet-assoc}$

lemmas $L4 = \text{join-meet-absorb meet-join-absorb}$

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** ($\text{simp add: } L4$)

also have $\dots = x \sqcup ((x \sqcap z) \sqcup (y \sqcap z))$ **by** ($\text{simp add: } L2$)

also have $\dots = x \sqcup ((x \sqcup y) \sqcap z)$ **by** ($\text{simp add: } L1\ \text{meet-distr}$)

also have $\dots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$ **by** ($\text{simp add: } L1\ L4$)

also have $\dots = (x \sqcup y) \sqcap (x \sqcup z)$ **by** ($\text{simp add: meet-distr}$)

finally show $?thesis$.

qed

end

locale $dlo = dpo +$

assumes $\text{total: } x \sqsubseteq y \mid y \sqsubseteq x$

begin

lemma $\text{less-total: } x \sqsubseteq y \mid x = y \mid y \sqsubseteq x$

using total

by (unfold less-def) blast

end

interpretation $dlo < dlat$

proof unfold-locales

fix $x\ y$

from total **have** $\text{is-inf } x\ y$ ($\text{if } x \sqsubseteq y \text{ then } x \text{ else } y$) **by** ($\text{auto simp: is-inf-def}$)

then show $EX\ \text{inf. is-inf } x\ y\ \text{inf}$ **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

```

```

interpretation dlo < dlat
proof unfold-locales
  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 z  $\sqsubseteq$  x
    from c have ?l = y  $\sqcup$  z
      by (metis c join-connection2 join-related2 meet-connection meet-related2
total)
    also from c have ... = ?r by (metis meet-related2)
    finally have ?l = ?r . }
  moreover
  { assume c: x  $\sqsubseteq$  y | x  $\sqsubseteq$  z
    from c have ?l = x
      by (metis join-connection2 join-related2 meet-connection total trans)
    also from c have ... = ?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 <= :: [int, int] => bool]
  where (dpo.less (op <=) (x::int) y) = (x < y)

```

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

```

proof -
  show dpo (op <= :: [int, int] => bool)
    by unfold-locales auto
  then interpret int: dpo [op <= :: [int, int] => bool] .

```

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

```

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

```

```

thm int.circular

```

```

lemma  $\llbracket (x::int) \leq y; y \leq z; z \leq x \rrbracket \implies x = y \wedge y = z$ 
  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 ( $op \leq :: [int, int] \Rightarrow bool$ )
    apply unfold-locales
    apply (unfold int.is-inf-def int.is-sup-def)
    apply arith+
    done
  then interpret int: dlat [ $op \leq :: [int, int] \Rightarrow bool$ ] .

```

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

```

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

```

```

interpretation int: dlo [ $op \leq :: [int, int] \Rightarrow bool$ ]
  by unfold-locales 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 \leq on nat

interpretation $\text{nat}: \text{dpo} [\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool}]$
where $\text{dpo.less} (\text{op} \leq) (x::\text{nat}) y = (x < y)$

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

proof –
show $\text{dpo} (\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool})$
by *unfold-locales auto*
then interpret $\text{nat}: \text{dpo} [\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool}]$.

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

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

interpretation $\text{nat}: \text{dlat} [\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool}]$
where $\text{dlat.meet} (\text{op} \leq) (x::\text{nat}) y = \min x y$
and $\text{dlat.join} (\text{op} \leq) (x::\text{nat}) y = \max x y$

proof –
show $\text{dlat} (\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool})$
apply *unfold-locales*
apply (*unfold nat.is-inf-def nat.is-sup-def*)
apply *arith+*
done
then interpret $\text{nat}: \text{dlat} [\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool}]$.

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

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

interpretation $\text{nat}: \text{dlo} [\text{op} \leq :: [\text{nat}, \text{nat}] \Rightarrow \text{bool}]$
by *unfold-locales 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 dlat

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)
    by unfold-locales (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 lcm-dvd2 lcm-least)
    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 lcm-dvd2 lcm-least)
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
  apply (rule nat-dvd.meet-left) done

```

```

print-interps dpo

```

```

print-interps dlat

```

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 ==>  $\exists y. y ** x = one$ 
  by (unfold unit-def) auto

lemma unit-r-inv-ex:
  unit x ==>  $\exists y. x ** y = one$ 
  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  $G$ :  $\text{unit } x$ 
  then show  $x ** y = x ** z$  by simp
qed

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

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

lemma unit-inv-comm:
  assumes  $\text{inv}: x ** y = \text{one}$ 
  and  $G$ :  $\text{unit } x$   $\text{unit } y$ 
  shows  $y ** x = \text{one}$ 
proof -
  from  $G$  have  $x ** y ** x = x ** \text{one}$  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]:  $\text{Dmonoid.unit } (op **) \text{one } x$ 

begin

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

lemma r-inv-ex [simp]:
   $\exists y. x ** y = \text{one}$ 
  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  $y ** (x ** inv\ x) = z ** (x ** inv\ x)$ 
    by (simp add: assoc [symmetric] del: r-inv)
  then show  $y = z$  by simp
qed simp

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

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

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

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

```

```

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

lemma inv-equality:
   $y ** x = one \implies inv\ x = y$ 
  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 = Dgrp prod (infixl ** 65) one + Dgrp sum (infixl +++ 60) 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) by unfold-locales (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  $(g \circ f) x = (g \circ f) y$  by simp
    with id2 show  $x = y$  by simp
  qed
next
  show surj f
  proof (rule surjI)
    fix x
    from id1 have  $(f \circ g) x = x$  by simp
    then show  $f (g x) = x$  by simp
  qed
qed
next
  fix f
  assume bij: bij f
  then
    have  $inv: f \circ Hilbert-Choice.inv f = id \ \& \ Hilbert-Choice.inv f \circ f = id$ 
      by (simp add: bij-def surj-iff inj-iff)
    show  $EX g. f \circ g = id \ \& \ g \circ f = id$  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) by unfold-locales (simp-all add: o-assoc)
  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] inv-def)
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 = (∀ x y z.
      times M (times M x y) z = times M x (times M y z) ∧
      times M (one M) x = x ∧ times M x (one M) = x)

```

```

definition
  group :: (| times :: 'a => 'a => 'a, one :: 'a, inv :: 'a => 'a, ... :: 'b |) => bool
where
  group G = (monoid G ∧ (∀ x. times G (inv G x) x = one G))

```

```

definition
  reverse :: (| times :: 'a => 'a => 'a, one :: 'a, ... :: 'b |) =>
    (| times :: 'a => 'a => 'a, one :: 'a, ... :: 'b |) where
    reverse M = M (| times := λ x y. times M y x |)

```

```

end

```

14 Binary arithmetic examples

theory *BinEx* **imports** *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 $2*y + 3*z + 6*w + 2*y + 3*z + 2*u = 2*y' + 3*z' + 6*w' + 2*y'$
 $+ 3*z' + u + (vv::int)$
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 $!!a::int. [a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k;$

$k \leq l$ $||$
 $==> a \leq l$
by *arith*

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

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

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

lemma $!!a::int.$ $[[a+b+c+d \leq i+j+k+l; a \leq b; b \leq c; c \leq d; i \leq j; j \leq k;$
 $k \leq l]]$
 $==> 6*a \leq 5*l+i$
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 ≠ 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) ≤ (1000001 + 1) - 2
  by simp

lemma (1234567::int) ≤ 1234567
  by simp

No integer overflow!

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

Quotient and Remainder

lemma (10::int) div 3 = 3
  by simp

lemma (10::int) mod 3 = 1
  by simp

A negative divisor

lemma (10::int) div -3 = -4
  by simp

lemma (10::int) mod -3 = -2
  by simp

A negative dividend1

```

¹The definition agrees with mathematical convention and with ML, but not with the

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

A few bigger examples

lemma $(8452::int) \text{ mod } 3 = 1$
 by *simp*

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

lemma $(1000006::int) \text{ mod } 10 = 6$
 by *simp*

Division by shifting

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

lemma $10000001 \text{ mod } 2 = (1::int)$
 by *simp*

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

lemma $10000055 \text{ mod } 32 = (23::int)$
 by *simp*

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

lemma $100094 \text{ mod } 144 = (14::int)$
 by *simp*

Powers

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

hardware of most computers

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 $Suc\ 9999 = 10000$
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 $(10000::nat) \text{ mod } 9 = 1$
by *simp*

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

lemma $(10000::nat) \text{ mod } 16 = 0$
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

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  $0x0A + 0x10 = (0x1A :: nat)$  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::nat) by simp
lemma 0b0000 = (0::int) by simp

end

```

16 Antiquotations

theory *Antiquote* **imports** *Main* **begin**

A simple example on quote / antiquote in higher-order abstract syntax.

```

syntax
  -Expr :: 'a => 'a                                (EXPR - [1000] 999)

constdefs
  var :: 'a => ('a => nat) => nat                    (VAR - [1000] 999)
  var x env == env x

  Expr :: (('a => nat) => nat) => ('a => nat) => nat
  Expr exp env == exp env

parse-translation << [Syntax.quote-antiquote-tr -Expr var Expr] >>
print-translation << [Syntax.quote-antiquote-tr' -Expr var 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)
term Expr (λenv. f env)
term Expr (λenv. f env + env x)
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 (-antiquote, -) \$ (t as Const (-antiquote, -) \$ -)) =
 skip-antiquote-tr i t
 | antiquote-tr i (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 (-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 [(-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.

consts

eqv :: $'a \Rightarrow 'a \Rightarrow \text{bool}$ (infixl \sim 50)

axclass *partial-equiv* < type

partial-equiv-sym [elim?]: $x \sim y \implies y \sim x$

partial-equiv-trans [trans]: $x \sim y \implies y \sim z \implies x \sim z$

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 :: $'a :: \text{partial-equiv set}$ **where**
domain = $\{x. x \sim x\}$

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*: $x \sim y$

also from *xy* **have** $y \sim x$..

finally show $x \sim x$.

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.

defs (overloaded)

eqv-fun-def: $f \sim g \implies \forall x \in \text{domain}. \forall y \in \text{domain}. x \sim y \implies f x \sim g y$

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 fun :: (partial-equiv, partial-equiv) partial-equiv
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 ∈ domain and y: y ∈ domain
    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 ~ h
show f ~ h
proof
  fix x y :: 'a
  assume x: x ∈ domain and y: y ∈ domain and x ~ y
  with fg have f x ~ g y ..
  also from y have y ~ y ..
  with gh y y have g y ~ h y ..
  finally show f x ~ h y .
qed
qed

```

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.

```

axclass equiv < partial-equiv
  eqv-refl [intro]: x ~ x

```

On total equivalences all elements are reflexive, and congruence holds unconditionally.

```

theorem equiv-domain [intro]: (x::'a::equiv) ∈ domain
proof
  show x ~ x ..
qed

```

```

theorem equiv-cong [dest?]: f ~ g ==> x ~ y ==> f x ~ g (y::'a::equiv)
proof –
  assume f ~ g
  moreover have x ∈ domain ..
  moreover have y ∈ domain ..
  moreover assume x ~ 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 ~ x} | a::'a. True }
by blast
```

```
lemma quotI [intro]: {x. a ~ x} ∈ quot
unfolding quot-def by blast
```

```
lemma quotE [elim]: R ∈ quot ==> (!!a. R = {x. a ~ x} ==> C) ==> 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   ([_]) where
[a] = Abs-quot {x. a ~ x}
```

```
theorem quot-rep: ∃ a. A = [a]
```

```
proof (cases A)
```

```
  fix R assume R: A = Abs-quot R
  assume R ∈ quot then have ∃ a. R = {x. a ~ x} by blast
  with R have ∃ a. A = Abs-quot {x. a ~ x} by blast
  then show ?thesis by (unfold eqv-class-def)
```

qed

```
lemma quot-cases [cases type: quot]:
```

```
  obtains (rep) a where A = [a]
  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 ~ b ==> [a] = [b]
```

```
proof -
```

```
  assume ab: a ~ b
```

```
  have {x. a ~ x} = {x. b ~ x}
```

```
  proof (rule Collect-cong)
```

```
    fix x show (a ~ x) = (b ~ x)
```

```
  proof
```

```
    from ab have b ~ a ..
```

```
    also assume a ~ x
```

```
    finally show b ~ 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  $\text{Abs-quot } \{x. a \sim x\} = \text{Abs-quot } \{x. b \sim x\}$ 
  then have  $\{x. a \sim x\} = \{x. b \sim x\}$  by (simp only: Abs-quot-inject quotI)
  moreover assume  $a \in \text{domain}$  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  $a \in \text{domain}$  ..
qed

lemma eqv-class-eq' [simp]:  $a \in \text{domain} \implies (\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$ 
  using eqv-class-eqI eqv-class-eqD' by blast

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

$\text{pick} :: 'a::\text{partial-equiv} \text{ quot} \Rightarrow 'a$ where
 $\text{pick } A = (\text{SOME } a. A = \lfloor a \rfloor)$

theorem $\text{pick-eqv}' [\text{intro?}, \text{simp}]: a \in \text{domain} \implies \text{pick } \lfloor a \rfloor \sim a$

proof (unfold pick-def)
 assume $a: a \in \text{domain}$
 show $(\text{SOME } x. \lfloor a \rfloor = \lfloor x \rfloor) \sim a$
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 $\text{pick-eqv} [\text{intro}, \text{simp}]: \text{pick } \lfloor a \rfloor \sim (a::'a::equiv)$

```

proof (rule pick-eqv')
  show  $a \in \text{domain}$  ..
qed

theorem pick-inverse:  $\lfloor \text{pick } A \rfloor = (A :: 'a :: \text{equiv } \text{quot})$ 
proof (cases  $A$ )
  fix  $a$  assume  $a: A = \lfloor a \rfloor$ 
  then have  $\text{pick } A \sim a$  by simp
  then have  $\lfloor \text{pick } A \rfloor = \lfloor a \rfloor$  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*: $(\sum i=0..n. i)^2 = (\sum i=0..n::\text{nat}. i^3)$

proof(*induct n*)

case 0 **show** ?case **by** *simp*

next

case (*Suc n*)

have $(\sum i = 0.. \text{Suc } n. i)^2 =$

$(\sum i = 0..n. i^3) + (2 * (\sum i = 0..n. i) * (n+1) + (n+1)^2)$

(**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 + $(n+1)^3 = (\sum i=0.. \text{Suc } n. i^3)$ **by** *simp*

finally show ?case .

qed

Sum of fourth powers: three versions.

lemma *sum-of-fourth-powers*:

$30 * (\sum i=0..n. i * i * i * i) =$

$n * \text{Suc } n * \text{Suc } (2 * n) * (3 * n * n + 3 * n - 1)$

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*:

$30 * \text{int } (\sum i=0..<m. i * i * i * i) =$

$\text{int } m * (\text{int } m - 1) * (\text{int } (2 * m) - 1) *$

$(\text{int } (3 * m * m) - \text{int } (3 * m) - 1)$

by (*induct m*) (*simp-all add: int-mult*)

lemma *of-nat-sum-of-fourth-powers*:

$30 * \text{of-nat } (\sum i=0..<m. i * i * i * i) =$

$\text{of-nat } m * (\text{of-nat } m - 1) * (\text{of-nat } (2 * m) - 1) *$

$(\text{of-nat } (3 * m * m) - \text{of-nat } (3 * m) - (1::\text{int}))$

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 $\forall x. a \text{ dvd } A x \implies a \text{ dvd } (\sum_{x<n.} A x * D x)$

```

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) 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 $(3 \text{ dvd } (\sum x < nd. D x * 10^x)) = (3 \text{ dvd } (\sum x < nd. D x))$
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*.

```
consts nlen :: nat  $\Rightarrow$  nat
recdef nlen measure id
  nlen 0 = 0
  nlen x = 1 + nlen (x div 10)
```

The function *sumdig* returns the sum of all digits in some number *n*.

```
definition
  sumdig :: nat  $\Rightarrow$  nat where
  sumdig n = ( $\sum$  x < nlen n. n div 10x mod 10)
```

Some properties of these functions follow.

```
lemma nlen-zero:
  0 = nlen x  $\implies$  x = 0
by (induct x rule: nlen.induct) auto
```

```
lemma nlen-suc:
  Suc m = nlen n  $\implies$  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 = ( $\sum$  x < nlen m. (m div (10::nat)x mod 10) * 10x)
proof (induct nd  $\equiv$  nlen m arbitrary: m)
  case 0 thus ?case by (simp add: nlen-zero)
next
  case (Suc nd)
  hence IH:
    nd = nlen (m div 10)  $\implies$ 
    m div 10 = ( $\sum$  x < nd. m div 10 div 10x mod 10 * 10x)
    by blast
  have  $\exists$  c. m = 10*(m div 10) + c  $\wedge$  c < 10 by presburger
  then obtain c where mexp: m = 10*(m div 10) + c  $\wedge$  c < 10 ..
  then have cdef: c = m mod 10 by arith
  show m = ( $\sum$  x < nlen m. m div 10x mod 10 * 10x)
proof -
  from  $\langle$ Suc nd = nlen m $\rangle$ 
  have nd = nlen (m div 10) by (rule nlen-suc)
  with IH have
    m div 10 = ( $\sum$  x < nd. m div 10 div 10x mod 10 * 10x) by simp
  with mexp have
    m = 10*( $\sum$  x < nd. m div 10 div 10x mod 10 * 10x) + c by simp
```

```

also have
... = (∑ x < nd. m div 10 div 10^x 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 10^x mod 10 * 10^x) + c
by (simp only: setsum-shift-bounds-Suc-ivl)
(simp add: atLeast0LessThan)
also have
... = (∑ x < Suc nd. m div 10^x mod 10 * 10^x)
by (simp add: setsum-head-upt cdef)
also note (Suc nd = nlen m)
finally
show m = (∑ x < nlen m. m div 10^x mod 10 * 10^x) .
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 dvd n) = (3 dvd sumdig n)
proof (unfold sumdig-def)
  have n = (∑ x < nlen n. (n div (10::nat)^x mod 10) * 10^x)
  by (rule exp-exists)
  moreover
  have 3 dvd (∑ x < nlen n. (n div (10::nat)^x mod 10) * 10^x) =
    (3 dvd (∑ x < nlen n. n div 10^x mod 10))
  by (rule three-div-general)
  ultimately
  show 3 dvd n = (3 dvd (∑ x < nlen n. n div 10^x 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 $\sim\sim ((\sim P \multimap Q) \multimap (\sim P \multimap \sim Q) \multimap P)$
by *iprover*

lemma $(\sim\sim(P \multimap Q)) = (\sim\sim P \multimap \sim\sim Q)$
by *iprover*

lemma $(\sim\sim\sim P) = (\sim P)$
by *iprover*

lemma $\sim\sim((P \multimap Q \mid R) \multimap (P \multimap Q) \mid (P \multimap R))$
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 $P \dashv\dashv \sim \sim P$
by *iprover*

lemma $\sim\sim(\sim\sim P \dashrightarrow P)$
by *iprover*

lemma $\sim\sim P \ \& \ \sim\sim(P \dashv\dashv Q) \dashv\dashv \sim\sim Q$
by *iprover*

$$\begin{array}{l} \text{lemma } ((P=Q) \dashv\dashv P \& Q \& R) \& \\ \quad ((Q=R) \dashv\dashv P \& Q \& R) \& \\ \quad ((R=P) \dashv\dashv P \& Q \& R) \dashv\dashv P \& Q \& R \\ \text{by } iprover \end{array}$$

lemma $((P=Q) \dashrightarrow P \& Q \& R \& S \& T) \&$
 $((Q=R) \dashrightarrow P \& Q \& R \& S \& T) \&$
 $((R=S) \dashrightarrow P \& Q \& R \& S \& T) \&$
 $((S=T) \dashrightarrow P \& Q \& R \& S \& T) \&$
 $((T=P) \dashrightarrow P \& Q \& R \& S \& T) \dashrightarrow P \& Q \& R \& S \& T$
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 $\sim\sim(P \mid \sim P)$
by *iprover*

lemma $\sim\sim(P \mid \sim\sim\sim P)$
by *iprover*

lemma $\sim\sim((P \multimap Q) \multimap P) \multimap P$
by *iprover*

lemma $((P|Q) \& (\sim P|Q) \& (P|\sim Q)) \multimap \sim (\sim P|\sim Q)$
by *iprover*

lemma $(Q \multimap R) \multimap (R \multimap P \& Q) \multimap (P \multimap (Q|R)) \multimap (P=Q)$
by *iprover*

lemma $P=P$
by *iprover*

lemma $\sim\sim((P = Q) = R) = (P = (Q = R))$
by *iprover*

lemma $((P = Q) = R) \multimap \sim\sim(P = (Q = R))$
by *iprover*

lemma $(P|(Q \& R)) = ((P|Q) \& (P|R))$
by *iprover*

lemma $\sim\sim((P = Q) = ((Q|\sim P) \& (\sim Q|P)))$
by *iprover*

lemma $\sim\sim((P \multimap Q) = (\sim P|Q))$
by *iprover*

lemma $\sim\sim((P \multimap Q) |(Q \multimap P))$
by *iprover*

lemma $\sim\sim(((P \& (Q \multimap R)) \multimap S) = ((\sim P|Q|S) \& (\sim P|\sim R|S)))$
oops

lemma $(P \& Q) = (P = (Q = (P|Q)))$
by *iprover*

lemma $(EX\ x.\ P(x) \dashrightarrow Q) \dashrightarrow (ALL\ x.\ P(x)) \dashrightarrow Q$
by *iprover*

lemma $((ALL\ x.\ P(x)) \dashrightarrow Q) \dashrightarrow \sim (ALL\ x.\ P(x) \ \& \ \sim Q)$
by *iprover*

lemma $((ALL\ x.\ \sim P(x)) \dashrightarrow Q) \dashrightarrow \sim (ALL\ x.\ \sim (P(x) | Q))$
by *iprover*

lemma $(ALL\ x.\ P(x)) | Q \dashrightarrow (ALL\ x.\ P(x) | Q)$
by *iprover*

lemma $(EX\ x.\ P \dashrightarrow Q(x)) \dashrightarrow (P \dashrightarrow (EX\ x.\ Q(x)))$
by *iprover*

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 $\sim\sim ((ALL\ x.\ P | Q(x)) = (P | (ALL\ x.\ Q(x))))$
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 $((EX\ x.\ P(x)) \& (EX\ y.\ Q(y))) \dashrightarrow$
 $((ALL\ x.\ (P(x) \dashrightarrow R(x))) \& (ALL\ y.\ (Q(y) \dashrightarrow S(y)))) =$
 $(ALL\ x\ y.\ ((P(x) \& Q(y)) \dashrightarrow (R(x) \& S(y))))$
by *iprover*

lemma $(ALL\ x.\ (P(x) \mid Q(x)) \dashrightarrow \sim R(x)) \&$
 $(ALL\ x.\ (Q(x) \dashrightarrow \sim S(x)) \dashrightarrow P(x) \& R(x))$
 $\dashrightarrow (ALL\ x.\ \sim \sim S(x))$
by *iprover*

lemma $\sim (EX\ x.\ P(x) \& (Q(x) \mid R(x))) \&$
 $(EX\ x.\ L(x) \& P(x)) \&$
 $(ALL\ x.\ \sim R(x) \dashrightarrow M(x))$
 $\dashrightarrow (EX\ x.\ L(x) \& M(x))$
by *iprover*

lemma $(ALL\ x.\ P(x) \& (Q(x) \mid R(x)) \dashrightarrow S(x)) \&$
 $(ALL\ x.\ S(x) \& R(x) \dashrightarrow L(x)) \&$
 $(ALL\ x.\ M(x) \dashrightarrow R(x))$
 $\dashrightarrow (ALL\ x.\ P(x) \& M(x) \dashrightarrow L(x))$
by *iprover*

lemma $(ALL\ x.\ \sim \sim (P(a) \& (P(x) \dashrightarrow P(b)) \dashrightarrow P(c))) =$
 $(ALL\ x.\ \sim \sim ((\sim P(a) \mid P(x) \mid P(c)) \& (\sim P(a) \mid \sim P(b) \mid P(c))))$
oops

lemma

$(ALL\ x.\ EX\ y.\ J\ x\ y) \ \&$
 $(ALL\ x.\ EX\ y.\ G\ x\ y) \ \&$
 $(ALL\ x\ y.\ J\ x\ y \mid G\ x\ y \dashrightarrow (ALL\ z.\ J\ y\ z \mid G\ y\ z \dashrightarrow H\ x\ z))$
 $\dashrightarrow (ALL\ x.\ EX\ y.\ H\ x\ y)$
by *iprover*

lemma $\sim (EX\ x.\ ALL\ y.\ F\ y\ x = (\sim F\ y\ y))$

by *iprover*

lemma $(EX\ y.\ ALL\ x.\ F\ x\ y = F\ x\ x) \dashrightarrow$

$\sim (ALL\ x.\ EX\ y.\ ALL\ z.\ F\ z\ y = (\sim F\ z\ x))$

by *iprover*

lemma $(ALL\ x.\ f(x) \dashrightarrow$

$(EX\ y.\ g(y) \ \& \ h\ x\ y \ \& \ (EX\ y.\ g(y) \ \& \ \sim h\ x\ y))) \ \&$

$(EX\ x.\ j(x) \ \& \ (ALL\ y.\ g(y) \dashrightarrow h\ x\ y))$

$\dashrightarrow (EX\ x.\ j(x) \ \& \ \sim f(x))$

by *iprover*

lemma $(a=b \mid c=d) \ \& \ (a=c \mid b=d) \dashrightarrow a=d \mid b=c$

by *iprover*

lemma $((EX\ z\ w.\ (ALL\ x\ y.\ (P\ x\ y = ((x = z) \ \& \ (y = w))))) \dashrightarrow$

$(EX\ z.\ (ALL\ x.\ (EX\ w.\ ((ALL\ y.\ (P\ x\ y = (y = w))) = (x = z)))))$

by *iprover*

lemma $((EX\ z\ w.\ (ALL\ x\ y.\ (P\ x\ y = ((x = z) \ \& \ (y = w))))) \dashrightarrow$

$(EX\ w.\ (ALL\ y.\ (EX\ z.\ ((ALL\ x.\ (P\ x\ y = (x = z))) = (y = w)))))$

by *iprover*

lemma $(ALL\ x.\ (EX\ y.\ P(y) \ \& \ x=f(y)) \dashrightarrow P(x)) = (ALL\ x.\ P(x) \dashrightarrow$

$P(f(x)))$

by *iprover*

lemma $P\ (f\ a\ b)\ (f\ b\ c) \ \& \ P\ (f\ b\ c)\ (f\ a\ c) \ \&$

$(ALL\ x\ y\ z.\ P\ x\ y \ \& \ P\ y\ z \dashrightarrow P\ x\ z) \dashrightarrow P\ (f\ a\ b)\ (f\ a\ c)$

by *iprover*

```

lemma  $ALL\ x.\ P\ x\ (f\ x) = (EX\ y.\ (ALL\ z.\ P\ z\ y \dashrightarrow P\ z\ (f\ x)) \ \&\ P\ x\ y)$ 
  by iprover

end

```

22 CTL formulae

theory *CTL* **imports** *Main* **begin**

We formalize basic concepts of Computational Tree Logic (CTL) [4, 3] 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 operation 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 $\mathcal{M} :: ('a \times 'a)\ set$

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 [4].

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 $lfp-gfp$: $lfp \ f = - \ gfp \ (\lambda s.::'a \ set. \ - \ (f \ (- \ s)))$

proof

show $lfp \ f \subseteq - \ gfp \ (\lambda s. \ - \ f \ (- \ s))$

proof

fix x **assume** l : $x \in lfp \ f$

show $x \in - \ gfp \ (\lambda s. \ - \ f \ (- \ s))$

proof

assume $x \in gfp \ (\lambda s. \ - \ f \ (- \ s))$

then obtain u **where** $x \in u$ **and** $u \subseteq - \ f \ (- \ u)$

by (*auto simp add: gfp-def Sup-set-eq*)

then have $f \ (- \ u) \subseteq - \ u$ **by** *auto*

then have $lfp \ f \subseteq - \ u$ **by** (*rule lfp-lowerbound*)

from l **and this have** $x \notin u$ **by** *auto*

with $\langle x \in u \rangle$ **show** *False* **by** *contradiction*

qed

qed

show $- \ gfp \ (\lambda s. \ - \ f \ (- \ s)) \subseteq lfp \ f$

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 $- \ u \subseteq gfp \ (\lambda s. \ - \ f \ (- \ s))$ **by** (*rule gfp-upperbound*)

then show $- \ gfp \ (\lambda s. \ - \ f \ (- \ s)) \subseteq u$ **by** *auto*

qed

qed

lemma $lfp-gfp'$: $- \ lfp \ f = gfp \ (\lambda s.::'a \ set. \ - \ (f \ (- \ s)))$

by (*simp add: lfp-gfp*)

lemma $gfp-lfp'$: $- \ gfp \ f = lfp \ (\lambda s.::'a \ set. \ - \ (f \ (- \ s)))$

by (*simp add: lfp-gfp*)

in order to give dual fixed point representations of $\text{AF } p$ and $\text{AG } p$:

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 $AG\ p$.

lemma *AG-I*: $q \subseteq p \cap AX\ q \implies q \subseteq AG\ p$
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*: $AX\ (p \cap q) = AX\ p \cap AX\ q$ **by** *auto*

lemma *AX-mono*: $p \subseteq q \implies AX\ p \subseteq AX\ q$ **by** *auto*

lemma *AG-mono*: $p \subseteq q \implies AG\ p \subseteq AG\ q$
by (*simp only: AG-gfp, rule gfp-mono*) *auto*

The formula $AG\ p$ implies $AX\ p$ (we use substitution of \subseteq with monotonicity).

lemma *AG-AX*: $AG\ p \subseteq AX\ p$

proof –

have $AG\ p \subseteq AX\ AG\ p$ **by** (*rule AG-fp-2*)

also have $AG\ p \subseteq p$ **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*: $AG\ AG\ p = AG\ p$

proof

show $AG\ AG\ p \subseteq AG\ p$ **by** (*rule AG-fp-1*)

next

show $AG\ p \subseteq AG\ AG\ p$

proof (*rule AG-I*)

have $AG\ p \subseteq AG\ p$..

moreover have $AG\ p \subseteq AX\ AG\ p$ **by** (*rule AG-fp-2*)

ultimately show $AG\ p \subseteq AG\ p \cap AX\ AG\ p$..

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*: $p \cap AG\ (p \rightarrow AX\ p) = AG\ p$

proof

show $p \cap AG\ (p \rightarrow AX\ p) \subseteq AG\ p$ (*is ?lhs \subseteq -*)


```

proof (rule AG-I)
  show  $?lhs \subseteq p \cap AX \ ?lhs$ 
  proof
    show  $?lhs \subseteq p$  ..
    show  $?lhs \subseteq AX \ ?lhs$ 
    proof –
      {
        have  $AG (p \rightarrow AX \ p) \subseteq p \rightarrow AX \ p$  by (rule AG-fp-1)
        moreover have  $p \cap p \rightarrow AX \ p \subseteq AX \ p$  ..
        ultimately have  $?lhs \subseteq AX \ p$  by auto
      }
    moreover
      {
        have  $p \cap AG (p \rightarrow AX \ p) \subseteq AG (p \rightarrow AX \ p)$  ..
        also have  $\dots \subseteq AX \ \dots$  by (rule AG-fp-2)
        finally have  $?lhs \subseteq AX \ AG (p \rightarrow AX \ p)$  .
      }
    ultimately have  $?lhs \subseteq AX \ p \cap AX \ AG (p \rightarrow AX \ p)$ 
      by (rule Int-greatest)
    also have  $\dots = AX \ ?lhs$  by (simp only: AX-int)
    finally show  $?thesis$  .
  qed
qed
qed
next
  show  $AG \ p \subseteq p \cap AG (p \rightarrow AX \ p)$ 
  proof
    show  $AG \ p \subseteq p$  by (rule AG-fp-1)
    show  $AG \ p \subseteq AG (p \rightarrow AX \ p)$ 
    proof –
      have  $AG \ p = AG \ AG \ p$  by (simp only: AG-AG)
      also have  $AG \ p \subseteq AX \ p$  by (rule AG-AX) moreover note AG-mono
      also have  $AX \ p \subseteq (p \rightarrow AX \ p)$  .. 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  $AG \ AX \ p = AX \ p \cap AX \ AG \ AX \ p$  by (rule AG-fp)
  also have  $\dots = AX \ (p \cap AG \ AX \ p)$  by (simp only: AX-int)
  also have  $p \cap AG \ AX \ p = AG \ p$  (is  $?lhs = -$ )
  proof
    have  $AX \ p \subseteq p \rightarrow AX \ p$  ..
  
```

```

    also have  $p \cap \text{AG } (p \rightarrow \text{AX } p) = \text{AG } p$  by (rule AG-induct)
    also note Int-mono AG-mono
    ultimately show  $?lhs \subseteq \text{AG } p$  by fast
next
  have  $\text{AG } p \subseteq p$  by (rule AG-fp-1)
  moreover
  {
    have  $\text{AG } p = \text{AG } \text{AG } p$  by (simp only: AG-AG)
    also have  $\text{AG } p \subseteq \text{AX } p$  by (rule AG-AX)
    also note AG-mono
    ultimately have  $\text{AG } p \subseteq \text{AG } \text{AX } p$  .
  }
  ultimately show  $\text{AG } p \subseteq ?lhs$  ..
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:

fast_arith_tac is a very basic version of the tactic. It performs no meta-to-object-logic conversion, and only some splitting of operators. **simple_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 `LinArith.lin_arith_simproc`), which—for performance reasons—however does even less splitting than **fast_arith_tac** at the moment (namely inequalities only). (On the other hand, it does take apart conjunctions, which **fast_arith_tac** currently does not do.)

23.1 Splitting of Operators: *max*, *min*, *abs*, *op −*, *nat*, *op mod*, *op div*

```

lemma ( $i::\text{nat}$ )  $\leq \max i j$ 
  by (tactic  $\ll$  fast-arith-tac @{context} 1  $\gg$ )

```

```

lemma ( $i::\text{int}$ )  $\leq \max i j$ 
  by (tactic  $\ll$  fast-arith-tac @{context} 1  $\gg$ )

```

lemma $\min i j \leq (i::nat)$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $\min i j \leq (i::int)$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $\min (i::nat) j \leq \max i j$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $\min (i::int) j \leq \max i j$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $\min (i::nat) j + \max i j = i + j$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $\min (i::int) j + \max i j = i + j$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(i::nat) < j \implies \min i j < \max i j$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(i::int) < j \implies \min i j < \max i j$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(0::int) \leq \text{abs } i$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(i::int) \leq \text{abs } i$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $\text{abs } (\text{abs } (i::int)) = \text{abs } i$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

Also testing subgoals with bound variables.

lemma $!!x. (x::nat) \leq y \implies x - y = 0$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $!!x. (x::nat) - y = 0 \implies x \leq y$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $!!x. ((x::nat) \leq y) = (x - y = 0)$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) < y; d < 1] \implies x - y = d$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) < y; d < 1] \implies x - y - x = d - x$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

```

lemma (x::int) < y ==> x - y < 0
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma nat (i + j) <= nat i + nat j
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma i < j ==> nat (i - j) = 0
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma (i::nat) mod 0 = i

  apply (subst nat-numeral-0-eq-0 [symmetric])
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma (i::nat) mod 1 = 0

  apply (subst nat-numeral-1-eq-1 [symmetric])
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma (i::nat) mod 42 <= 41
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma (i::int) mod 0 = i

  apply (subst numeral-0-eq-0 [symmetric])
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma (i::int) mod 1 = 0

  apply (subst numeral-1-eq-1 [symmetric])

  apply (tactic << lin-arith-pre-tac @{context} 1 >>)
oops

lemma (i::int) mod 42 <= 41

  apply (tactic << lin-arith-pre-tac @{context} 1 >>)
oops

lemma -(i::int) * 1 = 0 ==> i = 0
  by (tactic << fast-arith-tac @{context} 1 >>)

lemma [] (0::int) < abs i; abs i * 1 < abs i * j [] ==> 1 < abs i * j
  by (tactic << fast-arith-tac @{context} 1 >>)

```

23.2 Meta-Logic

```

lemma x < Suc y == x <= y
  by (tactic << simple-arith-tac @{context} 1 >>)

```

lemma $((x::nat) == z ==> x \sim y) ==> x \sim y \mid z \sim y$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

23.3 Various Other Examples

lemma $(x < Suc\ y) = (x <= y)$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) < y; y < z] ==> x < z$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(x::nat) < y \ \& \ y < z ==> x < z$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

This example involves no arithmetic at all, but is solved by preprocessing (i.e. NNF normalization) alone.

lemma $(P::bool) = Q ==> Q = P$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

lemma $[P = (x = 0); (\sim P) = (y = 0)] ==> \min\ (x::nat)\ y = 0$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

lemma $[P = (x = 0); (\sim P) = (y = 0)] ==> \max\ (x::nat)\ y = x + y$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) \sim y; a + 2 = b; a < y; y < b; a < x; x < b] ==> False$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) > y; y > z; z > x] ==> False$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(x::nat) - 5 > y ==> y < x$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(x::nat) \sim 0 ==> 0 < x$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) \sim y; x <= y] ==> x < y$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(x::nat) < y; P\ (x - y)] ==> P\ 0$
by (*tactic* \ll *simple-arith-tac* $@\{context\}$ 1 \gg)

lemma $(x - y) - (x::nat) = (x - x) - y$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $[(a::nat) < b; c < d] ==> (a - b) = (c - d)$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $((a::nat) - (b - (c - (d - e)))) = (a - (b - (c - (d - e))))$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(n < m \ \& \ m < n' \mid (n < m \ \& \ m = n') \mid (n < n' \ \& \ n' < m) \mid$
 $(n = n' \ \& \ n' < m) \mid (n = m \ \& \ m < n') \mid$
 $(n' < m \ \& \ m < n) \mid (n' < m \ \& \ m = n) \mid$
 $(n' < n \ \& \ n < m) \mid (n' = n \ \& \ n < m) \mid (n' = m \ \& \ m < n) \mid$
 $(m < n \ \& \ n < n') \mid (m < n \ \& \ n' = n) \mid (m < n' \ \& \ n' < n) \mid$
 $(m = n \ \& \ n < n') \mid (m = n' \ \& \ n' < n) \mid$
 $(n' = m \ \& \ m = (n::nat))$

oops

lemma $2 * (x::nat) \sim = 1$

oops

Constants.

lemma $(0::nat) < 1$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(0::int) < 1$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(47::nat) + 11 < 08 * 15$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

lemma $(47::int) + 11 < 08 * 15$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

Splitting of inequalities of different type.

lemma $[(a::nat) \sim = b; (i::int) \sim = j; a < 2; b < 2] ==>$
 $a + b <= nat \ (max \ (abs \ i) \ (abs \ j))$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

Again, but different order.

lemma $[(i::int) \sim = j; (a::nat) \sim = b; a < 2; b < 2] ==>$
 $a + b <= nat \ (max \ (abs \ i) \ (abs \ j))$
by (*tactic* \ll *fast-arith-tac* $@\{context\}$ 1 \gg)

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 :: ('a => 'b) => ('a bt => 'b bt)
preorder :: 'a bt => 'a list
inorder :: 'a bt => 'a list
postorder :: 'a bt => 'a list
append :: 'a bt => 'a bt => 'a 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 Sorting: Basic Theory

```

theory Sorting
imports Main Multiset
begin

consts
  sorted1:: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool
  sorted :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool

primrec
  sorted1 le [] = True
  sorted1 le (x#xs) = ((case xs of [] => True | y#ys => le x y) &
    sorted1 le xs)

primrec
  sorted le [] = True
  sorted le (x#xs) = (( $\forall$  y  $\in$  set xs. le x y) & sorted le xs)

definition
  total :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  bool where
    total r = ( $\forall$  x y. r x y | r y x)

definition
  transf :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  bool where
    transf f = ( $\forall$  x y z. f x y & f y z  $\longrightarrow$  f x z)

lemma sorted1-is-sorted: transf(le)  $\implies$  sorted1 le xs = sorted le xs
apply(induct xs)

```

```

  apply simp
  apply (simp split: list.split)
  apply (unfold transf-def)
  apply (blast)
done

lemma sorted-append [simp]:
  sorted le (xs@ys) =
    (sorted le xs & sorted le ys & ( $\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{le } x \ y$ ))
  by (induct xs) auto

end

```

26 Merge Sort

```

theory MergeSort
imports Sorting
begin

consts merge :: ('a::linorder)list * 'a list  $\Rightarrow$  'a list

recdef merge measure (%(xs,ys). size xs + size ys)
  merge(x#xs, y#ys) =
    (if  $x \leq y$  then  $x \# \text{merge}(xs, y\#ys)$  else  $y \# \text{merge}(x\#xs, ys)$ )

  merge(xs,[]) = xs

  merge([],ys) = ys

lemma multiset-of-merge[simp]:
  multiset-of (merge(xs,ys)) = multiset-of xs + multiset-of ys
  apply (induct xs ys rule: merge.induct)
  apply (auto simp: union-ac)
done

lemma set-merge[simp]:  $\text{set}(\text{merge}(xs,ys)) = \text{set } xs \cup \text{set } ys$ 
  apply (induct xs ys rule: merge.induct)
  apply auto
done

lemma sorted-merge[simp]:
  sorted ( $op \leq$ ) (merge(xs,ys)) = (sorted ( $op \leq$ ) xs & sorted ( $op \leq$ ) ys)
  apply (induct xs ys rule: merge.induct)
  apply (simp-all add: ball-Un linorder-not-le order-less-le)
  apply (blast intro: order-trans)
done

consts msort :: ('a::linorder) list  $\Rightarrow$  'a list

```

```

recdef msort measure size
  msort [] = []
  msort [x] = [x]
  msort xs = merge(msort(take (size xs div 2) xs),
                  msort(drop (size xs div 2) xs))

theorem sorted-msort: sorted (op ≤) (msort xs)
by (induct xs rule: msort.induct) simp-all

theorem multiset-of-msort: multiset-of (msort xs) = multiset-of xs
apply (induct xs rule: msort.induct)
  apply simp-all
apply (subst union-commute)
apply (simp del:multiset-of-append add:multiset-of-append[symmetric] union-assoc)
apply (simp add: union-ac)
done

end

```

27 A question from “Bundeswettbewerb Mathematik”

```

theory Puzzle imports Main begin

consts f :: nat => nat

specification (f)
  f-ax [intro!]: f(f(n)) < f(Suc(n))
  by (rule exI [of - id], simp)

lemma lemma0 [rule-format]: ∀ n. k=f(n) --> n <= f(n)
apply (induct-tac k rule: nat-less-induct)
apply (rule allI)
apply (rename-tac i)
apply (case-tac i)
  apply simp
apply (blast intro!: Suc-leI intro: le-less-trans)
done

lemma lemma1: n <= f(n)
by (blast intro: lemma0)

lemma f-mono [rule-format (no-asm)]: m <= n --> f(m) <= f(n)
apply (induct-tac n)
  apply simp
apply (metis f-ax le-SucE le-trans lemma0 nat-le-linear nat-less-le)

```

```

done

lemma f-id: f(n) = n
apply (rule order-antisym)
apply (rule-tac [2] lemma1)
apply (blast intro: leI dest: leD f-mono Suc-leI)
done

end

```

28 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 $sq :: 'a::times \Rightarrow 'a$ **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 $\langle\langle$
Delsimprocs [*ab-group-add-cancel.sum-conv*, *ab-group-add-cancel.rel-conv*]
 $\rangle\rangle$

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 add: sq-def ring-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 add: sq-def ring-simps)

end

```

29 Groebner Basis Examples

```

theory Groebner-Examples
imports Groebner-Basis
begin

```

29.1 Basic examples

```

lemma 3 ^ 3 == (?X::'a::{number-ring,recpower})
  by sring-norm

```

```

lemma (x - (-2)) ^ 5 == ?X::int
  by sring-norm

```

```

lemma (x - (-2)) ^ 5 * (y - 78) ^ 8 == ?X::int
  by sring-norm

```

```

lemma ((-3) ^ (Suc (Suc (Suc 0)))) == (X::'a::{number-ring,recpower})
  apply (simp only: power-Suc power-0)
  apply (simp only: comp-arith)
  oops

```

```

lemma ((x::int) + y) ^ 3 - 1 = (x - z) ^ 2 - 10 ==> x = z + 3 ==> 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 $x::'a::\{idom,recpower,number-ring\}$
shows $x^2*y = x^2 \ \& \ x*y^2 = y^2 \longleftrightarrow x=1 \ \& \ y=1 \mid x=0 \ \& \ y=0$
by *algebra*

29.2 Lemmas for Lagrange's theorem

definition
 $sq :: 'a::times \Rightarrow 'a$ **where**
 $sq \ x == x*x$

lemma
fixes $x1 :: 'a::\{idom,recpower,number-ring\}$
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 $p1 :: 'a::\{idom,recpower,number-ring\}$
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*)

29.3 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

30 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-const-inj: $t\text{-const}(c1) = t\text{-const}(c2) \implies c1 = c2$
t-fun-inj: $t11 \rightarrow t12 = t21 \rightarrow t22 \implies t11 = t21 \ \& \ t12 = t22$

t-ind:

$$\begin{aligned} & [\quad !!p. P(t\text{-const}\ p); !!t1\ t2. P(t1) \implies P(t2) \implies P(t\text{-fun}\ t1\ t2) \] \\ & \implies P(t) \end{aligned}$$

v-const-inj: $v\text{-const}(c1) = v\text{-const}(c2) \implies c1 = c2$

v-clos-inj:

$v\text{-clos}(<|ev1, e1, ve1|>) = v\text{-clos}(<|ev2, e2, ve2|>) \implies$
 $ev1 = ev2 \ \& \ e1 = e2 \ \& \ ve1 = ve2$

v-disj-const-clos: $\sim v\text{-const}(c) = v\text{-clos}(cl)$

ve-dom-owr: $ve\text{-dom}(ve + \{ev \mid\!-\!> v\}) = ve\text{-dom}(ve) \cup \{ev\}$

ve-app-owr1: $ve\text{-app}(ve + \{ev \mid\!-\!> v\}) \text{ ev} = v$

ve-app-owr2: $\sim ev1 = ev2 \implies ve\text{-app}(ve + \{ev1 \mid\!-\!> v\}) \text{ ev2} = ve\text{-app } ve \text{ ev2}$

te-dom-owr: $te\text{-dom}(te + \{ev \mid\!=> t\}) = te\text{-dom}(te) \cup \{ev\}$

te-app-owr1: $te\text{-app}(te + \{ev \mid\!=> t\}) \text{ ev} = t$

te-app-owr2: $\sim ev1 = ev2 \implies te\text{-app}(te + \{ev1 \mid\!=> t\}) \text{ ev2} = te\text{-app } te \text{ ev2}$

defs

eval-fun-def:

$eval\text{-fun}(s) ==$

{ *pp*.

($? \text{ ve } c. \text{ pp} = ((ve, e\text{-const}(c)), v\text{-const}(c))$) |

($? \text{ ve } x. \text{ pp} = ((ve, e\text{-var}(x)), ve\text{-app } ve \ x) \ \& \ x : ve\text{-dom}(ve)$) |

($? \text{ ve } e \ x. \text{ pp} = ((ve, fn \ x \ => \ e), v\text{-clos}(<|x, e, ve|>))$) |

($? \text{ ve } e \ x \ f \ cl.$

$\text{ pp} = ((ve, fix \ f(x) = e), v\text{-clos}(cl)) \ \&$

$cl = <|x, e, ve + \{f \mid\!-\!> v\text{-clos}(cl)\}|>$

) |

($? \text{ ve } e1 \ e2 \ c1 \ c2.$

$\text{ pp} = ((ve, e1 \ @\!@ \ e2), v\text{-const}(c\text{-app } c1 \ c2)) \ \&$

$((ve, e1), v\text{-const}(c1)) : s \ \& \ ((ve, e2), v\text{-const}(c2)) : s$

) |

```

( ? 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: $ve \mid- e \dashrightarrow v == ((ve,e),v):eval-rel$

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: $te \mid- e \implies t == ((te,e),t):elab-rel$

isof-env-def:
 $ve \ isofenv \ te ==$
 $ve-dom(ve) = te-dom(te) \ \&$
(! x .
 $x:ve-dom(ve) \dashrightarrow$
(? c . $ve-app \ ve \ x = v-const(c) \ \& \ c \ isof \ te-app \ te \ x$)
)

axioms

isof-app: $[\mid c1 \ isof \ t1->t2; \ c2 \ isof \ t1 \mid] \implies c-app \ c1 \ c2 \ isof \ t2$

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 <<
val infsys-mono-tac = REPEAT (ares-tac (@{thms basic-monos} @ [allI, impI])
1)
>>

```

```

lemma infsys-p1: P a b ==> P (fst (a,b)) (snd (a,b))
by simp

```

```

lemma infsys-p2: P (fst (a,b)) (snd (a,b)) ==> P a b
by simp

```

```

lemma infsys-pp1: P a b c ==> 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)) ==> P
a b c
by simp

```

```

lemma lfp-intro2: [| mono(f); x:f(lfp(f)) |] ==> x:lfp(f)
apply (rule subsetD)
apply (rule lfp-lemma2)
apply assumption+
done

```

```

lemma lfp-elim2:
  assumes lfp: x:lfp(f)
    and mono: mono(f)
    and r: !!y. y:f(lfp(f)) ==> 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: mono(f)
    and r: !!y. y:f(lfp(f)) Int {x. P(x)} ==> 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 gfp: 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  $e\text{-injs} = e\text{-const-inj } e\text{-var-inj } e\text{-fn-inj } e\text{-fix-inj } e\text{-app-inj}$ 

```

```

lemmas  $e\text{-disjs} =$ 
   $e\text{-disj-const-var}$ 
   $e\text{-disj-const-fn}$ 
   $e\text{-disj-const-fix}$ 
   $e\text{-disj-const-app}$ 
   $e\text{-disj-var-fn}$ 
   $e\text{-disj-var-fix}$ 
   $e\text{-disj-var-app}$ 
   $e\text{-disj-fn-fix}$ 
   $e\text{-disj-fn-app}$ 
   $e\text{-disj-fix-app}$ 

```

```

lemmas  $e\text{-disj-si} = e\text{-disjs } e\text{-disjs [symmetric]}$ 

```

```

lemmas  $e\text{-disj-se} = e\text{-disj-si [THEN notE]}$ 

```

```

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  $eval\text{-fun-mono}$ :  $mono(eval\text{-fun})$ 
unfolding  $mono\text{-def } eval\text{-fun-def}$ 
apply (tactic  $infsys\text{-mono-tac}$ )

```

done

lemma *eval-const*: $ve \vdash e\text{-const}(c) \dashv\dashv\rightarrow v\text{-const}(c)$
unfolding *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-var2*:
 $ev:ve\text{-dom}(ve) \implies ve \vdash e\text{-var}(ev) \dashv\dashv\rightarrow ve\text{-app } ve \ ev$
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-fn*:
 $ve \vdash fn \ ev \Rightarrow e \dashv\dashv\rightarrow v\text{-clos}(<|ev,e,ve|>)$
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-fix*:
 $cl = <| ev1, e, ve + \{ev2 \mid\rightarrow v\text{-clos}(cl)\} |> \implies$
 $ve \vdash fix \ ev2(ev1) = e \dashv\dashv\rightarrow v\text{-clos}(cl)$
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*:
 $[| ve \vdash e1 \dashv\dashv\rightarrow v\text{-const}(c1); ve \vdash e2 \dashv\dashv\rightarrow v\text{-const}(c2) |] \implies$
 $ve \vdash e1 \ @\@ \ e2 \dashv\dashv\rightarrow v\text{-const}(c\text{-app } c1 \ c2)$
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*:

$$\begin{aligned} & \llbracket ve \mid e1 \dashrightarrow v\text{-clos}(<|xm,em,vem|>); \\ & \quad ve \mid e2 \dashrightarrow v2; \\ & \quad vem + \{xm \mid \rightarrow v2\} \mid em \dashrightarrow v \\ & \rrbracket ==> \\ & \quad ve \mid e1 \ @\@ e2 \dashrightarrow v \end{aligned}$$

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*:

$$\begin{aligned} & \llbracket ve \mid e \dashrightarrow v; \\ & \quad !!ve\ c.\ P(((ve,e\text{-const}(c)),v\text{-const}(c))); \\ & \quad !!ev\ ve.\ ev:ve\text{-dom}(ve) ==> P(((ve,e\text{-var}(ev)),ve\text{-app}\ ve\ ev)); \\ & \quad !!ev\ ve\ e.\ P(((ve,fn\ ev ==> e),v\text{-clos}(<|ev,e,ve|>))); \\ & \quad !!ev1\ ev2\ ve\ cl\ e. \\ & \quad \quad cl = <| ev1, e, ve + \{ev2 \mid \rightarrow v\text{-clos}(cl)\} |> ==> \\ & \quad \quad P(((ve,fix\ ev2(ev1) = e),v\text{-clos}(cl))); \\ & \quad !!ve\ c1\ c2\ e1\ e2. \\ & \quad \quad \llbracket P(((ve,e1),v\text{-const}(c1))); P(((ve,e2),v\text{-const}(c2))) \rrbracket ==> \\ & \quad \quad P(((ve,e1\ @\@ e2),v\text{-const}(c\text{-app}\ c1\ c2))); \\ & \quad !!ve\ vem\ xm\ e1\ e2\ em\ v\ v2. \\ & \quad \quad \llbracket P(((ve,e1),v\text{-clos}(<|xm,em,vem|>))); \\ & \quad \quad \quad P(((ve,e2),v2)); \\ & \quad \quad \quad P(((vem + \{xm \mid \rightarrow v2\},em),v)) \\ & \quad \rrbracket ==> \\ & \quad \quad P(((ve,e1\ @\@ e2),v)) \\ & \rrbracket ==> \\ & \quad P(((ve,e),v)) \end{aligned}$$

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*:

$$\begin{aligned} & \llbracket ve \mid e \dashrightarrow v; \\ & \quad !!ve\ c.\ P\ ve\ (e\text{-const}\ c)\ (v\text{-const}\ c); \\ & \quad !!ev\ ve.\ ev:ve\text{-dom}(ve) ==> P\ ve\ (e\text{-var}\ ev)\ (ve\text{-app}\ ve\ ev); \end{aligned}$$

```

!!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) ==> te |- e-var(x) ==> 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 | => ty1} |- e ==> ty2 ==> te |- fn x => e ==> ty1->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 | => ty1->ty2} + {x | => ty1} |- e ==> ty2 ==>
    te |- fix f(x) = e ==> ty1->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 |- e1 ==> ty1->ty2; te |- e2 ==> ty1 |] ==>
    te |- e1 @@ e2 ==> 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 |- e ==> t
    and 2: !!te c t. c isof t ==> 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 |- e ==> t
    and 2: !!te c t. c isof t ==> 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,fn x => e),t1->t2))
    and 5: !!te f x e t1 t2.
      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; te |- e2 ==> t1 |] ==>

```

```

      P(((te,e1 @@ 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:
  [| 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 (fn x => e) (t1->t2);
    !!te f x e t1 t2.
      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; te |- e2 ==> t1 |] ==>
        P te (e1 @@ e2) t2
  |] ==>
    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 |- e ==> t ==> (e = e-const(c) --> c 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 |- e-const(c) ==> t ==> c isof t
  apply (drule elab-const-elim-lem)
  apply blast
done

```

```

lemma elab-var-elim-lem:
  te |- e ==> t ==> (e = e-var(x) --> t=te-app te x & x:te-dom(te))
  apply (erule elab-elim)

```

apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-var-elim*: $te \mid - e\text{-var}(ev) \implies t \implies t = te\text{-app } te \text{ } ev \ \& \ ev :$
 $te\text{-dom}(te)$
apply (*drule elab-var-elim-lem*)
apply *blast*
done

lemma *elab-fn-elim-lem*:
 $te \mid - e \implies t \implies$
 $(e = fn \ x1 \Rightarrow e1 \ \longrightarrow$
 $(? \ t1 \ t2. \ t = t\text{-fun } t1 \ t2 \ \& \ te + \{x1 \mid \Rightarrow t1\} \mid - e1 \implies t2)$
 $)$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-fn-elim*: $te \mid - fn \ x1 \Rightarrow e1 \implies t \implies$
 $(? \ t1 \ t2. \ t = t1 \longrightarrow t2 \ \& \ te + \{x1 \mid \Rightarrow t1\} \mid - e1 \implies t2)$
apply (*drule elab-fn-elim-lem*)
apply *blast*
done

lemma *elab-fix-elim-lem*:
 $te \mid - e \implies t \implies$
 $(e = fix \ f(x) = e1 \ \longrightarrow$
 $(? \ t1 \ t2. \ t = t1 \longrightarrow t2 \ \& \ te + \{f \mid \Rightarrow t1 \longrightarrow t2\} + \{x \mid \Rightarrow t1\} \mid - e1 \implies t2))$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-fix-elim*: $te \mid - fix \ ev1(ev2) = e1 \implies t \implies$
 $(? \ t1 \ t2. \ t = t1 \longrightarrow t2 \ \& \ te + \{ev1 \mid \Rightarrow t1 \longrightarrow t2\} + \{ev2 \mid \Rightarrow t1\} \mid - e1 \implies$
 $t2)$
apply (*drule elab-fix-elim-lem*)
apply *blast*
done

lemma *elab-app-elim-lem*:
 $te \mid - e \implies t2 \implies$
 $(e = e1 \ @\@ \ e2 \ \longrightarrow \ (? \ t1 . te \mid - e1 \implies t1 \longrightarrow t2 \ \& \ te \mid - e2 \implies t1))$
apply (*erule elab-elim*)
apply (*fast intro!*: *e-disj-si elim!*: *e-disj-se dest!*: *e-injs*) +
done

lemma *elab-app-elim*: $te \mid - e1 \ @\@ \ e2 \implies t2 \implies (? \ t1 . te \mid - e1 \implies$
 $t1 \longrightarrow t2 \ \& \ te \mid - e2 \implies t1)$
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 isof t ==> v-const(c) hasty t
apply (unfold hasty-def)
apply (erule hasty-rel-const-coind)
done

```

```

lemma hasty-clos:
  te |- fn ev => e ==> t & ve hastysenv te ==> v-clos(<|ev,e,ve|>) 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:
   $v \text{ hasty } t \implies$ 
   $! x e ve.$ 
   $v=v\text{-clos}(<|x,e,ve|>) \dashrightarrow (? te. te \vdash \text{fn } x \Rightarrow e \implies t \ \& \ ve \text{ hastyenv } te)$ 
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:  $v\text{-clos}(<|ev,e,ve|>) \text{ hasty } t \implies$ 
   $? te. te \vdash \text{fn } ev \Rightarrow e \implies t \ \& \ ve \text{ hastyenv } te$ 
apply (drule hasty-elim-clos-lem)
apply blast
done

```

```

lemma hasty-env1:  $[| ve \text{ hastyenv } te; v \text{ hasty } t |] \implies$ 
   $ve + \{ev \mid\!-\!> v\} \text{ hastyenv } te + \{ev \mid\!-\!> t\}$ 
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:  $[| ve \text{ hastyenv } te ; te \vdash e\text{-const}(c) \implies t |] \implies$ 

```

```

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 |- e ----> v ==>
  (! t te. ve hastyenv te --> te |- e ==> t --> v 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 isofenv te ==> ve 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 (erule hasty-const)
apply (simp (no-asm-simp))
done

lemma basic-consistency:
  [| ve isofenv te; ve |- e ----> v-const(c); te |- e ===> t |] ==> c isof t
apply (rule hasty-elim-const)
apply (erule consistency)
apply (blast intro!: basic-consistency-lem)
done

end

```

31 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.

31.1 Basic definitions

```

datatype 'a trm =
  Var 'a
  | Const 'a
  | App 'a trm 'a trm (infix · 60)

```

types

$'a \text{ subst} = ('a \times 'a \text{ trm}) \text{ list}$

Applying a substitution to a variable:

```
fun assoc :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\times$  'b) list  $\Rightarrow$  '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  $\Rightarrow$  'a subst  $\Rightarrow$  'a trm (infixl  $\triangleleft$  60)
where
  (Var v)  $\triangleleft$  s = assoc v (Var v) s
| (Const c)  $\triangleleft$  s = (Const c)
| (M  $\cdot$  N)  $\triangleleft$  s = (M  $\triangleleft$  s)  $\cdot$  (N  $\triangleleft$  s)
```

Composition of substitutions:

```
fun
  compose :: 'a subst  $\Rightarrow$  'a subst  $\Rightarrow$  'a subst (infixl  $\cdot$  80)
where
  []  $\cdot$  bl = bl
| ((a,b) # al)  $\cdot$  bl = (a, b  $\triangleleft$  bl) # (al  $\cdot$  bl)
```

Equivalence of substitutions:

```
definition eqv (infix  $=_s$  50)
where
  s1  $=_s$  s2  $\equiv \forall t. t \triangleleft s1 = t \triangleleft s2$ 
```

31.2 Basic lemmas

```
lemma apply-empty[simp]: t  $\triangleleft$  [] = 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*

31.3 Specification: Most general unifiers

definition

Unifier $\sigma \ t \ u \equiv (t \triangleleft \sigma = u \triangleleft \sigma)$

definition

MGU $\sigma \ t \ u \equiv \text{Unifier } \sigma \ t \ u \wedge (\forall \vartheta. \text{Unifier } \vartheta \ t \ u \longrightarrow (\exists \gamma. \vartheta =_s \sigma \cdot \gamma))$

lemma *MGUI*[*intro*]:
 $\llbracket t \triangleleft \sigma = u \triangleleft \sigma; \bigwedge \vartheta. t \triangleleft \vartheta = u \triangleleft \vartheta \rrbracket \implies \exists \gamma. \vartheta =_s \sigma \cdot \gamma$
 $\implies \text{MGU } \sigma \ t \ u$
by (*simp only: Unifier-def MGU-def, auto*)

lemma *MGU-sym*[*sym*]:
 $\text{MGU } \sigma \ s \ t \implies \text{MGU } \sigma \ t \ s$
by (*auto simp: MGU-def Unifier-def*)

31.4 The unification algorithm

Occurs check: Proper subterm relation

fun *occ* :: $'a \ \text{trm} \Rightarrow 'a \ \text{trm} \Rightarrow \text{bool}$

where

$\text{occ } u \ (Var \ v) = False$
 $\mid \text{occ } u \ (Const \ c) = False$
 $\mid \text{occ } u \ (M \cdot N) = (u = M \vee u = N \vee \text{occ } u \ M \vee \text{occ } u \ N)$

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))
                           then None
                           else Some [(v, M  $\cdot$  N)])
| unify (Var v) M = (if (occ (Var v) M)
                     then None
                     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
  None  $\Rightarrow$  None |
  Some  $\vartheta$   $\Rightarrow$  (case unify (N  $\triangleleft$   $\vartheta$ ) (N'  $\triangleleft$   $\vartheta$ )
    of None  $\Rightarrow$  None |
    Some  $\sigma$   $\Rightarrow$  Some ( $\vartheta \cdot \sigma$ )))

by pat-completeness auto

```

31.5 Partial correctness

Some lemmas about occ and MGU:

```

lemma subst-no-occ:  $\neg \text{occ} \text{ (Var } v) \ t \implies \text{Var } v \neq t$ 
 $\implies t \triangleleft [(v, s)] = t$ 
by (induct t) auto

```

```

lemma MGU-Var[intro]:
  assumes no-occ:  $\neg \text{occ} \text{ (Var } v) \ t$ 
  shows MGU [(v,t)] (Var v) t
proof (intro MGUI exI)
  show Var v  $\triangleleft$  [(v,t)] = t  $\triangleleft$  [(v,t)] using no-occ
  by (cases Var v = t, auto simp:subst-no-occ)
next
  fix  $\vartheta$  assume th: 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  $M$   $N$  = 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  $M$   $M'$  = 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 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

```


31.6 Properties used in termination proof

The variables of a term:

```
fun vars-of:: 'a trm  $\Rightarrow$  'a set
where
  vars-of (Var v) = { v }
| vars-of (Const c) = {}
| vars-of (M · 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  $\equiv \forall t. v \notin \text{vars-of } (t \triangleleft \sigma)$ 
```

```
lemma elim-intro[intro]: ( $\bigwedge t. v \notin \text{vars-of } (t \triangleleft \sigma)$ )  $\implies$  elim  $\sigma$  v
by (auto simp:elim-def)
```

```
lemma elim-dest[dest]: elim  $\sigma$  v  $\implies v \notin \text{vars-of } (t \triangleleft \sigma)$ 
by (auto simp:elim-def)
```

```
lemma elim-equiv:  $\sigma =_s \vartheta \implies \text{elim } \sigma x = \text{elim } \vartheta x$ 
by (auto simp:elim-def equiv-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: (t = Var v) = ([ (v, t) ] =s [])
```

proof

```
assume t-v: t = Var v
```

```
thus [(v, t)] =s []
```

```
by auto
```

next

```
assume id: [(v, t)] =s []
```

```
show t = Var v
```

proof –

```
have t = Var v  $\triangleleft$  [(v, t)] by simp
```

```
also from id have ... = 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[symmetric])
  next
    assume  $\text{neg}: v \neq x$ 
    have  $\text{elim } [(v, \text{Var } x)] v$ 
      by (auto intro!: remove-var simp: neg)
    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[symmetric])
  with ih1 have  $\text{elim } [(v, M)] v$  by blast
  hence  $v \notin \text{vars-of } (\text{Var } v \triangleleft [(v, M)])$  ..
  hence not-in-M:  $v \notin \text{vars-of } M$  by simp

  from nonocc have  $\neg [(v, N)] =_s []$ 
    by (simp add: var-same[symmetric])
  with ih2 have  $\text{elim } [(v, N)] v$  by blast
  hence  $v \notin \text{vars-of } (\text{Var } v \triangleleft [(v, N)])$  ..
  hence not-in-N:  $v \notin \text{vars-of } N$  by simp

  have  $\text{elim } [(v, M \cdot N)] v$ 
  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 vars-of (t  $\triangleleft$   $\sigma$ )  $\subseteq$  vars-of M  $\cup$  vars-of N  $\cup$  vars-of t
  (is ?P M N  $\sigma$  t)
using assms
proof (induct M N arbitrary:  $\sigma$  t)
  case (3 c v)
  hence  $\sigma = [(v, \text{Const } c)]$  by simp
  thus ?case by (induct t) auto
next
  case (4 M N v)
  hence  $\neg \text{occ } (\text{Var } v) (M \cdot N)$  by (cases occ (Var v) (M · N), auto)
  with 4 have  $\sigma = [(v, M \cdot N)]$  by simp
  thus ?case by (induct t) auto
next
  case (5 v M)
  hence  $\neg \text{occ } (\text{Var } v) M$  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' = 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:  $\bigwedge t. ?P M M' \vartheta 1 t$ 
  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  $\in$  vars-of (t  $\triangleleft$   $\sigma$ )

  show v  $\in$  vars-of (M · N)  $\cup$  vars-of (M' · N')  $\cup$  vars-of t
  proof (cases v  $\notin$  vars-of M  $\wedge$  v  $\notin$  vars-of M'
     $\wedge$  v  $\notin$  vars-of N  $\wedge$  v  $\notin$  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 ( $\exists c v$ )
  have no-occ:  $\neg \text{occ } (\text{Var } v) (\text{Const } c)$  by simp
  with  $\exists$  have  $\sigma = [(v, \text{Const } c)]$  by simp
  with occ-elim[OF no-occ]
  show ?case by auto
next
  case ( $\lambda M N v$ )
  hence no-occ:  $\neg \text{occ } (\text{Var } v) (M \cdot N)$  by (cases occ (Var v) (M · N), auto)
  with  $\lambda$  have  $\sigma = [(v, M \cdot N)]$  by simp
  with occ-elim[OF no-occ]
  show ?case by auto
next
  case ( $\exists v M$ )
  hence no-occ:  $\neg \text{occ } (\text{Var } v) M$  by (cases occ (Var v) M, auto)
  with  $\exists$  have  $\sigma = [(v, M)]$  by simp
  with occ-elim[OF no-occ]
  show ?case by auto
next
  case ( $\exists c d$ ) thus ?case

```

```

    by (cases c = d) auto
next
case (γ M N M' N' σ)
then obtain ϑ1 ϑ2
  where unify M M' = Some ϑ1
  and unify (N < ϑ1) (N' < ϑ1) = Some ϑ2
  and σ: σ = ϑ1 · ϑ2
  and ih1: ?P M M' ϑ1
  and ih2: ?P (N < ϑ1) (N' < ϑ1) ϑ2
  by (auto split:option.split-asm)

from ⟨unify-dom (M · N, M' · N')⟩
have unify-dom (M, M')
  by (rule accp-downward) (rule unify-rel.intros)
hence no-new-vars:
  ∧ t. vars-of (t < ϑ1) ⊆ vars-of M ∪ vars-of M' ∪ vars-of t
  by (rule unify-vars) (rule ⟨unify M M' = Some ϑ1⟩)

from ih2 show ?case
proof
  assume ∃ v ∈ vars-of (N < ϑ1) ∪ vars-of (N' < ϑ1). elim ϑ2 v
  then obtain v
    where v ∈ vars-of (N < ϑ1) ∪ vars-of (N' < ϑ1)
    and el: elim ϑ2 v by auto
  with no-new-vars show ?thesis unfolding σ
    by (auto simp:elim-def)
next
  assume empty[simp]: ϑ2 =s []

  have σ =s (ϑ1 · []) unfolding σ
    by (rule compose-equiv) auto
  also have ... =s ϑ1 by auto
  finally have σ =s ϑ1 .

  from ih1 show ?thesis
  proof
    assume ∃ v ∈ vars-of M ∪ vars-of M'. elim ϑ1 v
    with elim-equiv[OF ⟨σ =s ϑ1⟩]
    show ?thesis by auto
  next
    note ⟨σ =s ϑ1⟩
    also assume ϑ1 =s []
    finally show ?thesis ..
  qed
qed
qed

```

31.7 Termination proof

```

termination unify
proof
  let  $?R = \text{measures } [\lambda(M, N). \text{card } (\text{vars-of } M \cup \text{vars-of } N),$ 
                                 $\lambda(M, N). \text{size } M]$ 
  show wf  $?R$  by simp

  fix  $M\ N\ M'\ N'$ 
  show  $((M, M'), (M \cdot N, M' \cdot N')) \in ?R$  — Inner call
    by (rule measures-lesseq) (auto intro: card-mono)

  fix  $\vartheta$  — Outer call
  assume inner: unify-dom  $(M, M')$ 
    unify  $M\ M' = \text{Some } \vartheta$ 

  from unify-eliminates [OF inner]
  show  $((N \triangleleft \vartheta, N' \triangleleft \vartheta), (M \cdot N, M' \cdot N')) \in ?R$ 
  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  $\text{vars-of } (N \triangleleft \vartheta) \cup \text{vars-of } (N' \triangleleft \vartheta)$ 
       $\subset \text{vars-of } (M \cdot N) \cup \text{vars-of } (M' \cdot N')$ 
      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

```

32 Some examples demonstrating the comm-ring method

```

theory Commutative-RingEx
imports Commutative-Ring

```

begin

lemma $4*(x::int)^5*y^3*x^2*3 + x*z + 3^5 = 12*x^7*y^3 + z*x + 243$
by *comm-ring*

lemma $((x::int) + y)^2 = x^2 + y^2 + 2*x*y$
by *comm-ring*

lemma $((x::int) + y)^3 = x^3 + y^3 + 3*x^2*y + 3*y^2*x$
by *comm-ring*

lemma $((x::int) - y)^3 = x^3 + 3*x*y^2 + (-3)*y*x^2 - y^3$
by *comm-ring*

lemma $((x::int) - y)^2 = x^2 + y^2 - 2*x*y$
by *comm-ring*

lemma $((a::int) + b + c)^2 = a^2 + b^2 + c^2 + 2*a*b + 2*b*c + 2*a*c$
by *comm-ring*

lemma $((a::int) - b - c)^2 = a^2 + b^2 + c^2 - 2*a*b + 2*b*c - 2*a*c$
by *comm-ring*

lemma $(a::int)*b + a*c = a*(b+c)$
by *comm-ring*

lemma $(a::int)^2 - b^2 = (a - b) * (a + b)$
by *comm-ring*

lemma $(a::int)^3 - b^3 = (a - b) * (a^2 + a*b + b^2)$
by *comm-ring*

lemma $(a::int)^3 + b^3 = (a + b) * (a^2 - a*b + b^2)$
by *comm-ring*

lemma $(a::int)^4 - b^4 = (a - b) * (a + b)*(a^2 + b^2)$
by *comm-ring*

lemma $(a::int)^{10} - b^{10} = (a - b) * (a^9 + a^8*b + a^7*b^2 + a^6*b^3 + a^5*b^4 + a^4*b^5 + a^3*b^6 + a^2*b^7 + a*b^8 + b^9)$
by *comm-ring*

end

33 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.

```
consts ack :: nat * nat => nat
recdef ack less-than <*<lex*> less-than
  ack (0, n) = Suc n
  ack (Suc m, 0) = ack (m, 1)
  ack (Suc m, Suc n) = ack (m, ack (Suc m, n))
```

```
consts list-add :: nat list => nat
primrec
  list-add [] = 0
  list-add (m # ms) = m + list-add ms
```

```
consts zeroHd :: nat list => nat
primrec
  zeroHd [] = 0
  zeroHd (m # ms) = m
```

The set of primitive recursive functions of type $\text{nat list} \Rightarrow \text{nat}$.

```
definition
  SC :: nat list => nat where
  SC l = Suc (zeroHd l)
```

```
definition
  CONSTANT :: nat => nat list => nat where
  CONSTANT k l = k
```

```
definition
  PROJ :: nat => nat list => nat where
  PROJ i l = zeroHd (drop i l)
```

```
definition
  COMP :: (nat list => nat) => (nat list => nat) list => nat list => nat where
  COMP g fs l = g (map ( $\lambda f$ . f l) fs)
```

```
definition
  PREC :: (nat list => nat) => (nat list => nat) => nat list => nat where
  PREC f g l =
    (case l of
      [] => 0
    | x # l' => 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 list => nat) => bool
  where
    SC: PRIMREC SC
  | CONSTANT: PRIMREC (CONSTANT k)
  | PROJ: PRIMREC (PROJ i)
  | COMP: PRIMREC g ==>  $\forall f \in \text{set fs. PRIMREC } f ==> \text{PRIMREC } (\text{COMP } g \text{ fs})$ 
  | PREC: PRIMREC f ==> PRIMREC g ==> PRIMREC (PREC f g)

```

Useful special cases of evaluation

```

lemma SC [simp]: SC (x # l) = Suc x
apply (simp add: SC-def)
done

```

```

lemma CONSTANT [simp]: CONSTANT k l = k
apply (simp add: CONSTANT-def)
done

```

```

lemma PROJ-0 [simp]: PROJ 0 (x # l) = x
apply (simp add: PROJ-def)
done

```

```

lemma COMP-1 [simp]: COMP g [f] l = g [f l]
apply (simp add: COMP-def)
done

```

```

lemma PREC-0 [simp]: PREC f g (0 # l) = f l
apply (simp add: PREC-def)
done

```

```

lemma PREC-Suc [simp]: PREC f g (Suc x # l) = g (PREC f g (x # l) # x # l)
apply (simp add: PREC-def)
done

```

PROPERTY A 4

```

lemma less-ack2 [iff]: j < ack (i, j)
apply (induct i j rule: ack.induct)
apply simp-all
done

```

PROPERTY A 5-, the single-step lemma

```

lemma ack-less-ack-Suc2 [iff]: ack(i, j) < ack (i, Suc j)
apply (induct i j rule: ack.induct)
apply simp-all
done

```

PROPERTY A 5, monotonicity for <

```

lemma ack-less-mono2: j < k ==> ack (i, j) < ack (i, k)

```

```

apply (induct i k rule: ack.induct)
  apply simp-all
apply (blast elim!: less-SucE intro: less-trans)
done

```

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]:  $\text{ack } (i, \text{Suc } j) \leq \text{ack } (\text{Suc } i, j)$ 
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]:  $\text{ack } (i, j) < \text{ack } (\text{Suc } i, j)$ 
  apply (blast intro: ack-less-mono2 less-le-trans)
done

```

PROPERTY A 4'? Extra lemma needed for *CONSTANT* case, constant functions

```

lemma less-ack1 [iff]:  $i < \text{ack } (i, j)$ 
  apply (induct i)
  apply simp-all
  apply (blast intro: Suc-leI le-less-trans)
done

```

PROPERTY A 8

```

lemma ack-1 [simp]:  $\text{ack } (\text{Suc } 0, j) = j + 2$ 
  apply (induct j)
  apply simp-all
done

```

PROPERTY A 9. The unary 1 and 2 in *ack* is essential for the rewriting.

```

lemma ack-2 [simp]:  $\text{ack } (\text{Suc } (\text{Suc } 0), j) = 2 * j + 3$ 
  apply (induct j)
  apply simp-all
done

```

PROPERTY A 7, monotonicity for $<$ [not clear why *ack-1* is now needed first!]

```

lemma ack-less-mono1-aux:  $ack\ i\ k < ack\ (Suc\ (i + i'), k)$ 
  apply (induct i k rule: ack.induct)
    apply simp-all
    prefer 2
    apply (blast intro: less-trans ack-less-mono2)
    apply (induct-tac i' n rule: ack.induct)
    apply simp-all
    apply (blast intro: Suc-leI [THEN le-less-trans] ack-less-mono2)
  done

```

```

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:  $i \leq j ==> ack\ i\ k \leq ack\ j\ k$ 
  apply (simp add: order-le-less)
  apply (blast intro: ack-less-mono1)
done

```

PROPERTY A 10

```

lemma ack-nest-bound:  $ack\ i1\ (ack\ i2\ j) < ack\ (2 + (i1 + i2), j)$ 
  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:  $ack\ i1\ j + ack\ i2\ j < ack\ (4 + (i1 + i2), j)$ 
  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:  $i < ack\ k\ j ==> i + j < ack\ (4 + k, j)$ 
  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

```

Inductive definition of the *PR* functions

MAIN RESULT

```

lemma SC-case: SC l < ack (1, list-add 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, list-add l)
by simp

```

```

lemma PROJ-case [rule-format]:  $\forall i. PROJ\ i\ l < ack\ (0, list-add\ l)$ 
apply (simp add: PROJ-def)
apply (induct l)
apply (auto simp add: drop-Cons split: nat.split)
apply (blast intro: less-le-trans le-add2)
done

```

COMP case

```

lemma COMP-map-aux:  $\forall f \in set\ fs. PRIMREC\ f \wedge (\exists kf. \forall l. f\ l < ack\ (kf,$ 
list-add l))
==>  $\exists k. \forall l. list-add\ (map\ (\lambda f. f\ l)\ fs) < ack\ (k, list-add\ l)$ 
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:

```

 $\forall l. g\ l < ack\ (kg, list-add\ l) ==>$ 
 $\forall f \in set\ fs. PRIMREC\ f \wedge (\exists kf. \forall l. f\ l < ack\ (kf, list-add\ l))$ 
==>  $\exists k. \forall l. COMP\ g\ fs\ l < ack\ (k, list-add\ l)$ 
apply (unfold COMP-def)
— Now, if meson tolerated map, we could finish with (drule COMP-map-aux,
meson ack-less-mono2 ack-nest-bound less-trans)
apply (erule COMP-map-aux [THEN exE])
apply (rule exI)
apply (rule allI)
apply (drule spec)+
apply (erule less-trans)
apply (blast intro: ack-less-mono2 ack-nest-bound less-trans)
done

```

PREC case

lemma *PREC-case-aux*:

```

  ∀ l. f l + list-add l < ack (kf, list-add l) ==>
    ∀ l. g l + list-add l < ack (kg, list-add l) ==>
      PREC f g l + list-add l < ack (Suc (kf + kg), list-add 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*:

```

  ∀ l. f l < ack (kf, list-add l) ==>
    ∀ l. g l < ack (kg, list-add l) ==>
      ∃ k. ∀ l. PREC f g l < ack (k, list-add l)
by (metis le-less-trans [OF le-add1 PREC-case-aux] ack-add-bound2)

```

lemma *ack-bounds-PRIMREC*: $\text{PRIMREC } f \implies \exists k. \forall l. f l < \text{ack } (k, \text{list-add } l)$

```

apply (erule PRIMREC.induct)
apply (blast intro: SC-case CONSTANT-case PROJ-case COMP-case PREC-case)+
done

```

lemma *ack-not-PRIMREC*: $\neg \text{PRIMREC } (\lambda l. \text{case } l \text{ of } [] \Rightarrow 0 \mid x \# l' \Rightarrow \text{ack } (x, x))$

```

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

34 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* &
 $(\forall y \in \text{pset } po. P\ y \longrightarrow (x, y): \text{order } po)$)

definition

greatest :: [*'a* => *bool*, *'a potype*] => *'a* **where**
greatest *P po* = (*SOME* *x. x: pset po* & *P x* &
 $(\forall y \in \text{pset } po. P\ y \longrightarrow (y, x): \text{order } po)$)

definition

lub :: [*'a set*, *'a potype*] => *'a* **where**
lub *S po* = *least* ($\%x. \forall y \in S. (y, x): \text{order } po$) *po*

definition

glb :: [*'a set*, *'a potype*] => *'a* **where**
glb *S po* = *greatest* ($\%x. \forall y \in S. (x, y): \text{order } po$) *po*

definition

isLub :: [*'a set*, *'a potype*, *'a*] => *bool* **where**
isLub *S po* = ($\%L. (L: \text{pset } po \ \& \ (\forall y \in S. (y, L): \text{order } po) \ \& \ (\forall z \in \text{pset } po. (\forall y \in S. (y, z): \text{order } po) \longrightarrow (L, z): \text{order } po))$)

definition

isGlb :: [*'a set*, *'a potype*, *'a*] => *bool* **where**
isGlb *S po* = ($\%G. (G: \text{pset } po \ \& \ (\forall y \in S. (G, y): \text{order } po) \ \& \ (\forall z \in \text{pset } po. (\forall y \in S. (z, y): \text{order } po) \longrightarrow (z, G): \text{order } po))$)

definition

$fix :: [('a \Rightarrow 'a), 'a\ set] \Rightarrow 'a\ set$ **where**
 $fix\ f\ A = \{x. x: A \ \&\ f\ x = x\}$

definition

$interval :: [('a * 'a)\ set, 'a, 'a] \Rightarrow 'a\ set$ **where**
 $interval\ r\ a\ b = \{x. (a,x): r \ \&\ (x,b): r\}$

definition

$Bot :: 'a\ potype \Rightarrow 'a$ **where**
 $Bot\ po = least\ (\%x. True)\ po$

definition

$Top :: 'a\ potype \Rightarrow 'a$ **where**
 $Top\ po = greatest\ (\%x. True)\ po$

definition

$PartialOrder :: ('a\ potype)\ set$ **where**
 $PartialOrder = \{P. refl\ (pset\ P)\ (order\ P) \ \&\ antisym\ (order\ P) \ \&\ trans\ (order\ P)\}$

definition

$CompleteLattice :: ('a\ potype)\ set$ **where**
 $CompleteLattice = \{cl. cl: PartialOrder \ \&\$
 $(\forall S. S \subseteq pset\ cl \longrightarrow (\exists L. isLub\ S\ cl\ L)) \ \&\$
 $(\forall S. S \subseteq pset\ cl \longrightarrow (\exists G. isGlb\ S\ cl\ G))\}$

definition

$CLF :: ('a\ potype * ('a \Rightarrow 'a))\ set$ **where**
 $CLF = (SIGMA\ cl: CompleteLattice.$
 $\{f. f: pset\ cl \rightarrow pset\ cl \ \&\ monotone\ f\ (pset\ cl)\ (order\ cl)\})$

definition

$induced :: ['a\ set, ('a * 'a)\ set] \Rightarrow ('a * 'a)\ set$ **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 \ \&\$
 $(| pset = S, order = induced\ S\ (order\ cl)\ |): CompleteLattice\})$

abbreviation

$sublat :: ['a\ set, 'a\ potype] \Rightarrow bool\ (- <=&= - [51,50]50)$ **where**
 $S <=&= cl == S : sublattice\ \text{“}\ \{cl\}$

definition

dual :: 'a potype => 'a potype **where**
dual po = (| *pset* = *pset po*, *order* = *converse (order po)* |)

locale (open) *PO* =
fixes *cl* :: 'a potype
and *A* :: 'a set
and *r* :: ('a * 'a) set
assumes *cl-po*: *cl* : *PartialOrder*
defines *A-def*: *A* == *pset cl*
and *r-def*: *r* == *order cl*

locale (open) *CL* = *PO* +
assumes *cl-co*: *cl* : *CompleteLattice*

locale (open) *CLF* = *CL* +
fixes *f* :: 'a => 'a
and *P* :: 'a set
assumes *f-cl*: (*cl*,*f*) : *CLF*
defines *P-def*: *P* == *fix f A*

locale (open) *Tarski* = *CLF* +
fixes *Y* :: 'a set
and *intY1* :: 'a set
and *v* :: 'a
assumes
Y-ss: *Y* ⊆ *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}*
(| *pset*=*intY1*, *order*=*induced intY1 r* |)

34.1 Partial Order

lemma (in *PO*) *PO-imp-refl*: *refl A r*
apply (*insert cl-po*)
apply (*simp add: PartialOrder-def A-def r-def*)
done

lemma (in *PO*) *PO-imp-sym*: *antisym r*
apply (*insert cl-po*)
apply (*simp add: PartialOrder-def r-def*)
done

lemma (in *PO*) *PO-imp-trans*: *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-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-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-converse
      trans-converse antisym-converse)
done

lemma Rdual:
   $\forall S. (S \subseteq A \dashrightarrow (\exists L. \text{isLub } S \text{ (| pset = A, order = r|) } L))$ 
   $\implies \forall S. (S \subseteq A \dashrightarrow (\exists G. \text{isGlb } S \text{ (| pset = A, order = r|) } G))$ 
apply safe
apply (rule-tac x = lub {y. y ∈ A & (∀ k ∈ S. (y, k) ∈ r)}
      (|pset = A, order = r|) in exI)
apply (drule-tac x = {y. y ∈ A & (∀ k ∈ S. (y,k) ∈ 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]

declare CL-imp-PO [THEN PO.PO-imp-refl, simp]
declare CL-imp-PO [THEN PO.PO-imp-sym, simp]
declare CL-imp-PO [THEN PO.PO-imp-trans, simp]

lemma (in CL) CO-refl:  $\text{refl } A \text{ r}$ 
by (rule PO-imp-refl)

```

```

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) interval-not-empty:
  [| trans r; interval r a b ≠ {} |] ==> (a, b) ∈ r
apply (simp add: interval-def)
apply (unfold trans-def, blast)
done

lemma (in PO) interval-imp-mem: x ∈ interval r a b ==> (a, x) ∈ r
by (simp add: interval-def)

lemma (in PO) left-in-interval:
  [| a ∈ A; b ∈ A; interval r a b ≠ {} |] ==> a ∈ 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

34.2 sublattice

lemma (**in** *PO*) *sublattice-imp-CL*:
 $S \leqslant cl \iff (| \text{pset} = S, \text{order} = \text{induced } S \ r |) \in \text{CompleteLattice}$
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*)

34.3 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*:
 $[[S \subseteq A; L \in A; \forall x \in S. (x, L) \in r]] \implies (\text{lub } S \ cl, L) \in r$
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*: $S \subseteq A \implies \text{lub } S \ cl \in A$

```

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:
  [|  $S \subseteq A$ ;  $L \in A$ ;  $\forall x \in S. (x, L) \in r$ ;
     $\forall z \in A. (\forall y \in S. (y, z) \in r) \longrightarrow (L, z) \in r$  |] ==>  $L = \text{lub } S \text{ cl}$ 
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: [|  $S \subseteq A$ ; isLub  $S \text{ cl } L$  |] ==>  $L = \text{lub } S \text{ cl}$ 
by (simp add: lubI isLub-def A-def r-def)

```

```

lemma (in CL) isLub-in-lattice: isLub  $S \text{ cl } L$  ==>  $L \in A$ 
by (simp add: isLub-def A-def)

```

```

lemma (in CL) isLub-upper: [|isLub  $S \text{ cl } L$ ;  $y \in S$ |] ==>  $(y, L) \in r$ 
by (simp add: isLub-def r-def)

```

```

lemma (in CL) isLub-least:
  [| isLub  $S \text{ cl } L$ ;  $z \in A$ ;  $\forall y \in S. (y, z) \in r$  |] ==>  $(L, z) \in r$ 
by (simp add: isLub-def A-def r-def)

```

```

lemma (in CL) isLubI:
  [|  $L \in A$ ;  $\forall y \in S. (y, L) \in r$ ;
     $(\forall z \in A. (\forall y \in S. (y, z) \in r) \longrightarrow (L, z) \in r)$  |] ==> isLub  $S \text{ cl } L$ 
by (simp add: isLub-def A-def r-def)

```

34.4 glb

```

lemma (in CL) glb-in-lattice:  $S \subseteq A$  ==> glb  $S \text{ cl} \in A$ 
apply (subst glb-dual-lub)
apply (simp add: A-def)
apply (rule dualA-iff [THEN subst])
apply (rule CL.lub-in-lattice)
apply (rule dualPO)
apply (rule CL-dualCL)
apply (simp add: dualA-iff)
done

```

```

lemma (in CL) glb-lower: [ $|S \subseteq A; x \in S| \implies (\text{glb } S \text{ } cl, x) \in r$ 
apply (subst glb-dual-lub)
apply (simp add: r-def)
apply (rule dualr-iff [THEN subst])
apply (rule CL.lub-upper)
apply (rule dualPO)
apply (rule CL-dualCL)
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  $\rightarrow$  pset cl & monotone f (pset cl) (order cl)
apply (insert f-cl)
apply (simp add: CLF-def)
done

```

```

declare (in CLF) f-cl [simp]

```

```

lemma (in CLF) f-in-funcset:  $f \in A \rightarrow A$ 
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)  $\in$  CLF
apply (simp add: CLF-def CL-dualCL monotone-dual)
apply (simp add: dualA-iff)
done

```

34.5 fixed points

```

lemma fix-subset:  $\text{fix } f \text{ } A \subseteq A$ 
by (simp add: fix-def, fast)

```

```

lemma fix-imp-eq:  $x \in \text{fix } f \text{ } A \implies f \text{ } x = x$ 
by (simp add: fix-def)

```

```

lemma fixf-subset:
  [ $|A \subseteq B; x \in \text{fix } (\%y. A. f \text{ } y) \text{ } A| \implies x \in \text{fix } f \text{ } B$ 
by (simp add: fix-def, auto)

```

34.6 lemmas for Tarski, lub

```

lemma (in CLF) lubH-le-flubH:
   $H = \{x. (x, f \text{ } x) \in r \text{ \& } x \in A\} \implies (\text{lub } H \text{ } cl, f \text{ } (\text{lub } H \text{ } cl)) \in r$ 

```

```

apply (rule lub-least, fast)
apply (rule f-in-funcset [THEN funcset-mem])
apply (rule lub-in-lattice, fast)
—  $\forall x:H. (x, f (\text{lub } H \ r)) \in r$ 
apply (rule ballI)
apply (rule transE)
— instantiates  $(x, ???z) \in \text{order } cl \text{ to } (x, f \ x)$ ,
— because of the def of  $H$ 
apply fast
— 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:
  [|  $H = \{x. (x, f \ x) \in r \ \& \ x \in A\}$  |] ==>  $(f (\text{lub } H \ cl), \text{lub } H \ cl) \in r$ 
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:
   $H = \{x. (x, f \ x) \in r \ \& \ x \in A\} ==> \text{lub } H \ cl \in \text{fix } f \ A$ 
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:
  [|  $H = \{x. (x, f \ x) \in r \ \& \ x \in A\}; \ x \in P$  |] ==>  $x \in H$ 
by (simp add: P-def fix-imp-eq [of -  $f \ A$ ] reflE CO-refl
      fix-subset [of  $f \ A$ , THEN subsetD])

```

```

lemma (in CLF) fixf-le-lubH:

```

```

       $H = \{x. (x, f x) \in r \ \& \ x \in A\} \implies \forall x \in \text{fix } f \ A. (x, \text{lub } H \ cl) \in r$ 
    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:
   $H = \{x. (x, f x) \in r \ \& \ x \in A\}$ 
   $\implies \forall L. (\forall y \in \text{fix } f \ A. (y, L) \in r) \longrightarrow (\text{lub } H \ cl, L) \in r$ 
  apply (rule allI)
  apply (rule impI)
  apply (erule bspec)
  apply (rule lubH-is-fixp, assumption)
done

```

34.7 Tarski fixpoint theorem 1, first part

```

lemma (in CLF) T-thm-1-lub:  $\text{lub } P \ cl = \text{lub } \{x. (x, f x) \in r \ \& \ x \in A\} \ cl$ 
  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:  $H = \{x. (f x, x) \in r \ \& \ x \in A\} \implies \text{glb } H \ cl \in P$ 
  — 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 dualPO)
  apply (rule CL-dualCL)
  apply (rule CLF-dual)
  apply (simp add: dualr-iff dualA-iff)
done

```

```

lemma (in CLF) T-thm-1-glb:  $\text{glb } P \ cl = \text{glb } \{x. (f x, x) \in r \ \& \ x \in A\} \ cl$ 
  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 dualPO CL-dualCL]
    dualPO CL-dualCL CLF-dual dualr-iff)
done

```

34.8 interval

```

lemma (in CLF) rel-imp-elem:  $(x, y) \in r \implies x \in A$ 
  apply (insert CO-refl)

```


apply (*simp add: refl-def, blast*)
done

lemma (*in CLF*) *interval-subset*: $[[a \in A; b \in A]] \implies \text{interval } r \ a \ b \subseteq A$
apply (*simp add: interval-def*)
apply (*blast intro: rel-imp-elem*)
done

lemma (*in CLF*) *intervalI*:
 $[[(a, x) \in r; (x, b) \in r]] \implies x \in \text{interval } r \ a \ b$
by (*simp add: interval-def*)

lemma (*in CLF*) *interval-lemma1*:
 $[[S \subseteq \text{interval } r \ a \ b; x \in S]] \implies (a, x) \in r$
by (*unfold interval-def, fast*)

lemma (*in CLF*) *interval-lemma2*:
 $[[S \subseteq \text{interval } r \ a \ b; x \in S]] \implies (x, b) \in r$
by (*unfold interval-def, fast*)

lemma (*in CLF*) *a-less-lub*:
 $[[S \subseteq A; S \neq \{\} ; \forall x \in S. (a, x) \in r; \forall y \in S. (y, L) \in r]] \implies (a, L) \in r$
by (*blast intro: transE*)

lemma (*in CLF*) *glb-less-b*:
 $[[S \subseteq A; S \neq \{\} ; \forall x \in S. (x, b) \in r; \forall y \in S. (G, y) \in r]] \implies (G, b) \in r$
by (*blast intro: transE*)

lemma (*in CLF*) *S-intv-cl*:
 $[[a \in A; b \in A; S \subseteq \text{interval } r \ a \ b]] \implies S \subseteq A$
by (*simp add: subset-trans [OF - interval-subset]*)

lemma (*in CLF*) *L-in-interval*:
 $[[a \in A; b \in A; S \subseteq \text{interval } r \ a \ b; S \neq \{\} ; \text{isLub } S \text{ cl } L; \text{interval } r \ a \ b \neq \{\}]] \implies L \in \text{interval } r \ a \ b$
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*)
 $— (L, b) \in r$
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]
    dualA-iff A-def dualPO CL-dualCL CLF-dual 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 ∈ S = False ==> everything
  apply fast
  — S ≠ {}
  apply simp
  — ∀ y:S. (y, L) ∈ induced (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)
  — ∀ z:interval r a b. (∀ y:S. (y, z) ∈ induced (interval r a b) r → (if S = {} then
    a else L, z) ∈ induced (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 (rule CO-trans)
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) intv-CL-glb = intv-CL-lub [THEN Rdual]

lemma (in CLF) interval-is-sublattice:
  [|  $a \in A$ ;  $b \in A$ ; interval  $r\ a\ b \neq \{\}$  |]
  ==> interval  $r\ a\ b <=<= cl$ 
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) interv-is-compl-latt =
  interval-is-sublattice [THEN sublattice-imp-CL]

```

34.9 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 ∈ 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 ∈ A
apply (simp add: Top-dual-Bot A-def)
apply (rule dualA-iff [THEN subst])
apply (blast intro!: CLF.Bot-in-lattice dualPO CL-dualCL CLF-dual)
done

```

```

lemma (in CLF) Top-prop: x ∈ A ==> (x, Top cl) ∈ 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 ∈ A ==> (Bot cl, x) ∈ r
apply (simp add: Bot-dual-Top r-def)
apply (rule dualr-iff [THEN subst])
apply (simp add: CLF.Top-prop [of - f]
               dualA-iff A-def dualPO CL-dualCL CLF-dual)
done

```

```

lemma (in CLF) Top-intv-not-empty: x ∈ A ==> interval r x (Top cl) ≠ {}
apply (rule notI)
apply (drule-tac a = Top cl in equals0D)
apply (simp add: interval-def)
apply (simp add: refl-def Top-in-lattice Top-prop)
done

```

```

lemma (in CLF) Bot-intv-not-empty: x ∈ A ==> interval r (Bot cl) x ≠ {}
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 (blast intro!: CLF.Top-in-lattice dualPO CL-dualCL CLF-dual)
apply (simp add: CLF.Top-intv-not-empty [of - f]
               dualA-iff A-def dualPO CL-dualCL CLF-dual)
done

```

34.10 fixed points form a partial order

```

lemma (in CLF) fixf-po: (| pset = P, order = induced P r|) ∈ PartialOrder
by (simp add: P-def fix-subset po-subset-po)

```

```

lemma (in Tarski) Y-subset-A: Y ⊆ 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)
—  $(f x, \text{Top cl}) \in r$ 
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-func:  $(\%x: \text{intY1}. f x) \in \text{intY1} \rightarrow \text{intY1}$ 
apply (rule restrictI)
apply (erule intY1-f-closed)

```

done

lemma (in Tarski) intY1-mono:
 monotone (%x: 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 fix-subset)
apply (rule intY1-subset)
apply (simp add: CLF.glbH-is-fixp [OF - intY1-is-cl, simplified]
 v-def CL-imp-PO intY1-is-cl CLF-def intY1-func 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)

```

—  $v \in P$ 
apply (simp add: v-in-P)
apply (rule conjI)
—  $v$  is lub
— 1.  $\forall y:Y. (y, v) \in \text{induced } P \ r$ 
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 - intY1-is-cl, simplified])
  apply (simp add: CL-imp-PO intY1-is-cl, force)
—  $v$  is LEAST ub
apply clarify
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 - 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:
  [| ( $\mid pset = A, order = r \mid$ )  $\in$  PartialOrder;
     $\forall S. S \subseteq A \longrightarrow (\exists L. isLub \ S \ (\mid pset = A, order = r \mid) \ L)$  |]
  ==> ( $\mid pset = A, order = r \mid$ )  $\in$  CompleteLattice
by (simp add: CompleteLatticeI Rdual)

theorem (in CLF) Tarski-full:
  ( $\mid pset = P, order = induced \ P \ r \mid$ )  $\in$  CompleteLattice
apply (rule CompleteLatticeI-simp)
apply (rule fixf-po, clarify)
apply (simp add: P-def A-def r-def)
apply (blast intro!: Tarski.tarski-full-lemma cl-po cl-co f-cl)
done

end

```

35 Implementation of carry chain incrementor and adder

theory *Adder* **imports** *Main Word* **begin**

lemma *[simp]*: *bv-to-nat* [b] = *bitval* b
by (*simp add: bv-to-nat-helper*)

lemma *bv-to-nat-helper'*:
 $bv \neq [] \implies bv\text{-to-nat } bv = \text{bitval } (hd\ bv) * 2^{length\ bv - 1} + bv\text{-to-nat } (tl\ bv)$
by (*cases bv*) (*simp-all add: bv-to-nat-helper*)

definition

half-adder :: [bit, bit] => bit list **where**
half-adder a b = [a bitand b, a bitxor b]

lemma *half-adder-correct*: *bv-to-nat* (*half-adder* a b) = *bitval* a + *bitval* b
apply (*simp add: half-adder-def*)
apply (*cases a, auto*)
apply (*cases b, auto*)
done

lemma *[simp]*: *length* (*half-adder* a b) = 2
by (*simp add: half-adder-def*)

definition

full-adder :: [bit, bit, bit] => bit list **where**
full-adder a b c =
 (let x = a bitxor b in [a bitand b bitor c bitand x, x bitxor c])

lemma *full-adder-correct*:
 $bv\text{-to-nat } (full\text{-adder } a\ b\ c) = \text{bitval } a + \text{bitval } b + \text{bitval } c$
apply (*simp add: full-adder-def Let-def*)
apply (*cases a, auto*)
apply (*case-tac [!] b, auto*)
apply (*case-tac [!] c, auto*)
done

lemma *[simp]*: *length* (*full-adder* a b c) = 2
by (*simp add: full-adder-def Let-def*)

35.1 Carry chain incrementor

consts

carry-chain-inc :: [bit list, bit] => bit list

primrec

carry-chain-inc [] c = [c]
carry-chain-inc (a#as) c =


```

    (let chain = carry-chain-inc as c
      in half-adder a (hd chain) @ tl chain)

lemma cci-nonnul: carry-chain-inc as c ≠ []
  by (cases as) (auto simp add: Let-def half-adder-def)

lemma cci-length [simp]: length (carry-chain-inc as c) = length as + 1
  by (induct as) (simp-all add: Let-def)

lemma cci-correct: bv-to-nat (carry-chain-inc as c) = bv-to-nat as + bitval c
  apply (induct as)
  apply (cases c, simp-all add: Let-def bv-to-nat-dist-append)
  apply (simp add: half-adder-correct bv-to-nat-helper' [OF cci-nonnul]
    ring-distrib bv-to-nat-helper)
  done

consts
  carry-chain-adder :: [bit list, bit list, bit] => bit list
primrec
  carry-chain-adder [] bs c = [c]
  carry-chain-adder (a # as) bs c =
    (let chain = carry-chain-adder as (tl bs) c
      in full-adder a (hd bs) (hd chain) @ tl chain)

lemma cca-nonnul: carry-chain-adder as bs c ≠ []
  by (cases as) (auto simp add: full-adder-def Let-def)

lemma cca-length: length as = length bs ==>
  length (carry-chain-adder as bs c) = Suc (length bs)
  by (induct as arbitrary: bs) (auto simp add: Let-def)

theorem cca-correct:
  length as = length bs ==>
  bv-to-nat (carry-chain-adder as bs c) =
  bv-to-nat as + bv-to-nat bs + bitval c
proof (induct as arbitrary: bs)
  case Nil
  then show ?case by simp
next
  case (Cons a as xs)
  note ind = Cons.hyps
  from Cons.prem have len: Suc (length as) = length xs by simp
  show ?case
  proof (cases xs)
  case Nil
  with len show ?thesis by simp
  next
  case (Cons b bs)
  with len have len': length as = length bs by simp

```

```

    then have bv-to-nat (carry-chain-adder as bs c) = bv-to-nat as + bv-to-nat bs
+ bitval c
    by (rule ind)
    with len' and Cons
    show ?thesis
    apply (simp add: Let-def)
    apply (subst bv-to-nat-dist-append)
    apply (simp add: full-adder-correct bv-to-nat-helper' [OF cca-nonnull]
        ring-distrib bv-to-nat-helper cca-length)
    done
qed
qed
end

```

36 Classical Predicate Calculus Problems

theory *Classical* **imports** *Main* **begin**

36.1 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  $(P \rightarrow Q \mid R) \rightarrow (P \rightarrow Q) \mid (P \rightarrow R)$ 
by blast

```

If and only if

```

lemma  $(P=Q) = (Q = (P::bool))$ 
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.

36.1.1 Pelletier's examples

1

```

lemma  $(P \rightarrow Q) = (\sim Q \rightarrow \sim P)$ 

```

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 $P=(P::\text{bool})$

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*

36.1.2 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*

36.1.3 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 $\exists x. P(x) \longrightarrow P(a) \ \& \ P(b)$

by *blast*

lemma $\exists z. P(z) \longrightarrow (\forall x. P(x))$

by *blast*

lemma $\exists x. (\exists y. P(y)) \longrightarrow P(x)$

by *blast*

36.1.4 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 $(\exists x. P(x)) \ \&$

$$\begin{aligned}
& (\forall x. L(x) \dashrightarrow \sim (M(x) \& R(x))) \& \\
& (\forall x. P(x) \dashrightarrow (M(x) \& L(x))) \& \\
& ((\forall x. P(x) \dashrightarrow Q(x)) \mid (\exists x. P(x) \& R(x))) \\
& \dashrightarrow (\exists x. Q(x) \& P(x))
\end{aligned}$$

by *blast*

Problem 26

$$\begin{aligned}
\textbf{lemma } & ((\exists x. p(x)) = (\exists x. q(x))) \& \\
& (\forall x. \forall y. p(x) \& q(y) \dashrightarrow (r(x) = s(y))) \\
& \dashrightarrow ((\forall x. p(x) \dashrightarrow r(x)) = (\forall x. q(x) \dashrightarrow s(x)))
\end{aligned}$$

by *blast*

Problem 27

$$\begin{aligned}
\textbf{lemma } & (\exists x. P(x) \& \sim Q(x)) \& \\
& (\forall x. P(x) \dashrightarrow R(x)) \& \\
& (\forall x. M(x) \& L(x) \dashrightarrow P(x)) \& \\
& ((\exists x. R(x) \& \sim Q(x)) \dashrightarrow (\forall x. L(x) \dashrightarrow \sim R(x))) \\
& \dashrightarrow (\forall x. M(x) \dashrightarrow \sim L(x))
\end{aligned}$$

by *blast*

Problem 28. AMENDED

$$\begin{aligned}
\textbf{lemma } & (\forall x. P(x) \dashrightarrow (\forall x. Q(x))) \& \\
& ((\forall x. Q(x) \mid R(x)) \dashrightarrow (\exists x. Q(x) \& S(x))) \& \\
& ((\exists x. S(x)) \dashrightarrow (\forall x. L(x) \dashrightarrow M(x))) \\
& \dashrightarrow (\forall x. P(x) \& L(x) \dashrightarrow M(x))
\end{aligned}$$

by *blast*

Problem 29. Essentially the same as Principia Mathematica *11.71

$$\begin{aligned}
\textbf{lemma } & (\exists x. F(x)) \& (\exists y. G(y)) \\
& \dashrightarrow ((\forall x. F(x) \dashrightarrow H(x)) \& (\forall y. G(y) \dashrightarrow J(y))) = \\
& (\forall x y. F(x) \& G(y) \dashrightarrow H(x) \& J(y))
\end{aligned}$$

by *blast*

Problem 30

$$\begin{aligned}
\textbf{lemma } & (\forall x. P(x) \mid Q(x) \dashrightarrow \sim R(x)) \& \\
& (\forall x. (Q(x) \dashrightarrow \sim S(x)) \dashrightarrow P(x) \& R(x)) \\
& \dashrightarrow (\forall x. S(x))
\end{aligned}$$

by *blast*

Problem 31

$$\begin{aligned}
\textbf{lemma } & \sim(\exists x. P(x) \& (Q(x) \mid R(x))) \& \\
& (\exists x. L(x) \& P(x)) \& \\
& (\forall x. \sim R(x) \dashrightarrow M(x)) \\
& \dashrightarrow (\exists x. L(x) \& M(x))
\end{aligned}$$

by *blast*

Problem 32

$$\textbf{lemma } (\forall x. P(x) \& (Q(x) \mid R(x)) \dashrightarrow S(x)) \&$$

$$(\forall x. S(x) \ \& \ R(x) \ \longrightarrow \ L(x)) \ \& \\
(\forall x. M(x) \ \longrightarrow \ R(x)) \\
\longrightarrow (\forall x. P(x) \ \& \ M(x) \ \longrightarrow \ L(x))$$
by *blast*

Problem 33

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 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 $\exists x y. P \ x \ y \ \longrightarrow \ (\forall u \ v. P \ u \ v)$
by *blast*

Problem 36

lemma $(\forall x. \exists y. J \ x \ y) \ \& \\
(\forall x. \exists y. G \ x \ y) \ \& \\
(\forall x \ y. J \ x \ y \mid G \ x \ y \ \longrightarrow \\
(\forall z. J \ y \ z \mid G \ y \ z \ \longrightarrow H \ x \ z)) \\
\longrightarrow (\forall x. \exists y. H \ x \ y)$
by *blast*

Problem 37

lemma $(\forall z. \exists w. \forall x. \exists y. \\
(P \ x \ z \ \longrightarrow P \ y \ w) \ \& \ P \ y \ z \ \& \ (P \ y \ w \ \longrightarrow (\exists u. Q \ u \ w))) \ \& \\
(\forall x \ z. \sim(P \ x \ z) \ \longrightarrow (\exists y. Q \ y \ z)) \ \& \\
((\exists x \ y. Q \ x \ y) \ \longrightarrow (\forall x. R \ x \ x)) \\
\longrightarrow (\forall x. \exists y. R \ x \ y)$
by *blast*

Problem 38

lemma $(\forall x. p(a) \ \& \ (p(x) \longrightarrow (\exists y. p(y) \ \& \ r \ x \ y)) \ \longrightarrow \\
(\exists z. \exists w. p(z) \ \& \ r \ x \ w \ \& \ r \ w \ z)) =$
 $(\forall x. (\sim p(a) \mid p(x) \mid (\exists z. \exists w. p(z) \ \& \ r \ x \ w \ \& \ r \ w \ z)) \ \& \\
(\sim p(a) \mid \sim(\exists y. p(y) \ \& \ r \ x \ y) \mid \\
(\exists z. \exists w. p(z) \ \& \ r \ x \ w \ \& \ r \ w \ z)))$
by *blast*

Problem 39

lemma $\sim (\exists x. \forall y. F y x = (\sim F y y))$
by *blast*

Problem 40. AMENDED

lemma $(\exists y. \forall x. F x y = F x x)$
 $\longrightarrow \sim (\forall x. \exists y. \forall z. F z y = (\sim F z x))$
by *blast*

Problem 41

lemma $(\forall z. \exists y. \forall x. f x y = (f x z \ \& \ \sim f x x))$
 $\longrightarrow \sim (\exists z. \forall x. f x z)$
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 $(\forall x::'a. \forall y::'a. q x y = (\forall z. p z x = (p z y::bool)))$
 $\longrightarrow (\forall x. (\forall y. q x y = (q y x::bool)))$
by *blast*

Problem 44

lemma $(\forall x. f(x) \longrightarrow$
 $(\exists y. g(y) \ \& \ h x y \ \& \ (\exists y. g(y) \ \& \ \sim h x y))) \ \&$
 $(\exists x. j(x) \ \& \ (\forall y. g(y) \longrightarrow h x y))$
 $\longrightarrow (\exists x. j(x) \ \& \ \sim f(x))$
by *blast*

Problem 45

lemma $(\forall x. f(x) \ \& \ (\forall y. g(y) \ \& \ h x y \longrightarrow j x y)$
 $\longrightarrow (\forall y. g(y) \ \& \ h x y \longrightarrow k(y))) \ \&$
 $\sim (\exists y. l(y) \ \& \ k(y)) \ \&$
 $(\exists x. f(x) \ \& \ (\forall y. h x y \longrightarrow l(y))$
 $\ \& \ (\forall y. g(y) \ \& \ h x y \longrightarrow j x y))$
 $\longrightarrow (\exists x. f(x) \ \& \ \sim (\exists y. g(y) \ \& \ h x y))$
by *blast*

36.1.5 Problems (mainly) involving equality or functions

Problem 48

lemma $(a=b \mid c=d) \ \& \ (a=c \mid b=d) \longrightarrow a=d \mid b=c$
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 $(\forall x. P \ a \ x \mid (\forall y. P \ x \ y)) ---> (\exists x. \forall y. P \ x \ y)$

by *blast*

Problem 51

lemma $(\exists z \ w. \forall x \ y. P \ x \ y = (x=z \ \& \ y=w)) --->$
 $(\exists z. \forall x. \exists w. (\forall y. P \ x \ y = (y=w)) = (x=z))$

by *blast*

Problem 52. Almost the same as 51.

lemma $(\exists z \ w. \forall x \ y. P \ x \ y = (x=z \ \& \ y=w)) --->$
 $(\exists z. \forall y. \exists x. (\forall x. P \ x \ y = (x=z)) = (y=w))$

by *blast*

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988).
 fast DISCOVERS who killed Agatha.

lemma $lives(agatha) \ \& \ lives(butler) \ \& \ lives(charles) \ \&$
 $(killed \ agatha \ agatha \mid killed \ butler \ agatha \mid killed \ charles \ agatha) \ \&$
 $(\forall x \ y. killed \ x \ y ---> hates \ x \ y \ \& \ \sim richer \ x \ y) \ \&$
 $(\forall x. hates \ agatha \ x ---> \sim hates \ charles \ x) \ \&$
 $(hates \ agatha \ agatha \ \& \ hates \ agatha \ charles) \ \&$
 $(\forall x. lives(x) \ \& \ \sim richer \ x \ agatha ---> hates \ butler \ x) \ \&$
 $(\forall x. hates \ agatha \ x ---> hates \ butler \ x) \ \&$
 $(\forall x. \sim hates \ x \ agatha \mid \sim hates \ x \ butler \mid \sim hates \ x \ charles) --->$
 $killed \ ?who \ agatha$

by *fast*

Problem 56

lemma $(\forall x. (\exists y. P(y) \ \& \ x=f(y)) ---> P(x)) = (\forall x. P(x) ---> P(f(x)))$

by *blast*

Problem 57

lemma $P \ (f \ a \ b) \ (f \ b \ c) \ \& \ P \ (f \ b \ c) \ (f \ a \ c) \ \&$
 $(\forall x \ y \ z. P \ x \ y \ \& \ P \ y \ z ---> P \ x \ z) ---> P \ (f \ a \ b) \ (f \ a \ c)$

by *blast*

Problem 58 NOT PROVED AUTOMATICALLY

lemma $(\forall x \ y. f(x)=g(y)) ---> (\forall x \ y. f(f(x))=f(g(y)))$

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 $\forall x. P\ x\ (f\ x) = (\exists y. (\forall z. P\ z\ y \dashv\vdash P\ z\ (f\ x)) \ \& \ P\ x\ y)$
by *blast*

Problem 62 as corrected in JAR 18 (1997), page 135

lemma $(\forall x. p\ a \ \& \ (p\ x \dashv\vdash p(f\ x)) \dashv\vdash p(f(f\ x))) =$
 $(\forall x. (\sim p\ a \mid p\ x \mid p(f(f\ x))) \ \&$
 $(\sim p\ a \mid \sim p(f\ x) \mid p(f(f\ x))))$
by *blast*

From Davis, Obvious Logical Inferences, IJCAI-81, 530-531 fast indeed copes!

lemma $(\forall x. F(x) \ \& \ \sim G(x) \dashv\vdash (\exists y. H(x,y) \ \& \ J(y))) \ \&$
 $(\exists x. K(x) \ \& \ F(x) \ \& \ (\forall y. H(x,y) \dashv\vdash K(y))) \ \&$
 $(\forall x. K(x) \dashv\vdash \sim G(x)) \dashv\vdash (\exists x. K(x) \ \& \ J(x))$
by *fast*

From Rudnicki, Obvious Inferences, JAR 3 (1987), 383-393. It does seem obvious!

lemma $(\forall x. F(x) \ \& \ \sim G(x) \dashv\vdash (\exists y. H(x,y) \ \& \ J(y))) \ \&$
 $(\exists x. K(x) \ \& \ F(x) \ \& \ (\forall y. H(x,y) \dashv\vdash K(y))) \ \&$
 $(\forall x. K(x) \dashv\vdash \sim G(x)) \dashv\vdash (\exists x. K(x) \dashv\vdash \sim G(x))$
by *fast*

Attributed to Lewis Carroll by S. G. Pulman. The first or last assumption can be deleted.

lemma $(\forall x. honest(x) \ \& \ industrious(x) \dashv\vdash healthy(x)) \ \&$
 $\sim (\exists x. grocer(x) \ \& \ healthy(x)) \ \&$
 $(\forall x. industrious(x) \ \& \ grocer(x) \dashv\vdash honest(x)) \ \&$
 $(\forall x. cyclist(x) \dashv\vdash industrious(x)) \ \&$
 $(\forall x. \sim healthy(x) \ \& \ cyclist(x) \dashv\vdash \sim honest(x))$
 $\dashv\vdash (\forall x. grocer(x) \dashv\vdash \sim cyclist(x))$
by *blast*

lemma $(\forall x\ y. R(x,y) \mid R(y,x)) \ \&$
 $(\forall x\ y. S(x,y) \ \& \ S(y,x) \dashv\vdash x=y) \ \&$
 $(\forall x\ y. R(x,y) \dashv\vdash S(x,y)) \dashv\vdash (\forall x\ y. S(x,y) \dashv\vdash R(x,y))$
by *blast*

36.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y \ \& \ y < z \dashv\vdash \sim (z < (x::nat))$
by (*meson order-less-irrefl order-less-trans*)

The "small example" from Bezem, Hendriks and de Nivelde, 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 $(tactic\langle\langle Meson.safe-best-meson-tac\ 1 \rangle\rangle)$
— In contrast, *meson* is SLOW: 7.6s on griffon

36.2.1 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 $P | \sim P$
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 $(Q \multimap R) \ \& \ (R \multimap P \ \& \ Q) \ \& \ (P \multimap Q \mid R) \multimap (P = Q)$
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 $(P \multimap Q) = (\sim P \mid Q)$
by *blast*

16

lemma $(P \multimap Q) \mid (Q \multimap P)$
by *blast*

17

lemma $((P \ \& \ (Q \multimap R)) \multimap S) = ((\sim P \mid Q \mid S) \ \& \ (\sim P \mid \sim R \mid S))$
by *blast*

36.2.2 Classical Logic: examples with quantifiers

lemma $(\forall x. P \ x \ \& \ Q \ x) = ((\forall x. P \ x) \ \& \ (\forall x. Q \ x))$
by *blast*

lemma $(\exists x. P \multimap Q \ x) = (P \multimap (\exists x. Q \ x))$
by *blast*

lemma $(\exists x. P \ x \multimap Q) = ((\forall x. P \ x) \multimap Q)$
by *blast*

lemma $((\forall x. P \ x) \mid Q) = (\forall x. P \ x \mid Q)$
by *blast*

lemma $(\forall x. P \ x \multimap P(f \ x)) \ \& \ P \ d \multimap P(f(f \ d))$

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*

36.2.3 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 $(\exists x. P x) \ \&$
 $(\forall x. L x \longrightarrow \sim (M x \ \& \ R x)) \ \&$
 $(\forall x. P x \longrightarrow (M x \ \& \ L x)) \ \&$
 $((\forall x. P x \longrightarrow Q x) \mid (\exists x. P x \ \& \ R x))$
 $\longrightarrow (\exists x. Q x \ \& \ P x)$
by *blast*

Problem 26; has 24 Horn clauses

lemma $((\exists x. p x) = (\exists x. q x)) \ \&$
 $(\forall x. \forall y. p x \ \& \ q y \longrightarrow (r x = s y))$
 $\longrightarrow ((\forall x. p x \longrightarrow r x) = (\forall x. q x \longrightarrow s x))$
by *blast*

Problem 27; has 13 Horn clauses

lemma $(\exists x. P x \ \& \ \sim Q x) \ \&$
 $(\forall x. P x \longrightarrow R x) \ \&$
 $(\forall x. M x \ \& \ L x \longrightarrow P x) \ \&$
 $((\exists x. R x \ \& \ \sim Q x) \longrightarrow (\forall x. L x \longrightarrow \sim R x))$
 $\longrightarrow (\forall x. M x \longrightarrow \sim L x)$
by *blast*

Problem 28. AMENDED; has 14 Horn clauses

lemma $(\forall x. P x \longrightarrow (\forall x. Q x)) \ \&$
 $((\forall x. Q x \mid R x) \longrightarrow (\exists x. Q x \ \& \ S x)) \ \&$
 $((\exists x. S x) \longrightarrow (\forall x. L x \longrightarrow M x))$
 $\longrightarrow (\forall x. P x \ \& \ L x \longrightarrow M x)$
by *blast*

Problem 29. Essentially the same as Principia Mathematica *11.71. 62 Horn clauses

lemma $(\exists x. F x) \ \& \ (\exists y. G y)$
 $\longrightarrow (((\forall x. F x \longrightarrow H x) \ \& \ (\forall y. G y \longrightarrow J y)) =$
 $(\forall x y. F x \ \& \ G y \longrightarrow H x \ \& \ J y))$
by *blast*

Problem 30

lemma $(\forall x. P x \mid Q x \longrightarrow \sim R x) \ \& \ (\forall x. (Q x \longrightarrow \sim S x) \longrightarrow P x \ \& \ R x)$
 $\longrightarrow (\forall x. S x)$
by *blast*

Problem 31; has 10 Horn clauses; first negative clauses is useless

lemma $\sim(\exists x. P x \ \& \ (Q x \mid R x)) \ \&$
 $(\exists x. L x \ \& \ P x) \ \&$
 $(\forall x. \sim R x \longrightarrow M x)$
 $\longrightarrow (\exists x. L x \ \& \ M x)$
by *blast*

Problem 32

lemma $(\forall x. P x \ \& \ (Q x \mid R x) \multimap S x) \ \& \$
 $(\forall x. S x \ \& \ R x \multimap L x) \ \& \$
 $(\forall x. M x \multimap R x)$
 $\multimap (\forall x. P x \ \& \ M x \multimap L x)$
by *blast*

Problem 33; has 55 Horn clauses

lemma $(\forall x. P a \ \& \ (P x \multimap P b) \multimap P c) =$
 $(\forall x. (\sim P a \mid P x \mid P c) \ \& \ (\sim P a \mid \sim P b \mid P c))$
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 $\exists x y. P x y \multimap (\forall u v. P u v)$
by *blast*

Problem 36; has 15 Horn clauses

lemma $(\forall x. \exists y. J x y) \ \& \ (\forall x. \exists y. G x y) \ \& \$
 $(\forall x y. J x y \mid G x y \multimap (\forall z. J y z \mid G y z \multimap H x z))$
 $\multimap (\forall x. \exists y. H x y)$
by *blast*

Problem 37; has 10 Horn clauses

lemma $(\forall z. \exists w. \forall x. \exists y.$
 $(P x z \multimap P y w) \ \& \ P y z \ \& \ (P y w \multimap (\exists u. Q u w))) \ \& \$
 $(\forall x z. \sim P x z \multimap (\exists y. Q y z)) \ \& \$
 $((\exists x y. Q x y) \multimap (\forall x. R x x))$
 $\multimap (\forall x. \exists y. R x y)$
by *blast* — causes unification tracing messages

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x. p a \ \& \ (p x \multimap (\exists y. p y \ \& \ r x y)) \multimap$
 $(\exists z. \exists w. p z \ \& \ r x w \ \& \ r w z)) =$
 $(\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 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 $(\forall x. wolf x \longrightarrow animal x) \ \& \ (\exists x. wolf x) \ \&$
 $(\forall x. fox x \longrightarrow animal x) \ \& \ (\exists x. fox x) \ \&$
 $(\forall x. bird x \longrightarrow animal x) \ \& \ (\exists x. bird x) \ \&$

$(\forall x. \text{caterpillar } x \longrightarrow \text{animal } x) \ \& \ (\exists x. \text{caterpillar } x) \ \&$
 $(\forall x. \text{snail } x \longrightarrow \text{animal } x) \ \& \ (\exists x. \text{snail } x) \ \&$
 $(\forall x. \text{grain } x \longrightarrow \text{plant } x) \ \& \ (\exists x. \text{grain } x) \ \&$
 $(\forall x. \text{animal } x \longrightarrow$
 $\quad ((\forall y. \text{plant } y \longrightarrow \text{eats } x \ y) \ \vee$
 $\quad (\forall y. \text{animal } y \ \& \ \text{smaller-than } y \ x \ \&$
 $\quad \quad (\exists z. \text{plant } z \ \& \ \text{eats } y \ z) \longrightarrow \text{eats } x \ y))) \ \&$
 $(\forall x \ y. \text{bird } y \ \& \ (\text{snail } x \vee \text{caterpillar } x) \longrightarrow \text{smaller-than } x \ y) \ \&$
 $(\forall x \ y. \text{bird } x \ \& \ \text{fox } y \longrightarrow \text{smaller-than } x \ y) \ \&$
 $(\forall x \ y. \text{fox } x \ \& \ \text{wolf } y \longrightarrow \text{smaller-than } x \ y) \ \&$
 $(\forall x \ y. \text{wolf } x \ \& \ (\text{fox } y \vee \text{grain } y) \longrightarrow \sim \text{eats } x \ y) \ \&$
 $(\forall x \ y. \text{bird } x \ \& \ \text{caterpillar } y \longrightarrow \text{eats } x \ y) \ \&$
 $(\forall x \ y. \text{bird } x \ \& \ \text{snail } y \longrightarrow \sim \text{eats } x \ y) \ \&$
 $(\forall x. (\text{caterpillar } x \vee \text{snail } x) \longrightarrow (\exists y. \text{plant } y \ \& \ \text{eats } x \ y))$
 $\longrightarrow (\exists x \ y. \text{animal } x \ \& \ \text{animal } y \ \& \ (\exists z. \text{grain } z \ \& \ \text{eats } y \ z \ \& \ \text{eats } x \ y))$
by (*tactic*⟨*Meson.safe-best-meson-tac 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 & lives butler & lives charles &*
(killed agatha agatha | killed butler agatha | killed charles agatha) &
($\forall x y. \text{killed } x y \longrightarrow \text{hates } x y \ \& \ \sim \text{richer } x y$) &
($\forall x. \text{hates agatha } x \longrightarrow \sim \text{hates charles } x$) &
(hates agatha agatha & hates agatha charles) &
($\forall x. \text{lives } x \ \& \ \sim \text{richer } x \text{ agatha} \longrightarrow \text{hates butler } x$) &
($\forall x. \text{hates agatha } x \longrightarrow \text{hates butler } x$) &
($\forall x. \sim \text{hates } x \text{ agatha} \mid \sim \text{hates } x \text{ butler} \mid \sim \text{hates } x \text{ charles}$) \longrightarrow
($\exists x. \text{killed } x \text{ agatha}$)

by *meson*

Problem 57

lemma $P(f a b) (f b c) \ \& \ P(f b c) (f a c) \ \&$
 $(\forall x y z. P x y \ \& \ P y z \longrightarrow P x z) \longrightarrow P(f a b) (f a c)$
by *blast*

Problem 58: Challenge found on info-hol

lemma $\forall P Q R x. \exists v w. \forall y z. P x \ \& \ Q y \longrightarrow (P v \mid R w) \ \& \ (R z \longrightarrow Q v)$
by *blast*

Problem 59

lemma $(\forall x. P x = (\sim P(f x))) \longrightarrow (\exists x. P x \ \& \ \sim P(f x))$
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*

* 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 $d: \forall x y. T(i x y) \ \& \ T x \longrightarrow T y$

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

37 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!

lemma $a \neq b \implies a \in ?X \wedge b \notin ?X$
by *blast*

37.1 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*

37.2 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*

lemma $\forall f:: 'a \Rightarrow 'a \text{ set}. \forall x. f x \neq ?S f$
— This form displays the diagonal term.
by *best*

lemma $?S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$
— This form exploits the set constructs.
by (*rule notI, erule rangeE, best*)

lemma $?S \notin \text{range } (f :: 'a \Rightarrow 'a \text{ set})$
— Or just this!
by *best*

37.3 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*:

```

inj (f :: 'a ⇒ 'b) ⇒ inj (g :: 'b ⇒ 'a)
  ⇒ ∃ h :: 'a ⇒ 'b. inj h ∧ surj h
apply (rule decomposition [where f=f and g=g, THEN exE])
apply (rule-tac x = (λz. if z ∈ x then f z else 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)
  apply (erule subset-inj-on [OF - subset-UNIV])
apply blast
apply (erule ssbst, subst double-complement, erule inv-image-comp [symmetric])
done

```

37.4 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 ¬ inj-on (%a. card(R “ {a} - {a})) A
proof —
  let ?N = %a. card(R “ {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..?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 ~ inj-on ?N A (is ~ ?I)
  proof
    assume ?I
    hence ?n = card(?N ‘ A) by (rule card-image[symmetric])
    with sub ⟨finite A⟩ have 2[simp]: ?N ‘ A = {0..?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  $b \notin R$  “  $\{a\}$  using  $\langle a \neq b \rangle$  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  $\langle a \neq b \rangle$   $\langle \text{finite } A \rangle$  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 $\exists A. (\forall x \in A. x \leq (0::\text{int}))$
— 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 $a < b \wedge b < (c::\text{int}) \implies \exists A. a \notin A \wedge b \in A \wedge c \notin A$
— 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 $(\forall u \ v. u < (0::\text{int}) \longrightarrow u \neq \text{abs } v)$
 $\longrightarrow (\exists A::\text{int set}. (\forall y. \text{abs } y \notin A) \wedge -2 \in A)$
— 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.
apply *clarify*
apply (*drule-tac* $x = \{x. P \ x\}$ **in** *spec*)
apply *force*
done

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 (*case-tac* *v*, *auto*)
apply (*drule-tac* $x = \text{Suc } v$ **and** $P = \lambda x. ?a \ x \neq ?b \ x$ **in** *spec*, *force*)
done

end

38 Meson test cases

theory *Meson-Test*
imports *Main*
begin

WARNING: there are many potential conflicts between variables used below
 and constants declared in HOL!

hide *const subset member quotient*

Test data for the MESON proof procedure (Excludes the equality problems
 51, 52, 56, 58)

38.1 Interactive examples

ML $\langle\langle$
writeln *Problem 25*;
 $\text{Goal } (\exists x. P \ x) \ \& \ (\forall x. L \ x \ \longrightarrow \sim (M \ x \ \& \ R \ x)) \ \& \ (\forall x. P \ x \ \longrightarrow (M \ x \ \& \ L \ x)) \ \& \ ((\forall x. P \ x \ \longrightarrow Q \ x) \mid (\exists x. P \ x \ \& \ R \ x)) \ \longrightarrow (\exists x. Q \ x \ \& \ P \ x);$

```

by (rtac ccontr 1);
val [prem25] = gethyps 1;
val nnf25 = Meson.make-nnf prem25;
val xsko25 = Meson.skolemize 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 prem26;
val xsko26 = Meson.skolemize 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 prem43;
val xsko43 = Meson.skolemize 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 *BO002-0-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 $\&$
 $(\text{defined}(a::'a, b)) \ \&$

```

(defined(b::'a,c)) &
(~defined(a::'a,compos(b::'a,c))) --> False
oops

```

lemma *COL001-2*:
$$\begin{aligned}
& EQU001-0-\alpha x \text{ equal } \& \\
& (\forall X Y Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(s::'a,X),Y),Z),\text{apply}(\text{apply}(X::'a,Z),\text{apply}(Y::'a,Z)))) \\
& \& \\
& (\forall Y X. \text{equal}(\text{apply}(\text{apply}(k::'a,X),Y),X)) \& \\
& (\forall X Y Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(b::'a,X),Y),Z),\text{apply}(X::'a,\text{apply}(Y::'a,Z)))) \\
& \& \\
& (\forall X. \text{equal}(\text{apply}(i::'a,X),X)) \& \\
& (\forall A B C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(\text{apply}(A::'a,C),\text{apply}(B::'a,C))) \& \\
& (\forall D F' E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(\text{apply}(F'::'a,D),\text{apply}(F'::'a,E))) \& \\
& (\forall X. \text{equal}(\text{apply}(\text{apply}(\text{apply}(s::'a,\text{apply}(b::'a,X)),i),\text{apply}(\text{apply}(s::'a,\text{apply}(b::'a,X)),i)),\text{apply}(x::'a,\text{apply}(y::'a,\text{apply}(z::'a,w::'a,v::'a,u::'a,t::'a,s::'a,r::'a,q::'a,p::'a,o::'a,n::'a,m::'a,l::'a,k::'a,j::'a,i::'a,h::'a,g::'a,f::'a,e::'a,d::'a,c::'a,b::'a,a::'a)))) \\
& \& \\
& (\forall Y. \sim \text{equal}(Y::'a,\text{apply}(\text{combinator}::'a,Y))) \longrightarrow False \\
& \textbf{by meson}
\end{aligned}$$
lemma *COL023-1*:
$$\begin{aligned}
& \text{EQU001-0-ax equal \&} \\
& (\forall X Y Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(b::'a,X),Y),Z),\text{apply}(X::'a,\text{apply}(Y::'a,Z)))) \\
& \& \\
& (\forall X Y Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(n::'a,X),Y),Z),\text{apply}(\text{apply}(\text{apply}(X::'a,Z),Y),Z))) \\
& \& \\
& (\forall A B C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(\text{apply}(A::'a,C),\text{apply}(B::'a,C))) \& \\
& (\forall D F' E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(\text{apply}(F'::'a,D),\text{apply}(F'::'a,E))) \& \\
& (\forall Y. \sim \text{equal}(Y::'a,\text{apply}(\text{combinator}::'a,Y))) \longrightarrow \text{False} \\
& \text{by meson}
\end{aligned}$$

lemma *COL032-1*:

$$\begin{aligned}
& EQU001-0\text{-}ax \text{ equal \&} \\
& (\forall X. \text{equal}(\text{apply}(m::'a,X),\text{apply}(X::'a,X))) \& \\
& (\forall Y X Z. \text{equal}(\text{apply}(\text{apply}(\text{apply}(q::'a,X),Y),Z),\text{apply}(Y::'a,\text{apply}(X::'a,Z)))) \\
& \& \\
& (\forall A B C. \text{equal}(A::'a,B) \longrightarrow \text{equal}(\text{apply}(A::'a,C),\text{apply}(B::'a,C))) \& \\
& (\forall D F' E. \text{equal}(D::'a,E) \longrightarrow \text{equal}(\text{apply}(F'::'a,D),\text{apply}(F'::'a,E))) \& \\
& (\forall G H. \text{equal}(G::'a,H) \longrightarrow \text{equal}(f(G),f(H))) \& \\
& (\forall Y. \sim \text{equal}(\text{apply}(Y::'a,f(Y)),\text{apply}(f(Y),\text{apply}(Y::'a,f(Y))))) \longrightarrow \text{False} \\
& \textbf{by meson}
\end{aligned}$$
lemma *COL052-2*:
$$\begin{aligned} & EOU001-0-ax \text{ equal } \& \\ & (\forall X \ Y \ W. \text{equal}(\text{response}(\text{compos}(X::'a, Y), W), \text{response}(X::'a, \text{response}(Y::'a, W)))) \\ & \& \end{aligned}$$

$(\forall X Y. \text{agreeable}(X) \longrightarrow \text{equal}(\text{response}(X::'a, \text{common-bird}(Y)), \text{response}(Y::'a, \text{common-bird}(Y))))$
 $\&$
 $(\forall Z X. \text{equal}(\text{response}(X::'a, Z), \text{response}(\text{compatible}(X), Z)) \longrightarrow \text{agreeable}(X))$
 $\&$
 $(\forall A B. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{common-bird}(A), \text{common-bird}(B))) \&$
 $(\forall C D. \text{equal}(C::'a, D) \longrightarrow \text{equal}(\text{compatible}(C), \text{compatible}(D))) \&$
 $(\forall Q R. \text{equal}(Q::'a, R) \& \text{agreeable}(Q) \longrightarrow \text{agreeable}(R)) \&$
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{compos}(A::'a, C), \text{compos}(B::'a, C))) \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{compos}(F'::'a, D), \text{compos}(F'::'a, E))) \&$
 $(\forall G H I'. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{response}(G::'a, I'), \text{response}(H::'a, I'))) \&$
 $(\forall J L K'. \text{equal}(J::'a, K') \longrightarrow \text{equal}(\text{response}(L::'a, J), \text{response}(L::'a, K'))) \&$
 $(\text{agreeable}(c)) \&$
 $(\sim \text{agreeable}(a)) \&$
 $(\text{equal}(c::'a, \text{compos}(a::'a, b))) \longrightarrow \text{False}$
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:*

$\text{EQU001-0-ax equal} \ \&$
 $\text{GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2}$
 $\text{lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal}$
 $\text{between} \ \&$
 $\text{GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between}$
 $\text{equal} \ \&$
 $(\sim \text{between}(a::'a,b,b)) \dashrightarrow \text{False}$
by *meson*

abbreviation *GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3*

$\text{lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension}$
 $\text{between equal equidistant} \equiv$
 $(\forall Y\ X. \text{equidistant}(X::'a,Y,Y,X)) \ \&$
 $(\forall X\ Y\ Z\ V\ V2\ W. \text{equidistant}(X::'a,Y,Z,V) \ \& \ \text{equidistant}(X::'a,Y,V2,W))$
 $\dashrightarrow \text{equidistant}(Z::'a,V,V2,W)) \ \&$
 $(\forall Z\ X\ Y. \text{equidistant}(X::'a,Y,Z,Z) \dashrightarrow \text{equal}(X::'a,Y)) \ \&$
 $(\forall X\ Y\ W\ V. \text{between}(X::'a,Y,\text{extension}(X::'a,Y,W,V))) \ \&$
 $(\forall X\ Y\ W\ V. \text{equidistant}(Y::'a,\text{extension}(X::'a,Y,W,V),W,V)) \ \&$
 $(\forall X1\ Y1\ X\ Y\ Z\ V\ Z1\ V1. \text{equidistant}(X::'a,Y,X1,Y1) \ \& \ \text{equidistant}(Y::'a,Z,Y1,Z1))$

$\& \text{equidistant}(X::'a, V, X1, V1) \& \text{equidistant}(Y::'a, V, Y1, V1) \& \text{between}(X::'a, Y, Z)$
 $\& \text{between}(X1::'a, Y1, Z1) \dashrightarrow \text{equal}(X::'a, Y) \mid \text{equidistant}(Z::'a, V, Z1, V1)) \&$
 $(\forall X Y. \text{between}(X::'a, Y, X) \dashrightarrow \text{equal}(X::'a, Y)) \&$
 $(\forall U V W X Y. \text{between}(U::'a, V, W) \& \text{between}(Y::'a, X, W) \dashrightarrow \text{between}(V::'a, \text{inner-pasch}(U::'a, V, W, X,$
 $\&$
 $(\forall V W X Y U. \text{between}(U::'a, V, W) \& \text{between}(Y::'a, X, W) \dashrightarrow \text{between}(X::'a, \text{inner-pasch}(U::'a, V, W, X,$
 $\&$
 $(\sim \text{between}(\text{lower-dimension-point-1}::'a, \text{lower-dimension-point-2}, \text{lower-dimension-point-3}))$
 $\&$
 $(\sim \text{between}(\text{lower-dimension-point-2}::'a, \text{lower-dimension-point-3}, \text{lower-dimension-point-1}))$
 $\&$
 $(\sim \text{between}(\text{lower-dimension-point-3}::'a, \text{lower-dimension-point-1}, \text{lower-dimension-point-2}))$
 $\&$
 $(\forall Z X Y W V. \text{equidistant}(X::'a, W, X, V) \& \text{equidistant}(Y::'a, W, Y, V) \& \text{equidis-}$
 $\text{tant}(Z::'a, W, Z, V) \dashrightarrow \text{between}(X::'a, Y, Z) \mid \text{between}(Y::'a, Z, X) \mid \text{between}(Z::'a, X, Y)$
 $\mid \text{equal}(W::'a, V)) \&$
 $(\forall U V W X Y. \text{between}(U::'a, W, Y) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equal}(U::'a, W)$
 $\mid \text{between}(U::'a, V, \text{euclid1}(U::'a, V, W, X, Y))) \&$
 $(\forall U V W X Y. \text{between}(U::'a, W, Y) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equal}(U::'a, W)$
 $\mid \text{between}(U::'a, X, \text{euclid2}(U::'a, V, W, X, Y))) \&$
 $(\forall U V W X Y. \text{between}(U::'a, W, Y) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equal}(U::'a, W)$
 $\mid \text{between}(\text{euclid1}(U::'a, V, W, X, Y), Y, \text{euclid2}(U::'a, V, W, X, Y))) \&$
 $(\forall U V V1 W X X1. \text{equidistant}(U::'a, V, U, V1) \& \text{equidistant}(U::'a, X, U, X1) \&$
 $\text{between}(U::'a, V, X) \& \text{between}(V::'a, W, X) \dashrightarrow \text{between}(V1::'a, \text{continuous}(U::'a, V, V1, W, X, X1), X1))$
 $\&$
 $(\forall U V V1 W X X1. \text{equidistant}(U::'a, V, U, V1) \& \text{equidistant}(U::'a, X, U, X1) \&$
 $\text{between}(U::'a, V, X) \& \text{between}(V::'a, W, X) \dashrightarrow \text{equidistant}(U::'a, W, U, \text{continuous}(U::'a, V, V1, W, X, X1)))$
 $\&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{between}(X::'a, W, Z) \dashrightarrow \text{between}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{between}(W::'a, X, Z) \dashrightarrow \text{between}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{between}(W::'a, Z, X) \dashrightarrow \text{between}(W::'a, Z, Y))$
 $\&$
 $(\forall X Y V W Z. \text{equal}(X::'a, Y) \& \text{equidistant}(X::'a, V, W, Z) \dashrightarrow \text{equidis-}$
 $\text{tant}(Y::'a, V, W, Z)) \&$
 $(\forall X V Y W Z. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, X, W, Z) \dashrightarrow \text{equidis-}$
 $\text{tant}(V::'a, Y, W, Z)) \&$
 $(\forall X V W Y Z. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, W, X, Z) \dashrightarrow \text{equidis-}$
 $\text{tant}(V::'a, W, Y, Z)) \&$
 $(\forall X V W Z Y. \text{equal}(X::'a, Y) \& \text{equidistant}(V::'a, W, Z, X) \dashrightarrow \text{equidis-}$
 $\text{tant}(V::'a, W, Z, Y)) \&$
 $(\forall X Y V1 V2 V3 V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(X::'a, V1, V2, V3, V4), \text{inner-pasch}(Y::'a, V1, V$
 $\&$
 $(\forall X V1 Y V2 V3 V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(V1::'a, X, V2, V3, V4), \text{inner-pasch}(V1::'a, Y, V$
 $\&$
 $(\forall X V1 V2 Y V3 V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(V1::'a, V2, X, V3, V4), \text{inner-pasch}(V1::'a, V2, Y$
 $\&$
 $(\forall X V1 V2 V3 Y V4. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{inner-pasch}(V1::'a, V2, V3, X, V4), \text{inner-pasch}(V1::'a, V2, Y$

$\&$
 $(\forall X V1 V2 V3 V4 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{inner-pasch}(V1::'a, V2, V3, V4, X), \text{inner-pasch}(V1::'a, V2,$
 $\&$
 $(\forall A B C D E F'. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{euclid1}(A::'a, C, D, E, F'), \text{euclid1}(B::'a, C, D, E, F'))))$
 $\&$
 $(\forall G I' H J K' L. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{euclid1}(I'::'a, G, J, K', L), \text{euclid1}(I'::'a, H, J, K', L)))$
 $\&$
 $(\forall M O' P N Q R. \text{equal}(M::'a, N) \longrightarrow \text{equal}(\text{euclid1}(O'::'a, P, M, Q, R), \text{euclid1}(O'::'a, P, N, Q, R)))$
 $\&$
 $(\forall S' U V W T' X. \text{equal}(S'::'a, T') \longrightarrow \text{equal}(\text{euclid1}(U::'a, V, W, S', X), \text{euclid1}(U::'a, V, W, T', X)))$
 $\&$
 $(\forall Y A1 B1 C1 D1 Z. \text{equal}(Y::'a, Z) \longrightarrow \text{equal}(\text{euclid1}(A1::'a, B1, C1, D1, Y), \text{euclid1}(A1::'a, B1, C1, D1, Z)))$
 $\&$
 $(\forall E1 F1 G1 H1 I1 J1. \text{equal}(E1::'a, F1) \longrightarrow \text{equal}(\text{euclid2}(E1::'a, G1, H1, I1, J1), \text{euclid2}(F1::'a, G1, H1, I1, J1)))$
 $\&$
 $(\forall K1 M1 L1 N1 O1 P1. \text{equal}(K1::'a, L1) \longrightarrow \text{equal}(\text{euclid2}(M1::'a, K1, N1, O1, P1), \text{euclid2}(M1::'a, L1, N1, O1, P1)))$
 $\&$
 $(\forall Q1 S1 T1 R1 U1 V1. \text{equal}(Q1::'a, R1) \longrightarrow \text{equal}(\text{euclid2}(S1::'a, T1, Q1, U1, V1), \text{euclid2}(S1::'a, T1, R1, U1, V1)))$
 $\&$
 $(\forall W1 Y1 Z1 A2 X1 B2. \text{equal}(W1::'a, X1) \longrightarrow \text{equal}(\text{euclid2}(Y1::'a, Z1, A2, W1, B2), \text{euclid2}(Y1::'a, Z1, A2, W1, B2)))$
 $\&$
 $(\forall C2 E2 F2 G2 H2 D2. \text{equal}(C2::'a, D2) \longrightarrow \text{equal}(\text{euclid2}(E2::'a, F2, G2, H2, C2), \text{euclid2}(E2::'a, F2, G2, H2, C2)))$
 $\&$
 $(\forall X Y V1 V2 V3. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(X::'a, V1, V2, V3), \text{extension}(Y::'a, V1, V2, V3)))$
 $\&$
 $(\forall X V1 Y V2 V3. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(V1::'a, X, V2, V3), \text{extension}(V1::'a, Y, V2, V3)))$
 $\&$
 $(\forall X V1 V2 Y V3. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(V1::'a, V2, X, V3), \text{extension}(V1::'a, V2, Y, V3)))$
 $\&$
 $(\forall X V1 V2 V3 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{extension}(V1::'a, V2, V3, X), \text{extension}(V1::'a, V2, V3, Y)))$
 $\&$
 $(\forall X Y V1 V2 V3 V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(X::'a, V1, V2, V3, V4, V5), \text{continuous}(Y::'a, V1, V2, V3, V4, V5)))$
 $\&$
 $(\forall X V1 Y V2 V3 V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, X, V2, V3, V4, V5), \text{continuous}(V1::'a, Y, V2, V3, V4, V5)))$
 $\&$
 $(\forall X V1 V2 Y V3 V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, X, V3, V4, V5), \text{continuous}(V1::'a, V2, Y, V3, V4, V5)))$
 $\&$
 $(\forall X V1 V2 V3 Y V4 V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, V3, X, V4, V5), \text{continuous}(V1::'a, V2, V3, Y, V4, V5)))$
 $\&$
 $(\forall X V1 V2 V3 V4 Y V5. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, V3, V4, X, V5), \text{continuous}(V1::'a, V2, V3, V4, Y, V5)))$
 $\&$
 $(\forall X V1 V2 V3 V4 V5 Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{continuous}(V1::'a, V2, V3, V4, V5, X), \text{continuous}(V1::'a, V2, V3, V4, V5, Y)))$

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:*

$\text{EQU001-0-ax equal} \&$
 $\text{GRP004-0-ax INVERSE identity multiply equal} \&$
 $\text{GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal} \&$
 $(\text{equal}(\text{least-upper-bound}(a::'a, b), b)) \&$
 $(\sim \text{equal}(\text{greatest-lower-bound}(\text{multiply}(a::'a, c), \text{multiply}(b::'a, c)), \text{multiply}(a::'a, c)))$
 $\longrightarrow \text{False}$
by *meson*

lemma *GRP168-1:*

$\text{EQU001-0-ax equal} \&$
 $\text{GRP004-0-ax INVERSE identity multiply equal} \&$
 $\text{GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal} \&$
 $(\text{equal}(\text{least-upper-bound}(a::'a, b), b)) \&$
 $(\sim \text{equal}(\text{least-upper-bound}(\text{multiply}(\text{INVERSE}(c), \text{multiply}(a::'a, c)), \text{multiply}(\text{INVERSE}(c), \text{multiply}(b::'a, c))))$
 $\longrightarrow \text{False}$
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:

$(\forall A. equal(A::'a,A)) \ \&$
 $(\forall B A C. equal(A::'a,B) \ \& \ equal(B::'a,C) \longrightarrow equal(A::'a,C)) \ \&$
 $(\forall B A. equal(add(A::'a,B),add(B::'a,A))) \ \&$
 $(\forall A B C. equal(add(A::'a,add(B::'a,C)),add(add(A::'a,B),C))) \ \&$
 $(\forall B A. equal(subtract(add(A::'a,B),B),A)) \ \&$
 $(\forall A B. equal(A::'a,subtract(add(A::'a,B),B))) \ \&$
 $(\forall A C B. equal(add(subtract(A::'a,B),C),subtract(add(A::'a,C),B))) \ \&$
 $(\forall A C B. equal(subtract(add(A::'a,B),C),add(subtract(A::'a,C),B))) \ \&$
 $(\forall A C B D. equal(A::'a,B) \ \& \ equal(C::'a,add(A::'a,D)) \longrightarrow equal(C::'a,add(B::'a,D)))$
 $\ \&$
 $(\forall A C D B. equal(A::'a,B) \ \& \ equal(C::'a,add(D::'a,A)) \longrightarrow equal(C::'a,add(D::'a,B)))$
 $\ \&$
 $(\forall A C B D. equal(A::'a,B) \ \& \ equal(C::'a,subtract(A::'a,D)) \longrightarrow equal(C::'a,subtract(B::'a,D)))$

$\&$
 $(\forall A \ C \ D \ B. \text{equal}(A::'a,B) \ \& \ \text{equal}(C::'a,\text{subtract}(D::'a,A)) \ \longrightarrow \ \text{equal}(C::'a,\text{subtract}(D::'a,B)))$
 $\&$
 $(\sim \text{equal}(\text{add}(\text{add}(a::'a,b),c),\text{add}(a::'a,\text{add}(b::'a,c)))) \ \longrightarrow \ \text{False}$
by *meson*

abbreviation *NUM001-0-ax multiply successor num0 add equal* \equiv
 $(\forall A. \text{equal}(\text{add}(A::'a,\text{num0}),A)) \ \&$
 $(\forall A \ B. \text{equal}(\text{add}(A::'a,\text{successor}(B)),\text{successor}(\text{add}(A::'a,B)))) \ \&$
 $(\forall A. \text{equal}(\text{multiply}(A::'a,\text{num0}),\text{num0})) \ \&$
 $(\forall B \ A. \text{equal}(\text{multiply}(A::'a,\text{successor}(B)),\text{add}(\text{multiply}(A::'a,B),A))) \ \&$
 $(\forall A \ B. \text{equal}(\text{successor}(A),\text{successor}(B)) \ \longrightarrow \ \text{equal}(A::'a,B)) \ \&$
 $(\forall A \ B. \text{equal}(A::'a,B) \ \longrightarrow \ \text{equal}(\text{successor}(A),\text{successor}(B)))$

abbreviation *NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless*
 \equiv
 $(\forall A \ C \ B. \text{mless}(A::'a,B) \ \& \ \text{mless}(C::'a,A) \ \longrightarrow \ \text{mless}(C::'a,B)) \ \&$
 $(\forall A \ B \ C. \text{equal}(\text{add}(\text{successor}(A),B),C) \ \longrightarrow \ \text{mless}(B::'a,C)) \ \&$
 $(\forall A \ B. \text{mless}(A::'a,B) \ \longrightarrow \ \text{equal}(\text{add}(\text{successor}(\text{predecessor-of-1st-minus-2nd}(B::'a,A)),A),B))$

abbreviation *NUM001-2-ax equal mless divides* \equiv
 $(\forall A \ B. \text{divides}(A::'a,B) \ \longrightarrow \ \text{mless}(A::'a,B) \ \mid \ \text{equal}(A::'a,B)) \ \&$
 $(\forall A \ B. \text{mless}(A::'a,B) \ \longrightarrow \ \text{divides}(A::'a,B)) \ \&$
 $(\forall A \ B. \text{equal}(A::'a,B) \ \longrightarrow \ \text{divides}(A::'a,B))$

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 $\&$
 $(\text{mless}(b::'a,c)) \ \&$
 $(\sim \text{mless}(b::'a,a)) \ \&$
 $(\text{divides}(c::'a,a)) \ \&$
 $(\forall A. \sim \text{equal}(\text{successor}(A),\text{num0})) \ \longrightarrow \ \text{False}$
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 $\&$
 $(\forall B \ A. \text{equal}(\text{add}(A::'a,B),\text{add}(B::'a,A))) \ \&$
 $(\forall B \ A \ C. \text{equal}(\text{add}(A::'a,B),\text{add}(C::'a,B)) \ \longrightarrow \ \text{equal}(A::'a,C)) \ \&$
 $(\text{mless}(a::'a,a)) \ \&$
 $(\forall A. \sim \text{equal}(\text{successor}(A),\text{num0})) \ \longrightarrow \ \text{False}$
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
 $(\forall X U Y. \text{subclass}(X::'a, Y) \ \& \ \text{member}(U::'a, X) \longrightarrow \text{member}(U::'a, Y)) \ \&$
 $(\forall X Y. \text{member}(\text{not-subclass-element}(X::'a, Y), X) \mid \text{subclass}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{member}(\text{not-subclass-element}(X::'a, Y), Y) \longrightarrow \text{subclass}(X::'a, Y)) \ \&$
 $(\forall X. \text{subclass}(X::'a, \text{universal-class})) \ \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \longrightarrow \text{subclass}(X::'a, Y)) \ \&$
 $(\forall Y X. \text{equal}(X::'a, Y) \longrightarrow \text{subclass}(Y::'a, X)) \ \&$
 $(\forall X Y. \text{subclass}(X::'a, Y) \ \& \ \text{subclass}(Y::'a, X) \longrightarrow \text{equal}(X::'a, Y)) \ \&$
 $(\forall X U Y. \text{member}(U::'a, \text{unordered-pair}(X::'a, Y)) \longrightarrow \text{equal}(U::'a, X) \mid \text{equal}(U::'a, Y))$
 $\ \&$
 $(\forall X Y. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(X::'a, \text{unordered-pair}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y. \text{member}(Y::'a, \text{universal-class}) \longrightarrow \text{member}(Y::'a, \text{unordered-pair}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y. \text{member}(\text{unordered-pair}(X::'a, Y), \text{universal-class})) \ \&$
 $(\forall X. \text{equal}(\text{unordered-pair}(X::'a, X), \text{singleton}(X))) \ \&$
 $(\forall X Y. \text{equal}(\text{unordered-pair}(\text{singleton}(X), \text{unordered-pair}(X::'a, \text{singleton}(Y))), \text{ordered-pair}(X::'a, Y)))$
 $\ \&$
 $(\forall V Y U X. \text{member}(\text{ordered-pair}(U::'a, V), \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(U::'a, X)) \ \&$
 $(\forall U X V Y. \text{member}(\text{ordered-pair}(U::'a, V), \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(V::'a, Y)) \ \&$
 $(\forall U V X Y. \text{member}(U::'a, X) \ \& \ \text{member}(V::'a, Y) \longrightarrow \text{member}(\text{ordered-pair}(U::'a, V), \text{cross-product}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y Z. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{equal}(\text{ordered-pair}(\text{first}(Z), \text{second}(Z)), Z))$
 $\ \&$
 $(\text{subclass}(\text{element-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\ \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{element-relation}) \longrightarrow \text{member}(X::'a, Y))$
 $\ \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))$
 $\ \& \ \text{member}(X::'a, Y) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, Y), \text{element-relation})) \ \&$
 $(\forall Y Z X. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, X)) \ \&$
 $(\forall X Z Y. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, Y)) \ \&$
 $(\forall Z X Y. \text{member}(Z::'a, X) \ \& \ \text{member}(Z::'a, Y) \longrightarrow \text{member}(Z::'a, \text{intersection}(X::'a, Y)))$
 $\ \&$
 $(\forall Z X. \sim(\text{member}(Z::'a, \text{complement}(X)) \ \& \ \text{member}(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{universal-class}) \longrightarrow \text{member}(Z::'a, \text{complement}(X)) \mid$
 $\text{member}(Z::'a, X)) \ \&$
 $(\forall X Y. \text{equal}(\text{complement}(\text{intersection}(\text{complement}(X), \text{complement}(Y))), \text{union}(X::'a, Y)))$
 $\ \&$
 $(\forall X Y. \text{equal}(\text{intersection}(\text{complement}(\text{intersection}(X::'a, Y)), \text{complement}(\text{intersection}(\text{complement}(X), \text{complement}(Y)))))$

$\&$
 $(\forall Xr X Y. \text{equal}(\text{intersection}(Xr::'a, \text{cross-product}(X::'a, Y)), \text{restrct}(Xr::'a, X, Y)))$
 $\&$
 $(\forall Xr X Y. \text{equal}(\text{intersection}(\text{cross-product}(X::'a, Y), Xr), \text{restrct}(Xr::'a, X, Y)))$
 $\&$
 $(\forall Z X. \sim(\text{equal}(\text{restrct}(X::'a, \text{singleton}(Z), \text{universal-class}), \text{null-class}) \& \text{member}(Z::'a, \text{domain-of}(X)))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{universal-class}) \longrightarrow \text{equal}(\text{restrct}(X::'a, \text{singleton}(Z), \text{universal-class}), \text{null-class})$
 $| \text{member}(Z::'a, \text{domain-of}(X))) \&$
 $(\forall X. \text{subclass}(\text{rot}(X), \text{cross-product}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class}), \text{universal-class})))$
 $\&$
 $(\forall V W U X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{rot}(X)) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, W), U), X)) \&$
 $(\forall U V W X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, W), U), X) \& \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{rot}(X))) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{rot}(X))) \&$
 $(\forall X. \text{subclass}(\text{flip}(X), \text{cross-product}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class}), \text{universal-class})))$
 $\&$
 $(\forall V U W X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{flip}(X)) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, U), W), X)) \&$
 $(\forall U V W X. \text{member}(\text{ordered-pair}(\text{ordered-pair}(V::'a, U), W), X) \& \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{flip}(X))) \longrightarrow \text{member}(\text{ordered-pair}(\text{ordered-pair}(U::'a, V), W), \text{flip}(X))) \&$
 $(\forall Y. \text{equal}(\text{domain-of}(\text{flip}(\text{cross-product}(Y::'a, \text{universal-class}))), \text{INVERSE}(Y)))$
 $\&$
 $(\forall Z. \text{equal}(\text{domain-of}(\text{INVERSE}(Z)), \text{range-of}(Z))) \&$
 $(\forall Z X Y. \text{equal}(\text{first}(\text{not-subclass-element}(\text{restrct}(Z::'a, X, \text{singleton}(Y)), \text{null-class})), \text{domain}(Z::'a, X, Y)))$
 $\&$
 $(\forall Z X Y. \text{equal}(\text{second}(\text{not-subclass-element}(\text{restrct}(Z::'a, \text{singleton}(X), Y), \text{null-class})), \text{rng}(Z::'a, X, Y)))$
 $\&$
 $(\forall Xr X. \text{equal}(\text{range-of}(\text{restrct}(Xr::'a, X, \text{universal-class})), \text{image}'(Xr::'a, X))) \&$
 $(\forall X. \text{equal}(\text{union}(X::'a, \text{singleton}(X)), \text{successor}(X))) \&$
 $(\text{subclass}(\text{successor-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{successor-relation}) \longrightarrow \text{equal}(\text{successor}(X), Y))$
 $\&$
 $(\forall X Y. \text{equal}(\text{successor}(X), Y) \& \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, Y), \text{successor-relation})) \&$
 $(\forall X. \text{inductive}(X) \longrightarrow \text{member}(\text{null-class}::'a, X)) \&$
 $(\forall X. \text{inductive}(X) \longrightarrow \text{subclass}(\text{image}'(\text{successor-relation}::'a, X), X)) \&$
 $(\forall X. \text{member}(\text{null-class}::'a, X) \& \text{subclass}(\text{image}'(\text{successor-relation}::'a, X), X)) \longrightarrow \text{inductive}(X)) \&$
 $(\text{inductive}(\text{omega})) \&$
 $(\forall Y. \text{inductive}(Y) \longrightarrow \text{subclass}(\text{omega}::'a, Y)) \&$
 $(\text{member}(\text{omega}::'a, \text{universal-class})) \&$
 $(\forall X. \text{equal}(\text{domain-of}(\text{restrct}(\text{element-relation}::'a, \text{universal-class}, X)), \text{sum-class}(X)))$
 $\&$
 $(\forall X. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{sum-class}(X), \text{universal-class}))$
 $\&$
 $(\forall X. \text{equal}(\text{complement}(\text{image}'(\text{element-relation}::'a, \text{complement}(X))), \text{powerClass}(X)))$
 $\&$

$(\forall U. \text{member}(U::'a, \text{universal-class}) \longrightarrow \text{member}(\text{powerClass}(U), \text{universal-class}))$
 $\&$
 $(\forall Yr\ Xr. \text{subclass}(\text{compos}(Yr::'a, Xr), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall Z\ Yr\ Xr\ Y. \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compos}(Yr::'a, Xr)) \longrightarrow \text{member}(Z::'a, \text{image}'(Yr::'a, \text{image}'(Xr::'a, \text{singleton}(Y)))) \&$
 $(\forall Y\ Z\ Yr\ Xr. \text{member}(Z::'a, \text{image}'(Yr::'a, \text{image}'(Xr::'a, \text{singleton}(Y)))) \& \text{member}(\text{ordered-pair}(Y::'a, Z), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})) \longrightarrow \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compos}(Yr::'a, Xr))) \&$
 $(\forall X. \text{single-valued-class}(X) \longrightarrow \text{subclass}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation}))$
 $\&$
 $(\forall X. \text{subclass}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation}) \longrightarrow \text{single-valued-class}(X))$
 $\&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{subclass}(Xf::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{subclass}(\text{compos}(Xf::'a, \text{INVERSE}(Xf)), \text{identity-relation}))$
 $\&$
 $(\forall Xf. \text{subclass}(Xf::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class})) \& \text{subclass}(\text{compos}(Xf::'a, \text{INVERSE}(Xf)), \text{identity-relation}) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Xf\ X. \text{function}(Xf) \& \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{image}'(Xf::'a, X), \text{universal-class}))$
 $\&$
 $(\forall X. \text{equal}(X::'a, \text{null-class}) \mid \text{member}(\text{regular}(X), X)) \&$
 $(\forall X. \text{equal}(X::'a, \text{null-class}) \mid \text{equal}(\text{intersection}(X::'a, \text{regular}(X)), \text{null-class})) \&$
 $(\forall Xf\ Y. \text{equal}(\text{sum-class}(\text{image}'(Xf::'a, \text{singleton}(Y))), \text{apply}(Xf::'a, Y))) \&$
 $(\text{function}(\text{choice})) \&$
 $(\forall Y. \text{member}(Y::'a, \text{universal-class}) \longrightarrow \text{equal}(Y::'a, \text{null-class}) \mid \text{member}(\text{apply}(\text{choice}::'a, Y), Y))$
 $\&$
 $(\forall Xf. \text{one-to-one}(Xf) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Xf. \text{one-to-one}(Xf) \longrightarrow \text{function}(\text{INVERSE}(Xf))) \&$
 $(\forall Xf. \text{function}(\text{INVERSE}(Xf)) \& \text{function}(Xf) \longrightarrow \text{one-to-one}(Xf)) \&$
 $(\text{equal}(\text{intersection}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class}), \text{intersection}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class})), \text{intersection}(\text{cross-product}(\text{universal-class}::'a, \text{universal-class})), \text{identity-relation}))$
 $\&$
 $(\forall Xr. \text{equal}(\text{complement}(\text{domain-of}(\text{intersection}(Xr::'a, \text{identity-relation}))), \text{diagonalise}(Xr)))$
 $\&$
 $(\forall X. \text{equal}(\text{intersection}(\text{domain-of}(X), \text{diagonalise}(\text{compos}(\text{INVERSE}(\text{element-relation}), X))), \text{cantor}(X)))$
 $\&$
 $(\forall Xf. \text{operation}(Xf) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Xf. \text{operation}(Xf) \longrightarrow \text{equal}(\text{cross-product}(\text{domain-of}(\text{domain-of}(Xf)), \text{domain-of}(\text{domain-of}(Xf))), \text{domain-of}(\text{domain-of}(Xf))))$
 $\&$
 $(\forall Xf. \text{operation}(Xf) \longrightarrow \text{subclass}(\text{range-of}(Xf), \text{domain-of}(\text{domain-of}(Xf))))$
 $\&$
 $(\forall Xf. \text{function}(Xf) \& \text{equal}(\text{cross-product}(\text{domain-of}(\text{domain-of}(Xf)), \text{domain-of}(\text{domain-of}(Xf))), \text{domain-of}(\text{domain-of}(Xf))) \longrightarrow \text{operation}(Xf)) \&$
 $(\forall Xf1\ Xf2\ Xh. \text{compatible}(Xh::'a, Xf1, Xf2) \longrightarrow \text{function}(Xh)) \&$
 $(\forall Xf2\ Xf1\ Xh. \text{compatible}(Xh::'a, Xf1, Xf2) \longrightarrow \text{equal}(\text{domain-of}(\text{domain-of}(Xf1)), \text{domain-of}(Xh)))$
 $\&$
 $(\forall Xf1\ Xh\ Xf2. \text{compatible}(Xh::'a, Xf1, Xf2) \longrightarrow \text{subclass}(\text{range-of}(Xh), \text{domain-of}(\text{domain-of}(Xf2))))$

$\&$
 $(\forall Xh\ Xh1\ Xf1\ Xf2. \text{function}(Xh) \ \& \ \text{equal}(\text{domain-of}(\text{domain-of}(Xf1)), \text{domain-of}(Xh))$
 $\& \ \text{subclass}(\text{range-of}(Xh), \text{domain-of}(\text{domain-of}(Xf2))) \longrightarrow \text{compatible}(Xh1::'a, Xf1, Xf2))$
 $\&$
 $(\forall Xh\ Xf2\ Xf1. \text{homomorphism}(Xh::'a, Xf1, Xf2) \longrightarrow \text{operation}(Xf1)) \ \&$
 $(\forall Xh\ Xf1\ Xf2. \text{homomorphism}(Xh::'a, Xf1, Xf2) \longrightarrow \text{operation}(Xf2)) \ \&$
 $(\forall Xh\ Xf1\ Xf2. \text{homomorphism}(Xh::'a, Xf1, Xf2) \longrightarrow \text{compatible}(Xh::'a, Xf1, Xf2))$
 $\&$
 $(\forall Xf2\ Xh\ Xf1\ X\ Y. \text{homomorphism}(Xh::'a, Xf1, Xf2) \ \& \ \text{member}(\text{ordered-pair}(X::'a, Y), \text{domain-of}(Xf1))$
 $\longrightarrow \text{equal}(\text{apply}(Xf2::'a, \text{ordered-pair}(\text{apply}(Xh::'a, X), \text{apply}(Xh::'a, Y))), \text{apply}(Xh::'a, \text{apply}(Xf1::'a, \text{ordered-pair}(X::'a, Y))))$
 $\&$
 $(\forall Xh\ Xf1\ Xf2. \text{operation}(Xf1) \ \& \ \text{operation}(Xf2) \ \& \ \text{compatible}(Xh::'a, Xf1, Xf2)$
 $\longrightarrow \text{member}(\text{ordered-pair}(\text{not-homomorphism1}(Xh::'a, Xf1, Xf2), \text{not-homomorphism2}(Xh::'a, Xf1, Xf2)), \text{domain-of}(\text{homomorphism}(Xh::'a, Xf1, Xf2)))$
 $(\forall Xh\ Xf1\ Xf2. \text{operation}(Xf1) \ \& \ \text{operation}(Xf2) \ \& \ \text{compatible}(Xh::'a, Xf1, Xf2)$
 $\& \ \text{equal}(\text{apply}(Xf2::'a, \text{ordered-pair}(\text{apply}(Xh::'a, \text{not-homomorphism1}(Xh::'a, Xf1, Xf2)), \text{apply}(Xh::'a, \text{not-homomorphism2}(Xh::'a, Xf1, Xf2))))$
 $\longrightarrow \text{homomorphism}(Xh::'a, Xf1, Xf2))$

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

$(\forall X. \text{subclass}(\text{compose-class}(X), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$

$\&$

$(\forall X\ Y\ Z. \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compose-class}(X)) \longrightarrow \text{equal}(\text{compos}(X::'a, Y), Z))$

$\&$

$(\forall Y\ Z\ X. \text{member}(\text{ordered-pair}(Y::'a, Z), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$

$\& \text{equal}(\text{compos}(X::'a, Y), Z) \longrightarrow \text{member}(\text{ordered-pair}(Y::'a, Z), \text{compose-class}(X)))$

$\&$

$(\text{subclass}(\text{composition-function}::'a, \text{cross-product}(\text{universal-class}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universa$

$\&$
 $(\forall X Y Z. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{composition-function}))$
 $\longrightarrow \text{equal}(\text{compos}(X::'a, Y), Z)) \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{cross-product}(\text{universal-class}::'a, \text{universal-class})))$
 $\longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, \text{compos}(X::'a, Y))), \text{composition-function}))$
 $\&$
 $(\text{subclass}(\text{domain-relation}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))) \&$
 $(\forall X Y. \text{member}(\text{ordered-pair}(X::'a, Y), \text{domain-relation}) \longrightarrow \text{equal}(\text{domain-of}(X), Y))$
 $\&$
 $(\forall X. \text{member}(X::'a, \text{universal-class}) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{domain-of}(X)), \text{domain-relation}))$
 $\&$
 $(\forall X. \text{equal}(\text{first}(\text{not-subclass-element}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation})), \text{single-valued1}(X)))$
 $\&$
 $(\forall X. \text{equal}(\text{second}(\text{not-subclass-element}(\text{compos}(X::'a, \text{INVERSE}(X)), \text{identity-relation})), \text{single-valued2}(X)))$
 $\&$
 $(\forall X. \text{equal}(\text{domain}(X::'a, \text{image}'(\text{INVERSE}(X), \text{singleton}(\text{single-valued1}(X))), \text{single-valued2}(X)), \text{single-valued3}(X)))$
 $\&$
 $(\text{equal}(\text{intersection}(\text{complement}(\text{compos}(\text{element-relation}::'a, \text{complement}(\text{identity-relation}))), \text{element-relation}))$
 $\&$
 $(\text{subclass}(\text{application-function}::'a, \text{cross-product}(\text{universal-class}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))))$
 $\&$
 $(\forall Z Y X. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{application-function}))$
 $\longrightarrow \text{member}(Y::'a, \text{domain-of}(X))) \&$
 $(\forall X Y Z. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{application-function}))$
 $\longrightarrow \text{equal}(\text{apply}(X::'a, Y), Z)) \&$
 $(\forall Z X Y. \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, Z)), \text{cross-product}(\text{universal-class}::'a, \text{cross-product}(\text{universal-class}::'a, \text{universal-class}))))$
 $\& \text{member}(Y::'a, \text{domain-of}(X)) \longrightarrow \text{member}(\text{ordered-pair}(X::'a, \text{ordered-pair}(Y::'a, \text{apply}(X::'a, Y))), \text{application-function}))$
 $\&$
 $(\forall X Y Xf. \text{maps}(Xf::'a, X, Y) \longrightarrow \text{function}(Xf)) \&$
 $(\forall Y Xf X. \text{maps}(Xf::'a, X, Y) \longrightarrow \text{equal}(\text{domain-of}(Xf), X)) \&$
 $(\forall X Xf Y. \text{maps}(Xf::'a, X, Y) \longrightarrow \text{subclass}(\text{range-of}(Xf), Y)) \&$
 $(\forall Xf Y. \text{function}(Xf) \& \text{subclass}(\text{range-of}(Xf), Y) \longrightarrow \text{maps}(Xf::'a, \text{domain-of}(Xf), Y))$

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
 $(\forall D \ E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{integer-of}(D), \text{integer-of}(E))) \ \&$

$(\forall F' G H. \text{equal}(F'::'a, G) \dashrightarrow \text{equal}(\text{least}(F'::'a, H), \text{least}(G::'a, H))) \ \&$
 $(\forall I' K' J. \text{equal}(I'::'a, J) \dashrightarrow \text{equal}(\text{least}(K'::'a, I'), \text{least}(K'::'a, J))) \ \&$
 $(\forall L M N. \text{equal}(L::'a, M) \dashrightarrow \text{equal}(\text{not-well-ordering}(L::'a, N), \text{not-well-ordering}(M::'a, N)))$
 $\&$
 $(\forall O' Q P. \text{equal}(O'::'a, P) \dashrightarrow \text{equal}(\text{not-well-ordering}(Q::'a, O'), \text{not-well-ordering}(Q::'a, P)))$
 $\&$
 $(\forall R S' T'. \text{equal}(R::'a, S') \dashrightarrow \text{equal}(\text{ordinal-add}(R::'a, T'), \text{ordinal-add}(S'::'a, T')))$
 $\&$
 $(\forall U W V. \text{equal}(U::'a, V) \dashrightarrow \text{equal}(\text{ordinal-add}(W::'a, U), \text{ordinal-add}(W::'a, V)))$
 $\&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{ordinal-multiply}(X::'a, Z), \text{ordinal-multiply}(Y::'a, Z)))$
 $\&$
 $(\forall A1 C1 B1. \text{equal}(A1::'a, B1) \dashrightarrow \text{equal}(\text{ordinal-multiply}(C1::'a, A1), \text{ordinal-multiply}(C1::'a, B1)))$
 $\&$
 $(\forall F1 G1 H1 I1. \text{equal}(F1::'a, G1) \dashrightarrow \text{equal}(\text{recursion}(F1::'a, H1, I1), \text{recursion}(G1::'a, H1, I1)))$
 $\&$
 $(\forall J1 L1 K1 M1. \text{equal}(J1::'a, K1) \dashrightarrow \text{equal}(\text{recursion}(L1::'a, J1, M1), \text{recursion}(L1::'a, K1, M1)))$
 $\&$
 $(\forall N1 P1 Q1 O1. \text{equal}(N1::'a, O1) \dashrightarrow \text{equal}(\text{recursion}(P1::'a, Q1, N1), \text{recursion}(P1::'a, Q1, O1)))$
 $\&$
 $(\forall R1 S1. \text{equal}(R1::'a, S1) \dashrightarrow \text{equal}(\text{recursion-equation-functions}(R1), \text{recursion-equation-functions}(S1)))$
 $\&$
 $(\forall T1 U1. \text{equal}(T1::'a, U1) \dashrightarrow \text{equal}(\text{rest-of}(T1), \text{rest-of}(U1))) \ \&$
 $(\forall V1 W1 X1 Y1. \text{equal}(V1::'a, W1) \dashrightarrow \text{equal}(\text{segment}(V1::'a, X1, Y1), \text{segment}(W1::'a, X1, Y1)))$
 $\&$
 $(\forall Z1 B2 A2 C2. \text{equal}(Z1::'a, A2) \dashrightarrow \text{equal}(\text{segment}(B2::'a, Z1, C2), \text{segment}(B2::'a, A2, C2)))$
 $\&$
 $(\forall D2 F2 G2 E2. \text{equal}(D2::'a, E2) \dashrightarrow \text{equal}(\text{segment}(F2::'a, G2, D2), \text{segment}(F2::'a, G2, E2)))$
 $\&$
 $(\forall H2 I2. \text{equal}(H2::'a, I2) \dashrightarrow \text{equal}(\text{symmetrization-of}(H2), \text{symmetrization-of}(I2)))$
 $\&$
 $(\forall J2 K2 L2. \text{equal}(J2::'a, K2) \ \& \ \text{asymmetric}(J2::'a, L2) \dashrightarrow \text{asymmetric}(K2::'a, L2))$
 $\&$
 $(\forall M2 O2 N2. \text{equal}(M2::'a, N2) \ \& \ \text{asymmetric}(O2::'a, M2) \dashrightarrow \text{asymmetric}(O2::'a, N2))$
 $\&$
 $(\forall P2 Q2 R2. \text{equal}(P2::'a, Q2) \ \& \ \text{connected}(P2::'a, R2) \dashrightarrow \text{connected}(Q2::'a, R2))$
 $\&$
 $(\forall S2 U2 T2. \text{equal}(S2::'a, T2) \ \& \ \text{connected}(U2::'a, S2) \dashrightarrow \text{connected}(U2::'a, T2))$
 $\&$
 $(\forall V2 W2 X2. \text{equal}(V2::'a, W2) \ \& \ \text{irreflexive}(V2::'a, X2) \dashrightarrow \text{irreflexive}(W2::'a, X2))$
 $\&$
 $(\forall Y2 A3 Z2. \text{equal}(Y2::'a, Z2) \ \& \ \text{irreflexive}(A3::'a, Y2) \dashrightarrow \text{irreflexive}(A3::'a, Z2))$
 $\&$
 $(\forall B3 C3 D3 E3. \text{equal}(B3::'a, C3) \ \& \ \text{section}(B3::'a, D3, E3) \dashrightarrow \text{section}(C3::'a, D3, E3))$
 $\&$
 $(\forall F3 H3 G3 I3. \text{equal}(F3::'a, G3) \ \& \ \text{section}(H3::'a, F3, I3) \dashrightarrow \text{section}(H3::'a, G3, I3))$
 $\&$
 $(\forall J3 L3 M3 K3. \text{equal}(J3::'a, K3) \ \& \ \text{section}(L3::'a, M3, J3) \dashrightarrow \text{section}(L3::'a, M3, K3))$
 $\&$

$(\forall N3\ O3\ P3. \text{equal}(N3::'a, O3) \ \& \ \text{transitive}(N3::'a, P3) \longrightarrow \text{transitive}(O3::'a, P3))$
 $\&$
 $(\forall Q3\ S3\ R3. \text{equal}(Q3::'a, R3) \ \& \ \text{transitive}(S3::'a, Q3) \longrightarrow \text{transitive}(S3::'a, R3))$
 $\&$
 $(\forall T3\ U3\ V3. \text{equal}(T3::'a, U3) \ \& \ \text{well-ordering}(T3::'a, V3) \longrightarrow \text{well-ordering}(U3::'a, V3))$
 $\&$
 $(\forall W3\ Y3\ X3. \text{equal}(W3::'a, X3) \ \& \ \text{well-ordering}(Y3::'a, W3) \longrightarrow \text{well-ordering}(Y3::'a, X3))$

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 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 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 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 function(z)) &
(\sim equal(recursion-equation-functions(z),null-class)) \longrightarrow False
by meson

lemma *PLA002-1*:

$$\begin{aligned}
& (\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} \\
& \textbf{by} \textit{meson}
\end{aligned}$$

abbreviation *PLA001-0-ax putdown on pickup do holding table differ clear EMPTY*

and' holds \equiv

$$\begin{aligned}
& (\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})))
\end{aligned}$$

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 &
 $(\forall A B. \text{equal}(A::'a, B) \dashv\vdash \text{equal}(\text{statement-by}(A), \text{statement-by}(B))) \&$
 $(\forall X. \text{person}(X) \dashv\vdash \text{knight}(X) \mid \text{knave}(X)) \&$
 $(\forall X. \sim(\text{person}(X) \& \text{knight}(X) \& \text{knave}(X))) \&$
 $(\forall X Y. \text{says}(X::'a, Y) \& \text{a-truth}(Y) \dashv\vdash \text{a-truth}(Y)) \&$
 $(\forall X Y. \sim(\text{says}(X::'a, Y) \& \text{equal}(X::'a, Y))) \&$
 $(\forall Y X. \text{says}(X::'a, Y) \dashv\vdash \text{equal}(Y::'a, \text{statement-by}(X))) \&$
 $(\forall X Y. \sim(\text{person}(X) \& \text{equal}(X::'a, \text{statement-by}(Y)))) \&$
 $(\forall X. \text{person}(X) \& \text{a-truth}(\text{statement-by}(X)) \dashv\vdash \text{knight}(X)) \&$
 $(\forall X. \text{person}(X) \dashv\vdash \text{a-truth}(\text{statement-by}(X)) \mid \text{knave}(X)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{knight}(X) \dashv\vdash \text{knight}(Y)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{knave}(X) \dashv\vdash \text{knave}(Y)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{person}(X) \dashv\vdash \text{person}(Y)) \&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) \& \text{says}(X::'a, Z) \dashv\vdash \text{says}(Y::'a, Z)) \&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \& \text{says}(Z::'a, X) \dashv\vdash \text{says}(Z::'a, Y)) \&$
 $(\forall X Y. \text{equal}(X::'a, Y) \& \text{a-truth}(X) \dashv\vdash \text{a-truth}(Y)) \&$
 $(\forall X Y. \text{knight}(X) \& \text{says}(X::'a, Y) \dashv\vdash \text{a-truth}(Y)) \&$
 $(\forall X Y. \sim(\text{knave}(X) \& \text{says}(X::'a, Y) \& \text{a-truth}(Y))) \&$
 $(\text{person}(\text{husband})) \&$
 $(\text{person}(\text{wife})) \&$
 $(\sim \text{equal}(\text{husband}::'a, \text{wife})) \&$
 $(\text{says}(\text{husband}::'a, \text{statement-by}(\text{husband}))) \&$
 $(\text{a-truth}(\text{statement-by}(\text{husband})) \& \text{knight}(\text{husband}) \dashv\vdash \text{knight}(\text{wife})) \&$
 $(\text{knight}(\text{husband}) \dashv\vdash \text{a-truth}(\text{statement-by}(\text{husband}))) \&$
 $(\text{a-truth}(\text{statement-by}(\text{husband})) \mid \text{knight}(\text{wife})) \&$
 $(\text{knight}(\text{wife}) \dashv\vdash \text{a-truth}(\text{statement-by}(\text{husband}))) \&$
 $(\sim \text{knight}(\text{husband})) \dashv\vdash \text{False}$
by meson

lemma PUZ025-1:

$(\forall X. \text{a-truth}(\text{truthteller}(X)) \mid \text{a-truth}(\text{liar}(X))) \&$
 $(\forall X. \sim(\text{a-truth}(\text{truthteller}(X)) \& \text{a-truth}(\text{liar}(X)))) \&$
 $(\forall \text{Truthteller Statement. a-truth}(\text{truthteller}(\text{Truthteller})) \& \text{a-truth}(\text{says}(\text{Truthteller}::'a, \text{Statement})))$
 $\dashv\vdash \text{a-truth}(\text{Statement})) \&$
 $(\forall \text{Liar Statement. } \sim(\text{a-truth}(\text{liar}(\text{Liar})) \& \text{a-truth}(\text{says}(\text{Liar}::'a, \text{Statement}))) \&$
 $\text{a-truth}(\text{Statement}))) \&$
 $(\forall \text{Statement Truthteller. a-truth}(\text{Statement}) \& \text{a-truth}(\text{says}(\text{Truthteller}::'a, \text{Statement})))$
 $\dashv\vdash \text{a-truth}(\text{truthteller}(\text{Truthteller}))) \&$
 $(\forall \text{Statement Liar. a-truth}(\text{says}(\text{Liar}::'a, \text{Statement})) \dashv\vdash \text{a-truth}(\text{Statement}) \mid$
 $\text{a-truth}(\text{liar}(\text{Liar}))) \&$
 $(\forall Z X Y. \text{people}(X::'a, Y, Z) \& \text{a-truth}(\text{liar}(X)) \& \text{a-truth}(\text{liar}(Y)) \dashv\vdash \text{a-truth}(\text{equal-type}(X::'a, Y)))$
 $\&$
 $(\forall Z X Y. \text{people}(X::'a, Y, Z) \& \text{a-truth}(\text{truthteller}(X)) \& \text{a-truth}(\text{truthteller}(Y)))$
 $\dashv\vdash \text{a-truth}(\text{equal-type}(X::'a, Y))) \&$
 $(\forall X Y. \text{a-truth}(\text{equal-type}(X::'a, Y)) \& \text{a-truth}(\text{truthteller}(X)) \dashv\vdash \text{a-truth}(\text{truthteller}(Y)))$
 $\&$
 $(\forall X Y. \text{a-truth}(\text{equal-type}(X::'a, Y)) \& \text{a-truth}(\text{liar}(X)) \dashv\vdash \text{a-truth}(\text{liar}(Y)))$
 $\&$

$(\forall X Y. a\text{-truth}(\text{truthteller}(X)) \dashv\dashv a\text{-truth}(\text{equal-type}(X::'a, Y)) \mid a\text{-truth}(\text{liar}(Y)))$
 $\&$
 $(\forall X Y. a\text{-truth}(\text{liar}(X)) \dashv\dashv a\text{-truth}(\text{equal-type}(X::'a, Y)) \mid a\text{-truth}(\text{truthteller}(Y)))$
 $\&$
 $(\forall Y X. a\text{-truth}(\text{equal-type}(X::'a, Y)) \dashv\dashv a\text{-truth}(\text{equal-type}(Y::'a, X))) \&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{truthteller}(X)) \& a\text{-truth}(Y) \dashv\dashv \text{answer}(\text{yes})) \&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{truthteller}(X)) \dashv\dashv a\text{-truth}(Y) \mid \text{answer}(\text{no})) \&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{liar}(X)) \& a\text{-truth}(Y) \dashv\dashv \text{answer}(\text{no}))$
 $\&$
 $(\forall X Y. \text{ask-1-if-2}(X::'a, Y) \& a\text{-truth}(\text{liar}(X)) \dashv\dashv a\text{-truth}(Y) \mid \text{answer}(\text{yes}))$
 $\&$
 $(\text{people}(b::'a, c, a)) \&$
 $(\text{people}(a::'a, b, a)) \&$
 $(\text{people}(a::'a, c, b)) \&$
 $(\text{people}(c::'a, b, a)) \&$
 $(a\text{-truth}(\text{says}(a::'a, \text{equal-type}(b::'a, c)))) \&$
 $(\text{ask-1-if-2}(c::'a, \text{equal-type}(a::'a, b))) \&$
 $(\forall \text{Answer}. \sim \text{answer}(\text{Answer})) \dashv\dashv \text{False}$
oops

lemma *PUZ029-1*:

$(\forall X. \text{dances-on-tightropes}(X) \mid \text{eats-pennybuns}(X) \mid \text{old}(X)) \&$
 $(\forall X. \text{pig}(X) \& \text{liable-to-giddiness}(X) \dashv\dashv \text{treated-with-respect}(X)) \&$
 $(\forall X. \text{wise}(X) \& \text{balloonist}(X) \dashv\dashv \text{has-umbrella}(X)) \&$
 $(\forall X. \sim(\text{looks-ridiculous}(X) \& \text{eats-pennybuns}(X) \& \text{eats-lunch-in-public}(X))) \&$
 $(\forall X. \text{balloonist}(X) \& \text{young}(X) \dashv\dashv \text{liable-to-giddiness}(X)) \&$
 $(\forall X. \text{fat}(X) \& \text{looks-ridiculous}(X) \dashv\dashv \text{dances-on-tightropes}(X) \mid \text{eats-lunch-in-public}(X))$
 $\&$
 $(\forall X. \sim(\text{liable-to-giddiness}(X) \& \text{wise}(X) \& \text{dances-on-tightropes}(X))) \&$
 $(\forall X. \text{pig}(X) \& \text{has-umbrella}(X) \dashv\dashv \text{looks-ridiculous}(X)) \&$
 $(\forall X. \text{treated-with-respect}(X) \dashv\dashv \text{dances-on-tightropes}(X) \mid \text{fat}(X)) \&$
 $(\forall X. \text{young}(X) \mid \text{old}(X)) \&$
 $(\forall X. \sim(\text{young}(X) \& \text{old}(X))) \&$
 $(\text{wise}(\text{piggy})) \&$
 $(\text{young}(\text{piggy})) \&$
 $(\text{pig}(\text{piggy})) \&$
 $(\text{balloonist}(\text{piggy})) \dashv\dashv \text{False}$
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:

$(\forall X. \text{sum}(\text{additive-identity}::'a, X, X)) \&$
 $(\forall X. \text{sum}(\text{additive-inverse}(X), X, \text{additive-identity})) \&$
 $(\forall Y U Z X V W. \text{sum}(X::'a, Y, U) \& \text{sum}(Y::'a, Z, V) \& \text{sum}(U::'a, Z, W) \longrightarrow$
 $\text{sum}(X::'a, V, W)) \&$
 $(\forall Y X V U Z W. \text{sum}(X::'a, Y, U) \& \text{sum}(Y::'a, Z, V) \& \text{sum}(X::'a, V, W) \longrightarrow$
 $\text{sum}(U::'a, Z, W)) \&$
 $(\forall X Y. \text{product}(X::'a, Y, \text{multiply}(X::'a, Y))) \&$
 $(\forall Y Z X V3 V1 V2 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$

$\& \text{product}(X::'a, V3, V4) \dashrightarrow \text{sum}(V1::'a, V2, V4)) \&$
 $(\forall Y Z V1 V2 X V3 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{sum}(V1::'a, V2, V4) \dashrightarrow \text{product}(X::'a, V3, V4)) \&$
 $(\sim \text{product}(a::'a, \text{additive-identity}, \text{additive-identity})) \dashrightarrow \text{False}$
oops

abbreviation *RNG-other-ax multiply add equal product additive-identity additive-inverse*

$\text{sum} \equiv$
 $(\forall X. \text{sum}(X::'a, \text{additive-inverse}(X), \text{additive-identity})) \&$
 $(\forall Y U Z X V W. \text{sum}(X::'a, Y, U) \& \text{sum}(Y::'a, Z, V) \& \text{sum}(U::'a, Z, W) \dashrightarrow$
 $\text{sum}(X::'a, V, W)) \&$
 $(\forall Y X V U Z W. \text{sum}(X::'a, Y, U) \& \text{sum}(Y::'a, Z, V) \& \text{sum}(X::'a, V, W) \dashrightarrow$
 $\text{sum}(U::'a, Z, W)) \&$
 $(\forall Y X Z. \text{sum}(X::'a, Y, Z) \dashrightarrow \text{sum}(Y::'a, X, Z)) \&$
 $(\forall Y U Z X V W. \text{product}(X::'a, Y, U) \& \text{product}(Y::'a, Z, V) \& \text{product}(U::'a, Z, W)$
 $\dashrightarrow \text{product}(X::'a, V, W)) \&$
 $(\forall Y X V U Z W. \text{product}(X::'a, Y, U) \& \text{product}(Y::'a, Z, V) \& \text{product}(X::'a, V, W)$
 $\dashrightarrow \text{product}(U::'a, Z, W)) \&$
 $(\forall Y Z X V3 V1 V2 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{product}(X::'a, V3, V4) \dashrightarrow \text{sum}(V1::'a, V2, V4)) \&$
 $(\forall Y Z V1 V2 X V3 V4. \text{product}(X::'a, Y, V1) \& \text{product}(X::'a, Z, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{sum}(V1::'a, V2, V4) \dashrightarrow \text{product}(X::'a, V3, V4)) \&$
 $(\forall Y Z V3 X V1 V2 V4. \text{product}(Y::'a, X, V1) \& \text{product}(Z::'a, X, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{product}(V3::'a, X, V4) \dashrightarrow \text{sum}(V1::'a, V2, V4)) \&$
 $(\forall Y Z V1 V2 V3 X V4. \text{product}(Y::'a, X, V1) \& \text{product}(Z::'a, X, V2) \& \text{sum}(Y::'a, Z, V3)$
 $\& \text{sum}(V1::'a, V2, V4) \dashrightarrow \text{product}(V3::'a, X, V4)) \&$
 $(\forall X Y U V. \text{sum}(X::'a, Y, U) \& \text{sum}(X::'a, Y, V) \dashrightarrow \text{equal}(U::'a, V)) \&$
 $(\forall X Y U V. \text{product}(X::'a, Y, U) \& \text{product}(X::'a, Y, V) \dashrightarrow \text{equal}(U::'a, V))$
 $\&$
 $(\forall X Y. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{additive-inverse}(X), \text{additive-inverse}(Y))) \&$
 $(\forall X Y W. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{add}(X::'a, W), \text{add}(Y::'a, W))) \&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{sum}(X::'a, W, Z) \dashrightarrow \text{sum}(Y::'a, W, Z)) \&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{sum}(W::'a, X, Z) \dashrightarrow \text{sum}(W::'a, Y, Z)) \&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{sum}(W::'a, Z, X) \dashrightarrow \text{sum}(W::'a, Z, Y)) \&$
 $(\forall X Y W. \text{equal}(X::'a, Y) \dashrightarrow \text{equal}(\text{multiply}(X::'a, W), \text{multiply}(Y::'a, W)))$
 $\&$
 $(\forall X Y W Z. \text{equal}(X::'a, Y) \& \text{product}(X::'a, W, Z) \dashrightarrow \text{product}(Y::'a, W, Z))$
 $\&$
 $(\forall X W Y Z. \text{equal}(X::'a, Y) \& \text{product}(W::'a, X, Z) \dashrightarrow \text{product}(W::'a, Y, Z))$
 $\&$
 $(\forall X W Z Y. \text{equal}(X::'a, Y) \& \text{product}(W::'a, Z, X) \dashrightarrow \text{product}(W::'a, Z, Y))$

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:

$\text{EQU001-0-ax equal} \ \&$
 $(\forall A B C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{add}(A::'a, C), \text{add}(B::'a, C))) \ \&$
 $(\forall D F' E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{add}(F'::'a, D), \text{add}(F'::'a, E))) \ \&$
 $(\forall G H. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{additive-inverse}(G), \text{additive-inverse}(H))) \ \&$
 $(\forall I' J K'. \text{equal}(I'::'a, J) \longrightarrow \text{equal}(\text{multiply}(I'::'a, K'), \text{multiply}(J::'a, K'))) \ \&$
 $(\forall L N M. \text{equal}(L::'a, M) \longrightarrow \text{equal}(\text{multiply}(N::'a, L), \text{multiply}(N::'a, M))) \ \&$
 $(\forall A B C D. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{associator}(A::'a, C, D), \text{associator}(B::'a, C, D)))$
 $\&$
 $(\forall E G F' H. \text{equal}(E::'a, F') \longrightarrow \text{equal}(\text{associator}(G::'a, E, H), \text{associator}(G::'a, F', H)))$
 $\&$
 $(\forall I' K' L J. \text{equal}(I'::'a, J) \longrightarrow \text{equal}(\text{associator}(K'::'a, L, I'), \text{associator}(K'::'a, L, J)))$
 $\&$
 $(\forall M N O'. \text{equal}(M::'a, N) \longrightarrow \text{equal}(\text{commutator}(M::'a, O'), \text{commutator}(N::'a, O')))$
 $\&$
 $(\forall P R Q. \text{equal}(P::'a, Q) \longrightarrow \text{equal}(\text{commutator}(R::'a, P), \text{commutator}(R::'a, Q)))$
 $\&$
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X))) \ \&$
 $(\forall X Y Z. \text{equal}(\text{add}(\text{add}(X::'a, Y), Z), \text{add}(X::'a, \text{add}(Y::'a, Z)))) \ \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-identity}), X)) \ \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-identity}::'a, X), X)) \ \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-inverse}(X)), \text{additive-identity})) \ \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-inverse}(X), X), \text{additive-identity})) \ \&$
 $(\text{equal}(\text{additive-inverse}(\text{additive-identity}), \text{additive-identity})) \ \&$
 $(\forall X Y. \text{equal}(\text{add}(X::'a, \text{add}(\text{additive-inverse}(X), Y)), Y)) \ \&$
 $(\forall X Y. \text{equal}(\text{additive-inverse}(\text{add}(X::'a, Y)), \text{add}(\text{additive-inverse}(X), \text{additive-inverse}(Y))))$
 $\&$
 $(\forall X. \text{equal}(\text{additive-inverse}(\text{additive-inverse}(X)), X)) \ \&$
 $(\forall X. \text{equal}(\text{multiply}(X::'a, \text{additive-identity}), \text{additive-identity})) \ \&$
 $(\forall X. \text{equal}(\text{multiply}(\text{additive-identity}::'a, X), \text{additive-identity})) \ \&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{additive-inverse}(X), \text{additive-inverse}(Y)), \text{multiply}(X::'a, Y)))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(X::'a, \text{additive-inverse}(Y)), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{additive-inverse}(X), Y), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall Y X Z. \text{equal}(\text{multiply}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{multiply}(\text{add}(X::'a, Y), Z), \text{add}(\text{multiply}(X::'a, Z), \text{multiply}(Y::'a, Z))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, Y), Y), \text{multiply}(X::'a, \text{multiply}(Y::'a, Y))))$

$\&$
 $(\forall X Y Z. \text{equal}(\text{associator}(X::'a, Y, Z), \text{add}(\text{multiply}(\text{multiply}(X::'a, Y), Z), \text{additive-inverse}(\text{multiply}(X::'a, Y, Z))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{commutator}(X::'a, Y), \text{add}(\text{multiply}(Y::'a, X), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(\text{associator}(X::'a, X, Y), X), \text{associator}(X::'a, X, Y)), \text{additive-identity}))$
 $\&$
 $(\sim \text{equal}(\text{multiply}(\text{multiply}(\text{associator}(a::'a, a, b), a), \text{associator}(a::'a, a, b)), \text{additive-identity}))$
 $---> \text{False}$
by meson

lemma RNG023-6:

$\text{EQU001-0-ax equal} \&$
 $(\forall Y X. \text{equal}(\text{add}(X::'a, Y), \text{add}(Y::'a, X))) \&$
 $(\forall X Y Z. \text{equal}(\text{add}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{add}(X::'a, Y), Z))) \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-identity}::'a, X), X)) \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-identity}), X)) \&$
 $(\forall X. \text{equal}(\text{multiply}(\text{additive-identity}::'a, X), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{multiply}(X::'a, \text{additive-identity}), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{add}(\text{additive-inverse}(X), X), \text{additive-identity})) \&$
 $(\forall X. \text{equal}(\text{add}(X::'a, \text{additive-inverse}(X)), \text{additive-identity})) \&$
 $(\forall Y X Z. \text{equal}(\text{multiply}(X::'a, \text{add}(Y::'a, Z)), \text{add}(\text{multiply}(X::'a, Y), \text{multiply}(X::'a, Z))))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{multiply}(\text{add}(X::'a, Y), Z), \text{add}(\text{multiply}(X::'a, Z), \text{multiply}(Y::'a, Z))))$
 $\&$
 $(\forall X. \text{equal}(\text{additive-inverse}(\text{additive-inverse}(X)), X)) \&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, Y), Y), \text{multiply}(X::'a, \text{multiply}(Y::'a, Y))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{multiply}(\text{multiply}(X::'a, X), Y), \text{multiply}(X::'a, \text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall X Y Z. \text{equal}(\text{associator}(X::'a, Y, Z), \text{add}(\text{multiply}(\text{multiply}(X::'a, Y), Z), \text{additive-inverse}(\text{multiply}(X::'a, Y, Z))))$
 $\&$
 $(\forall X Y. \text{equal}(\text{commutator}(X::'a, Y), \text{add}(\text{multiply}(Y::'a, X), \text{additive-inverse}(\text{multiply}(X::'a, Y))))$
 $\&$
 $(\forall D E F'. \text{equal}(D::'a, E) ---> \text{equal}(\text{add}(D::'a, F'), \text{add}(E::'a, F'))) \&$
 $(\forall G I' H. \text{equal}(G::'a, H) ---> \text{equal}(\text{add}(I'::'a, G), \text{add}(I'::'a, H))) \&$
 $(\forall J K'. \text{equal}(J::'a, K') ---> \text{equal}(\text{additive-inverse}(J), \text{additive-inverse}(K'))) \&$
 $(\forall L M N O'. \text{equal}(L::'a, M) ---> \text{equal}(\text{associator}(L::'a, N, O'), \text{associator}(M::'a, N, O')))$
 $\&$
 $(\forall P R Q S'. \text{equal}(P::'a, Q) ---> \text{equal}(\text{associator}(R::'a, P, S'), \text{associator}(R::'a, Q, S')))$
 $\&$
 $(\forall T' V W U. \text{equal}(T'::'a, U) ---> \text{equal}(\text{associator}(V::'a, W, T'), \text{associator}(V::'a, W, U)))$
 $\&$
 $(\forall X Y Z. \text{equal}(X::'a, Y) ---> \text{equal}(\text{commutator}(X::'a, Z), \text{commutator}(Y::'a, Z)))$
 $\&$
 $(\forall A1 C1 B1. \text{equal}(A1::'a, B1) ---> \text{equal}(\text{commutator}(C1::'a, A1), \text{commutator}(C1::'a, B1)))$
 $\&$
 $(\forall D1 E1 F1. \text{equal}(D1::'a, E1) ---> \text{equal}(\text{multiply}(D1::'a, F1), \text{multiply}(E1::'a, F1)))$

&
 ($\forall G1\ I1\ H1. \text{equal}(G1::'a, H1) \longrightarrow \text{equal}(\text{multiply}(I1::'a, G1), \text{multiply}(I1::'a, H1))$)
 &
 ($\sim \text{equal}(\text{associator}(x::'a, x, y), \text{additive-identity}) \longrightarrow \text{False}$)
by meson

lemma RNG028-2:

EQU001-0-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 *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 *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:

$\text{EQU001-0-ax equal} \&$
 $(\forall Y X. \text{member}(X::'a, Y) \longrightarrow \text{little-set}(X)) \&$
 $(\forall X Y. \text{little-set}(f1(X::'a, Y)) \mid \text{equal}(X::'a, Y)) \&$
 $(\forall X Y. \text{member}(f1(X::'a, Y), X) \mid \text{member}(f1(X::'a, Y), Y) \mid \text{equal}(X::'a, Y)) \&$
 $(\forall X Y. \text{member}(f1(X::'a, Y), X) \& \text{member}(f1(X::'a, Y), Y) \longrightarrow \text{equal}(X::'a, Y))$
 $\&$
 $(\forall X U Y. \text{member}(U::'a, \text{non-ordered-pair}(X::'a, Y)) \longrightarrow \text{equal}(U::'a, X) \mid \text{equal}(U::'a, Y))$
 $\&$
 $(\forall Y U X. \text{little-set}(U) \& \text{equal}(U::'a, X) \longrightarrow \text{member}(U::'a, \text{non-ordered-pair}(X::'a, Y)))$
 $\&$
 $(\forall X U Y. \text{little-set}(U) \& \text{equal}(U::'a, Y) \longrightarrow \text{member}(U::'a, \text{non-ordered-pair}(X::'a, Y)))$
 $\&$
 $(\forall X Y. \text{little-set}(\text{non-ordered-pair}(X::'a, Y))) \&$
 $(\forall X. \text{equal}(\text{singleton-set}(X), \text{non-ordered-pair}(X::'a, X))) \&$
 $(\forall X Y. \text{equal}(\text{ordered-pair}(X::'a, Y), \text{non-ordered-pair}(\text{singleton-set}(X), \text{non-ordered-pair}(X::'a, Y))))$
 $\&$
 $(\forall X. \text{ordered-pair-predicate}(X) \longrightarrow \text{little-set}(f2(X))) \&$
 $(\forall X. \text{ordered-pair-predicate}(X) \longrightarrow \text{little-set}(f3(X))) \&$

$(\forall X. \text{ordered-pair-predicate}(X) \longrightarrow \text{equal}(X::'a, \text{ordered-pair}(f2(X), f3(X)))) \&$
 $(\forall X Y Z. \text{little-set}(Y) \& \text{little-set}(Z) \& \text{equal}(X::'a, \text{ordered-pair}(Y::'a, Z)) \longrightarrow$
 $\text{ordered-pair-predicate}(X)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{little-set}(f4(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{little-set}(f5(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{equal}(X::'a, \text{ordered-pair}(f4(Z::'a, X), f5(Z::'a, X))))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{first}(X)) \longrightarrow \text{member}(Z::'a, f4(Z::'a, X))) \&$
 $(\forall X V Z U. \text{little-set}(U) \& \text{little-set}(V) \& \text{equal}(X::'a, \text{ordered-pair}(U::'a, V))$
 $\& \text{member}(Z::'a, U) \longrightarrow \text{member}(Z::'a, \text{first}(X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{little-set}(f6(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{little-set}(f7(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{equal}(X::'a, \text{ordered-pair}(f6(Z::'a, X), f7(Z::'a, X))))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{second}(X)) \longrightarrow \text{member}(Z::'a, f7(Z::'a, X))) \&$
 $(\forall X U Z V. \text{little-set}(U) \& \text{little-set}(V) \& \text{equal}(X::'a, \text{ordered-pair}(U::'a, V))$
 $\& \text{member}(Z::'a, V) \longrightarrow \text{member}(Z::'a, \text{second}(X))) \&$
 $(\forall Z. \text{member}(Z::'a, \text{estin}) \longrightarrow \text{ordered-pair-predicate}(Z)) \&$
 $(\forall Z. \text{member}(Z::'a, \text{estin}) \longrightarrow \text{member}(\text{first}(Z), \text{second}(Z))) \&$
 $(\forall Z. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Z) \& \text{member}(\text{first}(Z), \text{second}(Z))$
 $\longrightarrow \text{member}(Z::'a, \text{estin})) \&$
 $(\forall Y Z X. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, X)) \&$
 $(\forall X Z Y. \text{member}(Z::'a, \text{intersection}(X::'a, Y)) \longrightarrow \text{member}(Z::'a, Y)) \&$
 $(\forall X Z Y. \text{member}(Z::'a, X) \& \text{member}(Z::'a, Y) \longrightarrow \text{member}(Z::'a, \text{intersection}(X::'a, Y)))$
 $\&$
 $(\forall Z X. \sim(\text{member}(Z::'a, \text{complement}(X)) \& \text{member}(Z::'a, X))) \&$
 $(\forall Z X. \text{little-set}(Z) \longrightarrow \text{member}(Z::'a, \text{complement}(X)) \mid \text{member}(Z::'a, X)) \&$
 $(\forall X Y. \text{equal}(\text{union}(X::'a, Y), \text{complement}(\text{intersection}(\text{complement}(X), \text{complement}(Y))))))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{domain-of}(X)) \longrightarrow \text{ordered-pair-predicate}(f8(Z::'a, X)))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{domain-of}(X)) \longrightarrow \text{member}(f8(Z::'a, X), X)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{domain-of}(X)) \longrightarrow \text{equal}(Z::'a, \text{first}(f8(Z::'a, X)))) \&$
 $(\forall X Z Xp. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Xp) \& \text{member}(Xp::'a, X) \&$
 $\text{equal}(Z::'a, \text{first}(Xp)) \longrightarrow \text{member}(Z::'a, \text{domain-of}(X))) \&$
 $(\forall X Y Z. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{ordered-pair-predicate}(Z))$
 $\&$
 $(\forall Y Z X. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(\text{first}(Z), X)) \&$
 $(\forall X Z Y. \text{member}(Z::'a, \text{cross-product}(X::'a, Y)) \longrightarrow \text{member}(\text{second}(Z), Y))$
 $\&$
 $(\forall X Z Y. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Z) \& \text{member}(\text{first}(Z), X) \&$
 $\text{member}(\text{second}(Z), Y) \longrightarrow \text{member}(Z::'a, \text{cross-product}(X::'a, Y))) \&$
 $(\forall X Z. \text{member}(Z::'a, \text{inv1 } X) \longrightarrow \text{ordered-pair-predicate}(Z)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{inv1 } X) \longrightarrow \text{member}(\text{ordered-pair}(\text{second}(Z), \text{first}(Z)), X))$
 $\&$
 $(\forall Z X. \text{little-set}(Z) \& \text{ordered-pair-predicate}(Z) \& \text{member}(\text{ordered-pair}(\text{second}(Z), \text{first}(Z)), X)$
 $\longrightarrow \text{member}(Z::'a, \text{inv1 } X)) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{little-set}(f9(Z::'a, X))) \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{little-set}(f10(Z::'a, X))) \&$

$(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{little-set}(f11(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{equal}(Z::'a, \text{ordered-pair}(f9(Z::'a, X), \text{ordered-pair}(f10(Z::'a, X), f11(Z::'a, X)))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{rot-right}(X)) \longrightarrow \text{member}(\text{ordered-pair}(f10(Z::'a, X), \text{ordered-pair}(f11(Z::'a, X), f9(Z::'a, X)))) \ \&$
 $(\forall Z V W U X. \text{little-set}(Z) \ \& \ \text{little-set}(U) \ \& \ \text{little-set}(V) \ \& \ \text{little-set}(W) \ \& \ \text{equal}(Z::'a, \text{ordered-pair}(U::'a, \text{ordered-pair}(V::'a, W))) \ \& \ \text{member}(\text{ordered-pair}(V::'a, \text{ordered-pair}(W::'a, U))) \longrightarrow \text{member}(Z::'a, \text{rot-right}(X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{little-set}(f12(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{little-set}(f13(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{little-set}(f14(Z::'a, X))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{equal}(Z::'a, \text{ordered-pair}(f12(Z::'a, X), \text{ordered-pair}(f13(Z::'a, X), f14(Z::'a, X)))) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{flip-range-of}(X)) \longrightarrow \text{member}(\text{ordered-pair}(f12(Z::'a, X), \text{ordered-pair}(f14(Z::'a, X), f13(Z::'a, X)))) \ \&$
 $(\forall Z U W V X. \text{little-set}(Z) \ \& \ \text{little-set}(U) \ \& \ \text{little-set}(V) \ \& \ \text{little-set}(W) \ \& \ \text{equal}(Z::'a, \text{ordered-pair}(U::'a, \text{ordered-pair}(V::'a, W))) \ \& \ \text{member}(\text{ordered-pair}(U::'a, \text{ordered-pair}(W::'a, V))) \longrightarrow \text{member}(Z::'a, \text{flip-range-of}(X))) \ \&$
 $(\forall X. \text{equal}(\text{successor}(X), \text{union}(X::'a, \text{singleton-set}(X)))) \ \&$
 $(\forall Z. \sim \text{member}(Z::'a, \text{empty-set})) \ \&$
 $(\forall Z. \text{little-set}(Z) \longrightarrow \text{member}(Z::'a, \text{universal-set})) \ \&$
 $(\text{little-set}(\text{infinity})) \ \&$
 $(\text{member}(\text{empty-set}::'a, \text{infinity})) \ \&$
 $(\forall X. \text{member}(X::'a, \text{infinity}) \longrightarrow \text{member}(\text{successor}(X), \text{infinity})) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{sigma}(X)) \longrightarrow \text{member}(f16(Z::'a, X), X)) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{sigma}(X)) \longrightarrow \text{member}(Z::'a, f16(Z::'a, X))) \ \&$
 $(\forall X Z Y. \text{member}(Y::'a, X) \ \& \ \text{member}(Z::'a, Y) \longrightarrow \text{member}(Z::'a, \text{sigma}(X))) \ \&$
 $(\forall U. \text{little-set}(U) \longrightarrow \text{little-set}(\text{sigma}(U))) \ \&$
 $(\forall X U Y. \text{ssubset}(X::'a, Y) \ \& \ \text{member}(U::'a, X) \longrightarrow \text{member}(U::'a, Y)) \ \&$
 $(\forall Y X. \text{ssubset}(X::'a, Y) \mid \text{member}(f17(X::'a, Y), X)) \ \&$
 $(\forall X Y. \text{member}(f17(X::'a, Y), Y) \longrightarrow \text{ssubset}(X::'a, Y)) \ \&$
 $(\forall X Y. \text{proper-subset}(X::'a, Y) \longrightarrow \text{ssubset}(X::'a, Y)) \ \&$
 $(\forall X Y. \sim(\text{proper-subset}(X::'a, Y) \ \& \ \text{equal}(X::'a, Y))) \ \&$
 $(\forall X Y. \text{ssubset}(X::'a, Y) \longrightarrow \text{proper-subset}(X::'a, Y) \mid \text{equal}(X::'a, Y)) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{powerset}(X)) \longrightarrow \text{ssubset}(Z::'a, X)) \ \&$
 $(\forall Z X. \text{little-set}(Z) \ \& \ \text{ssubset}(Z::'a, X) \longrightarrow \text{member}(Z::'a, \text{powerset}(X))) \ \&$
 $(\forall U. \text{little-set}(U) \longrightarrow \text{little-set}(\text{powerset}(U))) \ \&$
 $(\forall Z X. \text{relation}(Z) \ \& \ \text{member}(X::'a, Z) \longrightarrow \text{ordered-pair-predicate}(X)) \ \&$
 $(\forall Z. \text{relation}(Z) \mid \text{member}(f18(Z), Z)) \ \&$
 $(\forall Z. \text{ordered-pair-predicate}(f18(Z)) \longrightarrow \text{relation}(Z)) \ \&$
 $(\forall U X V W. \text{single-valued-set}(X) \ \& \ \text{little-set}(U) \ \& \ \text{little-set}(V) \ \& \ \text{little-set}(W) \ \& \ \text{member}(\text{ordered-pair}(U::'a, V), X) \ \& \ \text{member}(\text{ordered-pair}(U::'a, W), X) \longrightarrow \text{equal}(V::'a, W)) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{little-set}(f19(X))) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{little-set}(f20(X))) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{little-set}(f21(X))) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{member}(\text{ordered-pair}(f19(X), f20(X)), X)) \ \&$
 $(\forall X. \text{single-valued-set}(X) \mid \text{member}(\text{ordered-pair}(f19(X), f21(X)), X)) \ \&$

$(\forall X. \text{equal}(f20(X), f21(X)) \longrightarrow \text{single-valued-set}(X)) \ \&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{relation}(Xf)) \ \&$
 $(\forall Xf. \text{function}(Xf) \longrightarrow \text{single-valued-set}(Xf)) \ \&$
 $(\forall Xf. \text{relation}(Xf) \ \& \ \text{single-valued-set}(Xf) \longrightarrow \text{function}(Xf)) \ \&$
 $(\forall Z X Xf. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{ordered-pair-predicate}(f22(Z::'a, X, Xf)))$
 $\&$
 $(\forall Z X Xf. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{member}(f22(Z::'a, X, Xf), Xf))$
 $\&$
 $(\forall Z Xf X. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{member}(\text{first}(f22(Z::'a, X, Xf)), X))$
 $\&$
 $(\forall X Xf Z. \text{member}(Z::'a, \text{image}'(X::'a, Xf)) \longrightarrow \text{equal}(\text{second}(f22(Z::'a, X, Xf)), Z))$
 $\&$
 $(\forall Xf X Y Z. \text{little-set}(Z) \ \& \ \text{ordered-pair-predicate}(Y) \ \& \ \text{member}(Y::'a, Xf) \ \&$
 $\text{member}(\text{first}(Y), X) \ \& \ \text{equal}(\text{second}(Y), Z) \longrightarrow \text{member}(Z::'a, \text{image}'(X::'a, Xf)))$
 $\&$
 $(\forall X Xf. \text{little-set}(X) \ \& \ \text{function}(Xf) \longrightarrow \text{little-set}(\text{image}'(X::'a, Xf))) \ \&$
 $(\forall X U Y. \sim(\text{disjoint}(X::'a, Y) \ \& \ \text{member}(U::'a, X) \ \& \ \text{member}(U::'a, Y))) \ \&$
 $(\forall Y X. \text{disjoint}(X::'a, Y) \mid \text{member}(f23(X::'a, Y), X)) \ \&$
 $(\forall X Y. \text{disjoint}(X::'a, Y) \mid \text{member}(f23(X::'a, Y), Y)) \ \&$
 $(\forall X. \text{equal}(X::'a, \text{empty-set}) \mid \text{member}(f24(X), X)) \ \&$
 $(\forall X. \text{equal}(X::'a, \text{empty-set}) \mid \text{disjoint}(f24(X), X)) \ \&$
 $(\text{function}(f25)) \ \&$
 $(\forall X. \text{little-set}(X) \longrightarrow \text{equal}(X::'a, \text{empty-set}) \mid \text{member}(f26(X), X)) \ \&$
 $(\forall X. \text{little-set}(X) \longrightarrow \text{equal}(X::'a, \text{empty-set}) \mid \text{member}(\text{ordered-pair}(X::'a, f26(X)), f25))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{range-of}(X)) \longrightarrow \text{ordered-pair-predicate}(f27(Z::'a, X)))$
 $\&$
 $(\forall Z X. \text{member}(Z::'a, \text{range-of}(X)) \longrightarrow \text{member}(f27(Z::'a, X), X)) \ \&$
 $(\forall Z X. \text{member}(Z::'a, \text{range-of}(X)) \longrightarrow \text{equal}(Z::'a, \text{second}(f27(Z::'a, X)))) \ \&$
 $(\forall X Z Xp. \text{little-set}(Z) \ \& \ \text{ordered-pair-predicate}(Xp) \ \& \ \text{member}(Xp::'a, X) \ \&$
 $\text{equal}(Z::'a, \text{second}(Xp)) \longrightarrow \text{member}(Z::'a, \text{range-of}(X))) \ \&$
 $(\forall Z. \text{member}(Z::'a, \text{identity-relation}) \longrightarrow \text{ordered-pair-predicate}(Z)) \ \&$
 $(\forall Z. \text{member}(Z::'a, \text{identity-relation}) \longrightarrow \text{equal}(\text{first}(Z), \text{second}(Z))) \ \&$
 $(\forall Z. \text{little-set}(Z) \ \& \ \text{ordered-pair-predicate}(Z) \ \& \ \text{equal}(\text{first}(Z), \text{second}(Z)) \longrightarrow$
 $\text{member}(Z::'a, \text{identity-relation})) \ \&$
 $(\forall X Y. \text{equal}(\text{restrct}(X::'a, Y), \text{intersection}(X::'a, \text{cross-product}(Y::'a, \text{universal-set}))))$
 $\&$
 $(\forall Xf. \text{one-to-one-function}(Xf) \longrightarrow \text{function}(Xf)) \ \&$
 $(\forall Xf. \text{one-to-one-function}(Xf) \longrightarrow \text{function}(\text{inv1 } Xf)) \ \&$
 $(\forall Xf. \text{function}(Xf) \ \& \ \text{function}(\text{inv1 } Xf) \longrightarrow \text{one-to-one-function}(Xf)) \ \&$
 $(\forall Z Xf Y. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{ordered-pair-predicate}(f28(Z::'a, Xf, Y)))$
 $\&$
 $(\forall Z Y Xf. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{member}(f28(Z::'a, Xf, Y), Xf))$
 $\&$
 $(\forall Z Xf Y. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{equal}(\text{first}(f28(Z::'a, Xf, Y)), Y))$
 $\&$
 $(\forall Z Xf Y. \text{member}(Z::'a, \text{apply}(Xf::'a, Y)) \longrightarrow \text{member}(Z::'a, \text{second}(f28(Z::'a, Xf, Y))))$
 $\&$
 $(\forall Xf Y Z W. \text{ordered-pair-predicate}(W) \ \& \ \text{member}(W::'a, Xf) \ \& \ \text{equal}(\text{first}(W), Y))$

$\& \text{ member}(Z::'a, \text{second}(W)) \longrightarrow \text{ member}(Z::'a, \text{apply}(Xf::'a, Y)) \&$
 $(\forall Xf X Y. \text{ equal}(\text{apply-to-two-arguments}(Xf::'a, X, Y), \text{apply}(Xf::'a, \text{ordered-pair}(X::'a, Y))))$
 $\&$
 $(\forall X Y Xf. \text{ maps}(Xf::'a, X, Y) \longrightarrow \text{ function}(Xf)) \&$
 $(\forall Y Xf X. \text{ maps}(Xf::'a, X, Y) \longrightarrow \text{ equal}(\text{domain-of}(Xf), X)) \&$
 $(\forall X Xf Y. \text{ maps}(Xf::'a, X, Y) \longrightarrow \text{ ssubset}(\text{range-of}(Xf), Y)) \&$
 $(\forall X Xf Y. \text{ function}(Xf) \& \text{ equal}(\text{domain-of}(Xf), X) \& \text{ ssubset}(\text{range-of}(Xf), Y))$
 $\longrightarrow \text{ maps}(Xf::'a, X, Y)) \&$
 $(\forall Xf Xs. \text{ closed}(Xs::'a, Xf) \longrightarrow \text{ little-set}(Xs)) \&$
 $(\forall Xs Xf. \text{ closed}(Xs::'a, Xf) \longrightarrow \text{ little-set}(Xf)) \&$
 $(\forall Xf Xs. \text{ closed}(Xs::'a, Xf) \longrightarrow \text{ maps}(Xf::'a, \text{cross-product}(Xs::'a, Xs), Xs)) \&$
 $(\forall Xf Xs. \text{ little-set}(Xs) \& \text{ little-set}(Xf) \& \text{ maps}(Xf::'a, \text{cross-product}(Xs::'a, Xs), Xs))$
 $\longrightarrow \text{ closed}(Xs::'a, Xf)) \&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ little-set}(f29(Z::'a, Xf, Xg)))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ little-set}(f30(Z::'a, Xf, Xg)))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ little-set}(f31(Z::'a, Xf, Xg)))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ equal}(Z::'a, \text{ordered-pair}(f29(Z::'a, Xf, Xg), f30(Z::'a, Xf, Xg))))$
 $\&$
 $(\forall Z Xg Xf. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ member}(\text{ordered-pair}(f29(Z::'a, Xf, Xg), f31(Z::'a, Xf, Xg))))$
 $\&$
 $(\forall Z Xf Xg. \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg)) \longrightarrow \text{ member}(\text{ordered-pair}(f31(Z::'a, Xf, Xg), f30(Z::'a, Xf, Xg))))$
 $\&$
 $(\forall Z X Xf W Y Xg. \text{ little-set}(Z) \& \text{ little-set}(X) \& \text{ little-set}(Y) \& \text{ little-set}(W) \&$
 $\text{ equal}(Z::'a, \text{ordered-pair}(X::'a, Y)) \& \text{ member}(\text{ordered-pair}(X::'a, W), Xf) \& \text{ mem-}$
 $\text{ ber}(\text{ordered-pair}(W::'a, Y), Xg) \longrightarrow \text{ member}(Z::'a, \text{composition}(Xf::'a, Xg))) \&$
 $(\forall Xh Xs2 Xf2 Xs1 Xf1. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \longrightarrow \text{ closed}(Xs1::'a, Xf1))$
 $\&$
 $(\forall Xh Xs1 Xf1 Xs2 Xf2. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \longrightarrow \text{ closed}(Xs2::'a, Xf2))$
 $\&$
 $(\forall Xf1 Xf2 Xh Xs1 Xs2. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \longrightarrow \text{ maps}(Xh::'a, Xs1, Xs2))$
 $\&$
 $(\forall Xs2 Xs1 Xf1 Xf2 X Xh Y. \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \& \text{ mem-}$
 $\text{ ber}(X::'a, Xs1) \& \text{ member}(Y::'a, Xs1) \longrightarrow \text{ equal}(\text{apply}(Xh::'a, \text{apply-to-two-arguments}(Xf1::'a, X, Y)), \text{apply-to-}$
 $\text{ two-arguments}(Xf1::'a, X, Y))) \&$
 $(\forall Xh Xf1 Xs2 Xf2 Xs1. \text{ closed}(Xs1::'a, Xf1) \& \text{ closed}(Xs2::'a, Xf2) \& \text{ maps}(Xh::'a, Xs1, Xs2)$
 $\longrightarrow \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \mid \text{ member}(f32(Xh::'a, Xs1, Xf1, Xs2, Xf2), Xs1))$
 $\&$
 $(\forall Xh Xf1 Xs2 Xf2 Xs1. \text{ closed}(Xs1::'a, Xf1) \& \text{ closed}(Xs2::'a, Xf2) \& \text{ maps}(Xh::'a, Xs1, Xs2)$
 $\longrightarrow \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2) \mid \text{ member}(f33(Xh::'a, Xs1, Xf1, Xs2, Xf2), Xs1))$
 $\&$
 $(\forall Xh Xs1 Xf1 Xs2 Xf2. \text{ closed}(Xs1::'a, Xf1) \& \text{ closed}(Xs2::'a, Xf2) \& \text{ maps}(Xh::'a, Xs1, Xs2)$
 $\& \text{ equal}(\text{apply}(Xh::'a, \text{apply-to-two-arguments}(Xf1::'a, f32(Xh::'a, Xs1, Xf1, Xs2, Xf2)), f33(Xh::'a, Xs1, Xf1, Xs2, Xf2)))$
 $\longrightarrow \text{ homomorphism}(Xh::'a, Xs1, Xf1, Xs2, Xf2)) \&$
 $(\forall A B C. \text{ equal}(A::'a, B) \longrightarrow \text{ equal}(f1(A::'a, C), f1(B::'a, C))) \&$
 $(\forall D F' E. \text{ equal}(D::'a, E) \longrightarrow \text{ equal}(f1(F'::'a, D), f1(F'::'a, E))) \&$
 $(\forall A2 B2. \text{ equal}(A2::'a, B2) \longrightarrow \text{ equal}(f2(A2), f2(B2))) \&$

$(\forall G4\ H4. \text{equal}(G4::'a,H4) \longrightarrow \text{equal}(f3(G4),f3(H4))) \ \&$
 $(\forall O7\ P7\ Q7. \text{equal}(O7::'a,P7) \longrightarrow \text{equal}(f4(O7::'a,Q7),f4(P7::'a,Q7))) \ \&$
 $(\forall R7\ T7\ S7. \text{equal}(R7::'a,S7) \longrightarrow \text{equal}(f4(T7::'a,R7),f4(T7::'a,S7))) \ \&$
 $(\forall U7\ V7\ W7. \text{equal}(U7::'a,V7) \longrightarrow \text{equal}(f5(U7::'a,W7),f5(V7::'a,W7))) \ \&$
 $(\forall X7\ Z7\ Y7. \text{equal}(X7::'a,Y7) \longrightarrow \text{equal}(f5(Z7::'a,X7),f5(Z7::'a,Y7))) \ \&$
 $(\forall A8\ B8\ C8. \text{equal}(A8::'a,B8) \longrightarrow \text{equal}(f6(A8::'a,C8),f6(B8::'a,C8))) \ \&$
 $(\forall D8\ F8\ E8. \text{equal}(D8::'a,E8) \longrightarrow \text{equal}(f6(F8::'a,D8),f6(F8::'a,E8))) \ \&$
 $(\forall G8\ H8\ I8. \text{equal}(G8::'a,H8) \longrightarrow \text{equal}(f7(G8::'a,I8),f7(H8::'a,I8))) \ \&$
 $(\forall J8\ L8\ K8. \text{equal}(J8::'a,K8) \longrightarrow \text{equal}(f7(L8::'a,J8),f7(L8::'a,K8))) \ \&$
 $(\forall M8\ N8\ O8. \text{equal}(M8::'a,N8) \longrightarrow \text{equal}(f8(M8::'a,O8),f8(N8::'a,O8))) \ \&$
 $(\forall P8\ R8\ Q8. \text{equal}(P8::'a,Q8) \longrightarrow \text{equal}(f8(R8::'a,P8),f8(R8::'a,Q8))) \ \&$
 $(\forall S8\ T8\ U8. \text{equal}(S8::'a,T8) \longrightarrow \text{equal}(f9(S8::'a,U8),f9(T8::'a,U8))) \ \&$
 $(\forall V8\ X8\ W8. \text{equal}(V8::'a,W8) \longrightarrow \text{equal}(f9(X8::'a,V8),f9(X8::'a,W8))) \ \&$
 $(\forall G\ H\ I'. \text{equal}(G::'a,H) \longrightarrow \text{equal}(f10(G::'a,I'),f10(H::'a,I'))) \ \&$
 $(\forall J\ L\ K'. \text{equal}(J::'a,K') \longrightarrow \text{equal}(f10(L::'a,J),f10(L::'a,K'))) \ \&$
 $(\forall M\ N\ O'. \text{equal}(M::'a,N) \longrightarrow \text{equal}(f11(M::'a,O'),f11(N::'a,O'))) \ \&$
 $(\forall P\ R\ Q. \text{equal}(P::'a,Q) \longrightarrow \text{equal}(f11(R::'a,P),f11(R::'a,Q))) \ \&$
 $(\forall S' \ T' \ U. \text{equal}(S'::'a,T') \longrightarrow \text{equal}(f12(S'::'a,U),f12(T'::'a,U))) \ \&$
 $(\forall V\ X\ W. \text{equal}(V::'a,W) \longrightarrow \text{equal}(f12(X::'a,V),f12(X::'a,W))) \ \&$
 $(\forall Y\ Z\ A1. \text{equal}(Y::'a,Z) \longrightarrow \text{equal}(f13(Y::'a,A1),f13(Z::'a,A1))) \ \&$
 $(\forall B1\ D1\ C1. \text{equal}(B1::'a,C1) \longrightarrow \text{equal}(f13(D1::'a,B1),f13(D1::'a,C1))) \ \&$
 $(\forall E1\ F1\ G1. \text{equal}(E1::'a,F1) \longrightarrow \text{equal}(f14(E1::'a,G1),f14(F1::'a,G1))) \ \&$
 $(\forall H1\ J1\ I1. \text{equal}(H1::'a,I1) \longrightarrow \text{equal}(f14(J1::'a,H1),f14(J1::'a,I1))) \ \&$
 $(\forall K1\ L1\ M1. \text{equal}(K1::'a,L1) \longrightarrow \text{equal}(f16(K1::'a,M1),f16(L1::'a,M1))) \ \&$
 $(\forall N1\ P1\ O1. \text{equal}(N1::'a,O1) \longrightarrow \text{equal}(f16(P1::'a,N1),f16(P1::'a,O1))) \ \&$
 $(\forall Q1\ R1\ S1. \text{equal}(Q1::'a,R1) \longrightarrow \text{equal}(f17(Q1::'a,S1),f17(R1::'a,S1))) \ \&$
 $(\forall T1\ V1\ U1. \text{equal}(T1::'a,U1) \longrightarrow \text{equal}(f17(V1::'a,T1),f17(V1::'a,U1))) \ \&$
 $(\forall W1\ X1. \text{equal}(W1::'a,X1) \longrightarrow \text{equal}(f18(W1),f18(X1))) \ \&$
 $(\forall Y1\ Z1. \text{equal}(Y1::'a,Z1) \longrightarrow \text{equal}(f19(Y1),f19(Z1))) \ \&$
 $(\forall C2\ D2. \text{equal}(C2::'a,D2) \longrightarrow \text{equal}(f20(C2),f20(D2))) \ \&$
 $(\forall E2\ F2. \text{equal}(E2::'a,F2) \longrightarrow \text{equal}(f21(E2),f21(F2))) \ \&$
 $(\forall G2\ H2\ I2\ J2. \text{equal}(G2::'a,H2) \longrightarrow \text{equal}(f22(G2::'a,I2,J2),f22(H2::'a,I2,J2)))$
 $\&$
 $(\forall K2\ M2\ L2\ N2. \text{equal}(K2::'a,L2) \longrightarrow \text{equal}(f22(M2::'a,K2,N2),f22(M2::'a,L2,N2)))$
 $\&$
 $(\forall O2\ Q2\ R2\ P2. \text{equal}(O2::'a,P2) \longrightarrow \text{equal}(f22(Q2::'a,R2,O2),f22(Q2::'a,R2,P2)))$
 $\&$
 $(\forall S2\ T2\ U2. \text{equal}(S2::'a,T2) \longrightarrow \text{equal}(f23(S2::'a,U2),f23(T2::'a,U2))) \ \&$
 $(\forall V2\ X2\ W2. \text{equal}(V2::'a,W2) \longrightarrow \text{equal}(f23(X2::'a,V2),f23(X2::'a,W2)))$
 $\&$
 $(\forall Y2\ Z2. \text{equal}(Y2::'a,Z2) \longrightarrow \text{equal}(f24(Y2),f24(Z2))) \ \&$
 $(\forall A3\ B3. \text{equal}(A3::'a,B3) \longrightarrow \text{equal}(f26(A3),f26(B3))) \ \&$
 $(\forall C3\ D3\ E3. \text{equal}(C3::'a,D3) \longrightarrow \text{equal}(f27(C3::'a,E3),f27(D3::'a,E3))) \ \&$
 $(\forall F3\ H3\ G3. \text{equal}(F3::'a,G3) \longrightarrow \text{equal}(f27(H3::'a,F3),f27(H3::'a,G3))) \ \&$
 $(\forall I3\ J3\ K3\ L3. \text{equal}(I3::'a,J3) \longrightarrow \text{equal}(f28(I3::'a,K3,L3),f28(J3::'a,K3,L3)))$
 $\&$
 $(\forall M3\ O3\ N3\ P3. \text{equal}(M3::'a,N3) \longrightarrow \text{equal}(f28(O3::'a,M3,P3),f28(O3::'a,N3,P3)))$
 $\&$
 $(\forall Q3\ S3\ T3\ R3. \text{equal}(Q3::'a,R3) \longrightarrow \text{equal}(f28(S3::'a,T3,Q3),f28(S3::'a,T3,R3)))$

$\&$
 $(\forall U3\ V3\ W3\ X3. \text{equal}(U3::'a, V3) \longrightarrow \text{equal}(f29(U3::'a, W3, X3), f29(V3::'a, W3, X3)))$
 $\&$
 $(\forall Y3\ A4\ Z3\ B4. \text{equal}(Y3::'a, Z3) \longrightarrow \text{equal}(f29(A4::'a, Y3, B4), f29(A4::'a, Z3, B4)))$
 $\&$
 $(\forall C4\ E4\ F4\ D4. \text{equal}(C4::'a, D4) \longrightarrow \text{equal}(f29(E4::'a, F4, C4), f29(E4::'a, F4, D4)))$
 $\&$
 $(\forall I4\ J4\ K4\ L4. \text{equal}(I4::'a, J4) \longrightarrow \text{equal}(f30(I4::'a, K4, L4), f30(J4::'a, K4, L4)))$
 $\&$
 $(\forall M4\ O4\ N4\ P4. \text{equal}(M4::'a, N4) \longrightarrow \text{equal}(f30(O4::'a, M4, P4), f30(O4::'a, N4, P4)))$
 $\&$
 $(\forall Q4\ S4\ T4\ R4. \text{equal}(Q4::'a, R4) \longrightarrow \text{equal}(f30(S4::'a, T4, Q4), f30(S4::'a, T4, R4)))$
 $\&$
 $(\forall U4\ V4\ W4\ X4. \text{equal}(U4::'a, V4) \longrightarrow \text{equal}(f31(U4::'a, W4, X4), f31(V4::'a, W4, X4)))$
 $\&$
 $(\forall Y4\ A5\ Z4\ B5. \text{equal}(Y4::'a, Z4) \longrightarrow \text{equal}(f31(A5::'a, Y4, B5), f31(A5::'a, Z4, B5)))$
 $\&$
 $(\forall C5\ E5\ F5\ D5. \text{equal}(C5::'a, D5) \longrightarrow \text{equal}(f31(E5::'a, F5, C5), f31(E5::'a, F5, D5)))$
 $\&$
 $(\forall G5\ H5\ I5\ J5\ K5\ L5. \text{equal}(G5::'a, H5) \longrightarrow \text{equal}(f32(G5::'a, I5, J5, K5, L5), f32(H5::'a, I5, J5, K5, L5)))$
 $\&$
 $(\forall M5\ O5\ N5\ P5\ Q5\ R5. \text{equal}(M5::'a, N5) \longrightarrow \text{equal}(f32(O5::'a, M5, P5, Q5, R5), f32(O5::'a, N5, P5, Q5, R5)))$
 $\&$
 $(\forall S5\ U5\ V5\ T5\ W5\ X5. \text{equal}(S5::'a, T5) \longrightarrow \text{equal}(f32(U5::'a, V5, S5, W5, X5), f32(U5::'a, V5, T5, W5, X5)))$
 $\&$
 $(\forall Y5\ A6\ B6\ C6\ Z5\ D6. \text{equal}(Y5::'a, Z5) \longrightarrow \text{equal}(f32(A6::'a, B6, C6, Y5, D6), f32(A6::'a, B6, C6, Z5, D6)))$
 $\&$
 $(\forall E6\ G6\ H6\ I6\ J6\ F6. \text{equal}(E6::'a, F6) \longrightarrow \text{equal}(f32(G6::'a, H6, I6, J6, E6), f32(G6::'a, H6, I6, J6, F6)))$
 $\&$
 $(\forall K6\ L6\ M6\ N6\ O6\ P6. \text{equal}(K6::'a, L6) \longrightarrow \text{equal}(f33(K6::'a, M6, N6, O6, P6), f33(L6::'a, M6, N6, O6, P6)))$
 $\&$
 $(\forall Q6\ S6\ R6\ T6\ U6\ V6. \text{equal}(Q6::'a, R6) \longrightarrow \text{equal}(f33(S6::'a, Q6, T6, U6, V6), f33(S6::'a, R6, T6, U6, V6)))$
 $\&$
 $(\forall W6\ Y6\ Z6\ X6\ A7\ B7. \text{equal}(W6::'a, X6) \longrightarrow \text{equal}(f33(Y6::'a, Z6, W6, A7, B7), f33(Y6::'a, Z6, X6, A7, B7)))$
 $\&$
 $(\forall C7\ E7\ F7\ G7\ D7\ H7. \text{equal}(C7::'a, D7) \longrightarrow \text{equal}(f33(E7::'a, F7, G7, C7, H7), f33(E7::'a, F7, G7, D7, H7)))$
 $\&$
 $(\forall I7\ K7\ L7\ M7\ N7\ J7. \text{equal}(I7::'a, J7) \longrightarrow \text{equal}(f33(K7::'a, L7, M7, N7, I7), f33(K7::'a, L7, M7, N7, J7)))$
 $\&$
 $(\forall A\ B\ C. \text{equal}(A::'a, B) \longrightarrow \text{equal}(\text{apply}(A::'a, C), \text{apply}(B::'a, C))) \ \&$
 $(\forall D\ F'\ E. \text{equal}(D::'a, E) \longrightarrow \text{equal}(\text{apply}(F'::'a, D), \text{apply}(F'::'a, E))) \ \&$
 $(\forall G\ H\ I'\ J. \text{equal}(G::'a, H) \longrightarrow \text{equal}(\text{apply-to-two-arguments}(G::'a, I', J), \text{apply-to-two-arguments}(H::'a, I', J)))$
 $\&$
 $(\forall K'\ M\ L\ N. \text{equal}(K'::'a, L) \longrightarrow \text{equal}(\text{apply-to-two-arguments}(M::'a, K', N), \text{apply-to-two-arguments}(M::'a, L, N)))$
 $\&$
 $(\forall O'\ Q\ R\ P. \text{equal}(O'::'a, P) \longrightarrow \text{equal}(\text{apply-to-two-arguments}(Q::'a, R, O'), \text{apply-to-two-arguments}(Q::'a, P, R)))$
 $\&$
 $(\forall S'\ T'. \text{equal}(S'::'a, T') \longrightarrow \text{equal}(\text{complement}(S'), \text{complement}(T'))) \ \&$
 $(\forall U\ V\ W. \text{equal}(U::'a, V) \longrightarrow \text{equal}(\text{composition}(U::'a, W), \text{composition}(V::'a, W)))$

$\&$
 $(\forall X Z Y. \text{equal}(X::'a, Y) \longrightarrow \text{equal}(\text{composition}(Z::'a, X), \text{composition}(Z::'a, Y)))$
 $\&$
 $(\forall A1 B1. \text{equal}(A1::'a, B1) \longrightarrow \text{equal}(\text{inv1 } A1, \text{inv1 } B1)) \&$
 $(\forall C1 D1 E1. \text{equal}(C1::'a, D1) \longrightarrow \text{equal}(\text{cross-product}(C1::'a, E1), \text{cross-product}(D1::'a, E1)))$
 $\&$
 $(\forall F1 H1 G1. \text{equal}(F1::'a, G1) \longrightarrow \text{equal}(\text{cross-product}(H1::'a, F1), \text{cross-product}(H1::'a, G1)))$
 $\&$
 $(\forall I1 J1. \text{equal}(I1::'a, J1) \longrightarrow \text{equal}(\text{domain-of}(I1), \text{domain-of}(J1))) \&$
 $(\forall I10 J10. \text{equal}(I10::'a, J10) \longrightarrow \text{equal}(\text{first}(I10), \text{first}(J10))) \&$
 $(\forall Q10 R10. \text{equal}(Q10::'a, R10) \longrightarrow \text{equal}(\text{flip-range-of}(Q10), \text{flip-range-of}(R10)))$
 $\&$
 $(\forall S10 T10 U10. \text{equal}(S10::'a, T10) \longrightarrow \text{equal}(\text{image}'(S10::'a, U10), \text{image}'(T10::'a, U10)))$
 $\&$
 $(\forall V10 X10 W10. \text{equal}(V10::'a, W10) \longrightarrow \text{equal}(\text{image}'(X10::'a, V10), \text{image}'(X10::'a, W10)))$
 $\&$
 $(\forall Y10 Z10 A11. \text{equal}(Y10::'a, Z10) \longrightarrow \text{equal}(\text{intersection}(Y10::'a, A11), \text{intersection}(Z10::'a, A11)))$
 $\&$
 $(\forall B11 D11 C11. \text{equal}(B11::'a, C11) \longrightarrow \text{equal}(\text{intersection}(D11::'a, B11), \text{intersection}(D11::'a, C11)))$
 $\&$
 $(\forall E11 F11 G11. \text{equal}(E11::'a, F11) \longrightarrow \text{equal}(\text{non-ordered-pair}(E11::'a, G11), \text{non-ordered-pair}(F11::'a, G11)))$
 $\&$
 $(\forall H11 J11 I11. \text{equal}(H11::'a, I11) \longrightarrow \text{equal}(\text{non-ordered-pair}(J11::'a, H11), \text{non-ordered-pair}(J11::'a, I11)))$
 $\&$
 $(\forall K11 L11 M11. \text{equal}(K11::'a, L11) \longrightarrow \text{equal}(\text{ordered-pair}(K11::'a, M11), \text{ordered-pair}(L11::'a, M11)))$
 $\&$
 $(\forall N11 P11 O11. \text{equal}(N11::'a, O11) \longrightarrow \text{equal}(\text{ordered-pair}(P11::'a, N11), \text{ordered-pair}(P11::'a, O11)))$
 $\&$
 $(\forall Q11 R11. \text{equal}(Q11::'a, R11) \longrightarrow \text{equal}(\text{powerset}(Q11), \text{powerset}(R11))) \&$
 $(\forall S11 T11. \text{equal}(S11::'a, T11) \longrightarrow \text{equal}(\text{range-of}(S11), \text{range-of}(T11))) \&$
 $(\forall U11 V11 W11. \text{equal}(U11::'a, V11) \longrightarrow \text{equal}(\text{restrct}(U11::'a, W11), \text{restrct}(V11::'a, W11)))$
 $\&$
 $(\forall X11 Z11 Y11. \text{equal}(X11::'a, Y11) \longrightarrow \text{equal}(\text{restrct}(Z11::'a, X11), \text{restrct}(Z11::'a, Y11)))$
 $\&$
 $(\forall A12 B12. \text{equal}(A12::'a, B12) \longrightarrow \text{equal}(\text{rot-right}(A12), \text{rot-right}(B12))) \&$
 $(\forall C12 D12. \text{equal}(C12::'a, D12) \longrightarrow \text{equal}(\text{second}(C12), \text{second}(D12))) \&$
 $(\forall K12 L12. \text{equal}(K12::'a, L12) \longrightarrow \text{equal}(\text{sigma}(K12), \text{sigma}(L12))) \&$
 $(\forall M12 N12. \text{equal}(M12::'a, N12) \longrightarrow \text{equal}(\text{singleton-set}(M12), \text{singleton-set}(N12)))$
 $\&$
 $(\forall O12 P12. \text{equal}(O12::'a, P12) \longrightarrow \text{equal}(\text{successor}(O12), \text{successor}(P12))) \&$
 $(\forall Q12 R12 S12. \text{equal}(Q12::'a, R12) \longrightarrow \text{equal}(\text{union}(Q12::'a, S12), \text{union}(R12::'a, S12)))$
 $\&$
 $(\forall T12 V12 U12. \text{equal}(T12::'a, U12) \longrightarrow \text{equal}(\text{union}(V12::'a, T12), \text{union}(V12::'a, U12)))$
 $\&$
 $(\forall W12 X12 Y12. \text{equal}(W12::'a, X12) \& \text{closed}(W12::'a, Y12) \longrightarrow \text{closed}(X12::'a, Y12))$
 $\&$
 $(\forall Z12 B13 A13. \text{equal}(Z12::'a, A13) \& \text{closed}(B13::'a, Z12) \longrightarrow \text{closed}(B13::'a, A13))$
 $\&$
 $(\forall C13 D13 E13. \text{equal}(C13::'a, D13) \& \text{disjoint}(C13::'a, E13) \longrightarrow \text{disjoint}(D13::'a, E13))$

&
 ($\forall F13\ H13\ G13. \text{equal}(F13::'a, G13) \ \& \ \text{disjoint}(H13::'a, F13) \longrightarrow \text{disjoint}(H13::'a, G13)$)
 &
 ($\forall I13\ J13. \text{equal}(I13::'a, J13) \ \& \ \text{function}(I13) \longrightarrow \text{function}(J13)$) &
 ($\forall K13\ L13\ M13\ N13\ O13\ P13. \text{equal}(K13::'a, L13) \ \& \ \text{homomorphism}(K13::'a, M13, N13, O13, P13) \longrightarrow \text{homomorphism}(L13::'a, M13, N13, O13, P13)$) &
 ($\forall Q13\ S13\ R13\ T13\ U13\ V13. \text{equal}(Q13::'a, R13) \ \& \ \text{homomorphism}(S13::'a, Q13, T13, U13, V13) \longrightarrow \text{homomorphism}(S13::'a, R13, T13, U13, V13)$) &
 ($\forall W13\ Y13\ Z13\ X13\ A14\ B14. \text{equal}(W13::'a, X13) \ \& \ \text{homomorphism}(Y13::'a, Z13, W13, A14, B14) \longrightarrow \text{homomorphism}(Y13::'a, Z13, X13, A14, B14)$) &
 ($\forall C14\ E14\ F14\ G14\ D14\ H14. \text{equal}(C14::'a, D14) \ \& \ \text{homomorphism}(E14::'a, F14, G14, C14, H14) \longrightarrow \text{homomorphism}(E14::'a, F14, G14, D14, H14)$) &
 ($\forall I14\ K14\ L14\ M14\ N14\ J14. \text{equal}(I14::'a, J14) \ \& \ \text{homomorphism}(K14::'a, L14, M14, N14, I14) \longrightarrow \text{homomorphism}(K14::'a, L14, M14, N14, J14)$) &
 ($\forall O14\ P14. \text{equal}(O14::'a, P14) \ \& \ \text{little-set}(O14) \longrightarrow \text{little-set}(P14)$) &
 ($\forall Q14\ R14\ S14\ T14. \text{equal}(Q14::'a, R14) \ \& \ \text{maps}(Q14::'a, S14, T14) \longrightarrow \text{maps}(R14::'a, S14, T14)$)
 &
 ($\forall U14\ W14\ V14\ X14. \text{equal}(U14::'a, V14) \ \& \ \text{maps}(W14::'a, U14, X14) \longrightarrow \text{maps}(W14::'a, V14, X14)$) &
 ($\forall Y14\ A15\ B15\ Z14. \text{equal}(Y14::'a, Z14) \ \& \ \text{maps}(A15::'a, B15, Y14) \longrightarrow \text{maps}(A15::'a, B15, Z14)$)
 &
 ($\forall C15\ D15\ E15. \text{equal}(C15::'a, D15) \ \& \ \text{member}(C15::'a, E15) \longrightarrow \text{member}(D15::'a, E15)$)
 &
 ($\forall F15\ H15\ G15. \text{equal}(F15::'a, G15) \ \& \ \text{member}(H15::'a, F15) \longrightarrow \text{member}(H15::'a, G15)$)
 &
 ($\forall I15\ J15. \text{equal}(I15::'a, J15) \ \& \ \text{one-to-one-function}(I15) \longrightarrow \text{one-to-one-function}(J15)$)
 &
 ($\forall K15\ L15. \text{equal}(K15::'a, L15) \ \& \ \text{ordered-pair-predicate}(K15) \longrightarrow \text{ordered-pair-predicate}(L15)$)
 &
 ($\forall M15\ N15\ O15. \text{equal}(M15::'a, N15) \ \& \ \text{proper-subset}(M15::'a, O15) \longrightarrow \text{proper-subset}(N15::'a, O15)$)
 &
 ($\forall P15\ R15\ Q15. \text{equal}(P15::'a, Q15) \ \& \ \text{proper-subset}(R15::'a, P15) \longrightarrow \text{proper-subset}(R15::'a, Q15)$)
 &
 ($\forall S15\ T15. \text{equal}(S15::'a, T15) \ \& \ \text{relation}(S15) \longrightarrow \text{relation}(T15)$) &
 ($\forall U15\ V15. \text{equal}(U15::'a, V15) \ \& \ \text{single-valued-set}(U15) \longrightarrow \text{single-valued-set}(V15)$)
 &
 ($\forall W15\ X15\ Y15. \text{equal}(W15::'a, X15) \ \& \ \text{ssubset}(W15::'a, Y15) \longrightarrow \text{ssubset}(X15::'a, Y15)$)
 &
 ($\forall Z15\ B16\ A16. \text{equal}(Z15::'a, A16) \ \& \ \text{ssubset}(B16::'a, Z15) \longrightarrow \text{ssubset}(B16::'a, A16)$)
 &
 ($\sim \text{little-set}(\text{ordered-pair}(a::'a, b))) \longrightarrow \text{False}$
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:

$(\forall Vf \ U. \text{element-of-set}(U::'a, \text{union-of-members}(Vf)) \dashrightarrow \text{element-of-set}(U::'a, f1(Vf::'a, U)))$

$\&$
 $(\forall U Vf. \text{element-of-set}(U::'a, \text{union-of-members}(Vf)) \longrightarrow \text{element-of-collection}(f1(Vf::'a, U), Vf))$
 $\&$
 $(\forall U Uu1 Vf. \text{element-of-set}(U::'a, Uu1) \ \& \ \text{element-of-collection}(Uu1::'a, Vf) \longrightarrow \text{element-of-set}(U::'a, \text{union-of-members}(Vf))) \ \&$
 $(\forall Vf U Va. \text{element-of-set}(U::'a, \text{intersection-of-members}(Vf)) \ \& \ \text{element-of-collection}(Va::'a, Vf) \longrightarrow \text{element-of-set}(U::'a, Va)) \ \&$
 $(\forall U Vf. \text{element-of-set}(U::'a, \text{intersection-of-members}(Vf)) \mid \text{element-of-collection}(f2(Vf::'a, U), Vf))$
 $\&$
 $(\forall Vf U. \text{element-of-set}(U::'a, f2(Vf::'a, U)) \longrightarrow \text{element-of-set}(U::'a, \text{intersection-of-members}(Vf)))$
 $\&$
 $(\forall Vt X. \text{topological-space}(X::'a, Vt) \longrightarrow \text{equal-sets}(\text{union-of-members}(Vt), X))$
 $\&$
 $(\forall X Vt. \text{topological-space}(X::'a, Vt) \longrightarrow \text{element-of-collection}(\text{empty-set}::'a, Vt))$
 $\&$
 $(\forall X Vt. \text{topological-space}(X::'a, Vt) \longrightarrow \text{element-of-collection}(X::'a, Vt)) \ \&$
 $(\forall X Y Z Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{element-of-collection}(Y::'a, Vt) \ \& \ \text{element-of-collection}(Z::'a, Vt) \longrightarrow \text{element-of-collection}(\text{intersection-of-sets}(Y::'a, Z), Vt))$
 $\&$
 $(\forall X Vf Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-collections}(Vf::'a, Vt) \longrightarrow \text{element-of-collection}(\text{union-of-members}(Vf), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f3(X::'a, Vt), Vt) \mid \text{subset-collections}(f5(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{union-of-members}(f5(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f3(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f4(X::'a, Vt), Vt) \mid \text{subset-collections}(f5(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{union-of-members}(f5(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{element-of-collection}(f4(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{intersection-of-sets}(f3(X::'a, Vt), f4(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt) \mid \text{subset-collections}(f5(X::'a, Vt), Vt)) \ \&$
 $(\forall X Vt. \text{equal-sets}(\text{union-of-members}(Vt), X) \ \& \ \text{element-of-collection}(\text{empty-set}::'a, Vt) \ \& \ \text{element-of-collection}(X::'a, Vt) \ \& \ \text{element-of-collection}(\text{intersection-of-sets}(f3(X::'a, Vt), f4(X::'a, Vt)), Vt) \ \& \ \text{element-of-collection}(\text{union-of-members}(f5(X::'a, Vt)), Vt) \longrightarrow \text{topological-space}(X::'a, Vt))$
 $\&$
 $(\forall U X Vt. \text{open}(U::'a, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall X U Vt. \text{open}(U::'a, X, Vt) \longrightarrow \text{element-of-collection}(U::'a, Vt)) \ \&$
 $(\forall X U Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{element-of-collection}(U::'a, Vt) \longrightarrow \text{open}(U::'a, X, Vt)) \ \&$
 $(\forall U X Vt. \text{closed}(U::'a, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall U X Vt. \text{closed}(U::'a, X, Vt) \longrightarrow \text{open}(\text{relative-complement-sets}(U::'a, X), X, Vt))$
 $\&$
 $(\forall U X Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{open}(\text{relative-complement-sets}(U::'a, X), X, Vt) \longrightarrow \text{closed}(U::'a, X, Vt)) \ \&$

$(\forall Vs X Vt. \text{finer}(Vt::'a, Vs, X) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall Vt X Vs. \text{finer}(Vt::'a, Vs, X) \longrightarrow \text{topological-space}(X::'a, Vs)) \ \&$
 $(\forall X Vs Vt. \text{finer}(Vt::'a, Vs, X) \longrightarrow \text{subset-collections}(Vs::'a, Vt)) \ \&$
 $(\forall X Vs Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{topological-space}(X::'a, Vs) \ \& \ \text{subset-collections}(Vs::'a, Vt) \longrightarrow \text{finer}(Vt::'a, Vs, X)) \ \&$
 $(\forall Vf X. \text{basis}(X::'a, Vf) \longrightarrow \text{equal-sets}(\text{union-of-members}(Vf), X)) \ \&$
 $(\forall X Vf Y Vb1 Vb2. \text{basis}(X::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, X) \ \& \ \text{element-of-collection}(Vb1::'a, Vf) \ \& \ \text{element-of-collection}(Vb2::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, \text{intersection-of-sets}(Vb1::'a, Vb2)) \longrightarrow \text{element-of-set}(Y::'a, f6(X::'a, Vf, Y, Vb1, Vb2))) \ \&$
 $(\forall X Y Vb1 Vb2 Vf. \text{basis}(X::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, X) \ \& \ \text{element-of-collection}(Vb1::'a, Vf) \ \& \ \text{element-of-collection}(Vb2::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, \text{intersection-of-sets}(Vb1::'a, Vb2)) \longrightarrow \text{element-of-collection}(f6(X::'a, Vf, Y, Vb1, Vb2), Vf)) \ \&$
 $(\forall X Vf Y Vb1 Vb2. \text{basis}(X::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, X) \ \& \ \text{element-of-collection}(Vb1::'a, Vf) \ \& \ \text{element-of-collection}(Vb2::'a, Vf) \ \& \ \text{element-of-set}(Y::'a, \text{intersection-of-sets}(Vb1::'a, Vb2)) \longrightarrow \text{subset-sets}(f6(X::'a, Vf, Y, Vb1, Vb2), \text{intersection-of-sets}(Vb1::'a, Vb2))) \ \&$
 $(\forall Vf X. \text{equal-sets}(\text{union-of-members}(Vf), X) \longrightarrow \text{basis}(X::'a, Vf) \mid \text{element-of-set}(f7(X::'a, Vf), X)) \ \&$
 $(\forall X Vf. \text{equal-sets}(\text{union-of-members}(Vf), X) \longrightarrow \text{basis}(X::'a, Vf) \mid \text{element-of-collection}(f8(X::'a, Vf), Vf)) \ \&$
 $(\forall X Vf. \text{equal-sets}(\text{union-of-members}(Vf), X) \longrightarrow \text{basis}(X::'a, Vf) \mid \text{element-of-collection}(f9(X::'a, Vf), Vf)) \ \&$
 $(\forall X Vf. \text{equal-sets}(\text{union-of-members}(Vf), X) \longrightarrow \text{basis}(X::'a, Vf) \mid \text{element-of-set}(f7(X::'a, Vf), \text{intersection-of-sets}(f8(X::'a, Vf), f9(X::'a, Vf)))) \ \&$
 $(\forall Uu9 X Vf. \text{equal-sets}(\text{union-of-members}(Vf), X) \ \& \ \text{element-of-set}(f7(X::'a, Vf), Uu9) \ \& \ \text{element-of-collection}(Uu9::'a, Vf) \ \& \ \text{subset-sets}(Uu9::'a, \text{intersection-of-sets}(f8(X::'a, Vf), f9(X::'a, Vf))) \longrightarrow \text{basis}(X::'a, Vf)) \ \&$
 $(\forall Vf U X. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \ \text{element-of-set}(X::'a, U) \longrightarrow \text{element-of-set}(X::'a, f10(Vf::'a, U, X))) \ \&$
 $(\forall U X Vf. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \ \text{element-of-set}(X::'a, U) \longrightarrow \text{element-of-collection}(f10(Vf::'a, U, X), Vf)) \ \&$
 $(\forall Vf X U. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \ \& \ \text{element-of-set}(X::'a, U) \longrightarrow \text{subset-sets}(f10(Vf::'a, U, X), U)) \ \&$
 $(\forall Vf U. \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf)) \mid \text{element-of-set}(f11(Vf::'a, U), U)) \ \&$
 $(\forall Vf Uu11 U. \text{element-of-set}(f11(Vf::'a, U), Uu11) \ \& \ \text{element-of-collection}(Uu11::'a, Vf) \ \& \ \text{subset-sets}(Uu11::'a, U) \longrightarrow \text{element-of-collection}(U::'a, \text{top-of-basis}(Vf))) \ \&$
 $(\forall U Y X Vt. \text{element-of-collection}(U::'a, \text{subspace-topology}(X::'a, Vt, Y)) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall U Vt Y X. \text{element-of-collection}(U::'a, \text{subspace-topology}(X::'a, Vt, Y)) \longrightarrow \text{subset-sets}(Y::'a, X)) \ \&$
 $(\forall X Y U Vt. \text{element-of-collection}(U::'a, \text{subspace-topology}(X::'a, Vt, Y)) \longrightarrow \text{element-of-collection}(f12(X::'a, Vt, Y, U), Vt)) \ \&$
 $(\forall X Vt Y U. \text{element-of-collection}(U::'a, \text{subspace-topology}(X::'a, Vt, Y)) \longrightarrow \text{equal-sets}(U::'a, \text{intersection-of-sets}(Y::'a, f12(X::'a, Vt, Y, U)))) \ \&$
 $(\forall X Vt U Y Uu12. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \ \& \ \text{element-of-collection}(Uu12::'a, Vt) \ \& \ \text{equal-sets}(U::'a, \text{intersection-of-sets}(Y::'a, Uu12)) \longrightarrow \text{element-of-collection}(U::'a, \text{subspace-topology}(X::'a, Vt, Y))) \ \&$
 $(\forall U Y X Vt. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$

$(\forall U \ Vt \ Y \ X. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{subset-sets}(Y::'a, X))$
 $\&$
 $(\forall Y \ X \ Vt \ U. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{element-of-set}(U::'a, f13(Y::'a, X, Vt, U)))$
 $\&$
 $(\forall X \ Vt \ U \ Y. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{subset-sets}(f13(Y::'a, X, Vt, U), Y))$
 $\&$
 $(\forall Y \ U \ X \ Vt. \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)) \longrightarrow \text{open}(f13(Y::'a, X, Vt, U), X, Vt))$
 $\&$
 $(\forall U \ Y \ Uu13 \ X \ Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \ \& \ \text{element-of-set}(U::'a, Uu13)$
 $\& \ \text{subset-sets}(Uu13::'a, Y) \ \& \ \text{open}(Uu13::'a, X, Vt) \longrightarrow \text{element-of-set}(U::'a, \text{interior}(Y::'a, X, Vt)))$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)) \longrightarrow \text{topological-space}(X::'a, Vt))$
 $\&$
 $(\forall U \ Vt \ Y \ X. \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)) \longrightarrow \text{subset-sets}(Y::'a, X))$
 $\&$
 $(\forall Y \ X \ Vt \ U \ V. \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)) \ \& \ \text{subset-sets}(Y::'a, V)$
 $\& \ \text{closed}(V::'a, X, Vt) \longrightarrow \text{element-of-set}(U::'a, V)) \ \&$
 $(\forall Y \ X \ Vt \ U. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X,$
 $| \ \text{subset-sets}(Y::'a, f14(Y::'a, X, Vt, U))) \ \&$
 $(\forall Y \ U \ X \ Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X,$
 $| \ \text{closed}(f14(Y::'a, X, Vt, U), X, Vt)) \ \&$
 $(\forall Y \ X \ Vt \ U. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \ \& \ \text{element-of-set}(U::'a, f14(Y::'a, X, Vt, U))$
 $\longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt))) \ \&$
 $(\forall U \ Y \ X \ Vt. \text{neighborhood}(U::'a, Y, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall Y \ U \ X \ Vt. \text{neighborhood}(U::'a, Y, X, Vt) \longrightarrow \text{open}(U::'a, X, Vt)) \ \&$
 $(\forall X \ Vt \ Y \ U. \text{neighborhood}(U::'a, Y, X, Vt) \longrightarrow \text{element-of-set}(Y::'a, U)) \ \&$
 $(\forall X \ Vt \ Y \ U. \text{topological-space}(X::'a, Vt) \ \& \ \text{open}(U::'a, X, Vt) \ \& \ \text{element-of-set}(Y::'a, U)$
 $\longrightarrow \text{neighborhood}(U::'a, Y, X, Vt)) \ \&$
 $(\forall Z \ Y \ X \ Vt. \text{limit-point}(Z::'a, Y, X, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall Z \ Vt \ Y \ X. \text{limit-point}(Z::'a, Y, X, Vt) \longrightarrow \text{subset-sets}(Y::'a, X)) \ \&$
 $(\forall Z \ X \ Vt \ U \ Y. \text{limit-point}(Z::'a, Y, X, Vt) \ \& \ \text{neighborhood}(U::'a, Z, X, Vt) \longrightarrow$
 $\text{element-of-set}(f15(Z::'a, Y, X, Vt, U), \text{intersection-of-sets}(U::'a, Y))) \ \&$
 $(\forall Y \ X \ Vt \ U \ Z. \sim(\text{limit-point}(Z::'a, Y, X, Vt) \ \& \ \text{neighborhood}(U::'a, Z, X, Vt) \ \&$
 $\text{eq-p}(f15(Z::'a, Y, X, Vt, U), Z))) \ \&$
 $(\forall Y \ Z \ X \ Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \longrightarrow \text{limit-point}(Z::'a, Y, X, Vt)$
 $| \ \text{neighborhood}(f16(Z::'a, Y, X, Vt), Z, X, Vt)) \ \&$
 $(\forall X \ Vt \ Y \ Uu16 \ Z. \text{topological-space}(X::'a, Vt) \ \& \ \text{subset-sets}(Y::'a, X) \ \& \ \text{element-of-set}(Uu16::'a, \text{intersection}$
 $\longrightarrow \text{limit-point}(Z::'a, Y, X, Vt) \ | \ \text{eq-p}(Uu16::'a, Z)) \ \&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{boundary}(Y::'a, X, Vt)) \longrightarrow \text{topological-space}(X::'a, Vt))$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{boundary}(Y::'a, X, Vt)) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt)))$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{element-of-set}(U::'a, \text{boundary}(Y::'a, X, Vt)) \longrightarrow \text{element-of-set}(U::'a, \text{closure}(\text{relative-complement}$
 $\&$
 $(\forall U \ Y \ X \ Vt. \text{topological-space}(X::'a, Vt) \ \& \ \text{element-of-set}(U::'a, \text{closure}(Y::'a, X, Vt))$
 $\& \ \text{element-of-set}(U::'a, \text{closure}(\text{relative-complement-sets}(Y::'a, X), X, Vt)) \longrightarrow \text{element-of-set}(U::'a, \text{boundary}$
 $\&$
 $(\forall X \ Vt. \text{hausdorff}(X::'a, Vt) \longrightarrow \text{topological-space}(X::'a, Vt)) \ \&$
 $(\forall X-2 \ X-1 \ X \ Vt. \text{hausdorff}(X::'a, Vt) \ \& \ \text{element-of-set}(X-1::'a, X) \ \& \ \text{element-of-set}(X-2::'a, X)$

$$\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

39 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).

39.1 Proof text

```
theorem tnd:  $A \vee \neg A$ 
proof –
  let  $?P = \lambda X. X = \text{False} \vee X = \text{True} \wedge A$ 
  let  $?Q = \lambda X. X = \text{False} \wedge A \vee X = \text{True}$ 

  have  $a$ :  $?P$  (Eps  $?P$ )
  proof (rule someI)
    have  $\text{False} = \text{False} \dots$ 
    thus  $?P$   $\text{False} \dots$ 
  qed
  have  $b$ :  $?Q$  (Eps  $?Q$ )
  proof (rule someI)
    have  $\text{True} = \text{True} \dots$ 
    thus  $?Q$   $\text{True} \dots$ 
  qed

  from  $a$  show  $?thesis$ 
  proof
    assume  $\text{Eps } ?P = \text{True} \wedge A$ 
    hence  $A \dots$ 
    thus  $?thesis \dots$ 
  next
    assume  $P$ :  $\text{Eps } ?P = \text{False}$ 
    from  $b$  show  $?thesis$ 
    proof
      assume  $\text{Eps } ?Q = \text{False} \wedge A$ 
      hence  $A \dots$ 
      thus  $?thesis \dots$ 
    next
      assume  $Q$ :  $\text{Eps } ?Q = \text{True}$ 
      have  $neg$ :  $?P \neq ?Q$ 
      proof
        assume  $?P = ?Q$ 
        hence  $\text{Eps } ?P = \text{Eps } ?Q$  by (rule arg-cong)
        also note  $P$ 
        also note  $Q$ 
        finally show  $\text{False}$  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 ∧ A ..
      thus ?Q x ..
    next
      assume x = True ∧ A
      hence x = True ..
      thus ?Q x ..
    qed
  next
    assume ?Q x
    thus ?P x
    proof
      assume x = False ∧ A
      hence x = False ..
      thus ?P x ..
    next
      assume x = True
      from this and a have x = True ∧ A ..
      thus ?P x ..
    qed
  qed
qed
qed
with neq show False by contradiction
qed
thus ?thesis ..
qed
qed
qed

```

39.2 Proof term of text

```

disjE · · · · ·
(someI · (λX. X = False ∨ X = True ∧ ?A) · · ·
  (disjI1 · · · · · (HOL.refl · -))) ·
(λH: -.
  disjE · · · · ·
    (someI · (λX. X = False ∧ ?A ∨ X = True) · · ·
      (disjI2 · · · · · (HOL.refl · -))) ·
    (λH: -. disjI1 · · · · · (conjE · · · · · H · (λ(H: -) H: -. H))) ·
    (λHa: -.
      disjI2 · · · · ·

```

```

(notI . . .
  (λHb: -.
    notE . . . . .
    (notI . . .
      (λHb: -.
        False-neq-True . . .
        (order-trans-rules-29 . . . . .
          (order-trans-rules-14 .
            (λa. a = (SOME X. X = False ∧ ?A ∨ X = True)) .
            . .
            . .
            (arg-cong · (λX. X = False ∨ X = True ∧ ?A) .
              (λX. X = False ∧ ?A ∨ X = True) .
              Eps .
              Hb) .
              H) .
              Ha))) .
    (ext . . . . .
      (λX. iffI . . . . .
        (λH: -.
          disjE . . . . . H .
          (λH: -. disjI1 . . . . . (conjI . . . . . H · Hb)) .
          (λH: -.
            disjI2 . . . . .
            (conjE . . . . . H · (λ(H: -) Ha: -. H)))) .
          (λH: -.
            disjE . . . . . H .
            (λH: -.
              disjI1 . . . . .
              (conjE . . . . . H · (λ(H: -) Ha: -. H))) .
              (λH: -.
                disjI2 . . . . . (conjI . . . . . H · Hb)))))))) .
        (λH: -. disjI1 . . . . . (conjE . . . . . H · (λ(H: -) H: -. H)))

```

39.3 Proof script

theorem *tnd'*: $A \vee \neg A$

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
  ( $\lambda x. (x = False) \vee (x = True) \wedge A$ )  $\neq$ 
  ( $\lambda x. (x = False) \wedge A \vee (x = True)$ ))
prefer 2
apply (rule notI)
apply (drule-tac  $f = \lambda y. \text{SOME } x. y \text{ } x$  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

```

39.4 Proof term of script

```

conjE . . . . .
(conjI . . . . .
  (someI . ( $\lambda x. x = False \vee x = True \wedge ?A$ ) . . .
    (disjI1 . . . . . (HOL.refl . -))) .
  (someI . ( $\lambda x. x = False \wedge ?A \vee x = True$ ) . . .
    (disjI2 . . . . . (HOL.refl . -)))) .
( $\lambda(H: -) Ha: -.$ 
  disjE . . . . . H .

```

```

( $\lambda H$ : -.
  disjE . . . . . Ha .
  ( $\lambda H$ : -. conjE . . . . . H . ( $\lambda H$ : -. disjI1 . . . . -)) .
  ( $\lambda Ha$ : -.
    disjI2 . . . . .
    (notI . . .
      ( $\lambda Hb$ : -.
        notE . . . . .
        (notI . . .
          ( $\lambda Hb$ : -.
            False-neg-True . . .
            (HOL.trans . . . . . (HOL.sym . . . . . H) .
              (HOL.trans . . . . .
                (arg-cong . ( $\lambda x$ .  $x = \text{False} \vee x = \text{True} \wedge ?A$ ) .
                  ( $\lambda x$ .  $x = \text{False} \wedge ?A \vee x = \text{True}$ ) .
                    Eps .
                    Hb) .
                    Ha)))))) .
            (ext . . . . .
              ( $\lambda x$ . iffI . . . . .
                ( $\lambda H$ : -.
                  disjE . . . . . H .
                  ( $\lambda H$ : -. disjI1 . . . . . (conjI . . . . . H . Hb)) .
                  ( $\lambda H$ : -.
                    conjE . . . . . H .
                    ( $\lambda (H: -)$  Ha: -. disjI2 . . . . . H))) .
                  ( $\lambda H$ : -.
                    disjE . . . . . H .
                    ( $\lambda H$ : -.
                      conjE . . . . . H .
                      ( $\lambda (H: -)$  Ha: -. disjI1 . . . . . H)) .
                      ( $\lambda H$ : -.
                        disjI2 . . . . . (conjI . . . . . H . Hb)))))))) .
                  ( $\lambda H$ : -. conjE . . . . . H . ( $\lambda H$ : -. disjI1 . . . . -)))
                )
              )
            )
          )
        )
      )
    )
  )
)

```

end

40 Dense linear order without endpoints and a quantifier elimination procedure in Ferrante and Rackoff style

theory Dense-Linear-Order

imports Arith-Tools

uses

~~/src/HOL/Tools/Qelim/qelim.ML

~~/src/HOL/Tools/Qelim/langford-data.ML

~~/src/HOL/Tools/Qelim/ferrante-rackoff-data.ML

```

( $\sim\sim$ /src/HOL/Tools/Qelim/langford.ML)
( $\sim\sim$ /src/HOL/Tools/Qelim/ferrante-rackoff.ML)
begin

setup Langford-Data.setup
setup Ferrante-Rackoff-Data.setup

context linorder
begin

lemma less-not-permute:  $\neg (x < y \wedge y < x)$  by (simp add: not-less linear)

lemma gather-simps:
  shows
     $(\exists x. (\forall y \in L. y < x) \wedge (\forall y \in U. x < y) \wedge x < u \wedge P x) \longleftrightarrow (\exists x. (\forall y \in L. y < x) \wedge (\forall y \in (\text{insert } u \ U). x < y) \wedge P x)$ 
    and  $(\exists x. (\forall y \in L. y < x) \wedge (\forall y \in U. x < y) \wedge l < x \wedge P x) \longleftrightarrow (\exists x. (\forall y \in (\text{insert } l \ L). y < x) \wedge (\forall y \in U. x < y) \wedge P x)$ 
     $(\exists x. (\forall y \in L. y < x) \wedge (\forall y \in U. x < y) \wedge x < u) \longleftrightarrow (\exists x. (\forall y \in L. y < x) \wedge (\forall y \in (\text{insert } u \ U). x < y))$ 
    and  $(\exists x. (\forall y \in L. y < x) \wedge (\forall y \in U. x < y) \wedge l < x) \longleftrightarrow (\exists x. (\forall y \in (\text{insert } l \ L). y < x) \wedge (\forall y \in U. x < y))$  by auto

lemma
  gather-start:  $(\exists x. P x) \equiv (\exists x. (\forall y \in \{\}. y < x) \wedge (\forall y \in \{\}. x < y) \wedge P x)$ 
  by simp

Theorems for  $\exists z. \forall x. x < z \longrightarrow (P x \longleftrightarrow P_{-\infty})$ 

lemma minf-lt:  $\exists z. \forall x. x < z \longrightarrow (x < t \longleftrightarrow \text{True})$  by auto
lemma minf-gt:  $\exists z. \forall x. x < z \longrightarrow (t < x \longleftrightarrow \text{False})$ 
  by (simp add: not-less) (rule exI[where x=t], auto simp add: less-le)

lemma minf-le:  $\exists z. \forall x. x < z \longrightarrow (x \leq t \longleftrightarrow \text{True})$  by (auto simp add: less-le)
lemma minf-ge:  $\exists z. \forall x. x < z \longrightarrow (t \leq x \longleftrightarrow \text{False})$ 
  by (auto simp add: less-le not-less not-le)
lemma minf-eq:  $\exists z. \forall x. x < z \longrightarrow (x = t \longleftrightarrow \text{False})$  by auto
lemma minf-neq:  $\exists z. \forall x. x < z \longrightarrow (x \neq t \longleftrightarrow \text{True})$  by auto
lemma minf-P:  $\exists z. \forall x. x < z \longrightarrow (P \longleftrightarrow P)$  by blast

Theorems for  $\exists z. \forall x. x < z \longrightarrow (P x \longleftrightarrow P_{+\infty})$ 

lemma pinf-gt:  $\exists z. \forall x. z < x \longrightarrow (t < x \longleftrightarrow \text{True})$  by auto
lemma pinf-lt:  $\exists z. \forall x. z < x \longrightarrow (x < t \longleftrightarrow \text{False})$ 
  by (simp add: not-less) (rule exI[where x=t], auto simp add: less-le)

lemma pinf-ge:  $\exists z. \forall x. z < x \longrightarrow (t \leq x \longleftrightarrow \text{True})$  by (auto simp add: less-le)
lemma pinf-le:  $\exists z. \forall x. z < x \longrightarrow (x \leq t \longleftrightarrow \text{False})$ 
  by (auto simp add: less-le not-less not-le)
lemma pinf-eq:  $\exists z. \forall x. z < x \longrightarrow (x = t \longleftrightarrow \text{False})$  by auto
lemma pinf-neq:  $\exists z. \forall x. z < x \longrightarrow (x \neq t \longleftrightarrow \text{True})$  by auto

```


lemma *pinf-P*: $\exists z. \forall x. z < x \longrightarrow (P \longleftrightarrow P)$ **by** *blast*

lemma *nmi-lt*: $t \in U \implies \forall x. \neg \text{True} \wedge x < t \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-gt*: $t \in U \implies \forall x. \neg \text{False} \wedge t < x \longrightarrow (\exists u \in U. u \leq x)$

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

lemma *nmi-le*: $t \in U \implies \forall x. \neg \text{True} \wedge x \leq t \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-ge*: $t \in U \implies \forall x. \neg \text{False} \wedge t \leq x \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-eq*: $t \in U \implies \forall x. \neg \text{False} \wedge x = t \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-neq*: $t \in U \implies \forall x. \neg \text{True} \wedge x \neq t \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-P*: $\forall x. \sim P \wedge P \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-conj*: $\llbracket \forall x. \neg P1' \wedge P1 x \longrightarrow (\exists u \in U. u \leq x) ;$

$\forall x. \neg P2' \wedge P2 x \longrightarrow (\exists u \in U. u \leq x) \rrbracket \implies$

$\forall x. \neg(P1' \wedge P2') \wedge (P1 x \wedge P2 x) \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *nmi-disj*: $\llbracket \forall x. \neg P1' \wedge P1 x \longrightarrow (\exists u \in U. u \leq x) ;$

$\forall x. \neg P2' \wedge P2 x \longrightarrow (\exists u \in U. u \leq x) \rrbracket \implies$

$\forall x. \neg(P1' \vee P2') \wedge (P1 x \vee P2 x) \longrightarrow (\exists u \in U. u \leq x)$ **by** *auto*

lemma *npi-lt*: $t \in U \implies \forall x. \neg \text{False} \wedge x < t \longrightarrow (\exists u \in U. x \leq u)$ **by** (*auto simp add: le-less*)

lemma *npi-gt*: $t \in U \implies \forall x. \neg \text{True} \wedge t < x \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-le*: $t \in U \implies \forall x. \neg \text{False} \wedge x \leq t \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-ge*: $t \in U \implies \forall x. \neg \text{True} \wedge t \leq x \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-eq*: $t \in U \implies \forall x. \neg \text{False} \wedge x = t \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-neq*: $t \in U \implies \forall x. \neg \text{True} \wedge x \neq t \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-P*: $\forall x. \sim P \wedge P \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-conj*: $\llbracket \forall x. \neg P1' \wedge P1 x \longrightarrow (\exists u \in U. x \leq u) ; \forall x. \neg P2' \wedge P2 x \longrightarrow (\exists u \in U. x \leq u) \rrbracket$

$\implies \forall x. \neg(P1' \wedge P2') \wedge (P1 x \wedge P2 x) \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *npi-disj*: $\llbracket \forall x. \neg P1' \wedge P1 x \longrightarrow (\exists u \in U. x \leq u) ; \forall x. \neg P2' \wedge P2 x \longrightarrow (\exists u \in U. x \leq u) \rrbracket$

$\implies \forall x. \neg(P1' \vee P2') \wedge (P1 x \vee P2 x) \longrightarrow (\exists u \in U. x \leq u)$ **by** *auto*

lemma *lin-dense-lt*: $t \in U \implies \forall x l u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge x < t \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow y < t)$

proof(*clarsimp*)

fix $x l u y$ **assume** $tU: t \in U$ **and** $noU: \forall t. l < t \wedge t < u \longrightarrow t \notin U$ **and** $lx: l < x$

and $xu: x < u$ **and** $px: x < t$ **and** $ly: l < y$ **and** $yu: y < u$

from $tU noU ly yu$ **have** $tny: t \neq y$ **by** *auto*

{assume $H: t < y$

from *less-trans*[*OF lx px*] *less-trans*[*OF H yu*]

have $l < t \wedge t < u$ **by** *simp*

with $tU noU$ **have** *False* **by** *auto*}

hence $\neg t < y$ **by** *auto* **hence** $y \leq t$ **by** (*simp add: not-less*)

thus $y < t$ **using** tny **by** (*simp add: less-le*)

qed

lemma *lin-dense-gt*: $t \in U \implies \forall x l u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge t < x \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow t < y)$

proof(*clarsimp*)
 fix $x\ l\ u\ y$
 assume $tU: t \in U$ and $noU: \forall t. l < t \wedge t < u \longrightarrow t \notin U$ and $lx: l < x$ and
 $xu: x < u$
 and $px: t < x$ and $ly: l < y$ and $yu: y < u$
 from $tU\ noU\ ly\ yu$ have $tny: t \neq y$ by *auto*
 {assume $H: y < t$
 from $less-trans[OF\ ly\ H]\ less-trans[OF\ px\ xu]$ have $l < t \wedge t < u$ by *simp*
 with $tU\ noU$ have *False* by *auto*}
 hence $\neg y < t$ by *auto* hence $t \leq y$ by (*auto simp add: not-less*)
 thus $t < y$ using tny by (*simp add: less-le*)
qed

lemma *lin-dense-le*: $t \in U \implies \forall x\ l\ u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge x \leq t \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow y \leq t)$

proof(*clarsimp*)
 fix $x\ l\ u\ y$
 assume $tU: t \in U$ and $noU: \forall t. l < t \wedge t < u \longrightarrow t \notin U$ and $lx: l < x$ and
 $xu: x < u$
 and $px: x \leq t$ and $ly: l < y$ and $yu: y < u$
 from $tU\ noU\ ly\ yu$ have $tny: t \neq y$ by *auto*
 {assume $H: t < y$
 from $less-le-trans[OF\ lx\ px]\ less-trans[OF\ H\ yu]$
 have $l < t \wedge t < u$ by *simp*
 with $tU\ noU$ have *False* by *auto*}
 hence $\neg t < y$ by *auto* thus $y \leq t$ by (*simp add: not-less*)
qed

lemma *lin-dense-ge*: $t \in U \implies \forall x\ l\ u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge t \leq x \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow t \leq y)$

proof(*clarsimp*)
 fix $x\ l\ u\ y$
 assume $tU: t \in U$ and $noU: \forall t. l < t \wedge t < u \longrightarrow t \notin U$ and $lx: l < x$ and
 $xu: x < u$
 and $px: t \leq x$ and $ly: l < y$ and $yu: y < u$
 from $tU\ noU\ ly\ yu$ have $tny: t \neq y$ by *auto*
 {assume $H: y < t$
 from $less-trans[OF\ ly\ H]\ le-less-trans[OF\ px\ xu]$
 have $l < t \wedge t < u$ by *simp*
 with $tU\ noU$ have *False* by *auto*}
 hence $\neg y < t$ by *auto* thus $t \leq y$ by (*simp add: not-less*)
qed

lemma *lin-dense-eq*: $t \in U \implies \forall x\ l\ u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge x = t \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow y = t)$ by *auto*

lemma *lin-dense-neg*: $t \in U \implies \forall x\ l\ u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge x \neq t \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow y \neq t)$ by *auto*

lemma *lin-dense-P*: $\forall x\ l\ u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow P)$ by *auto*

lemma *lin-dense-conj*:

$$\begin{aligned} & \llbracket \forall x \, l \, u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P1 \, x \\ & \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow P1 \, y) \ ; \\ & \forall x \, l \, u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P2 \, x \\ & \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow P2 \, y) \rrbracket \Longrightarrow \\ & \forall x \, l \, u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge (P1 \, x \wedge P2 \, x) \\ & \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow (P1 \, y \wedge P2 \, y)) \end{aligned}$$

by *blast*

lemma *lin-dense-disj*:

$$\begin{aligned} & \llbracket \forall x \, l \, u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P1 \, x \\ & \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow P1 \, y) \ ; \\ & \forall x \, l \, u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge P2 \, x \\ & \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow P2 \, y) \rrbracket \Longrightarrow \\ & \forall x \, l \, u. (\forall t. l < t \wedge t < u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge (P1 \, x \vee P2 \, x) \\ & \longrightarrow (\forall y. l < y \wedge y < u \longrightarrow (P1 \, y \vee P2 \, y)) \end{aligned}$$

by *blast*

lemma *npmibnd*: $\llbracket \forall x. \neg MP \wedge P \, x \longrightarrow (\exists u \in U. u \leq x); \forall x. \neg PP \wedge P \, x \longrightarrow (\exists u \in U. x \leq u) \rrbracket$

$\Longrightarrow \forall x. \neg MP \wedge \neg PP \wedge P \, x \longrightarrow (\exists u \in U. \exists u' \in U. u \leq x \wedge x \leq u')$

by *auto*

lemma *finite-set-intervals*:

assumes *px*: $P \, x$ **and** *lx*: $l \leq x$ **and** *xu*: $x \leq u$ **and** *linS*: $l \in S$

and *uinS*: $u \in S$ **and** *fS*: *finite* *S* **and** *lS*: $\forall x \in S. l \leq x$ **and** *Su*: $\forall x \in S. x \leq u$

shows $\exists a \in S. \exists b \in S. (\forall y. a < y \wedge y < b \longrightarrow y \notin S) \wedge a \leq x \wedge x \leq b \wedge P \, x$

proof–

let *?Mx* = $\{y. y \in S \wedge y \leq x\}$

let *?xM* = $\{y. y \in S \wedge x \leq y\}$

let *?a* = *Max* *?Mx*

let *?b* = *Min* *?xM*

have *MxS*: *?Mx* $\subseteq S$ **by** *blast*

hence *fMx*: *finite* *?Mx* **using** *fS* *finite-subset* **by** *auto*

from *lx* *linS* **have** *linMx*: $l \in ?Mx$ **by** *blast*

hence *Mxne*: *?Mx* $\neq \{\}$ **by** *blast*

have *xMS*: *?xM* $\subseteq S$ **by** *blast*

hence *fxM*: *finite* *?xM* **using** *fS* *finite-subset* **by** *auto*

from *xu* *uinS* **have** *linxM*: $u \in ?xM$ **by** *blast*

hence *xMne*: *?xM* $\neq \{\}$ **by** *blast*

have *ax*: $?a \leq x$ **using** *Mxne* *fMx* **by** *auto*

have *xb*: $x \leq ?b$ **using** *xMne* *fxM* **by** *auto*

have *?a* $\in ?Mx$ **using** *Max-in*[*OF* *fMx* *Mxne*] **by** *simp* **hence** *ainS*: *?a* $\in S$ **using** *MxS* **by** *blast*

have *?b* $\in ?xM$ **using** *Min-in*[*OF* *fxM* *xMne*] **by** *simp* **hence** *binS*: *?b* $\in S$ **using** *xMS* **by** *blast*

have *noy*: $\forall y. ?a < y \wedge y < ?b \longrightarrow y \notin S$

proof(*clarsimp*)

```

    fix y  assume ay: ?a < y and yb: y < ?b and yS: y ∈ S
    from yS have y ∈ ?Mx ∨ y ∈ ?xM by (auto simp add: linear)
    moreover {assume y ∈ ?Mx hence y ≤ ?a using Mxne fMx by auto with
ay have False by (simp add: not-le[symmetric])}
    moreover {assume y ∈ ?xM hence ?b ≤ y using xMne fMx by auto with
yb have False by (simp add: not-le[symmetric])}
    ultimately show False by blast
qed
from ainS binS noy ax xb px show ?thesis by blast
qed

```

lemma *finite-set-intervals2*:

```

    assumes px: P x and lx: l ≤ x and xu: x ≤ u and linS: l ∈ S
    and uinS: u ∈ S and fS: finite S and lS: ∀ x ∈ S. l ≤ x and Su: ∀ x ∈ S. x ≤
u
    shows (∃ s ∈ S. P s) ∨ (∃ a ∈ S. ∃ b ∈ S. (∀ y. a < y ∧ y < b ⟶ y ∉ S) ∧
a < x ∧ x < b ∧ P x)
proof-
  from finite-set-intervals[where P=P, OF px lx xu linS uinS fS lS Su]
  obtain a and b where
    as: a ∈ S and bs: b ∈ S and noS: ∀ y. a < y ∧ y < b ⟶ y ∉ S
    and axb: a ≤ x ∧ x ≤ b ∧ P x by auto
  from axb have x = a ∨ x = b ∨ (a < x ∧ x < b) by (auto simp add: le-less)
  thus ?thesis using px as bs noS by blast
qed

```

end

41 The classical QE after Langford for dense linear orders

context *dense-linear-order*
begin

lemma *dlo-qe-bnds*:

```

    assumes ne: L ≠ {} and neU: U ≠ {} and fL: finite L and fU: finite U
    shows (∃ x. (∀ y ∈ L. y < x) ∧ (∀ y ∈ U. x < y)) ≡ (∀ l ∈ L. ∀ u ∈ U. l < u)
proof (simp only: atomize-eq, rule iffI)
  assume H: ∃ x. (∀ y ∈ L. y < x) ∧ (∀ y ∈ U. x < y)
  then obtain x where xL: ∀ y ∈ L. y < x and xU: ∀ y ∈ U. x < y by blast
  {fix l u assume l: l ∈ L and u: u ∈ U
    have l < x using xL l by blast
    also have x < u using xU u by blast
    finally (less-trans) have l < u .}
  thus ∀ l ∈ L. ∀ u ∈ U. l < u by blast
next
  assume H: ∀ l ∈ L. ∀ u ∈ U. l < u
  let ?ML = Max L

```

```

let ?MU = Min U
from fL ne have th1: ?ML ∈ L and th1': ∀ l ∈ L. l ≤ ?ML by auto
from fU neU have th2: ?MU ∈ U and th2': ∀ u ∈ U. ?MU ≤ u by auto
from th1 th2 H have ?ML < ?MU by auto
with dense obtain w where th3: ?ML < w and th4: w < ?MU by blast
from th3 th1' have ∀ l ∈ L. l < w by auto
moreover from th4 th2' have ∀ u ∈ U. w < u by auto
ultimately show ∃ x. (∀ y ∈ L. y < x) ∧ (∀ y ∈ U. x < y) by auto
qed

```

```

lemma dlo-qe-noub:
  assumes ne: L ≠ {} and fL: finite L
  shows (∃ x. (∀ y ∈ L. y < x) ∧ (∀ y ∈ {}. x < y)) ≡ True
proof(simp add: atomize-eq)
  from gt-ex[of Max L] obtain M where M: Max L < M by blast
  from ne fL have ∀ x ∈ L. x ≤ Max L by simp
  with M have ∀ x ∈ L. x < M by (auto intro: le-less-trans)
  thus ∃ x. ∀ y ∈ L. y < x by blast
qed

```

```

lemma dlo-qe-nolb:
  assumes ne: U ≠ {} and fU: finite U
  shows (∃ x. (∀ y ∈ {}. y < x) ∧ (∀ y ∈ U. x < y)) ≡ True
proof(simp add: atomize-eq)
  from lt-ex[of Min U] obtain M where M: M < Min U by blast
  from ne fU have ∀ x ∈ U. Min U ≤ x by simp
  with M have ∀ x ∈ U. M < x by (auto intro: less-le-trans)
  thus ∃ x. ∀ y ∈ U. x < y by blast
qed

```

```

lemma exists-neq: ∃ (x::'a). x ≠ t ∃ (x::'a). t ≠ x
  using gt-ex[of t] by auto

```

```

lemmas dlo-simps = order-refl less-irrefl not-less not-le exists-neq
  le-less neq-iff linear less-not-permute

```

```

lemma axiom: dense-linear-order (op ≤) (op <) by (rule dense-linear-order-axioms)

```

```

lemma atoms:
  includes meta-term-syntax
  shows TERM (less :: 'a ⇒ -)
    and TERM (less-eq :: 'a ⇒ -)
    and TERM (op = :: 'a ⇒ -) .

```

```

declare axiom[langford qe: dlo-qe-bnds dlo-qe-nolb dlo-qe-noub gather: gather-start
gather-simps atoms: atoms]
declare dlo-simps[langfordsimp]

```

```

end

```

```

lemma dnf:
   $(P \ \& \ (Q \mid R)) = ((P \& Q) \mid (P \& R))$ 
   $((Q \mid R) \ \& \ P) = ((Q \& P) \mid (R \& P))$ 
by blast+

lemmas weak-dnf-simps = simp-thms dnf

lemma nnf-simps:
   $(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$   $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$   $(P \longrightarrow Q) = (\neg P \vee Q)$ 
   $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))$   $(\neg \neg(P)) = P$ 
by blast+

lemma ex-distrib:  $(\exists x. P \ x \vee Q \ x) \longleftrightarrow ((\exists x. P \ x) \vee (\exists x. Q \ x))$  by blast

lemmas dnf-simps = weak-dnf-simps nnf-simps ex-distrib

use  $\sim\sim$  /src/HOL/Tools/Qelim/langford.ML
method-setup dlo =  $\langle\langle$ 
  Method.txt-args (Method.SIMPLE-METHOD' o LangfordQE.dlo-tac)
 $\rangle\rangle$  Langford's algorithm for quantifier elimination in dense linear orders

```

42 Constructive dense linear orders yield QE for linear arithmetic over ordered Fields – see *Arith-Tools.thy*

Linear order without upper bounds

```

locale linorder-stupid-syntax = linorder
begin
notation
  less-eq (op  $\sqsubseteq$ ) and
  less-eq  $((-/ \sqsubseteq -) [51, 51] 50)$  and
  less (op  $\sqsubset$ ) and
  less  $((-/ \sqsubset -) [51, 51] 50)$ 

end

locale linorder-no-ub = linorder-stupid-syntax +
  assumes gt-ex:  $\exists y. \text{less } x \ y$ 
begin
lemma ge-ex:  $\exists y. x \sqsubseteq y$  using gt-ex by auto

```

Theorems for $\exists z. \forall x. z \sqsubset x \longrightarrow (P \ x \longleftrightarrow P_{+\infty})$

```

lemma pinf-conj:
  assumes ex1:  $\exists z1. \forall x. z1 \sqsubset x \longrightarrow (P1 \ x \longleftrightarrow P1')$ 
  and ex2:  $\exists z2. \forall x. z2 \sqsubset x \longrightarrow (P2 \ x \longleftrightarrow P2')$ 
  shows  $\exists z. \forall x. z \sqsubset x \longrightarrow ((P1 \ x \wedge P2 \ x) \longleftrightarrow (P1' \wedge P2'))$ 
proof –

```

```

from ex1 ex2 obtain z1 and z2 where z1:  $\forall x. z1 \sqsubseteq x \longrightarrow (P1\ x \longleftrightarrow P1')$ 
  and z2:  $\forall x. z2 \sqsubseteq x \longrightarrow (P2\ x \longleftrightarrow P2')$  by blast
from gt-ex obtain z where z: ord.max less-eq z1 z2  $\sqsubseteq z$  by blast
from z have zz1:  $z1 \sqsubseteq z$  and zz2:  $z2 \sqsubseteq z$  by simp-all
{fix x assume H:  $z \sqsubseteq x$ 
  from less-trans[OF zz1 H] less-trans[OF zz2 H]
  have  $(P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2')$  using z1 zz1 z2 zz2 by auto
}
thus ?thesis by blast
qed

```

```

lemma pinf-disj:
  assumes ex1:  $\exists z1. \forall x. z1 \sqsubseteq x \longrightarrow (P1\ x \longleftrightarrow P1')$ 
  and ex2:  $\exists z2. \forall x. z2 \sqsubseteq x \longrightarrow (P2\ x \longleftrightarrow P2')$ 
  shows  $\exists z. \forall x. z \sqsubseteq x \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$ 
proof-
  from ex1 ex2 obtain z1 and z2 where z1:  $\forall x. z1 \sqsubseteq x \longrightarrow (P1\ x \longleftrightarrow P1')$ 
    and z2:  $\forall x. z2 \sqsubseteq x \longrightarrow (P2\ x \longleftrightarrow P2')$  by blast
  from gt-ex obtain z where z: ord.max less-eq z1 z2  $\sqsubseteq z$  by blast
  from z have zz1:  $z1 \sqsubseteq z$  and zz2:  $z2 \sqsubseteq z$  by simp-all
  {fix x assume H:  $z \sqsubseteq x$ 
    from less-trans[OF zz1 H] less-trans[OF zz2 H]
    have  $(P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2')$  using z1 zz1 z2 zz2 by auto
  }
  thus ?thesis by blast
qed

```

```

lemma pinf-ex: assumes ex:  $\exists z. \forall x. z \sqsubseteq x \longrightarrow (P\ x \longleftrightarrow P1)$  and p1: P1 shows
 $\exists x. P\ x$ 
proof-
  from ex obtain z where z:  $\forall x. z \sqsubseteq x \longrightarrow (P\ x \longleftrightarrow P1)$  by blast
  from gt-ex obtain x where x:  $z \sqsubseteq x$  by blast
  from z x p1 show ?thesis by blast
qed

```

end

Linear order without upper bounds

```

locale linorder-no-lb = linorder-stupid-syntax +
  assumes lt-ex:  $\exists y. less\ y\ x$ 
begin
lemma le-ex:  $\exists y. y \sqsubseteq x$  using lt-ex by auto

```

Theorems for $\exists z. \forall x. x \sqsubseteq z \longrightarrow (P\ x \longleftrightarrow P_{-\infty})$

```

lemma minf-conj:
  assumes ex1:  $\exists z1. \forall x. x \sqsubseteq z1 \longrightarrow (P1\ x \longleftrightarrow P1')$ 
  and ex2:  $\exists z2. \forall x. x \sqsubseteq z2 \longrightarrow (P2\ x \longleftrightarrow P2')$ 
  shows  $\exists z. \forall x. x \sqsubseteq z \longrightarrow ((P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2'))$ 
proof-

```

```

    from ex1 ex2 obtain z1 and z2 where z1:  $\forall x. x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$  and
    z2:  $\forall x. x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$  by blast
    from lt-ex obtain z where z:  $z \sqsubset \text{ord.min less-eq } z1\ z2$  by blast
    from z have zz1:  $z \sqsubset z1$  and zz2:  $z \sqsubset z2$  by simp-all
    {fix x assume H:  $x \sqsubset z$ 
     from less-trans[OF H zz1] less-trans[OF H zz2]
     have  $(P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2')$  using z1 zz1 z2 zz2 by auto
    }
    thus ?thesis by blast
qed

```

lemma minf-disj:

```

    assumes ex1:  $\exists z1. \forall x. x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$ 
    and ex2:  $\exists z2. \forall x. x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$ 
    shows  $\exists z. \forall x. x \sqsubset z \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$ 
proof-
    from ex1 ex2 obtain z1 and z2 where z1:  $\forall x. x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$  and
    z2:  $\forall x. x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$  by blast
    from lt-ex obtain z where z:  $z \sqsubset \text{ord.min less-eq } z1\ z2$  by blast
    from z have zz1:  $z \sqsubset z1$  and zz2:  $z \sqsubset z2$  by simp-all
    {fix x assume H:  $x \sqsubset z$ 
     from less-trans[OF H zz1] less-trans[OF H zz2]
     have  $(P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2')$  using z1 zz1 z2 zz2 by auto
    }
    thus ?thesis by blast
qed

```

lemma minf-ex: assumes ex: $\exists z. \forall x. x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$ and p1: $P1$
shows $\exists x. P\ x$

```

proof-
    from ex obtain z where z:  $\forall x. x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$  by blast
    from lt-ex obtain x where x:  $x \sqsubset z$  by blast
    from z x p1 show ?thesis by blast
qed

```

end

```

locale constr-dense-linear-order = linorder-no-lb + linorder-no-ub +
  fixes between
  assumes between-less:  $\text{less } x\ y \implies \text{less } x\ (\text{between } x\ y) \wedge \text{less } (\text{between } x\ y)\ y$ 
  and between-same:  $\text{between } x\ x = x$ 

```

```

interpretation constr-dense-linear-order < dense-linear-order
  apply unfold-locales
  using gt-ex lt-ex between-less
  by (auto, rule-tac x=between x y in exI, simp)

```

```

context constr-dense-linear-order

```


begin

lemma *rinf-U*:

assumes *fU*: *finite U*
and *lin-dense*: $\forall x \ l \ u. (\forall \ t. \ l \sqsubset t \wedge t \sqsubset u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P \ x$
 $\longrightarrow (\forall \ y. \ l \sqsubset y \wedge y \sqsubset u \longrightarrow P \ y)$
and *npmiU*: $\forall x. \neg MP \wedge \neg PP \wedge P \ x \longrightarrow (\exists \ u \in U. \exists \ u' \in U. u \sqsubseteq x \wedge x \sqsubseteq u')$
and *nmi*: $\neg MP$ **and** *npi*: $\neg PP$ **and** *ex*: $\exists x. \ P \ x$
shows $\exists \ u \in U. \exists \ u' \in U. \ P \ (between \ u \ u')$
proof–
from *ex* **obtain** *x* **where** *px*: $P \ x$ **by** *blast*
from *px* *nmi* *npi* *npmiU* **have** $\exists \ u \in U. \exists \ u' \in U. u \sqsubseteq x \wedge x \sqsubseteq u'$ **by** *auto*
then obtain *u* **and** *u'* **where** *uU*: $u \in U$ **and** *uU'*: $u' \in U$ **and** *ux*: $u \sqsubseteq x$ **and**
xu': $x \sqsubseteq u'$ **by** *auto*
from *uU* **have** *Une*: $U \neq \{\}$ **by** *auto*
term *linorder.Min less-eq*
let *?l* = *linorder.Min less-eq U*
let *?u* = *linorder.Max less-eq U*
have *linM*: $?l \in U$ **using** *fU Une* **by** *simp*
have *uinM*: $?u \in U$ **using** *fU Une* **by** *simp*
have *lM*: $\forall \ t \in U. \ ?l \sqsubseteq t$ **using** *Une fU* **by** *auto*
have *Mu*: $\forall \ t \in U. \ t \sqsubseteq ?u$ **using** *Une fU* **by** *auto*
have *th*: $?l \sqsubseteq u$ **using** *uU Une lM* **by** *auto*
from *order-trans[OF th ux]* **have** *lx*: $?l \sqsubseteq x$.
have *th*: $u' \sqsubseteq ?u$ **using** *uU' Une Mu* **by** *simp*
from *order-trans[OF xu' th]* **have** *xu*: $x \sqsubseteq ?u$.
from *finite-set-intervals2[where P=P,OF px lx xu linM uinM fU lM Mu]*
have $(\exists \ s \in U. \ P \ s) \vee$
 $(\exists \ t1 \in U. \exists \ t2 \in U. (\forall \ y. \ t1 \sqsubset y \wedge y \sqsubset t2 \longrightarrow y \notin U) \wedge t1 \sqsubset x \wedge x \sqsubset t2 \wedge P \ x)$.
moreover { **fix** *u* **assume** *um*: $u \in U$ **and** *pu*: $P \ u$
have *between u u = u* **by** (*simp add: between-same*)
with *um pu* **have** $P \ (between \ u \ u)$ **by** *simp*
with *um* **have** *?thesis* **by** *blast* }
moreover{
assume $\exists \ t1 \in U. \exists \ t2 \in U. (\forall \ y. \ t1 \sqsubset y \wedge y \sqsubset t2 \longrightarrow y \notin U) \wedge t1 \sqsubset x \wedge$
 $x \sqsubset t2 \wedge P \ x$
then obtain *t1* **and** *t2* **where** *t1M*: $t1 \in U$ **and** *t2M*: $t2 \in U$
and *noM*: $\forall \ y. \ t1 \sqsubset y \wedge y \sqsubset t2 \longrightarrow y \notin U$ **and** *t1x*: $t1 \sqsubset x$ **and** *xt2*: $x \sqsubset t2$ **and** *px*: $P \ x$
by *blast*
from *less-trans[OF t1x xt2]* **have** *t1t2*: $t1 \sqsubset t2$.
let *?u* = *between t1 t2*
from *between-less t1t2* **have** *t1lu*: $t1 \sqsubset ?u$ **and** *ut2*: $?u \sqsubset t2$ **by** *auto*
from *lin-dense noM t1x xt2 px t1lu ut2* **have** $P \ ?u$ **by** *blast*
with *t1M t2M* **have** *?thesis* **by** *blast* }
ultimately show *?thesis* **by** *blast*
qed

theorem *fr-eq*:

assumes *fU*: *finite U*
and *lin-dense*: $\forall x \ l \ u. (\forall \ t. \ l \sqsubset t \wedge t \sqsubset u \longrightarrow t \notin U) \wedge l \sqsubset x \wedge x \sqsubset u \wedge P \ x$
 $\longrightarrow (\forall \ y. \ l \sqsubset y \wedge y \sqsubset u \longrightarrow P \ y)$
and *nmibnd*: $\forall x. \neg MP \wedge P \ x \longrightarrow (\exists \ u \in U. \ u \sqsubseteq x)$
and *npibnd*: $\forall x. \neg PP \wedge P \ x \longrightarrow (\exists \ u \in U. \ x \sqsubseteq u)$
and *mi*: $\exists z. \forall x. \ x \sqsubset z \longrightarrow (P \ x = MP)$ **and** *pi*: $\exists z. \forall x. \ z \sqsubset x \longrightarrow (P \ x = PP)$
shows $(\exists \ x. \ P \ x) \equiv (MP \vee PP \vee (\exists \ u \in U. \exists \ u' \in U. \ P \ (\text{between } u \ u')))$
(is $- \equiv (- \vee - \vee ?F)$ **is** $?E \equiv ?D)$
proof –
{
assume *px*: $\exists \ x. \ P \ x$
have $MP \vee PP \vee (\neg MP \wedge \neg PP)$ **by** *blast*
moreover {**assume** $MP \vee PP$ **hence** $?D$ **by** *blast*}
moreover {**assume** *nmi*: $\neg MP$ **and** *npi*: $\neg PP$
from *npmibnd*[*OF nmibnd npibnd*]
have *npmiU*: $\forall x. \neg MP \wedge \neg PP \wedge P \ x \longrightarrow (\exists \ u \in U. \exists \ u' \in U. \ u \sqsubseteq x \wedge x \sqsubseteq u')$
from *rinf-U*[*OF fU lin-dense npmiU nmi npi px*] **have** $?D$ **by** *blast*}
ultimately have $?D$ **by** *blast*}
moreover
{ **assume** $?D$
moreover {**assume** *m*: MP **from** *minf-ex*[*OF mi m*] **have** $?E$.}
moreover {**assume** *p*: PP **from** *pinf-ex*[*OF pi p*] **have** $?E$. }
moreover {**assume** *f*: $?F$ **hence** $?E$ **by** *blast*}
ultimately have $?E$ **by** *blast*}
ultimately have $?E = ?D$ **by** *blast* **thus** $?E \equiv ?D$ **by** *simp*
qed

lemmas *minf-thms* = *minf-conj minf-disj minf-eq minf-neq minf-lt minf-le minf-gt minf-ge minf-P*
lemmas *pinf-thms* = *pinf-conj pinf-disj pinf-eq pinf-neq pinf-lt pinf-le pinf-gt pinf-ge pinf-P*

lemmas *nmi-thms* = *nmi-conj nmi-disj nmi-eq nmi-neq nmi-lt nmi-le nmi-gt nmi-ge nmi-P*
lemmas *npi-thms* = *npi-conj npi-disj npi-eq npi-neq npi-lt npi-le npi-gt npi-ge npi-P*
lemmas *lin-dense-thms* = *lin-dense-conj lin-dense-disj lin-dense-eq lin-dense-neq lin-dense-lt lin-dense-le lin-dense-gt lin-dense-ge lin-dense-P*

lemma *ferrack-axiom*: *constr-dense-linear-order less-eq less between*
by (*rule constr-dense-linear-order-axioms*)

lemma *atoms*:
includes *meta-term-syntax*
shows *TERM* (*less* :: '*a* \Rightarrow -')
and *TERM* (*less-eq* :: '*a* \Rightarrow -')

```

and TERM (op = :: 'a ⇒ -) .

declare ferrack-axiom [ferrack minf: minf-thms pinf: pinf-thms
  nmi: nmi-thms npi: npi-thms lindense:
  lin-dense-thms qe: fr-eq atoms: atoms]

declaration ⟨⟨
  let
  fun simps phi = map (Morphism.thm phi) [@{thm not-less}, @{thm not-le}]
  fun generic-whatis phi =
  let
    val [lt, le] = map (Morphism.term phi) [@{term op □}, @{term op ⊑}]
    fun h x t =
      case term-of t of
        Const(op =, -)$y$z => if term-of x aconv y then Ferrante-Rackoff-Data.Eq
          else Ferrante-Rackoff-Data.Nox
      | @{term Not}$(Const(op =, -)$y$z) => if term-of x aconv y then Ferrante-Rackoff-Data.NEq
          else Ferrante-Rackoff-Data.Nox
      | b$y$z => if Term.could-unify (b, lt) then
          if term-of x aconv y then Ferrante-Rackoff-Data.Lt
          else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
          else Ferrante-Rackoff-Data.Nox
        else if Term.could-unify (b, le) then
          if term-of x aconv y then Ferrante-Rackoff-Data.Le
          else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
          else Ferrante-Rackoff-Data.Nox
        else Ferrante-Rackoff-Data.Nox
      | - => Ferrante-Rackoff-Data.Nox
    in h end
  fun ss phi = HOL-ss addsimps (simps phi)
  in
    Ferrante-Rackoff-Data.funs @{thm ferrack-axiom}
    {isolate-conv = K (K (K Thm.reflexive)), whatis = generic-whatis, simpset =
    ss}
  end
  ⟩⟩

end

use ~~/src/HOL/Tools/Qelim/ferrante-rackoff.ML

method-setup ferrack = ⟨⟨
  Method.txt-args (Method.SIMPLE-METHOD' o FerranteRackoff.dlo-tac)
  ⟩⟩ Ferrante and Rackoff's algorithm for quantifier elimination in dense linear orders

```

42.1 Ferrante and Rackoff algorithm over ordered fields

lemma *neg-prod-lt*:(*c*::'*a*::ordered-field) < 0 ⇒ ((*c***x* < 0) == (*x* > 0))

proof–

assume $H: c < 0$
have $c*x < 0 = (0/c < x)$ **by** (*simp only: neg-divide-less-eq*[$OF\ H$] *ring-simps*)
also have $\dots = (0 < x)$ **by** *simp*
finally show $(c*x < 0) == (x > 0)$ **by** *simp*
qed

lemma *pos-prod-lt*: $(c::'a::ordered-field) > 0 \implies ((c*x < 0) == (x < 0))$
proof–
assume $H: c > 0$
hence $c*x < 0 = (0/c > x)$ **by** (*simp only: pos-less-divide-eq*[$OF\ H$] *ring-simps*)
also have $\dots = (0 > x)$ **by** *simp*
finally show $(c*x < 0) == (x < 0)$ **by** *simp*
qed

lemma *neg-prod-sum-lt*: $(c::'a::ordered-field) < 0 \implies ((c*x + t < 0) == (x > (-1/c)*t))$
proof–
assume $H: c < 0$
have $c*x + t < 0 = (c*x < -t)$ **by** (*subst less-iff-diff-less-0* [*of* $c*x - t$], *simp*)
also have $\dots = (-t/c < x)$ **by** (*simp only: neg-divide-less-eq*[$OF\ H$] *ring-simps*)
also have $\dots = ((-1/c)*t < x)$ **by** *simp*
finally show $(c*x + t < 0) == (x > (-1/c)*t)$ **by** *simp*
qed

lemma *pos-prod-sum-lt*: $(c::'a::ordered-field) > 0 \implies ((c*x + t < 0) == (x < (-1/c)*t))$
proof–
assume $H: c > 0$
have $c*x + t < 0 = (c*x < -t)$ **by** (*subst less-iff-diff-less-0* [*of* $c*x - t$], *simp*)
also have $\dots = (-t/c > x)$ **by** (*simp only: pos-less-divide-eq*[$OF\ H$] *ring-simps*)
also have $\dots = ((-1/c)*t > x)$ **by** *simp*
finally show $(c*x + t < 0) == (x < (-1/c)*t)$ **by** *simp*
qed

lemma *sum-lt*: $((x::'a::pordered-ab-group-add) + t < 0) == (x < -t)$
using *less-diff-eq*[**where** $a = x$ **and** $b = t$ **and** $c = 0$] **by** *simp*

lemma *neg-prod-le*: $(c::'a::ordered-field) < 0 \implies ((c*x \leq 0) == (x \geq 0))$
proof–
assume $H: c < 0$
have $c*x \leq 0 = (0/c \leq x)$ **by** (*simp only: neg-divide-le-eq*[$OF\ H$] *ring-simps*)
also have $\dots = (0 \leq x)$ **by** *simp*
finally show $(c*x \leq 0) == (x \geq 0)$ **by** *simp*
qed

lemma *pos-prod-le*: $(c::'a::ordered-field) > 0 \implies ((c*x \leq 0) == (x \leq 0))$
proof–
assume $H: c > 0$
hence $c*x \leq 0 = (0/c \geq x)$ **by** (*simp only: pos-le-divide-eq*[$OF\ H$] *ring-simps*)

also have ... = (0 >= x) by simp
 finally show (c*x <= 0) == (x <= 0) by simp
 qed

lemma neg-prod-sum-le: (c::'a::ordered-field) < 0 ==> ((c*x + t <= 0) == (x >= (- 1/c)*t))

proof -

assume H: c < 0
 have c*x + t <= 0 = (c*x <= -t) by (subst le-iff-diff-le-0 [of c*x -t], simp)
 also have ... = (-t/c <= x) by (simp only: neg-divide-le-eq[OF H] ring-simps)
 also have ... = ((- 1/c)*t <= x) by simp
 finally show (c*x + t <= 0) == (x >= (- 1/c)*t) by simp
 qed

lemma pos-prod-sum-le: (c::'a::ordered-field) > 0 ==> ((c*x + t <= 0) == (x <= (- 1/c)*t))

proof -

assume H: c > 0
 have c*x + t <= 0 = (c*x <= -t) by (subst le-iff-diff-le-0 [of c*x -t], simp)
 also have ... = (-t/c >= x) by (simp only: pos-le-divide-eq[OF H] ring-simps)
 also have ... = ((- 1/c)*t >= x) by simp
 finally show (c*x + t <= 0) == (x <= (- 1/c)*t) by simp
 qed

lemma sum-le: ((x::'a::pordered-ab-group-add) + t <= 0) == (x <= - t)
 using le-diff-eq[where a=x and b=t and c=0] by simp

lemma nz-prod-eq: (c::'a::ordered-field) ≠ 0 ==> ((c*x = 0) == (x = 0)) by simp

lemma nz-prod-sum-eq: (c::'a::ordered-field) ≠ 0 ==> ((c*x + t = 0) == (x = (- 1/c)*t))

proof -

assume H: c ≠ 0
 have c*x + t = 0 = (c*x = -t) by (subst eq-iff-diff-eq-0 [of c*x -t], simp)
 also have ... = (x = -t/c) by (simp only: nonzero-eq-divide-eq[OF H] ring-simps)
 finally show (c*x + t = 0) == (x = (- 1/c)*t) by simp
 qed

lemma sum-eq: ((x::'a::pordered-ab-group-add) + t = 0) == (x = - t)
 using eq-diff-eq[where a=x and b=t and c=0] by simp

interpretation class-ordered-field-dense-linear-order: constr-dense-linear-order

[op <= op <
 λ x y. 1/2 * ((x::'a::{ordered-field,recpower,number-ring}) + y)]

proof (unfold-locales, dlo, dlo, auto)

fix x y::'a assume lt: x < y
 from less-half-sum[OF lt] show x < (x + y) / 2 by simp
 next

fix x y::'a assume lt: x < y
 from gt-half-sum[OF lt] show (x + y) / 2 < y by simp

qed

declaration⟨⟨

let

fun earlier [] x y = false

| earlier (h::t) x y =

if h aconvc y then false else if h aconvc x then true else earlier t x y;

fun dest-frac ct = case term-of ct of

Const (@{const-name HOL.divide},-) \$ a \$ b=>

Rat.rat-of-quotient (snd (HOLLogic.dest-number a), snd (HOLLogic.dest-number b))

| t=> Rat.rat-of-int (snd (HOLLogic.dest-number t))

fun mk-frac phi cT x =

let val (a, b) = Rat.quotient-of-rat x

in if b = 1 then Numeral.mk-cnumber cT a

else Thm.capply

(Thm.capply (Drule.ctrm-rule (instantiate' [SOME cT] []) @ {cpat op /})
(Numeral.mk-cnumber cT a))

(Numeral.mk-cnumber cT b)

end

fun whatis x ct = case term-of ct of

Const (@{const-name HOL.plus},-) \$(Const (@{const-name HOL.times},-) \$- \$y) \$- =>

if y aconv term-of x then (c*x+t, [(funpow 2 Thm.dest-arg1) ct, Thm.dest-arg ct])

else (Nox, [])

| Const (@{const-name HOL.plus},-) \$y \$- =>

if y aconv term-of x then (x+t, [Thm.dest-arg ct])

else (Nox, [])

| Const (@{const-name HOL.times},-) \$- \$y =>

if y aconv term-of x then (c*x, [Thm.dest-arg1 ct])

else (Nox, [])

| t=> if t aconv term-of x then (x, []) else (Nox, []);

fun xnormalize-conv ctxt [] ct = reflexive ct

| xnormalize-conv ctxt (vs as (x::-)) ct =

case term-of ct of

Const (@{const-name HOL.less},-) \$- \$Const (@{const-name HOL.zero},-) =>

(case whatis x (Thm.dest-arg1 ct) of

(c*x+t, [c, t]) =>

let

val cr = dest-frac c

val clt = Thm.dest-fun2 ct

val cz = Thm.dest-arg ct

val neg = cr </ Rat.zero

val cthp = Simplifier.rewrite (local-simpset-of ctxt)

```

      (Thm.capply @{cterm Trueprop}
        (if neg then Thm.capply (Thm.capply clt c) cz
          else Thm.capply (Thm.capply clt cz) c))
    val cth = equal-elim (symmetric cthp) TrueI
    val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x,t]))
      (if neg then @{thm neg-prod-sum-lt} else @{thm pos-prod-sum-lt})) cth
    val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
      (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
  in rth end
| (x+t,[t]) =>
  let
    val T = ctyp-of-term x
    val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-lt}
    val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
      (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
  in rth end
| (c*x,[c]) =>
  let
    val cr = dest-frac c
    val clt = Thm.dest-fun2 ct
    val cz = Thm.dest-arg ct
    val neg = cr </ Rat.zero
    val cthp = Simplifier.rewrite (local-simpset-of ctxt)
      (Thm.capply @{cterm Trueprop}
        (if neg then Thm.capply (Thm.capply clt c) cz
          else Thm.capply (Thm.capply clt cz) c))
    val cth = equal-elim (symmetric cthp) TrueI
    val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x]))
      (if neg then @{thm neg-prod-lt} else @{thm pos-prod-lt})) cth
    val rth = th
  in rth end
| - => reflexive ct)

| Const(@{const-name HOL.less-eq},-)$-Const(@{const-name HOL.zero},-) =>
(case whatis x (Thm.dest-arg1 ct) of
(c*x+t,[c,t]) =>
  let
    val T = ctyp-of-term x
    val cr = dest-frac c
    val clt = Drule.cterm-rule (instantiate' [SOME T] []) @{cpat op <}
    val cz = Thm.dest-arg ct
    val neg = cr </ Rat.zero
    val cthp = Simplifier.rewrite (local-simpset-of ctxt)
      (Thm.capply @{cterm Trueprop}
        (if neg then Thm.capply (Thm.capply clt c) cz
          else Thm.capply (Thm.capply clt cz) c))

```

```

    val cth = equal-elim (symmetric cthp) TrueI
    val th = implies-elim (instantiate' [SOME T] (map SOME [c,x,t])
      (if neg then @{thm neg-prod-sum-le} else @{thm pos-prod-sum-le})) cth
    val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
      (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
  in rth end
| (x+t,[t]) =>
  let
    val T = ctyp-of-term x
    val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-le}
    val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
      (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
  in rth end
| (c*x,[c]) =>
  let
    val T = ctyp-of-term x
    val cr = dest-frac c
    val clt = Drule.ctrm-rule (instantiate' [SOME T] []) @{cpat op <}
    val cz = Thm.dest-arg ct
    val neg = cr </ Rat.zero
    val cthp = Simplifier.rewrite (local-simpset-of ctxt)
      (Thm.capply @{ctrm Trueprop}
        (if neg then Thm.capply (Thm.capply clt c) cz
          else Thm.capply (Thm.capply clt cz) c))
    val cth = equal-elim (symmetric cthp) TrueI
    val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
      [c,x])
      (if neg then @{thm neg-prod-le} else @{thm pos-prod-le})) cth
    val rth = th
  in rth end
| - => reflexive ct)

| Const(op =,-)$-Const(@{const-name HOL.zero},-) =>
  (case whatis x (Thm.dest-arg1 ct) of
    (c*x+t,[c,t]) =>
      let
        val T = ctyp-of-term x
        val cr = dest-frac c
        val ceq = Thm.dest-fun2 ct
        val cz = Thm.dest-arg ct
        val cthp = Simplifier.rewrite (local-simpset-of ctxt)
          (Thm.capply @{ctrm Trueprop}
            (Thm.capply @{ctrm Not} (Thm.capply (Thm.capply ceq c) cz)))
        val cth = equal-elim (symmetric cthp) TrueI
        val th = implies-elim
          (instantiate' [SOME T] (map SOME [c,x,t]) @{thm nz-prod-sum-eq})
      cth
    val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
      (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
  )

```



```

    in rth end
  | (x+t,[t]) =>
    let
      val T = ctyp-of-term x
      val th = instantiate' [SOME T] [SOME x, SOME t] @ {thm sum-eq}
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
        (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
    in rth end
  | (c*x,[c]) =>
    let
      val T = ctyp-of-term x
      val cr = dest-frac c
      val ceq = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
        (Thm.capply @ {cterm Trueprop}
          (Thm.capply @ {cterm Not} (Thm.capply (Thm.capply ceq c) cz)))
      val cth = equal-elim (symmetric cthp) TrueI
      val rth = implies-elim
        (instantiate' [SOME T] (map SOME [c,x]) @ {thm nz-prod-eq}) cth
    in rth end
  | - => reflexive ct);

local
  val less-iff-diff-less-0 = mk-meta-eq @ {thm less-iff-diff-less-0}
  val le-iff-diff-le-0 = mk-meta-eq @ {thm le-iff-diff-le-0}
  val eq-iff-diff-eq-0 = mk-meta-eq @ {thm eq-iff-diff-eq-0}
in
  fun field-isolate-conv phi ctxt vs ct = case term-of ct of
    Const(@ {const-name HOL.less},-) $a $b =>
      let val (ca,cb) = Thm.dest-binop ct
          val T = ctyp-of-term ca
          val th = instantiate' [SOME T] [SOME ca, SOME cb] less-iff-diff-less-0
          val nth = Conv.fconv-rule
            (Conv.arg-conv (Conv.arg1-conv
              (Normalizer.semiring-normalize-ord-conv @ {context} (earlier vs)))) th
          val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
        in rth end
    | Const(@ {const-name HOL.less-eq},-) $a $b =>
      let val (ca,cb) = Thm.dest-binop ct
          val T = ctyp-of-term ca
          val th = instantiate' [SOME T] [SOME ca, SOME cb] le-iff-diff-le-0
          val nth = Conv.fconv-rule
            (Conv.arg-conv (Conv.arg1-conv
              (Normalizer.semiring-normalize-ord-conv @ {context} (earlier vs)))) th
          val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
        in rth end
  | Const(op =, -) $a $b =>

```

```

let val (ca,cb) = Thm.dest_binop ct
    val T = ctyp-of-term ca
    val th = instantiate' [SOME T] [SOME ca, SOME cb] eq-iff-diff-eq-0
    val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @ {context} (earlier vs)))) th
    val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
in rth end
| @ {term Not} $(Const (op =, -) $a $b) => Conv.arg-conv (field-isolate-conv phi ctxt
vs) ct
| - => reflexive ct
end;

fun classfield-what is phi =
let
fun h x t =
case term-of t of
Const (op =, -) $y $z => if term-of x aconv y then Ferrante-Rackoff-Data.Eq
else Ferrante-Rackoff-Data.Nox
| @ {term Not} $(Const (op =, -) $y $z) => if term-of x aconv y then Ferrante-Rackoff-Data.NEq
else Ferrante-Rackoff-Data.Nox
| Const (@ {const-name HOL.less}, -) $y $z =>
if term-of x aconv y then Ferrante-Rackoff-Data.Lt
else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
else Ferrante-Rackoff-Data.Nox
| Const (@ {const-name HOL.less-eq}, -) $y $z =>
if term-of x aconv y then Ferrante-Rackoff-Data.Le
else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
else Ferrante-Rackoff-Data.Nox
| - => Ferrante-Rackoff-Data.Nox
in h end;
fun class-field-ss phi =
HOL-basic-ss addsimps ([@ {thm linorder-not-less}, @ {thm linorder-not-le}])
addsplits [@ {thm abs-split}, @ {thm split-max}, @ {thm split-min}]

in
Ferrante-Rackoff-Data.funs @ {thm class-ordered-field-dense-linear-order.ferrack-axiom}
{isolate-conv = field-isolate-conv, whatis = classfield-what is, simpset = class-field-ss}
end
>>

end

```

43 Examples for Ferrante and Rackoff's quantifier elimination procedure

theory *Dense-Linear-Order-Ex*
imports *Dense-Linear-Order Main*
begin

lemma

$\exists (y::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) < 2. x + 3 * y < 0 \wedge x - y > 0$
by *ferrack*

lemma $\sim (ALL x (y::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}). x < y \longrightarrow 10 * x < 11 * y)$
by *ferrack*

lemma $ALL (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) y. x < y \longrightarrow (10 * (x + 5 * y + -1) < 60 * y)$
by *ferrack*

lemma $EX (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) y. x \sim = y \longrightarrow x < y$
by *ferrack*

lemma $EX (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) y. (x \sim = y \ \& \ 10 * x \sim = 9 * y \ \& \ 10 * x < y) \longrightarrow x < y$
by *ferrack*

lemma $ALL (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) y. (x \sim = y \ \& \ 5 * x <= y) \longrightarrow 500 * x <= 100 * y$
by *ferrack*

lemma $ALL (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}). (EX (y::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}). 4 * x + 3 * y <= 0 \ \& \ 4 * x + 3 * y >= -1)$
by *ferrack*

lemma $ALL (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) < 0. (EX (y::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) > 0. 7 * x + y > 0 \ \& \ x - y <= 9)$
by *ferrack*

lemma $EX (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}). (0 < x \ \& \ x < 1) \longrightarrow (ALL y > 1. x + y \sim = 1)$
by *ferrack*

lemma $EX x. (ALL (y::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}). y < 2 \longrightarrow 2 * (y - x) \leq 0)$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). $x < 10 \mid x > 20 \mid (\text{EX } y. y \geq 0 \ \& \ y \leq 10 \ \& \ x+y = 20)$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z. x + y < z \longrightarrow y \geq z \longrightarrow x < 0$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z. x + 7*y < 5*z \ \& \ 5*y \geq 7*z \ \& \ x < 0$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z. \text{abs } (x + y) \leq z \longrightarrow (\text{abs } z = z)$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z. x + 7*y - 5*z < 0 \ \& \ 5*y + 7*z + 3*x < 0$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z. (\text{abs } (5*x+3*y+z) \leq 5*x+3*y+z \ \& \ \text{abs } (5*x+3*y+z) \geq -(5*x+3*y+z)) \mid (\text{abs } (5*x+3*y+z) \geq 5*x+3*y+z \ \& \ \text{abs } (5*x+3*y+z) \leq -(5*x+3*y+z))$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y. x < y \longrightarrow (\text{EX } z > 0. x+z = y)$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y. x < y \longrightarrow (\text{EX } z > 0. x+z = y)$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y. (\text{EX } z > 0. \text{abs } (x - y) \leq z)$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y. (\text{ALL } z < 0. (z < x \longrightarrow z \leq y) \ \& \ (z > y \longrightarrow z \geq x))$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y. (\text{ALL } z \geq 0. \text{abs } (3*x+7*y) \leq 2*z + 1)$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y. (\text{ALL } z < 0. (z < x \longrightarrow z \leq y) \ \& \ (z > y \longrightarrow z \geq x))$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) > 0 . (*ALL* y . (*EX* z . $13 * \text{abs } z \neq \text{abs } (12 * y - x) \ \& \ 5 * x - 3 * (\text{abs } y) \leq 7 * z$))
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). $\text{abs } (4 * x + 17) < 4 \ \& \ (\text{ALL } y . \text{abs } (x * 34 - 34 * y - 9) \neq 0 \longrightarrow (\text{EX } z . 5 * x - 3 * \text{abs } y \leq 7 * z))$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*EX* $y > \text{abs } (23 * x - 9)$. (*ALL* $z > \text{abs } (3 * y - 19 * \text{abs } x)$. $x + z > 2 * y$))
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*EX* $y < \text{abs } (3 * x - 1)$. (*ALL* $z \geq (3 * \text{abs } x - 1)$. $\text{abs } (12 * x - 13 * y + 19 * z) > \text{abs } (23 * x)$))
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). $\text{abs } x < 100 \ \& \ (\text{ALL } y > x . (\text{EX } z < 2 * y - x . 5 * x - 3 * y \leq 7 * z))$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z \ w$. $7 * x < 3 * y \longrightarrow 5 * y < 7 * z \longrightarrow z < 2 * w \longrightarrow 7 * (2 * w - x) > 2 * y$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z \ w$. $5 * x + 3 * z - 17 * w + \text{abs } (y - 8 * x + z) \leq 89$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z \ w$. $5 * x + 3 * z - 17 * w + 7 * (y - 8 * x + z) \leq \max y (7 * z - x + w)$
by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z \ w$. $\min (5 * x + 3 * z) (17 * w) + 5 * \text{abs } (y - 8 * x + z) \leq \max y (7 * z - x + w)$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $y \ z$. (*EX* $w \geq (x + y + z)$. $w \leq \text{abs } x + \text{abs } y + \text{abs } z$)
by *ferrack*

lemma $\sim (\text{ALL } (x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}) . (\text{EX } y \ z \ w . 3 * x + z * 4 = 3 * y \ \& \ x + y < z \ \& \ x > w \ \& \ 3 * x < w + y))$
by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) y . (*EX* $z \ w$. $\text{abs } (x - y) = (z - w) \ \& \ z * 1234 < 233 * x \ \& \ w \sim = y$)

by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*EX* y z w . $\min (5*x + 3*z) (17*w) + 5* \text{abs} (y - 8*x + z) \leq \max y (7*z - x + w)$))

by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) y z . (*ALL* $w \geq \text{abs} (x+y+z)$. $w \geq \text{abs } x + \text{abs } y + \text{abs } z$)

by *ferrack*

lemma *EX* z . (*ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) y . (*EX* $w \geq (x+y+z)$. $w \leq \text{abs } x + \text{abs } y + \text{abs } z$))

by *ferrack*

lemma *EX* z . (*ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) $< \text{abs } z$. (*EX* y w . $x < y$ & $x < z$ & $x > w$ & $3*x < w + y$))

by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) y . (*EX* z . (*ALL* w . $\text{abs} (x-y) = \text{abs} (z-w) \longrightarrow z < x$ & $w \sim = y$))

by *ferrack*

lemma *EX* y . (*ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*EX* z w . $\min (5*x + 3*z) (17*w) + 5* \text{abs} (y - 8*x + z) \leq \max y (7*z - x + w)$))

by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) z . (*ALL* $w \geq 13*x - 4*z$. (*EX* y . $w \geq \text{abs } x + \text{abs } y + z$))

by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*ALL* $y < x$. (*EX* $z > (x+y)$.

(*ALL* w . $5*w + 10*x - z \geq y \longrightarrow w + 7*x + 3*z \geq 2*y$))

by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*ALL* y . (*EX* $z > y$.

(*ALL* w . $w < 13 \longrightarrow w + 10*x - z \geq y \longrightarrow 5*w + 7*x + 13*z \geq 2*y$))

by *ferrack*

lemma *EX* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$) y z w . $\min (5*x + 3*z) (17*w) + 5* \text{abs} (y - 8*x + z) \leq \max y (7*z - x + w)$

by *ferrack*

lemma *ALL* ($x::'a::\{\text{ordered-field}, \text{recpower}, \text{number-ring}, \text{division-by-zero}\}$). (*EX* y . (*ALL* $z > 19$. $y \leq x + z$ & (*EX* w . $\text{abs} (y - x) < w$))

```

by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX
y. (ALL z>19. y <= x + z & (EX w. abs (x + z) < w - y)))
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX
y. abs y ~ = abs x & (ALL z> max x y. (EX w. w ~ = y & w ~ = z & 3*w - z
>= x + y)))
by ferrack

end

```

44 Some examples for Presburger Arithmetic

```

theory PresburgerEx
imports Presburger
begin

```

```

lemma  $\bigwedge m\ n\ ja\ ia. \llbracket \neg m \leq j; \neg (n::nat) \leq i; (e::nat) \neq 0; Suc\ j \leq ja \rrbracket \implies \exists m. \forall ja\ ia. m \leq ja \longrightarrow (if\ j = ja \wedge i = ia\ then\ e\ else\ 0) = 0$  by presburger

```

```

lemma  $(0::nat) < emBits\ mod\ 8 \implies 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$ 

```

```
by presburger
```

```

lemma  $(0::nat) < emBits\ mod\ 8 \implies 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$ 

```

```
by presburger
```

```

theorem  $(\forall (y::int). 3\ dvd\ y) \implies \forall (x::int). b < x \longrightarrow a \leq x$ 
by presburger

```

```

theorem !!  $(y::int)\ (z::int)\ (n::int). 3\ dvd\ z \implies 2\ dvd\ (y::int) \implies (\exists (x::int). 2*x = y) \ \&\ (\exists (k::int). 3*k = z)$ 
by presburger

```

```

theorem !!  $(y::int)\ (z::int)\ n. Suc(n::nat) < 6 \implies 3\ dvd\ z \implies 2\ dvd\ (y::int) \implies (\exists (x::int). 2*x = y) \ \&\ (\exists (k::int). 3*k = z)$ 
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  $\forall (x::nat). \exists (y::nat). y = 5 + x \mid x\ div\ 6 + 1 = 2$ 
by presburger

```

```

theorem  $\exists (x::int). 0 < x$ 
by presburger

```

theorem $\forall (x::int) y. x < y \dashv\dashv 2 * x + 1 < 2 * y$
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 $\forall (x::int). b < x \dashv\dashv a \leq x$
apply (*presburger elim*)
oops

theorem $\sim (\exists (x::int). False)$
by *presburger*

theorem $\forall (x::int). (a::int) < 3 * x \dashv\dashv b < 3 * x$
apply (*presburger elim*)
oops

theorem $\forall (x::int). (2 \text{ dvd } x) \dashv\dashv (\exists (y::int). x = 2 * y)$
by *presburger*

theorem $\forall (x::int). (2 \text{ dvd } x) \dashv\dashv (\exists (y::int). x = 2 * y)$
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 $\sim (\forall (x::int). ((2 \text{ dvd } x) = (\forall (y::int). x \neq 2 * y + 1) \mid (\exists (q::int) (u::int) i. 3 * i + 2 * q - u < 17) \dashv\dashv 0 < x \mid ((\sim 3 \text{ dvd } x) \ \& \ (x + 8 = 0))))$
by *presburger*

theorem $\sim (\forall (i::int). 4 \leq i \dashv\dashv (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i))$
by *presburger*

theorem $\forall (i::int). 8 \leq i \dashv\dashv (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i)$
by *presburger*

theorem $\exists (j::int). \forall i. j \leq i \dashv\dashv (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i)$

by *presburger*

theorem $\sim (\forall j (i::int). j \leq i \dashv\vdash (\exists x y. 0 \leq x \ \& \ 0 \leq y \ \& \ 3 * x + 5 * y = i))$

by *presburger*

Slow: about 5 seconds on a 1.6GHz machine.

theorem $(\exists m::nat. n = 2 * m) \dashv\vdash (n + 1) \text{ div } 2 = n \text{ div } 2$

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

theory *Reflected-Presburger*

imports *Main GCD Efficient-Nat*

uses (*coopereif.ML*) (*coopertac.ML*)

begin

function

iupt :: *int* \Rightarrow *int* \Rightarrow *int list*

where

iupt *i j* = (if *j* < *i* then [] else *i* # *iupt* (*i*+1) *j*)

by *pat-completeness auto*

termination by (*relation measure* ($\lambda (i, j). \text{nat } (j-i+1)$))) *auto*

lemma *iupt-set*: *set* (*iupt* *i j*) = {*i..j*}

by (*induct rule: iupt.induct*) (*simp add: simp-from-to*)

datatype *num* = *C int* | *Bound nat* | *CN nat int num* | *Neg num* | *Add num num* |
Sub num num
| *Mul int num*

consts *num-size* :: *num* \Rightarrow *nat*

primrec

num-size (*C* *c*) = 1
num-size (*Bound* *n*) = 1
num-size (*Neg* *a*) = 1 + *num-size* *a*
num-size (*Add* *a* *b*) = 1 + *num-size* *a* + *num-size* *b*
num-size (*Sub* *a* *b*) = 3 + *num-size* *a* + *num-size* *b*
num-size (*CN* *n* *c* *a*) = 4 + *num-size* *a*
num-size (*Mul* *c* *a*) = 1 + *num-size* *a*

consts *Inum* :: *int* *list* \Rightarrow *num* \Rightarrow *int*

primrec

Inum *bs* (*C* *c*) = *c*
Inum *bs* (*Bound* *n*) = *bs*!*n*
Inum *bs* (*CN* *n* *c* *a*) = *c* * (*bs*!*n*) + (*Inum* *bs* *a*)
Inum *bs* (*Neg* *a*) = -(*Inum* *bs* *a*)
Inum *bs* (*Add* *a* *b*) = *Inum* *bs* *a* + *Inum* *bs* *b*
Inum *bs* (*Sub* *a* *b*) = *Inum* *bs* *a* - *Inum* *bs* *b*
Inum *bs* (*Mul* *c* *a*) = *c** *Inum* *bs* *a*

datatype *fm* =

T | *F* | *Lt* *num* | *Le* *num* | *Gt* *num* | *Ge* *num* | *Eq* *num* | *NEq* *num* | *Dvd* *int* *num* |
NDvd *int* *num* |
NOT *fm* | *And* *fm* *fm* | *Or* *fm* *fm* | *Imp* *fm* *fm* | *Iff* *fm* *fm* | *E* *fm* | *A* *fm*
| *Closed* *nat* | *NClosed* *nat*

consts *fmsize* :: *fm* \Rightarrow *nat*

recdef *fmsize* *measure* *size*

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*) = 3 + *fmsize* *p* + *fmsize* *q*
fmsize (*Iff* *p* *q*) = 3 + 2*(*fmsize* *p* + *fmsize* *q*)
fmsize (*E* *p*) = 1 + *fmsize* *p*
fmsize (*A* *p*) = 4 + *fmsize* *p*
fmsize (*Dvd* *i* *t*) = 2
fmsize (*NDvd* *i* *t*) = 2
fmsize *p* = 1

lemma *fmsize-pos*: *fmsize* *p* > 0

by (*induct* *p* *rule*: *fmsize.induct*) *simp-all*

consts *Ifm* :: *bool* *list* \Rightarrow *int* *list* \Rightarrow *fm* \Rightarrow *bool*

primrec

$\text{Ifm } bbs \ bs \ T = \text{True}$
 $\text{Ifm } bbs \ bs \ F = \text{False}$
 $\text{Ifm } bbs \ bs \ (Lt \ a) = (\text{Inum } bs \ a < 0)$
 $\text{Ifm } bbs \ bs \ (Gt \ a) = (\text{Inum } bs \ a > 0)$
 $\text{Ifm } bbs \ bs \ (Le \ a) = (\text{Inum } bs \ a \leq 0)$
 $\text{Ifm } bbs \ bs \ (Ge \ a) = (\text{Inum } bs \ a \geq 0)$
 $\text{Ifm } bbs \ bs \ (Eq \ a) = (\text{Inum } bs \ a = 0)$
 $\text{Ifm } bbs \ bs \ (NEq \ a) = (\text{Inum } bs \ a \neq 0)$
 $\text{Ifm } bbs \ bs \ (Dvd \ i \ b) = (i \ \text{dvd} \ \text{Inum } bs \ b)$
 $\text{Ifm } bbs \ bs \ (NDvd \ i \ b) = (\neg(i \ \text{dvd} \ \text{Inum } bs \ b))$
 $\text{Ifm } bbs \ bs \ (NOT \ p) = (\neg (\text{Ifm } bbs \ bs \ p))$
 $\text{Ifm } bbs \ bs \ (And \ p \ q) = (\text{Ifm } bbs \ bs \ p \wedge \text{Ifm } bbs \ bs \ q)$
 $\text{Ifm } bbs \ bs \ (Or \ p \ q) = (\text{Ifm } bbs \ bs \ p \vee \text{Ifm } bbs \ bs \ q)$
 $\text{Ifm } bbs \ bs \ (Imp \ p \ q) = ((\text{Ifm } bbs \ bs \ p) \longrightarrow (\text{Ifm } bbs \ bs \ q))$
 $\text{Ifm } bbs \ bs \ (Iff \ p \ q) = (\text{Ifm } bbs \ bs \ p = \text{Ifm } bbs \ bs \ q)$
 $\text{Ifm } bbs \ bs \ (E \ p) = (\exists \ x. \ \text{Ifm } bbs \ (x\#bs) \ p)$
 $\text{Ifm } bbs \ bs \ (A \ p) = (\forall \ x. \ \text{Ifm } bbs \ (x\#bs) \ p)$
 $\text{Ifm } bbs \ bs \ (Closed \ n) = bbs!n$
 $\text{Ifm } bbs \ bs \ (NClosed \ n) = (\neg bbs!n)$

consts $\text{prep} :: \text{fm} \Rightarrow \text{fm}$

recdef prep measure fmsize

$\text{prep} \ (E \ T) = T$
 $\text{prep} \ (E \ F) = F$
 $\text{prep} \ (E \ (Or \ p \ q)) = Or \ (\text{prep} \ (E \ p)) \ (\text{prep} \ (E \ q))$
 $\text{prep} \ (E \ (Imp \ p \ q)) = Or \ (\text{prep} \ (E \ (NOT \ p))) \ (\text{prep} \ (E \ q))$
 $\text{prep} \ (E \ (Iff \ p \ q)) = Or \ (\text{prep} \ (E \ (And \ p \ q))) \ (\text{prep} \ (E \ (And \ (NOT \ p) \ (NOT \ q))))$
 $\text{prep} \ (E \ (NOT \ (And \ p \ q))) = Or \ (\text{prep} \ (E \ (NOT \ p))) \ (\text{prep} \ (E \ (NOT \ q)))$
 $\text{prep} \ (E \ (NOT \ (Imp \ p \ q))) = \text{prep} \ (E \ (And \ p \ (NOT \ q)))$
 $\text{prep} \ (E \ (NOT \ (Iff \ p \ q))) = Or \ (\text{prep} \ (E \ (And \ p \ (NOT \ q)))) \ (\text{prep} \ (E \ (And \ (NOT \ p) \ (NOT \ q))))$
 $\text{prep} \ (E \ p) = E \ (\text{prep} \ p)$
 $\text{prep} \ (A \ (And \ p \ q)) = And \ (\text{prep} \ (A \ p)) \ (\text{prep} \ (A \ q))$
 $\text{prep} \ (A \ p) = \text{prep} \ (NOT \ (E \ (NOT \ p)))$
 $\text{prep} \ (NOT \ (NOT \ p)) = \text{prep} \ p$
 $\text{prep} \ (NOT \ (And \ p \ q)) = Or \ (\text{prep} \ (NOT \ p)) \ (\text{prep} \ (NOT \ q))$
 $\text{prep} \ (NOT \ (A \ p)) = \text{prep} \ (E \ (NOT \ p))$
 $\text{prep} \ (NOT \ (Or \ p \ q)) = And \ (\text{prep} \ (NOT \ p)) \ (\text{prep} \ (NOT \ q))$
 $\text{prep} \ (NOT \ (Imp \ p \ q)) = And \ (\text{prep} \ p) \ (\text{prep} \ (NOT \ q))$
 $\text{prep} \ (NOT \ (Iff \ p \ q)) = Or \ (\text{prep} \ (And \ p \ (NOT \ q))) \ (\text{prep} \ (And \ (NOT \ p) \ q))$
 $\text{prep} \ (NOT \ p) = NOT \ (\text{prep} \ p)$
 $\text{prep} \ (Or \ p \ q) = Or \ (\text{prep} \ p) \ (\text{prep} \ q)$
 $\text{prep} \ (And \ p \ q) = And \ (\text{prep} \ p) \ (\text{prep} \ q)$
 $\text{prep} \ (Imp \ p \ q) = \text{prep} \ (Or \ (NOT \ p) \ q)$
 $\text{prep} \ (Iff \ p \ q) = Or \ (\text{prep} \ (And \ p \ q)) \ (\text{prep} \ (And \ (NOT \ p) \ (NOT \ q)))$
 $\text{prep} \ p = p$

(**hints** simp add: fmsize-pos)

lemma $\text{prep} : \text{Ifm } bbs \ bs \ (\text{prep} \ p) = \text{Ifm } bbs \ bs \ p$

by (*induct p arbitrary: bs rule: prep.induct, auto*)

consts *qfree*:: *fm* \Rightarrow *bool*
recdef *qfree* *measure size*
qfree (*E p*) = *False*
qfree (*A p*) = *False*
qfree (*NOT p*) = *qfree p*
qfree (*And p q*) = (*qfree p* \wedge *qfree q*)
qfree (*Or p q*) = (*qfree p* \wedge *qfree q*)
qfree (*Imp p q*) = (*qfree p* \wedge *qfree q*)
qfree (*Iff p q*) = (*qfree p* \wedge *qfree q*)
qfree p = *True*

consts
numbound0:: *num* \Rightarrow *bool*
bound0:: *fm* \Rightarrow *bool*
subst0:: *num* \Rightarrow *fm* \Rightarrow *fm*

primrec
numbound0 (*C c*) = *True*
numbound0 (*Bound n*) = (*n* > 0)
numbound0 (*CN n i a*) = (*n* > 0 \wedge *numbound0 a*)
numbound0 (*Neg a*) = *numbound0 a*
numbound0 (*Add a b*) = (*numbound0 a* \wedge *numbound0 b*)
numbound0 (*Sub a b*) = (*numbound0 a* \wedge *numbound0 b*)
numbound0 (*Mul i a*) = *numbound0 a*

lemma *numbound0-I*:
assumes *nb*: *numbound0 a*
shows *Inum (b#bs) a* = *Inum (b'#bs) a*
using *nb*
by (*induct a rule: numbound0.induct*) (*auto simp add: gr0-conv-Suc*)

primrec
bound0 T = *True*
bound0 F = *True*
bound0 (Lt a) = *numbound0 a*
bound0 (Le a) = *numbound0 a*
bound0 (Gt a) = *numbound0 a*
bound0 (Ge a) = *numbound0 a*
bound0 (Eq a) = *numbound0 a*
bound0 (NEq a) = *numbound0 a*
bound0 (Dvd i a) = *numbound0 a*
bound0 (NDvd i a) = *numbound0 a*
bound0 (NOT p) = *bound0 p*
bound0 (And p q) = (*bound0 p* \wedge *bound0 q*)
bound0 (Or p q) = (*bound0 p* \wedge *bound0 q*)

```

bound0 (Imp p q) = ((bound0 p) ∧ (bound0 q))
bound0 (Iff p q) = (bound0 p ∧ bound0 q)
bound0 (E p) = False
bound0 (A p) = False
bound0 (Closed P) = True
bound0 (NClosed P) = True
lemma bound0-I:
  assumes bp: bound0 p
  shows Ifm bbs (b#bs) p = Ifm bbs (b'#bs) p
using bp numbound0-I[where b=b and bs=bs and b'=b']
by (induct p rule: bound0.induct) (auto simp add: gr0-conv-Suc)

```

```

fun numsubst0:: num ⇒ num ⇒ num where
  numsubst0 t (C c) = (C c)
| numsubst0 t (Bound n) = (if n=0 then t else Bound n)
| numsubst0 t (CN 0 i a) = Add (Mul i t) (numsubst0 t a)
| numsubst0 t (CN n i a) = CN n i (numsubst0 t a)
| numsubst0 t (Neg a) = Neg (numsubst0 t a)
| numsubst0 t (Add a b) = Add (numsubst0 t a) (numsubst0 t b)
| numsubst0 t (Sub a b) = Sub (numsubst0 t a) (numsubst0 t b)
| numsubst0 t (Mul i a) = Mul i (numsubst0 t a)

```

```

lemma numsubst0-I:
  Inum (b#bs) (numsubst0 a t) = Inum ((Inum (b#bs) a)#bs) t
by (induct t rule: numsubst0.induct, auto simp: nth-Cons')

```

```

lemma numsubst0-I':
  numbound0 a ⇒ Inum (b#bs) (numsubst0 a t) = Inum ((Inum (b'#bs) a)#bs)
  t
by (induct t rule: numsubst0.induct, auto simp: nth-Cons' numbound0-I[where
  b=b and b'=b'])

```

```

primrec
  subst0 t T = T
  subst0 t F = F
  subst0 t (Lt a) = Lt (numsubst0 t a)
  subst0 t (Le a) = Le (numsubst0 t a)
  subst0 t (Gt a) = Gt (numsubst0 t a)
  subst0 t (Ge a) = Ge (numsubst0 t a)
  subst0 t (Eq a) = Eq (numsubst0 t a)
  subst0 t (NEq a) = NEq (numsubst0 t a)
  subst0 t (Dvd i a) = Dvd i (numsubst0 t a)
  subst0 t (NDvd i a) = NDvd i (numsubst0 t a)
  subst0 t (NOT p) = NOT (subst0 t p)
  subst0 t (And p q) = And (subst0 t p) (subst0 t q)
  subst0 t (Or p q) = Or (subst0 t p) (subst0 t q)
  subst0 t (Imp p q) = Imp (subst0 t p) (subst0 t q)
  subst0 t (Iff p q) = Iff (subst0 t p) (subst0 t q)
  subst0 t (Closed P) = (Closed P)

```

$subst0\ t\ (NClosed\ P) = (NClosed\ P)$

lemma *subst0-I*: **assumes** *qfp*: *qfree p*
shows *Ifm bbs (b#bs) (subst0 a p) = Ifm bbs ((Inum (b#bs) a)#bs) p*
using *qfp numsubst0-I* [**where** *b=b* **and** *bs=bs* **and** *a=a*]
by (*induct p*) (*simp-all add: gr0-conv-Suc*)

consts

decrnum:: *num* \Rightarrow *num*

decr :: *fm* \Rightarrow *fm*

recdef *decrnum measure size*

decrnum (*Bound n*) = *Bound (n - 1)*

decrnum (*Neg a*) = *Neg (decrnum a)*

decrnum (*Add a b*) = *Add (decrnum a) (decrnum b)*

decrnum (*Sub a b*) = *Sub (decrnum a) (decrnum b)*

decrnum (*Mul c a*) = *Mul c (decrnum a)*

decrnum (*CN n i a*) = (*CN (n - 1) i (decrnum a)*)

decrnum a = *a*

recdef *decr measure size*

decr (*Lt a*) = *Lt (decrnum a)*

decr (*Le a*) = *Le (decrnum a)*

decr (*Gt a*) = *Gt (decrnum a)*

decr (*Ge a*) = *Ge (decrnum a)*

decr (*Eq a*) = *Eq (decrnum a)*

decr (*NEq a*) = *NEq (decrnum a)*

decr (*Dvd i a*) = *Dvd i (decrnum a)*

decr (*NDvd i a*) = *NDvd i (decrnum a)*

decr (*NOT p*) = *NOT (decr p)*

decr (*And p q*) = *And (decr p) (decr q)*

decr (*Or p q*) = *Or (decr p) (decr q)*

decr (*Imp p q*) = *Imp (decr p) (decr q)*

decr (*Iff p q*) = *Iff (decr p) (decr q)*

decr p = *p*

lemma *decrnum*: **assumes** *nb*: *numbound0 t*

shows *Inum (x#bs) t = Inum bs (decrnum t)*

using *nb* **by** (*induct t rule: decrnum.induct, auto simp add: gr0-conv-Suc*)

lemma *decr*: **assumes** *nb*: *bound0 p*

shows *Ifm bbs (x#bs) p = Ifm bbs bs (decr p)*

using *nb*

by (*induct p rule: decr.induct, simp-all add: gr0-conv-Suc decrnum*)

lemma *decr-qf*: *bound0 p* \Longrightarrow *qfree (decr p)*

by (*induct p, simp-all*)

```

consts
  isatom :: fm  $\Rightarrow$  bool
recdef isatom measure size
  isatom T = True
  isatom F = True
  isatom (Lt a) = True
  isatom (Le a) = True
  isatom (Gt a) = True
  isatom (Ge a) = True
  isatom (Eq a) = True
  isatom (NEq a) = True
  isatom (Dvd i b) = True
  isatom (NDvd i b) = True
  isatom (Closed P) = True
  isatom (NClosed P) = True
  isatom p = False

lemma numsubst0-numbound0: assumes nb: numbound0 t
  shows numbound0 (numsubst0 t a)
using nb apply (induct a rule: numbound0.induct)
apply simp-all
apply (case-tac n, simp-all)
done

lemma subst0-bound0: assumes qf: qfree p and nb: numbound0 t
  shows bound0 (subst0 t p)
using qf numsubst0-numbound0[OF nb] by (induct p rule: subst0.induct, auto)

lemma bound0-qf: bound0 p  $\implies$  qfree p
by (induct p, simp-all)

constdefs djf:: ('a  $\Rightarrow$  fm)  $\Rightarrow$  'a  $\Rightarrow$  fm  $\Rightarrow$  fm
  djf f p q  $\equiv$  (if q=T then T else if q=F then f p else
    (let fp = f p in case fp of T  $\Rightarrow$  T | F  $\Rightarrow$  q | -  $\Rightarrow$  Or (f p) q))
constdefs evaldjf:: ('a  $\Rightarrow$  fm)  $\Rightarrow$  'a list  $\Rightarrow$  fm
  evaldjf f ps  $\equiv$  foldr (djf f) ps F

lemma djf-Or: Ifm bbs bs (djf f p q) = Ifm bbs bs (Or (f p) q)
by (cases q=T, simp add: djf-def, cases q=F, simp add: djf-def)
(cases f p, simp-all add: Let-def djf-def)

lemma evaldjf-ex: Ifm bbs bs (evaldjf f ps) = ( $\exists$  p  $\in$  set ps. Ifm bbs bs (f p))
by(induct ps, simp-all add: evaldjf-def djf-Or)

lemma evaldjf-bound0:
  assumes nb:  $\forall$  x  $\in$  set xs. bound0 (f x)
  shows bound0 (evaldjf f xs)
  using nb by (induct xs, auto simp add: evaldjf-def djf-def Let-def) (case-tac f a,

```

auto)

lemma *evaldjf-*qf**:

assumes *nb*: $\forall x \in \text{set } xs. \text{qfree } (f x)$

shows $\text{qfree } (\text{evaldjf } f \text{ } xs)$

using *nb* **by** (*induct xs, auto simp add: evaldjf-def djf-def Let-def*) (*case-tac f a, auto*)

consts *disjuncts* :: $fm \Rightarrow fm \text{ list}$

recdef *disjuncts* *measure size*

disjuncts (*Or p q*) = (*disjuncts p*) @ (*disjuncts q*)

disjuncts F = []

disjuncts p = [*p*]

lemma *disjuncts*: $(\exists q \in \text{set } (\text{disjuncts } p). \text{Ifm bbs bs } q) = \text{Ifm bbs bs } p$

by(*induct p rule: disjuncts.induct, auto*)

lemma *disjuncts-nb*: $\text{bound0 } p \implies \forall q \in \text{set } (\text{disjuncts } p). \text{bound0 } q$

proof–

assume *nb*: $\text{bound0 } p$

hence *list-all bound0 (disjuncts p)* **by** (*induct p rule: disjuncts.induct, auto*)

thus *?thesis* **by** (*simp only: list-all-iff*)

qed

lemma *disjuncts-qf*: $\text{qfree } p \implies \forall q \in \text{set } (\text{disjuncts } p). \text{qfree } q$

proof–

assume *qf*: $\text{qfree } p$

hence *list-all qfree (disjuncts p)*

by (*induct p rule: disjuncts.induct, auto*)

thus *?thesis* **by** (*simp only: list-all-iff*)

qed

constdefs *DJ* :: $(fm \Rightarrow fm) \Rightarrow fm \Rightarrow fm$

DJ f p $\equiv \text{evaldjf } f (\text{disjuncts } p)$

lemma *DJ*: **assumes** *fdj*: $\forall p q. f (\text{Or } p q) = \text{Or } (f p) (f q)$

and *fF*: $f F = F$

shows $\text{Ifm bbs bs } (DJ f p) = \text{Ifm bbs bs } (f p)$

proof–

have $\text{Ifm bbs bs } (DJ f p) = (\exists q \in \text{set } (\text{disjuncts } p). \text{Ifm bbs bs } (f q))$

by (*simp add: DJ-def evaldjf-ex*)

also have $\dots = \text{Ifm bbs bs } (f p)$ **using** *fdj fF* **by** (*induct p rule: disjuncts.induct, auto*)

finally show *?thesis* .

qed

lemma *DJ-qf*: **assumes**

fqf: $\forall p. \text{qfree } p \longrightarrow \text{qfree } (f p)$

shows $\forall p. \text{qfree } p \longrightarrow \text{qfree } (DJ f p)$


```

proof(clarify)
  fix p assume qf: qfree p
  have th: DJ f p = evaldjf f (disjuncts p) by (simp add: DJ-def)
  from disjuncts-qf[OF qf] have  $\forall q \in \text{set } (\text{disjuncts } p). \text{qfree } q$  .
  with fqq have th':  $\forall q \in \text{set } (\text{disjuncts } p). \text{qfree } (f q)$  by blast

  from evaldjf-qf[OF th'] th show qfree (DJ f p) by simp
qed

lemma DJ-qe: assumes qe:  $\forall bs p. \text{qfree } p \longrightarrow \text{qfree } (qe p) \wedge (\text{Ifm } bbs \ bs \ (qe p))$ 
= Ifm bbs bs (E p))
  shows  $\forall bs p. \text{qfree } p \longrightarrow \text{qfree } (DJ \ qe \ p) \wedge (\text{Ifm } bbs \ bs \ ((DJ \ qe \ p)) = \text{Ifm } bbs \ bs \ (E \ p))$ 
proof(clarify)
  fix p::fm and bs
  assume qf: qfree p
  from qe have qth:  $\forall p. \text{qfree } p \longrightarrow \text{qfree } (qe p)$  by blast
  from DJ-qf[OF qth] qf have qfth: qfree (DJ qe p) by auto
  have Ifm bbs bs (DJ qe p) =  $(\exists q \in \text{set } (\text{disjuncts } p). \text{Ifm } bbs \ bs \ (qe \ q))$ 
  by (simp add: DJ-def evaldjf-ex)
  also have ... =  $(\exists q \in \text{set } (\text{disjuncts } p). \text{Ifm } bbs \ bs \ (E \ q))$  using qe disjuncts-qf[OF qf] by auto
  also have ... = Ifm bbs bs (E p) by (induct p rule: disjuncts.induct, auto)
  finally show qfree (DJ qe p)  $\wedge$  Ifm bbs bs (DJ qe p) = Ifm bbs bs (E p) using
qfth by blast
qed

consts bnds:: num  $\Rightarrow$  nat list
  lex-ns:: nat list  $\times$  nat list  $\Rightarrow$  bool
recdef bnds measure size
  bnds (Bound n) = [n]
  bnds (CN n c a) = n#(bnds a)
  bnds (Neg a) = bnds a
  bnds (Add a b) = (bnds a)@(bnds b)
  bnds (Sub a b) = (bnds a)@(bnds b)
  bnds (Mul i a) = bnds a
  bnds a = []
recdef lex-ns measure  $(\lambda (xs,ys). \text{length } xs + \text{length } ys)$ 
  lex-ns ([], ms) = True
  lex-ns (ns, []) = False
  lex-ns (n#ns, m#ms) =  $(n < m \vee ((n = m) \wedge \text{lex-ns } (ns, ms)))$ 
constdefs lex-bnd :: num  $\Rightarrow$  num  $\Rightarrow$  bool
  lex-bnd t s  $\equiv$  lex-ns (bnds t, bnds s)

consts
  numadd:: num  $\times$  num  $\Rightarrow$  num
recdef numadd measure  $(\lambda (t,s). \text{num-size } t + \text{num-size } s)$ 

```

```

numadd (CN n1 c1 r1 ,CN n2 c2 r2) =
  (if n1=n2 then
    (let c = c1 + c2
     in (if c=0 then numadd(r1,r2) else CN n1 c (numadd (r1,r2))))
    else if n1 ≤ n2 then CN n1 c1 (numadd (r1,Add (Mul c2 (Bound n2)) r2))
    else CN n2 c2 (numadd (Add (Mul c1 (Bound n1)) r1,r2)))
numadd (CN n1 c1 r1, t) = CN n1 c1 (numadd (r1, t))
numadd (t,CN n2 c2 r2) = CN n2 c2 (numadd (t,r2))
numadd (C b1, C b2) = C (b1+b2)
numadd (a,b) = Add a b

```

```

lemma numadd: Inum bs (numadd (t,s)) = Inum bs (Add t s)
apply (induct t s rule: numadd.induct, simp-all add: Let-def)
apply (case-tac c1+c2 = 0, case-tac n1 ≤ n2, simp-all)
apply (case-tac n1 = n2)
apply (simp-all add: ring-simps)
apply (simp add: left-distrib[symmetric])
done

```

```

lemma numadd-nb: [| numbound0 t ; numbound0 s |] ==> numbound0 (numadd
(t,s))
by (induct t s rule: numadd.induct, auto simp add: Let-def)

```

```

fun
  nummul :: int => num => num
where
  nummul i (C j) = C (i * j)
  | nummul i (CN n c t) = CN n (c*i) (nummul i t)
  | nummul i t = Mul i t

```

```

lemma nummul: ∧ i. Inum bs (nummul i t) = Inum bs (Mul i t)
by (induct t rule: nummul.induct, auto simp add: ring-simps numadd)

```

```

lemma nummul-nb: ∧ i. numbound0 t ==> numbound0 (nummul i t)
by (induct t rule: nummul.induct, auto simp add: numadd-nb)

```

```

constdefs numneg :: num => num
  numneg t ≡ nummul (- 1) t

```

```

constdefs numsub :: num => num => num
  numsub s t ≡ (if s = t then C 0 else numadd (s, numneg t))

```

```

lemma numneg: Inum bs (numneg t) = Inum bs (Neg t)
using numneg-def nummul by simp

```

```

lemma numneg-nb: numbound0 t ==> numbound0 (numneg t)
using numneg-def nummul-nb by simp

```

```

lemma numsub:  $Inum\ bs\ (numsub\ a\ b) = Inum\ bs\ (Sub\ a\ b)$ 
using numneg numadd numsub-def by simp

lemma numsub-nb:  $\llbracket\ numbound0\ t\ ;\ numbound0\ s\ \rrbracket \implies numbound0\ (numsub\ t\ s)$ 
using numsub-def numadd-nb numneg-nb by simp

fun
  simpnum ::  $num \Rightarrow num$ 
where
  simpnum (C j) = C j
  | simpnum (Bound n) = CN n 1 (C 0)
  | simpnum (Neg t) = numneg (simpnum t)
  | simpnum (Add t s) = numadd (simpnum t, simpnum s)
  | simpnum (Sub t s) = numsub (simpnum t) (simpnum s)
  | simpnum (Mul i t) = (if i = 0 then C 0 else nummul i (simpnum t))
  | simpnum t = t

lemma simpnum-ci:  $Inum\ bs\ (simpnum\ t) = Inum\ bs\ t$ 
by (induct t rule: simpnum.induct, auto simp add: numneg numadd numsub nummul)

lemma simpnum-numbound0:
  numbound0 t  $\implies numbound0\ (simpnum\ t)$ 
by (induct t rule: simpnum.induct, auto simp add: numadd-nb numsub-nb nummul-nb numneg-nb)

fun
  not ::  $fm \Rightarrow fm$ 
where
  not (NOT p) = p
  | not T = F
  | not F = T
  | not p = NOT p
lemma not:  $Ifm\ bbs\ bs\ (not\ p) = Ifm\ bbs\ bs\ (NOT\ p)$ 
by (cases p) auto
lemma not-qf:  $qfree\ p \implies qfree\ (not\ p)$ 
by (cases p, auto)
lemma not-bn:  $bound0\ p \implies bound0\ (not\ p)$ 
by (cases p, auto)

constdefs conj ::  $fm \Rightarrow fm \Rightarrow fm$ 
  conj p q  $\equiv (if\ (p = F \vee q = F)\ then\ F\ else\ if\ p = T\ then\ q\ else\ if\ q = T\ then\ p\ else\ And\ p\ q)$ 
lemma conj:  $Ifm\ bbs\ bs\ (conj\ p\ q) = Ifm\ bbs\ bs\ (And\ p\ q)$ 
by (cases p=F  $\vee$  q=F, simp-all add: conj-def) (cases p, simp-all)

lemma conj-qf:  $\llbracket\ qfree\ p\ ;\ qfree\ q\ \rrbracket \implies qfree\ (conj\ p\ q)$ 
using conj-def by auto

```

lemma *conj-nb*: $\llbracket \text{bound0 } p ; \text{bound0 } q \rrbracket \implies \text{bound0 } (\text{conj } p \ q)$
using *conj-def* **by** *auto*

constdefs *disj* :: $fm \Rightarrow fm \Rightarrow fm$
disj $p \ q \equiv$ (if ($p = T \vee q = T$) then T else if $p = F$ then q else if $q = F$ then p else
 $Or \ p \ q$)

lemma *disj*: $Ifm \ bbs \ bs \ (\text{disj } p \ q) = Ifm \ bbs \ bs \ (Or \ p \ q)$
by (*cases* $p = T \vee q = T$, *simp-all* *add*: *disj-def*) (*cases* p , *simp-all*)
lemma *disj-qf*: $\llbracket qfree \ p ; qfree \ q \rrbracket \implies qfree \ (\text{disj } p \ q)$
using *disj-def* **by** *auto*
lemma *disj-nb*: $\llbracket \text{bound0 } p ; \text{bound0 } q \rrbracket \implies \text{bound0 } (\text{disj } p \ q)$
using *disj-def* **by** *auto*

constdefs *imp* :: $fm \Rightarrow fm \Rightarrow fm$
imp $p \ q \equiv$ (if ($p = F \vee q = T$) then T else if $p = T$ then q else if $q = F$ then $not \ p$
else $Imp \ p \ q$)
lemma *imp*: $Ifm \ bbs \ bs \ (\text{imp } p \ q) = Ifm \ bbs \ bs \ (Imp \ p \ q)$
by (*cases* $p = F \vee q = T$, *simp-all* *add*: *imp-def*, *cases* p) (*simp-all* *add*: *not*)
lemma *imp-qf*: $\llbracket qfree \ p ; qfree \ q \rrbracket \implies qfree \ (\text{imp } p \ q)$
using *imp-def* **by** (*cases* $p = F \vee q = T$, *simp-all* *add*: *imp-def*, *cases* p) (*simp-all*
add: *not-qf*)
lemma *imp-nb*: $\llbracket \text{bound0 } p ; \text{bound0 } q \rrbracket \implies \text{bound0 } (\text{imp } p \ q)$
using *imp-def* **by** (*cases* $p = F \vee q = T$, *simp-all* *add*: *imp-def*, *cases* p) *simp-all*

constdefs *iff* :: $fm \Rightarrow fm \Rightarrow fm$
iff $p \ q \equiv$ (if ($p = q$) then T else if ($p = not \ q \vee not \ p = q$) then F else
if $p = F$ then $not \ q$ else if $q = F$ then $not \ p$ else if $p = T$ then q else if $q = T$ then
 p else
 $Iff \ p \ q$)
lemma *iff*: $Ifm \ bbs \ bs \ (\text{iff } p \ q) = Ifm \ bbs \ bs \ (Iff \ p \ q)$
by (*unfold* *iff-def*, *cases* $p = q$, *simp*, *cases* $p = not \ q$, *simp* *add*: *not*)
(*cases* $not \ p = q$, *auto* *simp* *add*: *not*)
lemma *iff-qf*: $\llbracket qfree \ p ; qfree \ q \rrbracket \implies qfree \ (\text{iff } p \ q)$
by (*unfold* *iff-def*, *cases* $p = q$, *auto* *simp* *add*: *not-qf*)
lemma *iff-nb*: $\llbracket \text{bound0 } p ; \text{bound0 } q \rrbracket \implies \text{bound0 } (\text{iff } p \ q)$
using *iff-def* **by** (*unfold* *iff-def*, *cases* $p = q$, *auto* *simp* *add*: *not-bn*)

function (*sequential*)

simpfm :: $fm \Rightarrow fm$

where

simpfm ($And \ p \ q$) = *conj* (*simpfm* p) (*simpfm* q)
| *simpfm* ($Or \ p \ q$) = *disj* (*simpfm* p) (*simpfm* q)
| *simpfm* ($Imp \ p \ q$) = *imp* (*simpfm* p) (*simpfm* q)
| *simpfm* ($Iff \ p \ q$) = *iff* (*simpfm* p) (*simpfm* q)
| *simpfm* ($NOT \ p$) = *not* (*simpfm* p)
| *simpfm* ($Lt \ a$) = (let $a' = \text{simpnum } a$ in case a' of $C \ v \Rightarrow$ if ($v < 0$) then T
else F
| $- \Rightarrow Lt \ a'$)

$| \text{simpfm } (Le \ a) = (\text{let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (v \leq 0) \text{ then } T$
 $\text{else } F \mid - \Rightarrow Le \ a')$
 $| \text{simpfm } (Gt \ a) = (\text{let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (v > 0) \text{ then } T$
 $\text{else } F \mid - \Rightarrow Gt \ a')$
 $| \text{simpfm } (Ge \ a) = (\text{let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (v \geq 0) \text{ then } T$
 $\text{else } F \mid - \Rightarrow Ge \ a')$
 $| \text{simpfm } (Eq \ a) = (\text{let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (v = 0) \text{ then } T$
 $\text{else } F \mid - \Rightarrow Eq \ a')$
 $| \text{simpfm } (NEq \ a) = (\text{let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (v \neq 0) \text{ then } T$
 $\text{else } F \mid - \Rightarrow NEq \ a')$
 $| \text{simpfm } (Dvd \ i \ a) = (\text{if } i=0 \text{ then } \text{simpfm } (Eq \ a)$
 $\text{else if } (abs \ i = 1) \text{ then } T$
 $\text{else let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (i \text{ dvd } v) \text{ then } T \text{ else } F$
 $\mid - \Rightarrow Dvd \ i \ a')$
 $| \text{simpfm } (NDvd \ i \ a) = (\text{if } i=0 \text{ then } \text{simpfm } (NEq \ a)$
 $\text{else if } (abs \ i = 1) \text{ then } F$
 $\text{else let } a' = \text{simpnum } a \text{ in case } a' \text{ of } C \ v \Rightarrow \text{if } (\neg(i \text{ dvd } v)) \text{ then } T \text{ else } F$
 $\mid - \Rightarrow NDvd \ i \ a')$
 $| \text{simpfm } p = p$
by *pat-completeness auto*
termination by (*relation measure fmsize*) *auto*

lemma *simpfm*: *Ifm bbs bs (simpfm p) = Ifm bbs bs p*
proof(*induct p rule: simpfm.induct*)
 $\text{case } (6 \ a) \text{ let } ?sa = \text{simpnum } a \text{ from } \text{simpnum-ci} \text{ have } sa: \text{Inum } bs \ ?sa =$
 $\text{Inum } bs \ a \text{ by } \text{simp}$
 $\{ \text{fix } v \text{ assume } ?sa = C \ v \text{ hence } ?case \text{ using } sa \text{ by } \text{simp} \}$
 $\text{moreover } \{ \text{assume } \neg (\exists \ v. ?sa = C \ v) \text{ hence } ?case \text{ using } sa$
 $\text{by } (\text{cases } ?sa, \text{simp-all add: Let-def}) \}$
 $\text{ultimately show } ?case \text{ by } \text{blast}$
next
 $\text{case } (7 \ a) \text{ let } ?sa = \text{simpnum } a$
 $\text{from } \text{simpnum-ci} \text{ have } sa: \text{Inum } bs \ ?sa = \text{Inum } bs \ a \text{ by } \text{simp}$
 $\{ \text{fix } v \text{ assume } ?sa = C \ v \text{ hence } ?case \text{ using } sa \text{ by } \text{simp} \}$
 $\text{moreover } \{ \text{assume } \neg (\exists \ v. ?sa = C \ v) \text{ hence } ?case \text{ using } sa$
 $\text{by } (\text{cases } ?sa, \text{simp-all add: Let-def}) \}$
 $\text{ultimately show } ?case \text{ by } \text{blast}$
next
 $\text{case } (8 \ a) \text{ let } ?sa = \text{simpnum } a$
 $\text{from } \text{simpnum-ci} \text{ have } sa: \text{Inum } bs \ ?sa = \text{Inum } bs \ a \text{ by } \text{simp}$
 $\{ \text{fix } v \text{ assume } ?sa = C \ v \text{ hence } ?case \text{ using } sa \text{ by } \text{simp} \}$
 $\text{moreover } \{ \text{assume } \neg (\exists \ v. ?sa = C \ v) \text{ hence } ?case \text{ using } sa$
 $\text{by } (\text{cases } ?sa, \text{simp-all add: Let-def}) \}$
 $\text{ultimately show } ?case \text{ by } \text{blast}$
next
 $\text{case } (9 \ a) \text{ let } ?sa = \text{simpnum } a$
 $\text{from } \text{simpnum-ci} \text{ have } sa: \text{Inum } bs \ ?sa = \text{Inum } bs \ a \text{ by } \text{simp}$
 $\{ \text{fix } v \text{ assume } ?sa = C \ v \text{ hence } ?case \text{ using } sa \text{ by } \text{simp} \}$
 $\text{moreover } \{ \text{assume } \neg (\exists \ v. ?sa = C \ v) \text{ hence } ?case \text{ using } sa$

```

    by (cases ?sa, simp-all add: Let-def)}
  ultimately show ?case by blast
next
case (10 a) let ?sa = simpnum a
from simpnum-ci have sa: Inum bs ?sa = Inum bs a by simp
{fix v assume ?sa = C v hence ?case using sa by simp }
moreover {assume  $\neg (\exists v. ?sa = C v)$  hence ?case using sa
  by (cases ?sa, simp-all add: Let-def)}
ultimately show ?case by blast
next
case (11 a) let ?sa = simpnum a
from simpnum-ci have sa: Inum bs ?sa = Inum bs a by simp
{fix v assume ?sa = C v hence ?case using sa by simp }
moreover {assume  $\neg (\exists v. ?sa = C v)$  hence ?case using sa
  by (cases ?sa, simp-all add: Let-def)}
ultimately show ?case by blast
next
case (12 i a) let ?sa = simpnum a from simpnum-ci
have sa: Inum bs ?sa = Inum bs a by simp
have  $i=0 \vee \text{abs } i = 1 \vee (i \neq 0 \wedge (\text{abs } i \neq 1))$  by auto
{assume  $i=0$  hence ?case using 12.hyps by (simp add: dvd-def Let-def)}
moreover
{assume i1:  $\text{abs } i = 1$ 
  from zdvd-1-left[where  $m = \text{Inum bs } a$ ] uminus-dvd-conv[where  $d=1$  and
 $t=\text{Inum bs } a$ ]
  have ?case using i1 apply (cases  $i=0$ , simp-all add: Let-def)
  by (cases  $i > 0$ , simp-all)}
moreover
{assume inz:  $i \neq 0$  and cond:  $\text{abs } i \neq 1$ 
  {fix v assume ?sa = C v hence ?case using sa[symmetric] inz cond
    by (cases  $\text{abs } i = 1$ , auto) }
  moreover {assume  $\neg (\exists v. ?sa = C v)$ 
    hence  $\text{simpfm } (Dvd i a) = Dvd i ?sa$  using inz cond
    by (cases ?sa, auto simp add: Let-def)
    hence ?case using sa by simp}
  ultimately have ?case by blast}
ultimately show ?case by blast
next
case (13 i a) let ?sa = simpnum a from simpnum-ci
have sa: Inum bs ?sa = Inum bs a by simp
have  $i=0 \vee \text{abs } i = 1 \vee (i \neq 0 \wedge (\text{abs } i \neq 1))$  by auto
{assume  $i=0$  hence ?case using 13.hyps by (simp add: dvd-def Let-def)}
moreover
{assume i1:  $\text{abs } i = 1$ 
  from zdvd-1-left[where  $m = \text{Inum bs } a$ ] uminus-dvd-conv[where  $d=1$  and
 $t=\text{Inum bs } a$ ]
  have ?case using i1 apply (cases  $i=0$ , simp-all add: Let-def)
  apply (cases  $i > 0$ , simp-all) done}
moreover

```

```

{assume inz:  $i \neq 0$  and cond:  $\text{abs } i \neq 1$ 
 {fix v assume  $?sa = C \ v$  hence  $?case$  using  $sa[symmetric]$  inz cond
   by (cases  $\text{abs } i = 1$ , auto) }
moreover {assume  $\neg (\exists v. ?sa = C \ v)$ 
  hence  $\text{simpfm } (NDvd \ i \ a) = NDvd \ i \ ?sa$  using inz cond
    by (cases  $?sa$ , auto simp add: Let-def)
  hence  $?case$  using  $sa$  by simp}
ultimately have  $?case$  by blast}
ultimately show  $?case$  by blast
qed (induct p rule: simpfm.induct, simp-all add: conj disj imp iff not)

lemma simpfm-bound0:  $\text{bound0 } p \implies \text{bound0 } (\text{simpfm } p)$ 
proof (induct p rule: simpfm.induct)
  case (6 a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (7 a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (8 a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (9 a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (10 a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (11 a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (12 i a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
next
  case (13 i a) hence  $nb: \text{numbound0 } a$  by simp
  hence  $\text{numbound0 } (\text{simpnum } a)$  by (simp only: simpnum-numbound0[OF nb])
  thus  $?case$  by (cases simpnum a, auto simp add: Let-def)
qed (auto simp add: disj-def imp-def iff-def conj-def not-bn)

lemma simpfm-qf:  $qfree \ p \implies qfree \ (\text{simpfm } p)$ 
by (induct p rule: simpfm.induct, auto simp add: disj-qf imp-qf iff-qf conj-qf not-qf
  Let-def)

```

(*case-tac simpnum a,auto*)+

consts *qelim* :: *fm* \Rightarrow (*fm* \Rightarrow *fm*) \Rightarrow *fm*
recdef *qelim* *measure fmsize*
qelim (*E* *p*) = (λ *qe*. *DJ* *qe* (*qelim* *p* *qe*))
qelim (*A* *p*) = (λ *qe*. *not* (*qe* (*qelim* (*NOT* *p*) *qe*))))
qelim (*NOT* *p*) = (λ *qe*. *not* (*qelim* *p* *qe*))
qelim (*And* *p* *q*) = (λ *qe*. *conj* (*qelim* *p* *qe*) (*qelim* *q* *qe*))
qelim (*Or* *p* *q*) = (λ *qe*. *disj* (*qelim* *p* *qe*) (*qelim* *q* *qe*))
qelim (*Imp* *p* *q*) = (λ *qe*. *imp* (*qelim* *p* *qe*) (*qelim* *q* *qe*))
qelim (*Iff* *p* *q*) = (λ *qe*. *iff* (*qelim* *p* *qe*) (*qelim* *q* *qe*))
qelim *p* = (λ *y*. *simpfm* *p*)

lemma *qelim-ci*:

assumes *qe-inv*: \forall *bs* *p*. *qfree* *p* \longrightarrow *qfree* (*qe* *p*) \wedge (*Ifm* *bbs* *bs* (*qe* *p*) = *Ifm* *bbs* *bs* (*E* *p*))
shows \bigwedge *bs*. *qfree* (*qelim* *p* *qe*) \wedge (*Ifm* *bbs* *bs* (*qelim* *p* *qe*) = *Ifm* *bbs* *bs* *p*)
using *qe-inv* *DJ-qe*[*OF* *qe-inv*]
by(*induct* *p* *rule*: *qelim.induct*)
(*auto simp add*: *not disj conj iff imp not-qf disj-qf conj-qf imp-qf iff-qf*
simpfm simpfm-qf simp del: *simpfm.simps*)

fun

zsplitt0 :: *num* \Rightarrow *int* \times *num*

where

zsplitt0 (*C* *c*) = (*0*, *C* *c*)
| *zsplitt0* (*Bound* *n*) = (*if* *n*=*0* *then* (*1*, *C* *0*) *else* (*0*, *Bound* *n*))
| *zsplitt0* (*CN* *n* *i* *a*) =
(*let* (*i'*, *a'*) = *zsplitt0* *a*
in *if* *n*=*0* *then* (*i*+*i'*, *a'*) *else* (*i'*, *CN* *n* *i* *a'*))
| *zsplitt0* (*Neg* *a*) = (*let* (*i'*, *a'*) = *zsplitt0* *a* *in* (*-i'*, *Neg* *a'*))
| *zsplitt0* (*Add* *a* *b*) = (*let* (*ia*, *a'*) = *zsplitt0* *a* ;
(*ib*, *b'*) = *zsplitt0* *b*
in (*ia*+*ib*, *Add* *a'* *b'*))
| *zsplitt0* (*Sub* *a* *b*) = (*let* (*ia*, *a'*) = *zsplitt0* *a* ;
(*ib*, *b'*) = *zsplitt0* *b*
in (*ia*-*ib*, *Sub* *a'* *b'*))
| *zsplitt0* (*Mul* *i* *a*) = (*let* (*i'*, *a'*) = *zsplitt0* *a* *in* (*i***i'*, *Mul* *i* *a'*))

lemma *zsplitt0-I*:

shows \bigwedge *n* *a*. *zsplitt0* *t* = (*n*, *a*) \Longrightarrow (*Inum* ((*x*::*int*) #*bs*) (*CN* *0* *n* *a*) = *Inum* (*x* #*bs*) *t*) \wedge *numbound0* *a*
(*is* \bigwedge *n* *a*. *?S* *t* = (*n*, *a*) \Longrightarrow (*?I* *x* (*CN* *0* *n* *a*) = *?I* *x* *t*) \wedge *?N* *a*)
proof(*induct* *t* *rule*: *zsplitt0.induct*)
case (*1* *c* *n* *a*) **thus** *?case* **by** *auto*


```

next
  case (2 m n a) thus ?case by (cases m=0) auto
next
  case (3 m i a n a')
  let ?j = fst (zspl0 a)
  let ?b = snd (zspl0 a)
  have abj: zspl0 a = (?j, ?b) by simp
  {assume m≠0
   with prems(1)[OF abj] prems(2) have ?case by (auto simp add: Let-def
split-def)}
  moreover
  {assume m0: m = 0
   from abj have th: a'=?b ∧ n=i+?j using prems
   by (simp add: Let-def split-def)
   from abj prems have th2: (?I x (CN 0 ?j ?b) = ?I x a) ∧ ?N ?b by blast
   from th have ?I x (CN 0 n a') = ?I x (CN 0 (i+?j) ?b) by simp
   also from th2 have ... = ?I x (CN 0 i (CN 0 ?j ?b)) by (simp add: left-distrib)
   finally have ?I x (CN 0 n a') = ?I x (CN 0 i a) using th2 by simp
   with th2 th have ?case using m0 by blast}
ultimately show ?case by blast
next
  case (4 t n a)
  let ?nt = fst (zspl0 t)
  let ?at = snd (zspl0 t)
  have abj: zspl0 t = (?nt, ?at) by simp hence th: a=Neg ?at ∧ n=-?nt using
prems
  by (simp add: Let-def split-def)
  from abj prems have th2: (?I x (CN 0 ?nt ?at) = ?I x t) ∧ ?N ?at by blast
  from th2[simplified] th[simplified] show ?case by simp
next
  case (5 s t n a)
  let ?ns = fst (zspl0 s)
  let ?as = snd (zspl0 s)
  let ?nt = fst (zspl0 t)
  let ?at = snd (zspl0 t)
  have abjs: zspl0 s = (?ns, ?as) by simp
  moreover have abjt: zspl0 t = (?nt, ?at) by simp
  ultimately have th: a=Add ?as ?at ∧ n=?ns + ?nt using prems
  by (simp add: Let-def split-def)
  from abjs[symmetric] have bluddy: ∃ x y. (x,y) = zspl0 s by blast
  from prems have (∃ x y. (x,y) = zspl0 s) ⟶ (∀ xa xb. zspl0 t = (xa, xb)
⟶ Inum (x # bs) (CN 0 xa xb) = Inum (x # bs) t ∧ numbound0 xb) by auto
  with bluddy abjt have th3: (?I x (CN 0 ?nt ?at) = ?I x t) ∧ ?N ?at by blast
  from abjs prems have th2: (?I x (CN 0 ?ns ?as) = ?I x s) ∧ ?N ?as by blast
  from th3[simplified] th2[simplified] th[simplified] show ?case
  by (simp add: left-distrib)
next
  case (6 s t n a)
  let ?ns = fst (zspl0 s)

```

```

let ?as = snd (zsplit0 s)
let ?nt = fst (zsplit0 t)
let ?at = snd (zsplit0 t)
have abjs: zsplit0 s = (?ns, ?as) by simp
moreover have abjt: zsplit0 t = (?nt, ?at) by simp
ultimately have th: a=Sub ?as ?at ∧ n=?ns - ?nt using prems
  by (simp add: Let-def split-def)
from abjs[symmetric] have bluddy: ∃ x y. (x,y) = zsplit0 s by blast
from prems have (∃ x y. (x,y) = zsplit0 s) ⟶ (∀ xa xb. zsplit0 t = (xa, xb)
⟶ Inum (x # bs) (CN 0 xa xb) = Inum (x # bs) t ∧ numbound0 xb) by auto
with bluddy abjt have th3: (?I x (CN 0 ?nt ?at) = ?I x t) ∧ ?N ?at by blast
from abjs prems have th2: (?I x (CN 0 ?ns ?as) = ?I x s) ∧ ?N ?as by blast
from th3[simplified] th2[simplified] th[simplified] show ?case
  by (simp add: left-diff-distrib)
next
  case (7 i t n a)
  let ?nt = fst (zsplit0 t)
  let ?at = snd (zsplit0 t)
  have abj: zsplit0 t = (?nt, ?at) by simp hence th: a=Mul i ?at ∧ n=i*?nt
using prems
  by (simp add: Let-def split-def)
from abj prems have th2: (?I x (CN 0 ?nt ?at) = ?I x t) ∧ ?N ?at by blast
hence ?I x (Mul i t) = i * ?I x (CN 0 ?nt ?at) by simp
also have ... = ?I x (CN 0 (i*?nt) (Mul i ?at)) by (simp add: right-distrib)
finally show ?case using th th2 by simp
qed

consts
  iszlfm :: fm ⇒ bool
recdef iszlfm measure size
  iszlfm (And p q) = (iszlfm p ∧ iszlfm q)
  iszlfm (Or p q) = (iszlfm p ∧ iszlfm q)
  iszlfm (Eq (CN 0 c e)) = (c>0 ∧ numbound0 e)
  iszlfm (NEq (CN 0 c e)) = (c>0 ∧ numbound0 e)
  iszlfm (Lt (CN 0 c e)) = (c>0 ∧ numbound0 e)
  iszlfm (Le (CN 0 c e)) = (c>0 ∧ numbound0 e)
  iszlfm (Gt (CN 0 c e)) = (c>0 ∧ numbound0 e)
  iszlfm (Ge (CN 0 c e)) = (c>0 ∧ numbound0 e)
  iszlfm (Dvd i (CN 0 c e)) =
    (c>0 ∧ i>0 ∧ numbound0 e)
  iszlfm (NDvd i (CN 0 c e)) =
    (c>0 ∧ i>0 ∧ numbound0 e)
  iszlfm p = (isatom p ∧ (bound0 p))

lemma zlin-qfree: iszlfm p ⟹ qfree p
  by (induct p rule: iszlfm.induct) auto

consts
  zlfm :: fm ⇒ fm

```

```

recdef zlfm measure fmsize
  zlfm (And p q) = And (zlfm p) (zlfm q)
  zlfm (Or p q) = Or (zlfm p) (zlfm q)
  zlfm (Imp p q) = Or (zlfm (NOT p)) (zlfm q)
  zlfm (Iff p q) = Or (And (zlfm p) (zlfm q)) (And (zlfm (NOT p)) (zlfm (NOT
q)))
  zlfm (Lt a) = (let (c,r) = zsplt0 a in
    if c=0 then Lt r else
    if c>0 then (Lt (CN 0 c r)) else (Gt (CN 0 (- c) (Neg r))))
  zlfm (Le a) = (let (c,r) = zsplt0 a in
    if c=0 then Le r else
    if c>0 then (Le (CN 0 c r)) else (Ge (CN 0 (- c) (Neg r))))
  zlfm (Gt a) = (let (c,r) = zsplt0 a in
    if c=0 then Gt r else
    if c>0 then (Gt (CN 0 c r)) else (Lt (CN 0 (- c) (Neg r))))
  zlfm (Ge a) = (let (c,r) = zsplt0 a in
    if c=0 then Ge r else
    if c>0 then (Ge (CN 0 c r)) else (Le (CN 0 (- c) (Neg r))))
  zlfm (Eq a) = (let (c,r) = zsplt0 a in
    if c=0 then Eq r else
    if c>0 then (Eq (CN 0 c r)) else (Eq (CN 0 (- c) (Neg r))))
  zlfm (NEq a) = (let (c,r) = zsplt0 a in
    if c=0 then NEq r else
    if c>0 then (NEq (CN 0 c r)) else (NEq (CN 0 (- c) (Neg r))))
  zlfm (Dvd i a) = (if i=0 then zlfm (Eq a)
    else (let (c,r) = zsplt0 a in
      if c=0 then (Dvd (abs i) r) else
      if c>0 then (Dvd (abs i) (CN 0 c r))
      else (Dvd (abs i) (CN 0 (- c) (Neg r)))))
  zlfm (NDvd i a) = (if i=0 then zlfm (NEq a)
    else (let (c,r) = zsplt0 a in
      if c=0 then (NDvd (abs i) r) else
      if c>0 then (NDvd (abs i) (CN 0 c r))
      else (NDvd (abs i) (CN 0 (- c) (Neg r)))))
  zlfm (NOT (And p q)) = Or (zlfm (NOT p)) (zlfm (NOT q))
  zlfm (NOT (Or p q)) = And (zlfm (NOT p)) (zlfm (NOT q))
  zlfm (NOT (Imp p q)) = And (zlfm p) (zlfm (NOT q))
  zlfm (NOT (Iff p q)) = Or (And(zlfm p) (zlfm(NOT q))) (And (zlfm(NOT p))
(zlfm q))
  zlfm (NOT (NOT p)) = zlfm p
  zlfm (NOT T) = F
  zlfm (NOT F) = T
  zlfm (NOT (Lt a)) = zlfm (Ge a)
  zlfm (NOT (Le a)) = zlfm (Gt a)
  zlfm (NOT (Gt a)) = zlfm (Le a)
  zlfm (NOT (Ge a)) = zlfm (Lt a)
  zlfm (NOT (Eq a)) = zlfm (NEq a)
  zlfm (NOT (NEq a)) = zlfm (Eq a)
  zlfm (NOT (Dvd i a)) = zlfm (NDvd i a)

```

$zlfm (NOT (NDvd i a)) = zlfm (Dvd i a)$
 $zlfm (NOT (Closed P)) = NClosed P$
 $zlfm (NOT (NClosed P)) = Closed P$
 $zlfm p = p$ (**hints** simp add: fmsize-pos)

lemma *zlfm-I*:

assumes *qfp*: *qfree p*
shows $(Ifm\ bbs\ (i\#bs)\ (zlfm\ p) = Ifm\ bbs\ (i\#bs)\ p) \wedge iszlfm\ (zlfm\ p)$
 $(is\ (?I\ (?l\ p) = ?I\ p) \wedge ?L\ (?l\ p))$
using *qfp*
proof(*induct p rule: zlfm.induct*)
 case (5 *a*)
 let *?c* = *fst (zsplit0 a)*
 let *?r* = *snd (zsplit0 a)*
 have *spl*: *zsplit0 a = (?c, ?r)* **by** *simp*
 from *zsplit0-I[OF spl, where x=i and bs=bs]*
 have *Ia*: *Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r)* **and** *nb*: *numbound0 ?r*
by *auto*
 let *?N* = $\lambda t. Inum\ (i\#bs)\ t$
 from *prems Ia nb* **show** *?case*
 apply (*auto simp add: Let-def split-def ring-simps*)
 apply (*cases ?r, auto*)
 apply (*case-tac nat, auto*)
 done
next
 case (6 *a*)
 let *?c* = *fst (zsplit0 a)*
 let *?r* = *snd (zsplit0 a)*
 have *spl*: *zsplit0 a = (?c, ?r)* **by** *simp*
 from *zsplit0-I[OF spl, where x=i and bs=bs]*
 have *Ia*: *Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r)* **and** *nb*: *numbound0 ?r*
by *auto*
 let *?N* = $\lambda t. Inum\ (i\#bs)\ t$
 from *prems Ia nb* **show** *?case*
 apply (*auto simp add: Let-def split-def ring-simps*)
 apply (*cases ?r, auto*)
 apply (*case-tac nat, auto*)
 done
next
 case (7 *a*)
 let *?c* = *fst (zsplit0 a)*
 let *?r* = *snd (zsplit0 a)*
 have *spl*: *zsplit0 a = (?c, ?r)* **by** *simp*
 from *zsplit0-I[OF spl, where x=i and bs=bs]*
 have *Ia*: *Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r)* **and** *nb*: *numbound0 ?r*
by *auto*
 let *?N* = $\lambda t. Inum\ (i\#bs)\ t$
 from *prems Ia nb* **show** *?case*
 apply (*auto simp add: Let-def split-def ring-simps*)

```

    apply (cases ?r, auto)
    apply (case-tac nat, auto)
    done
next
case (8 a)
let ?c = fst (zsplit0 a)
let ?r = snd (zsplit0 a)
have spl: zsplit0 a = (?c, ?r) by simp
from zsplit0-I[OF spl, where x=i and bs=bs]
have Ia:Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r) and nb: numbound0 ?r
by auto
let ?N = λ t. Inum (i # bs) t
from prems Ia nb show ?case
  apply (auto simp add: Let-def split-def ring-simps)
  apply (cases ?r, auto)
  apply (case-tac nat, auto)
  done
next
case (9 a)
let ?c = fst (zsplit0 a)
let ?r = snd (zsplit0 a)
have spl: zsplit0 a = (?c, ?r) by simp
from zsplit0-I[OF spl, where x=i and bs=bs]
have Ia:Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r) and nb: numbound0 ?r
by auto
let ?N = λ t. Inum (i # bs) t
from prems Ia nb show ?case
  apply (auto simp add: Let-def split-def ring-simps)
  apply (cases ?r, auto)
  apply (case-tac nat, auto)
  done
next
case (10 a)
let ?c = fst (zsplit0 a)
let ?r = snd (zsplit0 a)
have spl: zsplit0 a = (?c, ?r) by simp
from zsplit0-I[OF spl, where x=i and bs=bs]
have Ia:Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r) and nb: numbound0 ?r
by auto
let ?N = λ t. Inum (i # bs) t
from prems Ia nb show ?case
  apply (auto simp add: Let-def split-def ring-simps)
  apply (cases ?r, auto)
  apply (case-tac nat, auto)
  done
next
case (11 j a)
let ?c = fst (zsplit0 a)
let ?r = snd (zsplit0 a)

```

```

have spl: zsplit0 a = (?c, ?r) by simp
from zsplit0-I[OF spl, where x=i and bs=bs]
have Ia:Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r) and nb: numbound0 ?r
by auto
let ?N = λ t. Inum (i # bs) t
have j=0 ∨ (j≠0 ∧ ?c = 0) ∨ (j≠0 ∧ ?c > 0) ∨ (j≠ 0 ∧ ?c < 0) by arith
moreover
{assume j=0 hence z: zlfm (Dvd j a) = (zlfm (Eq a)) by (simp add: Let-def)
  hence ?case using prems by (simp del: zlfm.simps add: zdvd-0-left)}
moreover
{assume ?c=0 and j≠0 hence ?case
  using zsplit0-I[OF spl, where x=i and bs=bs] zdvd-abs1[where i=j]
  apply (auto simp add: Let-def split-def ring-simps)
  apply (cases ?r, auto)
  apply (case-tac nat, auto)
  done}
moreover
{assume cp: ?c > 0 and jnz: j≠0 hence l: ?L (?l (Dvd j a))
  by (simp add: nb Let-def split-def)
  hence ?case using Ia cp jnz by (simp add: Let-def split-def
    zdvd-abs1[where i=j and j=(?c*i) + ?N ?r, symmetric])}
moreover
{assume cn: ?c < 0 and jnz: j≠0 hence l: ?L (?l (Dvd j a))
  by (simp add: nb Let-def split-def)
  hence ?case using Ia cn jnz zdvd-zminus-iff[where m=abs j and n=?c*i +
    ?N ?r ]
    by (simp add: Let-def split-def
      zdvd-abs1[where i=j and j=(?c*i) + ?N ?r, symmetric])}
ultimately show ?case by blast
next
case (12 j a)
let ?c = fst (zsplit0 a)
let ?r = snd (zsplit0 a)
have spl: zsplit0 a = (?c, ?r) by simp
from zsplit0-I[OF spl, where x=i and bs=bs]
have Ia:Inum (i # bs) a = Inum (i # bs) (CN 0 ?c ?r) and nb: numbound0 ?r
by auto
let ?N = λ t. Inum (i # bs) t
have j=0 ∨ (j≠0 ∧ ?c = 0) ∨ (j≠0 ∧ ?c > 0) ∨ (j≠ 0 ∧ ?c < 0) by arith
moreover
{assume j=0 hence z: zlfm (NDvd j a) = (zlfm (NEq a)) by (simp add: Let-def)
  hence ?case using prems by (simp del: zlfm.simps add: zdvd-0-left)}
moreover
{assume ?c=0 and j≠0 hence ?case
  using zsplit0-I[OF spl, where x=i and bs=bs] zdvd-abs1[where i=j]
  apply (auto simp add: Let-def split-def ring-simps)
  apply (cases ?r, auto)
  apply (case-tac nat, auto)
  done}

```

```

done}
moreover
{assume cp: ?c > 0 and jnz: j≠0 hence l: ?L (?l (Dvd j a))
  by (simp add: nb Let-def split-def)
  hence ?case using Ia cp jnz by (simp add: Let-def split-def
    zdvd-abs1[where i=j and j=(?c*i) + ?N ?r, symmetric])}
moreover
{assume cn: ?c < 0 and jnz: j≠0 hence l: ?L (?l (Dvd j a))
  by (simp add: nb Let-def split-def)
  hence ?case using Ia cn jnz zdvd-zminus-iff[where m=abs j and n=?c*i +
?N ?r ]
    by (simp add: Let-def split-def
      zdvd-abs1[where i=j and j=(?c*i) + ?N ?r, symmetric])}
ultimately show ?case by blast
qed auto

```

```

consts
  plusinf:: fm ⇒ fm
  minusinf:: fm ⇒ fm
  δ :: fm ⇒ int
  dδ :: fm ⇒ int ⇒ bool

```

```

recdef minusinf measure size
  minusinf (And p q) = And (minusinf p) (minusinf q)
  minusinf (Or p q) = Or (minusinf p) (minusinf q)
  minusinf (Eq (CN 0 c e)) = F
  minusinf (NEq (CN 0 c e)) = T
  minusinf (Lt (CN 0 c e)) = T
  minusinf (Le (CN 0 c e)) = T
  minusinf (Gt (CN 0 c e)) = F
  minusinf (Ge (CN 0 c e)) = F
  minusinf p = p

```

```

lemma minusinf-qfree: qfree p ⇒ qfree (minusinf p)
  by (induct p rule: minusinf.induct, auto)

```

```

recdef plusinf measure size
  plusinf (And p q) = And (plusinf p) (plusinf q)
  plusinf (Or p q) = Or (plusinf p) (plusinf q)
  plusinf (Eq (CN 0 c e)) = F
  plusinf (NEq (CN 0 c e)) = T
  plusinf (Lt (CN 0 c e)) = F
  plusinf (Le (CN 0 c e)) = F
  plusinf (Gt (CN 0 c e)) = T
  plusinf (Ge (CN 0 c e)) = T
  plusinf p = p

```

```

recdef δ measure size
  δ (And p q) = ilcm (δ p) (δ q)

```

```

 $\delta$  ( $Or$   $p$   $q$ ) =  $ilcm$  ( $\delta$   $p$ ) ( $\delta$   $q$ )
 $\delta$  ( $Dvd$   $i$  ( $CN$   $0$   $c$   $e$ )) =  $i$ 
 $\delta$  ( $NDvd$   $i$  ( $CN$   $0$   $c$   $e$ )) =  $i$ 
 $\delta$   $p$  =  $1$ 

recdef  $d\delta$  measure size
   $d\delta$  ( $And$   $p$   $q$ ) = ( $\lambda$   $d$ .  $d\delta$   $p$   $d \wedge d\delta$   $q$   $d$ )
   $d\delta$  ( $Or$   $p$   $q$ ) = ( $\lambda$   $d$ .  $d\delta$   $p$   $d \wedge d\delta$   $q$   $d$ )
   $d\delta$  ( $Dvd$   $i$  ( $CN$   $0$   $c$   $e$ )) = ( $\lambda$   $d$ .  $i$   $dvd$   $d$ )
   $d\delta$  ( $NDvd$   $i$  ( $CN$   $0$   $c$   $e$ )) = ( $\lambda$   $d$ .  $i$   $dvd$   $d$ )
   $d\delta$   $p$  = ( $\lambda$   $d$ .  $True$ )

lemma delta-mono:
  assumes  $lin$ :  $iszlfm$   $p$ 
  and  $d$ :  $d$   $dvd$   $d'$ 
  and  $ad$ :  $d\delta$   $p$   $d$ 
  shows  $d\delta$   $p$   $d'$ 
  using  $lin$   $ad$   $d$ 
proof(induct p rule: iszlfm.induct)
  case ( $9$   $i$   $c$   $e$ ) thus  $?case$  using  $d$ 
    by (simp add: zdvd-trans[where m=i and n=d and k=d'])
next
  case ( $10$   $i$   $c$   $e$ ) thus  $?case$  using  $d$ 
    by (simp add: zdvd-trans[where m=i and n=d and k=d'])
qed simp-all

lemma  $\delta$  : assumes  $lin$ : $iszlfm$   $p$ 
  shows  $d\delta$   $p$  ( $\delta$   $p$ )  $\wedge$   $\delta$   $p$   $>0$ 
using  $lin$ 
proof (induct p rule: iszlfm.induct)
  case ( $1$   $p$   $q$ )
    let  $?d = \delta$  ( $And$   $p$   $q$ )
    from prems ilcm-pos have  $dp$ :  $?d >0$  by simp
    have  $d1$ :  $\delta$   $p$   $dvd$   $\delta$  ( $And$   $p$   $q$ ) using prems by simp
    hence  $th$ :  $d\delta$   $p$   $?d$  using delta-mono prems ( $3-4$ ) by(simp del:dvd-ilcm-self1)
    have  $\delta$   $q$   $dvd$   $\delta$  ( $And$   $p$   $q$ ) using prems by simp
    hence  $th'$ :  $d\delta$   $q$   $?d$  using delta-mono prems by(simp del:dvd-ilcm-self2)
    from  $th$   $th'$   $dp$  show  $?case$  by simp
  next
  case ( $2$   $p$   $q$ )
    let  $?d = \delta$  ( $And$   $p$   $q$ )
    from prems ilcm-pos have  $dp$ :  $?d >0$  by simp
    have  $\delta$   $p$   $dvd$   $\delta$  ( $And$   $p$   $q$ ) using prems by simp
    hence  $th$ :  $d\delta$   $p$   $?d$  using delta-mono prems by(simp del:dvd-ilcm-self1)
    have  $\delta$   $q$   $dvd$   $\delta$  ( $And$   $p$   $q$ ) using prems by simp
    hence  $th'$ :  $d\delta$   $q$   $?d$  using delta-mono prems by(simp del:dvd-ilcm-self2)
    from  $th$   $th'$   $dp$  show  $?case$  by simp
qed simp-all

```


consts

$a\beta :: fm \Rightarrow int \Rightarrow fm$
 $d\beta :: fm \Rightarrow int \Rightarrow bool$
 $\zeta :: fm \Rightarrow int$
 $\beta :: fm \Rightarrow num\ list$
 $\alpha :: fm \Rightarrow num\ list$

recdef $a\beta$ *measure size*

$a\beta\ (And\ p\ q) = (\lambda\ k.\ And\ (a\beta\ p\ k)\ (a\beta\ q\ k))$
 $a\beta\ (Or\ p\ q) = (\lambda\ k.\ Or\ (a\beta\ p\ k)\ (a\beta\ q\ k))$
 $a\beta\ (Eq\ (CN\ 0\ c\ e)) = (\lambda\ k.\ Eq\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (NEq\ (CN\ 0\ c\ e)) = (\lambda\ k.\ NEq\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (Lt\ (CN\ 0\ c\ e)) = (\lambda\ k.\ Lt\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (Le\ (CN\ 0\ c\ e)) = (\lambda\ k.\ Le\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (Gt\ (CN\ 0\ c\ e)) = (\lambda\ k.\ Gt\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (Ge\ (CN\ 0\ c\ e)) = (\lambda\ k.\ Ge\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (Dvd\ i\ (CN\ 0\ c\ e)) = (\lambda\ k.\ Dvd\ ((k\ div\ c)*i)\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ (NDvd\ i\ (CN\ 0\ c\ e)) = (\lambda\ k.\ NDvd\ ((k\ div\ c)*i)\ (CN\ 0\ 1\ (Mul\ (k\ div\ c)\ e)))$
 $a\beta\ p = (\lambda\ k.\ p)$

recdef $d\beta$ *measure size*

$d\beta\ (And\ p\ q) = (\lambda\ k.\ (d\beta\ p\ k) \wedge (d\beta\ q\ k))$
 $d\beta\ (Or\ p\ q) = (\lambda\ k.\ (d\beta\ p\ k) \wedge (d\beta\ q\ k))$
 $d\beta\ (Eq\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (NEq\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (Lt\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (Le\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (Gt\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (Ge\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (Dvd\ i\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ (NDvd\ i\ (CN\ 0\ c\ e)) = (\lambda\ k.\ c\ dvd\ k)$
 $d\beta\ p = (\lambda\ k.\ True)$

recdef ζ *measure size*

$\zeta\ (And\ p\ q) = ilcm\ (\zeta\ p)\ (\zeta\ q)$
 $\zeta\ (Or\ p\ q) = ilcm\ (\zeta\ p)\ (\zeta\ q)$
 $\zeta\ (Eq\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (NEq\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (Lt\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (Le\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (Gt\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (Ge\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (Dvd\ i\ (CN\ 0\ c\ e)) = c$
 $\zeta\ (NDvd\ i\ (CN\ 0\ c\ e)) = c$
 $\zeta\ p = 1$

recdef β *measure size*

$\beta\ (And\ p\ q) = (\beta\ p\ @\ \beta\ q)$

```

 $\beta$  (Or p q) = ( $\beta$  p @  $\beta$  q)
 $\beta$  (Eq (CN 0 c e)) = [Sub (C -1) e]
 $\beta$  (NEq (CN 0 c e)) = [Neg e]
 $\beta$  (Lt (CN 0 c e)) = []
 $\beta$  (Le (CN 0 c e)) = []
 $\beta$  (Gt (CN 0 c e)) = [Neg e]
 $\beta$  (Ge (CN 0 c e)) = [Sub (C -1) e]
 $\beta$  p = []

```

recdef α measure size

```

 $\alpha$  (And p q) = ( $\alpha$  p @  $\alpha$  q)
 $\alpha$  (Or p q) = ( $\alpha$  p @  $\alpha$  q)
 $\alpha$  (Eq (CN 0 c e)) = [Add (C -1) e]
 $\alpha$  (NEq (CN 0 c e)) = [e]
 $\alpha$  (Lt (CN 0 c e)) = [e]
 $\alpha$  (Le (CN 0 c e)) = [Add (C -1) e]
 $\alpha$  (Gt (CN 0 c e)) = []
 $\alpha$  (Ge (CN 0 c e)) = []
 $\alpha$  p = []

```

consts mirror :: fm \Rightarrow fm

recdef mirror measure size

```

mirror (And p q) = And (mirror p) (mirror q)
mirror (Or p q) = Or (mirror p) (mirror q)
mirror (Eq (CN 0 c e)) = Eq (CN 0 c (Neg e))
mirror (NEq (CN 0 c e)) = NEq (CN 0 c (Neg e))
mirror (Lt (CN 0 c e)) = Gt (CN 0 c (Neg e))
mirror (Le (CN 0 c e)) = Ge (CN 0 c (Neg e))
mirror (Gt (CN 0 c e)) = Lt (CN 0 c (Neg e))
mirror (Ge (CN 0 c e)) = Le (CN 0 c (Neg e))
mirror (Dvd i (CN 0 c e)) = Dvd i (CN 0 c (Neg e))
mirror (NDvd i (CN 0 c e)) = NDvd i (CN 0 c (Neg e))
mirror p = p

```

lemma dvd1-eq1: $x > 0 \implies (x::int) \text{ dvd } 1 = (x = 1)$

by auto

lemma minusinf-inf:

assumes linp: iszlfm p

and u: $d\beta$ p 1

shows $\exists (z::int). \forall x < z. \text{Ifm } bbs (x\#bs) (\text{minusinf } p) = \text{Ifm } bbs (x\#bs) p$
(is ?P p is $\exists (z::int). \forall x < z. ?I x (?M p) = ?I x p)$

using linp u

proof (induct p rule: minusinf.induct)

case (1 p q) **thus** ?case

by (auto simp add: dvd1-eq1) (rule-tac $x=\min z \text{ za}$ **in** exI,simp)

next

case (2 p q) **thus** ?case

by (auto simp add: dvd1-eq1) (rule-tac $x=\min z \text{ za}$ **in** exI,simp)

next

```

case (3 c e) hence c1: c=1 and nb: numbound0 e using dvd1-eq1 by simp+
fix a
from 3 have  $\forall x < (- \text{Inum } (a \# bs) e). c * x + \text{Inum } (x \# bs) e \neq 0$ 
proof(clarsimp)
  fix x assume  $x < (- \text{Inum } (a \# bs) e)$  and  $x + \text{Inum } (x \# bs) e = 0$ 
  with numbound0-I[OF nb, where bs=bs and b=a and b'=x]
  show False by simp
qed
thus ?case by auto
next
case (4 c e) hence c1: c=1 and nb: numbound0 e using dvd1-eq1 by simp+
fix a
from 4 have  $\forall x < (- \text{Inum } (a \# bs) e). c * x + \text{Inum } (x \# bs) e \neq 0$ 
proof(clarsimp)
  fix x assume  $x < (- \text{Inum } (a \# bs) e)$  and  $x + \text{Inum } (x \# bs) e = 0$ 
  with numbound0-I[OF nb, where bs=bs and b=a and b'=x]
  show False by simp
qed
thus ?case by auto
next
case (5 c e) hence c1: c=1 and nb: numbound0 e using dvd1-eq1 by simp+
fix a
from 5 have  $\forall x < (- \text{Inum } (a \# bs) e). c * x + \text{Inum } (x \# bs) e < 0$ 
proof(clarsimp)
  fix x assume  $x < (- \text{Inum } (a \# bs) e)$ 
  with numbound0-I[OF nb, where bs=bs and b=a and b'=x]
  show  $x + \text{Inum } (x \# bs) e < 0$  by simp
qed
thus ?case by auto
next
case (6 c e) hence c1: c=1 and nb: numbound0 e using dvd1-eq1 by simp+
fix a
from 6 have  $\forall x < (- \text{Inum } (a \# bs) e). c * x + \text{Inum } (x \# bs) e \leq 0$ 
proof(clarsimp)
  fix x assume  $x < (- \text{Inum } (a \# bs) e)$ 
  with numbound0-I[OF nb, where bs=bs and b=a and b'=x]
  show  $x + \text{Inum } (x \# bs) e \leq 0$  by simp
qed
thus ?case by auto
next
case (7 c e) hence c1: c=1 and nb: numbound0 e using dvd1-eq1 by simp+
fix a
from 7 have  $\forall x < (- \text{Inum } (a \# bs) e). \neg (c * x + \text{Inum } (x \# bs) e > 0)$ 
proof(clarsimp)
  fix x assume  $x < (- \text{Inum } (a \# bs) e)$  and  $x + \text{Inum } (x \# bs) e > 0$ 
  with numbound0-I[OF nb, where bs=bs and b=a and b'=x]
  show False by simp
qed
thus ?case by auto

```

```

next
  case (8 c e) hence c1: c=1 and nb: numbound0 e using dvd1-eq1 by simp+
  fix a
  from 8 have  $\forall x < (- \text{Inum } (a \# bs) e). \neg (c * x + \text{Inum } (x \# bs) e \geq 0)$ 
  proof (clarsimp)
    fix x assume  $x < (- \text{Inum } (a \# bs) e)$  and  $x + \text{Inum } (x \# bs) e \geq 0$ 
    with numbound0-I[OF nb, where bs=bs and b=a and b'=x]
    show False by simp
  qed
  thus ?case by auto
qed auto

lemma minusinf-repeats:
  assumes d:  $d \delta p d$  and linp:  $\text{iszlfm } p$ 
  shows  $\text{Ifm } bbs ((x - k * d) \# bs) (\text{minusinf } p) = \text{Ifm } bbs (x \# bs) (\text{minusinf } p)$ 
using linp d
proof (induct p rule: iszlfm.induct)
  case (9 i c e) hence nbe: numbound0 e and id:  $i \text{ dvd } d$  by simp+
  hence  $\exists k. d = i * k$  by (simp add: dvd-def)
  then obtain di where di-def:  $d = i * di$  by blast
  show ?case
  proof (simp add: numbound0-I[OF nbe, where bs=bs and b=x - k * d and
    b'=x] right-diff-distrib, rule iffI)
    assume
       $i \text{ dvd } c * x - c * (k * d) + \text{Inum } (x \# bs) e$ 
      (is ?ri dvd ?rc*?rx - ?rc*(?rk*?rd) + ?I x e is ?ri dvd ?rt)
    hence  $\exists (l::\text{int}). ?rt = i * l$  by (simp add: dvd-def)
    hence  $\exists (l::\text{int}). c * x + ?I x e = i * l + c * (k * di)$ 
    by (simp add: ring-simps di-def)
    hence  $\exists (l::\text{int}). c * x + ?I x e = i * (l + c * k * di)$ 
    by (simp add: ring-simps)
    hence  $\exists (l::\text{int}). c * x + ?I x e = i * l$  by blast
    thus  $i \text{ dvd } c * x + \text{Inum } (x \# bs) e$  by (simp add: dvd-def)
  next
    assume
       $i \text{ dvd } c * x + \text{Inum } (x \# bs) e$  (is ?ri dvd ?rc*?rx+?e)
    hence  $\exists (l::\text{int}). c * x + ?e = i * l$  by (simp add: dvd-def)
    hence  $\exists (l::\text{int}). c * x - c * (k * d) + ?e = i * l - c * (k * d)$  by simp
    hence  $\exists (l::\text{int}). c * x - c * (k * d) + ?e = i * l - c * (k * i * di)$  by (simp add:
    di-def)
    hence  $\exists (l::\text{int}). c * x - c * (k * d) + ?e = i * (l - c * k * di)$  by (simp add:
    ring-simps)
    hence  $\exists (l::\text{int}). c * x - c * (k * d) + ?e = i * l$ 
    by blast
    thus  $i \text{ dvd } c * x - c * (k * d) + \text{Inum } (x \# bs) e$  by (simp add: dvd-def)
  qed
next
  case (10 i c e) hence nbe: numbound0 e and id:  $i \text{ dvd } d$  by simp+
  hence  $\exists k. d = i * k$  by (simp add: dvd-def)

```

```

then obtain di where di-def: d=i*di by blast
show ?case
proof(simp add: numbound0-I[OF nbe,where bs=bs and b=x - k * d and
b'=x] right-diff-distrib, rule iffI)
  assume
    i dvd c * x - c*(k*d) + Inum (x # bs) e
  (is ?ri dvd ?rc*?rx - ?rc*(?rk*?rd) + ?I x e is ?ri dvd ?rt)
  hence  $\exists (l::int). ?rt = i * l$  by (simp add: dvd-def)
  hence  $\exists (l::int). c*x + ?I x e = i*l + c*(k * i*di)$ 
    by (simp add: ring-simps di-def)
  hence  $\exists (l::int). c*x + ?I x e = i*(l + c*k*di)$ 
    by (simp add: ring-simps)
  hence  $\exists (l::int). c*x + ?I x e = i*l$  by blast
  thus i dvd c*x + Inum (x # bs) e by (simp add: dvd-def)
next
  assume
    i dvd c*x + Inum (x # bs) e (is ?ri dvd ?rc*?rx+?e)
  hence  $\exists (l::int). c*x + ?e = i*l$  by (simp add: dvd-def)
  hence  $\exists (l::int). c*x - c*(k*d) + ?e = i*l - c*(k*d)$  by simp
  hence  $\exists (l::int). c*x - c*(k*d) + ?e = i*l - c*(k*i*di)$  by (simp add:
di-def)
  hence  $\exists (l::int). c*x - c*(k*d) + ?e = i*((l - c*k*di))$  by (simp add:
ring-simps)
  hence  $\exists (l::int). c*x - c * (k*d) + ?e = i*l$ 
    by blast
  thus i dvd c*x - c*(k*d) + Inum (x # bs) e by (simp add: dvd-def)
qed
qed (auto simp add: gr0-conv-Suc numbound0-I[where bs=bs and b=x - k*d
and b'=x])

```

lemma mirror $\alpha\beta$:

```

assumes lp: iszlfm p
shows (Inum (i#bs)) ' set ( $\alpha$  p) = (Inum (i#bs)) ' set ( $\beta$  (mirror p))
using lp
by (induct p rule: mirror.induct, auto)

```

lemma mirror:

```

assumes lp: iszlfm p
shows Ifm bbs (x#bs) (mirror p) = Ifm bbs ((- x)#bs) p
using lp
proof(induct p rule: iszlfm.induct)
  case (9 j c e) hence nb: numbound0 e by simp
  have Ifm bbs (x#bs) (mirror (Dvd j (CN 0 c e))) = (j dvd c*x - Inum (x#bs)
e) (is - = (j dvd c*x - ?e)) by simp
  also have ... = (j dvd (- (c*x - ?e)))
    by (simp only: zdvd-zminus-iff)
  also have ... = (j dvd (c* (- x)) + ?e)
    by (simp only: minus-mult-right[symmetric] minus-mult-left[symmetric]
diff-def zadd-ac zminus-zadd-distrib)

```

```

    by (simp add: ring-simps)
  also have ... = Ifm bbs ((- x)#bs) (Dvd j (CN 0 c e))
    using numbound0-I[OF nb, where bs=bs and b=x and b'=- x]
    by simp
  finally show ?case .
next
  case (10 j c e) hence nb: numbound0 e by simp
  have Ifm bbs (x#bs) (mirror (Dvd j (CN 0 c e))) = (j dvd c*x - Inum (x#bs)
e) (is - = (j dvd c*x - ?e)) by simp
  also have ... = (j dvd (- (c*x - ?e)))
  by (simp only: zdvd-zminus-iff)
  also have ... = (j dvd (c* (- x)) + ?e)
  apply (simp only: minus-mult-right[symmetric] minus-mult-left[symmetric]
diff-def zadd-ac zminus-zadd-distrib)
  by (simp add: ring-simps)
  also have ... = Ifm bbs ((- x)#bs) (Dvd j (CN 0 c e))
    using numbound0-I[OF nb, where bs=bs and b=x and b'=- x]
    by simp
  finally show ?case by simp
qed (auto simp add: numbound0-I[where bs=bs and b=x and b'=- x] gr0-conv-Suc)

lemma mirror-l: iszlfm p  $\wedge$  d $\beta$  p 1
 $\implies$  iszlfm (mirror p)  $\wedge$  d $\beta$  (mirror p) 1
by (induct p rule: mirror.induct, auto)

lemma mirror- $\delta$ : iszlfm p  $\implies$   $\delta$  (mirror p) =  $\delta$  p
by (induct p rule: mirror.induct, auto)

lemma  $\beta$ -numbound0: assumes lp: iszlfm p
shows  $\forall$  b  $\in$  set ( $\beta$  p). numbound0 b
using lp by (induct p rule:  $\beta$ .induct, auto)

lemma d $\beta$ -mono:
assumes linp: iszlfm p
and dr: d $\beta$  p l
and d: l dvd l'
shows d $\beta$  p l'
using dr linp zdvd-trans[where n=l and k=l', simplified d]
by (induct p rule: iszlfm.induct) simp-all

lemma  $\alpha$ -l: assumes lp: iszlfm p
shows  $\forall$  b  $\in$  set ( $\alpha$  p). numbound0 b
using lp
by (induct p rule:  $\alpha$ .induct, auto)

lemma  $\zeta$ :
assumes linp: iszlfm p
shows  $\zeta$  p > 0  $\wedge$  d $\beta$  p ( $\zeta$  p)
using linp

```

```

proof(induct p rule: iszlfm.induct)
  case (1 p q)
    from prems have dl1:  $\zeta p \text{ dvd } \text{ilcm } (\zeta p) (\zeta q)$  by simp
    from prems have dl2:  $\zeta q \text{ dvd } \text{ilcm } (\zeta p) (\zeta q)$  by simp
    from prems  $d\beta\text{-mono}$ [where  $p = p$  and  $l = \zeta p$  and  $l' = \text{ilcm } (\zeta p) (\zeta q)$ ]
       $d\beta\text{-mono}$ [where  $p = q$  and  $l = \zeta q$  and  $l' = \text{ilcm } (\zeta p) (\zeta q)$ ]
      dl1 dl2 show ?case by (auto simp add: ilcm-pos)
  next
    case (2 p q)
      from prems have dl1:  $\zeta p \text{ dvd } \text{ilcm } (\zeta p) (\zeta q)$  by simp
      from prems have dl2:  $\zeta q \text{ dvd } \text{ilcm } (\zeta p) (\zeta q)$  by simp
      from prems  $d\beta\text{-mono}$ [where  $p = p$  and  $l = \zeta p$  and  $l' = \text{ilcm } (\zeta p) (\zeta q)$ ]
         $d\beta\text{-mono}$ [where  $p = q$  and  $l = \zeta q$  and  $l' = \text{ilcm } (\zeta p) (\zeta q)$ ]
        dl1 dl2 show ?case by (auto simp add: ilcm-pos)
  qed (auto simp add: ilcm-pos)

lemma  $a\beta$ : assumes linp:  $\text{iszlfm } p$  and d:  $d\beta p l$  and lp:  $l > 0$ 
  shows  $\text{iszlfm } (a\beta p l) \wedge d\beta (a\beta p l) 1 \wedge (\text{Ifm } bbs (l*x \# bs) (a\beta p l) = \text{Ifm } bbs (x \# bs) p)$ 
using linp d
proof (induct p rule: iszlfm.induct)
  case (5 c e) hence cp:  $c > 0$  and be:  $\text{numbound0 } e$  and d':  $c \text{ dvd } l$  by simp+
    from lp cp have clel:  $c \leq l$  by (simp add: zdvd-imp-le [OF d' lp])
    from cp have cnz:  $c \neq 0$  by simp
    have  $c \text{ div } c \leq l \text{ div } c$ 
      by (simp add: zdiv-mono1 [OF clel cp])
    then have ldcp:  $0 < l \text{ div } c$ 
      by (simp add: zdiv-self [OF cnz])
    have  $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$  using d'  $\text{zdvd-iff-zmod-eq-0}$  [where  $m=c$  and  $n=l$ ] by simp
    hence  $cl:c * (l \text{ div } c) = l$  using  $\text{zmod-zdiv-equality}$  [where  $a=l$  and  $b=c$ , symmetric]
    by simp
    hence  $(l*x + (l \text{ div } c) * \text{Inum } (x \# bs) e < 0) =$ 
       $((c * (l \text{ div } c)) * x + (l \text{ div } c) * \text{Inum } (x \# bs) e < 0)$ 
    by simp
    also have  $\dots = ((l \text{ div } c) * (c*x + \text{Inum } (x \# bs) e) < (l \text{ div } c) * 0)$  by (simp add: ring-simps)
    also have  $\dots = (c*x + \text{Inum } (x \# bs) e < 0)$ 
    using  $\text{mult-less-0-iff}$  [where  $a=(l \text{ div } c)$  and  $b=c*x + \text{Inum } (x \# bs) e$ ] ldcp
by simp
    finally show ?case using  $\text{numbound0-I}$  [OF be, where  $b=l*x$  and  $b'=x$  and  $bs=bs$ ] be by simp
  next
    case (6 c e) hence cp:  $c > 0$  and be:  $\text{numbound0 } e$  and d':  $c \text{ dvd } l$  by simp+
      from lp cp have clel:  $c \leq l$  by (simp add: zdvd-imp-le [OF d' lp])
      from cp have cnz:  $c \neq 0$  by simp
      have  $c \text{ div } c \leq l \text{ div } c$ 
        by (simp add: zdiv-mono1 [OF clel cp])

```

then have $ldcp:0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using $d' \text{ zdvd-iff-zmod-eq-0}$ [where $m=c$ and $n=l$] by simp
 hence $cl:c * (l \text{ div } c) = l$ using zmod-zdiv-equality [where $a=l$ and $b=c$, symmetric]
 by simp
 hence $(l*x + (l \text{ div } c) * Inum (x \# bs) e \leq 0) =$
 $((c * (l \text{ div } c)) * x + (l \text{ div } c) * Inum (x \# bs) e \leq 0)$
 by simp
 also have $\dots = ((l \text{ div } c) * (c * x + Inum (x \# bs) e) \leq ((l \text{ div } c)) * 0)$ by (simp add: ring-simps)
 also have $\dots = (c*x + Inum (x \# bs) e \leq 0)$
 using mult-le-0-iff [where $a=(l \text{ div } c)$ and $b=c*x + Inum (x \# bs) e$] ldcp
 by simp
 finally show ?case using numbound0-I[OF be, where $b=l*x$ and $b'=x$ and $bs=bs$] be by simp
 next
 case (7 c e) hence cp: $c > 0$ and be: numbound0 e and $d': c \text{ dvd } l$ by simp+
 from lp cp have clel: $c \leq l$ by (simp add: zdvd-imp-le [OF d' lp])
 from cp have cnz: $c \neq 0$ by simp
 have $c \text{ div } c \leq l \text{ div } c$
 by (simp add: zdiv-mono1[OF clel cp])
 then have $ldcp:0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using $d' \text{ zdvd-iff-zmod-eq-0}$ [where $m=c$ and $n=l$] by simp
 hence $cl:c * (l \text{ div } c) = l$ using zmod-zdiv-equality [where $a=l$ and $b=c$, symmetric]
 by simp
 hence $(l*x + (l \text{ div } c) * Inum (x \# bs) e > 0) =$
 $((c * (l \text{ div } c)) * x + (l \text{ div } c) * Inum (x \# bs) e > 0)$
 by simp
 also have $\dots = ((l \text{ div } c) * (c * x + Inum (x \# bs) e) > ((l \text{ div } c)) * 0)$ by (simp add: ring-simps)
 also have $\dots = (c * x + Inum (x \# bs) e > 0)$
 using zero-less-mult-iff [where $a=(l \text{ div } c)$ and $b=c * x + Inum (x \# bs) e$] ldcp by simp
 finally show ?case using numbound0-I[OF be, where $b=(l * x)$ and $b'=x$ and $bs=bs$] be by simp
 next
 case (8 c e) hence cp: $c > 0$ and be: numbound0 e and $d': c \text{ dvd } l$ by simp+
 from lp cp have clel: $c \leq l$ by (simp add: zdvd-imp-le [OF d' lp])
 from cp have cnz: $c \neq 0$ by simp
 have $c \text{ div } c \leq l \text{ div } c$
 by (simp add: zdiv-mono1[OF clel cp])
 then have $ldcp:0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using $d' \text{ zdvd-iff-zmod-eq-0}$ [where

$m=c$ and $n=l$] by simp
 hence $cl:c * (l \text{ div } c) = l$ using zmod-zdiv-equality[where $a=l$ and $b=c$, symmetric]
 by simp
 hence $(l*x + (l \text{ div } c)* \text{Inum } (x \# bs) \text{ e} \geq 0) =$
 $((c*(l \text{ div } c))*x + (l \text{ div } c)* \text{Inum } (x \# bs) \text{ e} \geq 0)$
 by simp
 also have $\dots = ((l \text{ div } c)*(c*x + \text{Inum } (x \# bs) \text{ e}) \geq ((l \text{ div } c)) * 0)$
 by (simp add: ring-simps)
 also have $\dots = (c*x + \text{Inum } (x \# bs) \text{ e} \geq 0)$ using ldcp
 zero-le-mult-iff [where $a=l \text{ div } c$ and $b=c*x + \text{Inum } (x \# bs) \text{ e}$] by simp
 finally show ?case using be numbound0-I[OF be,where $b=l*x$ and $b'=x$ and $bs=bs$]
 by simp
 next
 case (3 c e) hence cp: $c>0$ and be: numbound0 e and d': $c \text{ dvd } l$ by simp+
 from lp cp have clel: $c \leq l$ by (simp add: zdvd-imp-le [OF d' lp])
 from cp have cnz: $c \neq 0$ by simp
 have $c \text{ div } c \leq l \text{ div } c$
 by (simp add: zdiv-mono1[OF clel cp])
 then have ldcp: $0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using d' zdvd-iff-zmod-eq-0[where $m=c$ and $n=l$] by simp
 hence $cl:c * (l \text{ div } c) = l$ using zmod-zdiv-equality[where $a=l$ and $b=c$, symmetric]
 by simp
 hence $(l * x + (l \text{ div } c) * \text{Inum } (x \# bs) \text{ e} = 0) =$
 $((c * (l \text{ div } c)) * x + (l \text{ div } c) * \text{Inum } (x \# bs) \text{ e} = 0)$
 by simp
 also have $\dots = ((l \text{ div } c) * (c * x + \text{Inum } (x \# bs) \text{ e}) = ((l \text{ div } c)) * 0)$ by
 (simp add: ring-simps)
 also have $\dots = (c * x + \text{Inum } (x \# bs) \text{ e} = 0)$
 using mult-eq-0-iff [where $a=(l \text{ div } c)$ and $b=c * x + \text{Inum } (x \# bs) \text{ e}$] ldcp
 by simp
 finally show ?case using numbound0-I[OF be,where $b=(l * x)$ and $b'=x$ and $bs=bs$] be by simp
 next
 case (4 c e) hence cp: $c>0$ and be: numbound0 e and d': $c \text{ dvd } l$ by simp+
 from lp cp have clel: $c \leq l$ by (simp add: zdvd-imp-le [OF d' lp])
 from cp have cnz: $c \neq 0$ by simp
 have $c \text{ div } c \leq l \text{ div } c$
 by (simp add: zdiv-mono1[OF clel cp])
 then have ldcp: $0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using d' zdvd-iff-zmod-eq-0[where $m=c$ and $n=l$] by simp
 hence $cl:c * (l \text{ div } c) = l$ using zmod-zdiv-equality[where $a=l$ and $b=c$, symmetric]

by simp
 hence $(l * x + (l \text{ div } c) * \text{Inum } (x \# bs) \text{ e} \neq 0) =$
 $((c * (l \text{ div } c)) * x + (l \text{ div } c) * \text{Inum } (x \# bs) \text{ e} \neq 0)$
 by simp
 also have $\dots = ((l \text{ div } c) * (c * x + \text{Inum } (x \# bs) \text{ e}) \neq ((l \text{ div } c)) * 0)$ by
 (simp add: ring-simps)
 also have $\dots = (c * x + \text{Inum } (x \# bs) \text{ e} \neq 0)$
 using zero-le-mult-iff [where $a=(l \text{ div } c)$ and $b=c * x + \text{Inum } (x \# bs) \text{ e}$]
 ldcp by simp
 finally show ?case using numbound0-I[OF be, where $b=(l * x)$ and $b'=x$ and
 $bs=bs$] be by simp
 next
 case (9 j c e) hence cp: $c > 0$ and be: numbound0 e and jp: $j > 0$ and d': c
 dvd l by simp+
 from lp cp have clel: $c \leq l$ by (simp add: zdvd-imp-le [OF d' lp])
 from cp have cnz: $c \neq 0$ by simp
 have $c \text{ div } c \leq l \text{ div } c$
 by (simp add: zdiv-mono1[OF clel cp])
 then have ldcp: $0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using d' zdvd-iff-zmod-eq-0 [where
 $m=c$ and $n=l$] by simp
 hence cl: $c * (l \text{ div } c) = l$ using zmod-zdiv-equality [where $a=l$ and $b=c$,
 symmetric]
 by simp
 hence $(\exists (k::\text{int}). l * x + (l \text{ div } c) * \text{Inum } (x \# bs) \text{ e} = ((l \text{ div } c) * j) * k)$
 $= (\exists (k::\text{int}). (c * (l \text{ div } c)) * x + (l \text{ div } c) * \text{Inum } (x \# bs) \text{ e} = ((l \text{ div } c) * j) * k)$
 by simp
 also have $\dots = (\exists (k::\text{int}). (l \text{ div } c) * (c * x + \text{Inum } (x \# bs) \text{ e} - j * k) =$
 $(l \text{ div } c) * 0)$ by (simp add: ring-simps)
 also fix k have $\dots = (\exists (k::\text{int}). c * x + \text{Inum } (x \# bs) \text{ e} - j * k = 0)$
 using zero-le-mult-iff [where $a=(l \text{ div } c)$ and $b=c * x + \text{Inum } (x \# bs) \text{ e} -$
 $j * k$] ldcp by simp
 also have $\dots = (\exists (k::\text{int}). c * x + \text{Inum } (x \# bs) \text{ e} = j * k)$ by simp
 finally show ?case using numbound0-I[OF be, where $b=(l * x)$ and $b'=x$ and
 $bs=bs$] be mult-strict-mono[OF ldcp jp ldcp] by (simp add: dvd-def)
 next
 case (10 j c e) hence cp: $c > 0$ and be: numbound0 e and jp: $j > 0$ and d': c
 dvd l by simp+
 from lp cp have clel: $c \leq l$ by (simp add: zdvd-imp-le [OF d' lp])
 from cp have cnz: $c \neq 0$ by simp
 have $c \text{ div } c \leq l \text{ div } c$
 by (simp add: zdiv-mono1[OF clel cp])
 then have ldcp: $0 < l \text{ div } c$
 by (simp add: zdiv-self[OF cnz])
 have $c * (l \text{ div } c) = c * (l \text{ div } c) + l \text{ mod } c$ using d' zdvd-iff-zmod-eq-0 [where
 $m=c$ and $n=l$] by simp
 hence cl: $c * (l \text{ div } c) = l$ using zmod-zdiv-equality [where $a=l$ and $b=c$,
 symmetric]

by simp
 hence $(\exists (k::int). l * x + (l \text{ div } c) * \text{Inum } (x \# bs) \ e = ((l \text{ div } c) * j) * k)$
 $= (\exists (k::int). (c * (l \text{ div } c)) * x + (l \text{ div } c) * \text{Inum } (x \# bs) \ e = ((l \text{ div } c) * j) * k)$ by simp
 also have $\dots = (\exists (k::int). (l \text{ div } c) * (c * x + \text{Inum } (x \# bs) \ e - j * k) = (l \text{ div } c) * 0)$ by (simp add: ring-simps)
 also fix k have $\dots = (\exists (k::int). c * x + \text{Inum } (x \# bs) \ e - j * k = 0)$
 using zero-le-mult-iff [where $a=(l \text{ div } c)$ and $b=c * x + \text{Inum } (x \# bs) \ e - j * k$] ldc by simp
 also have $\dots = (\exists (k::int). c * x + \text{Inum } (x \# bs) \ e = j * k)$ by simp
 finally show ?case using numbound0-I[OF be, where $b=(l * x)$ and $b'=x$ and $bs=bs$] be mult-strict-mono[OF ldc jp ldc] by (simp add: dvd-def)
 qed (auto simp add: gr0-conv-Suc numbound0-I [where $bs=bs$ and $b=(l * x)$ and $b'=x$])

lemma $a\beta$ -ex: assumes $linp: \text{iszlfm } p$ and $d: d\beta \ p \ l$ and $lp: l > 0$
 shows $(\exists x. l \text{ dvd } x \wedge \text{Ifm } bbs \ (x \# bs) \ (a\beta \ p \ l)) = (\exists (x::int). \text{Ifm } bbs \ (x \# bs) \ p)$
 (is $(\exists x. l \text{ dvd } x \wedge ?P \ x) = (\exists x. ?P' \ x)$)
 proof-
 have $(\exists x. l \text{ dvd } x \wedge ?P \ x) = (\exists (x::int). ?P \ (l * x))$
 using unity-coeff-ex [where $l=l$ and $P=?P$, simplified] by simp
 also have $\dots = (\exists (x::int). ?P' \ x)$ using $a\beta$ [OF $linp \ d \ lp$] by simp
 finally show ?thesis .
 qed

lemma β :
 assumes $lp: \text{iszlfm } p$
 and $u: d\beta \ p \ 1$
 and $d: d\delta \ p \ d$
 and $dp: d > 0$
 and $nob: \neg(\exists (j::int) \in \{1 .. d\}. \exists b \in (\text{Inum } (a \# bs)) \ 'set(\beta \ p). x = b + j)$
 and $p: \text{Ifm } bbs \ (x \# bs) \ p$ (is $?P \ x$)
 shows $?P \ (x - d)$
 using $lp \ u \ d \ dp \ nob \ p$
 proof(induct p rule: iszlfm.induct)
 case (5 c e) hence $c1: c=1$ and $bn:\text{numbound0 } e$ using dvd1-eq1 [where $x=c$] by simp+
 with $dp \ p \ c1 \ \text{numbound0-I}$ [OF bn , where $b=(x-d)$ and $b'=x$ and $bs=bs$] prems
 show ?case by simp
 next
 case (6 c e) hence $c1: c=1$ and $bn:\text{numbound0 } e$ using dvd1-eq1 [where $x=c$] by simp+
 with $dp \ p \ c1 \ \text{numbound0-I}$ [OF bn , where $b=(x-d)$ and $b'=x$ and $bs=bs$] prems
 show ?case by simp
 next
 case (7 c e) hence $p: \text{Ifm } bbs \ (x \# bs) \ (Gt \ (CN \ 0 \ c \ e))$ and $c1: c=1$ and

```

bn:numbound0 e using dvd1-eq1[where x=c] by simp+
let ?e = Inum (x # bs) e
{assume (x-d) + ?e > 0 hence ?case using c1
  numbound0-I[OF bn,where b=(x-d) and b'=x and bs=bs] by simp}
moreover
{assume H: ¬ (x-d) + ?e > 0
  let ?v=Neg e
  have vb: ?v ∈ set (β (Gt (CN 0 c e))) by simp
  from prems(11)[simplified simp-thms Inum.simps β.simps set.simps be-simps
numbound0-I[OF bn,where b=a and b'=x and bs=bs]]
  have nob: ¬ (∃ j ∈ {1 .. d}. x = - ?e + j) by auto
  from H p have x + ?e > 0 ∧ x + ?e ≤ d by (simp add: c1)
  hence x + ?e ≥ 1 ∧ x + ?e ≤ d by simp
  hence ∃ (j::int) ∈ {1 .. d}. j = x + ?e by simp
  hence ∃ (j::int) ∈ {1 .. d}. x = (- ?e + j)
    by (simp add: ring-simps)
  with nob have ?case by auto}
ultimately show ?case by blast
next
case (8 c e) hence p: Ifm bbs (x #bs) (Ge (CN 0 c e)) and c1: c=1 and
bn:numbound0 e
  using dvd1-eq1[where x=c] by simp+
let ?e = Inum (x # bs) e
{assume (x-d) + ?e ≥ 0 hence ?case using c1
  numbound0-I[OF bn,where b=(x-d) and b'=x and bs=bs]
  by simp}
moreover
{assume H: ¬ (x-d) + ?e ≥ 0
  let ?v=Sub (C -1) e
  have vb: ?v ∈ set (β (Ge (CN 0 c e))) by simp
  from prems(11)[simplified simp-thms Inum.simps β.simps set.simps be-simps
numbound0-I[OF bn,where b=a and b'=x and bs=bs]]
  have nob: ¬ (∃ j ∈ {1 .. d}. x = - ?e - 1 + j) by auto
  from H p have x + ?e ≥ 0 ∧ x + ?e < d by (simp add: c1)
  hence x + ?e + 1 ≥ 1 ∧ x + ?e + 1 ≤ d by simp
  hence ∃ (j::int) ∈ {1 .. d}. j = x + ?e + 1 by simp
  hence ∃ (j::int) ∈ {1 .. d}. x = - ?e - 1 + j by (simp add: ring-simps)
  with nob have ?case by simp }
ultimately show ?case by blast
next
case (3 c e) hence p: Ifm bbs (x #bs) (Eq (CN 0 c e)) (is ?p x) and c1: c=1
and bn:numbound0 e using dvd1-eq1[where x=c] by simp+
let ?e = Inum (x # bs) e
let ?v=(Sub (C -1) e)
have vb: ?v ∈ set (β (Eq (CN 0 c e))) by simp
from p have x = - ?e by (simp add: c1) with prems(11) show ?case using
dp
  by simp (erule ballE[where x=1],
    simp-all add:ring-simps numbound0-I[OF bn,where b=x and b'=a and

```

```

bs=bs])
next
  case (4 c e) hence p: Ifm bbs (x #bs) (NEq (CN 0 c e)) (is ?p x) and c1: c=1
and bn:numbound0 e using dvd1-eq1[where x=c] by simp+
  let ?e = Inum (x # bs) e
  let ?v=Neg e
  have vb: ?v ∈ set (β (NEq (CN 0 c e))) by simp
  {assume x - d + Inum (((x - d)) # bs) e ≠ 0
   hence ?case by (simp add: c1)}
  moreover
  {assume H: x - d + Inum (((x - d)) # bs) e = 0
   hence x = - Inum (((x - d)) # bs) e + d by simp
   hence x = - Inum (a # bs) e + d
    by (simp add: numbound0-I[OF bn,where b=x - d and b'=a and bs=bs])
   with prems(11) have ?case using dp by simp}
  ultimately show ?case by blast
next
  case (9 j c e) hence p: Ifm bbs (x #bs) (Dvd j (CN 0 c e)) (is ?p x) and c1:
c=1 and bn:numbound0 e using dvd1-eq1[where x=c] by simp+
  let ?e = Inum (x # bs) e
  from prems have id: j dvd d by simp
  from c1 have ?p x = (j dvd (x + ?e)) by simp
  also have ... = (j dvd x - d + ?e)
    using zdvd-period[OF id, where x=x and c=-1 and t=?e] by simp
  finally show ?case
    using numbound0-I[OF bn,where b=(x-d) and b'=x and bs=bs] c1 p by
simp
next
  case (10 j c e) hence p: Ifm bbs (x #bs) (NDvd j (CN 0 c e)) (is ?p x) and
c1: c=1 and bn:numbound0 e using dvd1-eq1[where x=c] by simp+
  let ?e = Inum (x # bs) e
  from prems have id: j dvd d by simp
  from c1 have ?p x = (¬ j dvd (x + ?e)) by simp
  also have ... = (¬ j dvd x - d + ?e)
    using zdvd-period[OF id, where x=x and c=-1 and t=?e] by simp
  finally show ?case using numbound0-I[OF bn,where b=(x-d) and b'=x
and bs=bs] c1 p by simp
qed (auto simp add: numbound0-I[where bs=bs and b=(x - d) and b'=x]
gr0-conv-Suc)

lemma β':
  assumes lp: iszlfm p
  and u: dβ p 1
  and d: dδ p d
  and dp: d > 0
  shows ∀ x. ¬(∃ (j::int) ∈ {1 .. d}. ∃ b ∈ set(β p). Ifm bbs ((Inum (a#bs) b +
j) #bs) p) ⟶ Ifm bbs (x#bs) p ⟶ Ifm bbs ((x - d)#bs) p (is ∀ x. ?b ⟶ ?P
x ⟶ ?P (x - d))
proof(clarify)

```

```

fix x
assume nb:?b and px: ?P x
hence nb2:  $\neg(\exists (j::int) \in \{1..d\}. \exists b \in (Inum (a\#bs)) \text{ 'set}(\beta p). x = b + j)$ 
  by auto
from  $\beta[OF\ lp\ u\ d\ dp\ nb2\ px]$  show ?P (x - d) .
qed
lemma cpmi-eg:  $0 < D \implies (EX\ z::int. ALL\ x. x < z \dashrightarrow (P\ x = P1\ x))$ 
 $\implies ALL\ x. \sim(EX\ (j::int) : \{1..D\}. EX\ (b::int) : B. P(b+j)) \dashrightarrow P\ (x) \dashrightarrow$ 
 $P\ (x - D)$ 
 $\implies (ALL\ (x::int). ALL\ (k::int). ((P1\ x) = (P1\ (x - k * D))))$ 
 $\implies (EX\ (x::int). P(x)) = ((EX\ (j::int) : \{1..D\} . (P1(j))) \mid (EX\ (j::int) :$ 
 $\{1..D\}. EX\ (b::int) : B. P\ (b+j)))$ 
apply(rule iffI)
prefer 2
apply(drule minusinfinity)
apply assumption+
apply(fastsimp)
apply clarsimp
apply(subgoal-tac !!k.  $0 \leq k \implies !x. P\ x \longrightarrow P\ (x - k * D)$ )
apply(frule-tac x = x and z=z in decr-lemma)
apply(subgoal-tac  $P1(x - (|x - z| + 1) * D)$ )
prefer 2
apply(subgoal-tac  $0 \leq (|x - z| + 1)$ )
prefer 2 apply arith
  apply fastsimp
apply(drule (1) periodic-finite-ex)
apply blast
apply(blast dest:decr-mult-lemma)
done

theorem cp-thm:
  assumes lp: iszlfm p
  and u:  $d\beta\ p\ 1$ 
  and d:  $d\delta\ p\ d$ 
  and dp:  $d > 0$ 
  shows  $(\exists (x::int). Ifm\ bbs\ (x\ \#bs)\ p) = (\exists j \in \{1..d\}. Ifm\ bbs\ (j\ \#bs)\ (minusinf$ 
 $p) \vee (\exists b \in set\ (\beta\ p). Ifm\ bbs\ ((Inum\ (i\ \#bs)\ b + j)\ \#bs)\ p))$ 
  (is  $(\exists (x::int). ?P\ (x)) = (\exists j \in ?D. ?M\ j \vee (\exists b \in ?B. ?P\ (?I\ b + j))))$ )
proof-
  from minusinf-inf[OF lp u]
  have th:  $\exists (z::int). \forall x < z. ?P\ (x) = ?M\ x$  by blast
  let ?B' =  $\{ ?I\ b \mid b. b \in ?B \}$ 
  have BB':  $(\exists j \in ?D. \exists b \in ?B. ?P\ (?I\ b + j)) = (\exists j \in ?D. \exists b \in ?B'. ?P\ (b +$ 
 $j))$  by auto
  hence th2:  $\forall x. \neg (\exists j \in ?D. \exists b \in ?B'. ?P\ ((b + j))) \longrightarrow ?P\ (x) \longrightarrow ?P\ ((x$ 
 $- d))$ 
  using  $\beta'[OF\ lp\ u\ d\ dp, \text{ where } a=i \text{ and } bbs = bbs]$  by blast
  from minusinf-repeats[OF d lp]
  have th3:  $\forall x\ k. ?M\ x = ?M\ (x - k * d)$  by simp

```

from *cpmi-eq*[*OF dp th th2 th3*] *BB'* show *?thesis* by *blast*
qed

lemma *mirror-ex*:

assumes *lp: iszlfm p*
shows $(\exists x. \text{Ifm } bbs (x\#bs) (\text{mirror } p)) = (\exists x. \text{Ifm } bbs (x\#bs) p)$
(is $(\exists x. ?I x ?mp) = (\exists x. ?I x p)$)
proof(*auto*)
fix *x* assume *?I x ?mp* hence *?I (− x) p* using *mirror*[*OF lp*] by *blast*
thus $\exists x. ?I x p$ by *blast*
next
fix *x* assume *?I x p* hence *?I (− x) ?mp*
using *mirror*[*OF lp*, where *x = − x*, *symmetric*] by *auto*
thus $\exists x. ?I x ?mp$ by *blast*
qed

lemma *cp-thm'*:

assumes *lp: iszlfm p*
and *up: dβ p 1* and *dd: dδ p d* and *dp: d > 0*
shows $(\exists x. \text{Ifm } bbs (x\#bs) p) = ((\exists j \in \{1 .. d\}. \text{Ifm } bbs (j\#bs) (\text{minusinf } p)))$
 $\vee (\exists j \in \{1 .. d\}. \exists b \in (\text{Inum } (i\#bs)) \text{ 'set } (\beta p). \text{Ifm } bbs ((b+j)\#bs) p))$
using *cp-thm*[*OF lp up dd dp*, where *i=i*] by *auto*

constdefs *unit:: fm ⇒ fm × num list × int*

unit p ≡ (let *p' = zlfm p* ; *l = ζ p'* ; *q = And (Dvd l (CN 0 1 (C 0))) (aβ p' l)* ; *d = δ q* ;
B = remdups (map simpnum (β q)) ; *a = remdups (map simpnum (α q))*
in if length *B* ≤ length *a* then (*q, B, d*) else (*mirror q, a, d*))

lemma *unit: assumes qf: qfree p*

shows $\bigwedge q B d. \text{unit } p = (q, B, d) \implies ((\exists x. \text{Ifm } bbs (x\#bs) p) = (\exists x. \text{Ifm } bbs (x\#bs) q)) \wedge (\text{Inum } (i\#bs) \text{ 'set } B = (\text{Inum } (i\#bs) \text{ 'set } (\beta q) \wedge d\beta q 1 \wedge d\delta q d \wedge d > 0 \wedge \text{iszlfm } q \wedge (\forall b \in \text{set } B. \text{numbound0 } b))$

proof—

fix *q B d*

assume *qBd: unit p = (q, B, d)*

let *?thes* = $((\exists x. \text{Ifm } bbs (x\#bs) p) = (\exists x. \text{Ifm } bbs (x\#bs) q)) \wedge$

Inum (i#bs) 'set B = Inum (i#bs) 'set (β q) ∧

dβ q 1 ∧ dδ q d ∧ 0 < d ∧ iszlfm q ∧ (∀ b ∈ set B. numbound0 b)

let *?I* = $\lambda x p. \text{Ifm } bbs (x\#bs) p$

let *?p' = zlfm p*

let *?l = ζ ?p'*

let *?q = And (Dvd ?l (CN 0 1 (C 0))) (aβ ?p' ?l)*

let *?d = δ ?q*

let *?B = set (β ?q)*

let *?B' = remdups (map simpnum (β ?q))*

```

let ?A = set (α ?q)
let ?A' = remdups (map simpnum (α ?q))
from conjunct1[OF zlfm-I[OF qf, where bs=bs]]
have pp': ∀ i. ?I i ?p' = ?I i p by auto
from conjunct2[OF zlfm-I[OF qf, where bs=bs and i=i]]
have lp': iszlfm ?p' .
from lp' ζ[where p=?p'] have lp: ?l > 0 and dl: dβ ?p' ?l by auto
from aβ-ex[where p=?p' and l=?l and bs=bs, OF lp' dl lp] pp'
have pq-ex:(∃ (x::int). ?I x p) = (∃ x. ?I x ?q) by simp
from lp' lp aβ[OF lp' dl lp] have lq:iszlfm ?q and uq: dβ ?q 1 by auto
from δ[OF lq] have dp:?d > 0 and dd: dδ ?q ?d by blast+
let ?N = λ t. Inum (i#bs) t
have ?N ' set ?B' = ((?N o simpnum) ' ?B) by auto
also have ... = ?N ' ?B using simpnum-ci[where bs=i#bs] by auto
finally have BB': ?N ' set ?B' = ?N ' ?B .
have ?N ' set ?A' = ((?N o simpnum) ' ?A) by auto
also have ... = ?N ' ?A using simpnum-ci[where bs=i#bs] by auto
finally have AA': ?N ' set ?A' = ?N ' ?A .
from β-numbound0[OF lq] have B-nb:∀ b∈ set ?B'. numbound0 b
  by (simp add: simpnum-numbound0)
from α-l[OF lq] have A-nb:∀ b∈ set ?A'. numbound0 b
  by (simp add: simpnum-numbound0)
  {assume length ?B' ≤ length ?A'
   hence q:q=?q and B = ?B' and d:d = ?d
   using qBd by (auto simp add: Let-def unit-def)
   with BB' B-nb have b: ?N ' (set B) = ?N ' set (β q)
   and bn: ∀ b∈ set B. numbound0 b by simp+
with pq-ex dp uq dd lq q d have ?thes by simp}
moreover
  {assume ¬ (length ?B' ≤ length ?A')
   hence q:q=mirror ?q and B = ?A' and d:d = ?d
   using qBd by (auto simp add: Let-def unit-def)
   with AA' mirrorαβ[OF lq] A-nb have b: ?N ' (set B) = ?N ' set (β q)
   and bn: ∀ b∈ set B. numbound0 b by simp+
   from mirror-ex[OF lq] pq-ex q
   have pqm-eq:(∃ (x::int). ?I x p) = (∃ (x::int). ?I x q) by simp
   from lq uq q mirror-l[where p=?q]
   have lq': iszlfm q and uq: dβ q 1 by auto
   from δ[OF lq'] mirror-δ[OF lq] q d have dq:dδ q d by auto
   from pqm-eq b bn uq lq' dp dq q dp d have ?thes by simp
  }
ultimately show ?thes by blast
qed

```

```

constdefs cooper :: fm ⇒ fm
  cooper p ≡
    (let (q,B,d) = unit p; js = iupt 1 d;
     mq = simpfm (minusinf q);

```



```

      md = evaldjf (λ j. simpfm (subst0 (C j) mq)) js
in if md = T then T else
  (let qd = evaldjf (λ (b,j). simpfm (subst0 (Add b (C j)) q))
    [(b,j). b ← B, j ← js]
    in decr (disj md qd)))
lemma cooper: assumes qf: qfree p
shows ((∃ x. Ifm bbs (x#bs) p) = (Ifm bbs bs (cooper p))) ∧ qfree (cooper p)
(is (?lhs = ?rhs) ∧ -)
proof –
  let ?I = λ x p. Ifm bbs (x#bs) p
  let ?q = fst (unit p)
  let ?B = fst (snd (unit p))
  let ?d = snd (snd (unit p))
  let ?js = iupt 1 ?d
  let ?mq = minusinf ?q
  let ?smq = simpfm ?mq
  let ?md = evaldjf (λ j. simpfm (subst0 (C j) ?smq)) ?js
  fix i
  let ?N = λ t. Inum (i#bs) t
  let ?Bjs = [(b,j). b ← ?B, j ← ?js]
  let ?qd = evaldjf (λ (b,j). simpfm (subst0 (Add b (C j)) ?q)) ?Bjs
  have qbf: unit p = (?q, ?B, ?d) by simp
  from unit[OF qf qbf] have pq-ex: (∃ (x::int). ?I x p) = (∃ (x::int). ?I x ?q) and

    B: ?N ‘ set ?B = ?N ‘ set (β ?q) and
    uq: dβ ?q 1 and dd: dδ ?q ?d and dp: ?d > 0 and
    lq: iszlfm ?q and
    Bn: ∀ b ∈ set ?B. numbound0 b by auto
  from zlin-qfree[OF lq] have qfq: qfree ?q .
  from simpfm-qf[OF minusinf-qfree[OF qfq]] have qfmq: qfree ?smq.
  have jsnb: ∀ j ∈ set ?js. numbound0 (C j) by simp
  hence ∀ j ∈ set ?js. bound0 (subst0 (C j) ?smq)
    by (auto simp only: subst0-bound0[OF qfmq])
  hence th: ∀ j ∈ set ?js. bound0 (simpfm (subst0 (C j) ?smq))
    by (auto simp add: simpfm-bound0)
  from evaldjf-bound0[OF th] have mdb: bound0 ?md by simp
  from Bn jsnb have ∀ (b,j) ∈ set ?Bjs. numbound0 (Add b (C j))
    by simp
  hence ∀ (b,j) ∈ set ?Bjs. bound0 (subst0 (Add b (C j)) ?q)
    using subst0-bound0[OF qfq] by blast
  hence ∀ (b,j) ∈ set ?Bjs. bound0 (simpfm (subst0 (Add b (C j)) ?q))
    using simpfm-bound0 by blast
  hence th': ∀ x ∈ set ?Bjs. bound0 ((λ (b,j). simpfm (subst0 (Add b (C j)) ?q))
x)
    by auto
  from evaldjf-bound0 [OF th'] have qdb: bound0 ?qd by simp
  from mdb qdb
  have mdqdb: bound0 (disj ?md ?qd) by (simp only: disj-def, cases ?md=T ∨
?qd=T, simp-all)

```

from *trans* [*OF pq-ex cp-thm'*[*OF lq uq dd dp,where i=i*]] *B*
have *?lhs* = ($\exists j \in \{1.. ?d\}. ?I j ?mq \vee (\exists b \in ?N ' set ?B. Ifm bbs ((b + j) \# bs) ?q)$) **by** *auto*
also have ... = ($\exists j \in \{1.. ?d\}. ?I j ?mq \vee (\exists b \in set ?B. Ifm bbs ((?N b + j) \# bs) ?q)$) **by** *simp*
also have ... = ($(\exists j \in \{1.. ?d\}. ?I j ?mq) \vee (\exists j \in \{1.. ?d\}. \exists b \in set ?B. Ifm bbs ((?N (Add b (C j))) \# bs) ?q)$) **by** (*simp only: Inum.simps*) *blast*
also have ... = ($(\exists j \in \{1.. ?d\}. ?I j ?smq) \vee (\exists j \in \{1.. ?d\}. \exists b \in set ?B. Ifm bbs ((?N (Add b (C j))) \# bs) ?q)$) **by** (*simp add: simpfm*)
also have ... = ($(\exists j \in set ?js. (\lambda j. ?I i (simpfm (subst0 (C j) ?smq))) j) \vee (\exists j \in set ?js. \exists b \in set ?B. Ifm bbs ((?N (Add b (C j))) \# bs) ?q)$)
by (*simp only: simpfm subst0-I[OF qfmq] iupt-set*) *auto*
also have ... = ($?I i (evaldjf (\lambda j. simpfm (subst0 (C j) ?smq)) ?js) \vee (\exists j \in set ?js. \exists b \in set ?B. ?I i (subst0 (Add b (C j)) ?q))$)
by (*simp only: evaldjf-ex subst0-I[OF qfq]*)
also have ... = ($?I i ?md \vee (\exists (b,j) \in set ?Bjs. (\lambda (b,j). ?I i (simpfm (subst0 (Add b (C j)) ?q))) (b,j))$)
by (*simp only: simpfm set-concat set-map concat-map-singleton UN-simps*) *blast*
also have ... = ($?I i ?md \vee (?I i (evaldjf (\lambda (b,j). simpfm (subst0 (Add b (C j)) ?q)) ?Bjs))$)
by (*simp only: evaldjf-ex[where bs=i#bs and f=\lambda (b,j). simpfm (subst0 (Add b (C j)) ?q) and ps=?Bjs]*) (*auto simp add: split-def*)
finally have *mdqd: ?lhs* = ($?I i ?md \vee ?I i ?qd$) **by** *simp*
also have ... = ($?I i (disj ?md ?qd)$) **by** (*simp add: disj*)
also have ... = (*Ifm bbs bs (decr (disj ?md ?qd))*) **by** (*simp only: decr [OF mdqdb]*)
finally have *mdqd2: ?lhs* = (*Ifm bbs bs (decr (disj ?md ?qd))*) .
{assume *mdT: ?md = T*
hence *cT:cooper p = T*
by (*simp only: cooper-def unit-def split-def Let-def if-True*) *simp*
from *mdT* **have** *lhs: ?lhs* **using** *mdqd* **by** *simp*
from *mdT* **have** *?rhs* **by** (*simp add: cooper-def unit-def split-def*)
with *lhs cT* **have** *?thesis* **by** *simp* }
moreover
{assume *mdT: ?md \neq T* **hence** *cooper p = decr (disj ?md ?qd)*
by (*simp only: cooper-def unit-def split-def Let-def if-False*)
with *mdqd2 decr-qf[OF mdqdb]* **have** *?thesis* **by** *simp* }
ultimately show *?thesis* **by** *blast*
qed

constdefs *pa:: fm \Rightarrow fm*
pa $\equiv (\lambda p. qelim (prep p) cooper)$

theorem *mirqe: (Ifm bbs bs (pa p) = Ifm bbs bs p) \wedge qfree (pa p)*
using *qelim-ci cooper prep* **by** (*auto simp add: pa-def*)

definition

cooper-test :: unit \Rightarrow fm
where

```

cooper-test u = pa (E (A (Imp (Ge (Sub (Bound 0) (Bound 1))))
  (E (E (Eq (Sub (Add (Mul 3 (Bound 1)) (Mul 5 (Bound 0)))
    (Bound 2))))))))))

```

```

code-reserved SML oo
export-code pa cooper-test in SML module-name GeneratedCooper

```

```

ML << GeneratedCooper.cooper-test () >>
use coopereif.ML
oracle linzqe-oracle (term) = Coopereif.cooper-oracle
use coopertac.ML
setup LinZTac.setup

```

```

lemma  $\exists (j::int). \forall x \geq j. (\exists a b. x = 3*a + 5*b)$ 
by cooper

```

```

lemma ALL (x::int) >= 8. EX i j. 5*i + 3*j = x by cooper
theorem  $(\forall (y::int). 3 \text{ dvd } y) \implies \forall (x::int). b < x \longrightarrow a \leq x$ 
by cooper

```

```

theorem !! (y::int) (z::int) (n::int). 3 dvd z ==> 2 dvd (y::int) ==>
  ( $\exists (x::int). 2*x = y$ ) & ( $\exists (k::int). 3*k = z$ )
by cooper

```

```

theorem !! (y::int) (z::int) n. Suc(n::nat) < 6 ==> 3 dvd z ==>
  2 dvd (y::int) ==> ( $\exists (x::int). 2*x = y$ ) & ( $\exists (k::int). 3*k = z$ )
by cooper

```

```

theorem  $\forall (x::nat). \exists (y::nat). (0::nat) \leq 5 \longrightarrow y = 5 + x$ 
by cooper

```

```

lemma ALL (x::int) >= 8. EX i j. 5*i + 3*j = x by cooper
lemma ALL (y::int) (z::int) (n::int). 3 dvd z --> 2 dvd (y::int) --> (EX
  (x::int). 2*x = y) & (EX (k::int). 3*k = z) by cooper
lemma ALL(x::int) y. x < y --> 2 * x + 1 < 2 * y by cooper
lemma ALL(x::int) y. 2 * x + 1 ~ 2 * y by cooper
lemma EX(x::int) y. 0 < x & 0 <= y & 3 * x - 5 * y = 1 by cooper
lemma ~ (EX(x::int) (y::int) (z::int). 4*x + (-6::int)*y = 1) by cooper
lemma ALL(x::int). (2 dvd x) --> (EX(y::int). x = 2*y) by cooper
lemma ALL(x::int). (2 dvd x) --> (EX(y::int). x = 2*y) by cooper
lemma ALL(x::int). (2 dvd x) = (EX(y::int). x = 2*y) by cooper
lemma ALL(x::int). ((2 dvd x) = (ALL(y::int). x ~ 2*y + 1)) by cooper
lemma ~ (ALL(x::int). ((2 dvd x) = (ALL(y::int). x ~ 2*y + 1) | (EX(q::int)
  (u::int) i. 3*i + 2*q - u < 17) --> 0 < x | ((~ 3 dvd x) & (x + 8 = 0))))) by
  cooper
lemma ~ (ALL(i::int). 4 <= i --> (EX x y. 0 <= x & 0 <= y & 3 * x + 5
  * y = i))

```

by cooper
lemma $EX\ j. ALL\ (x::int)\ >= j. EX\ i\ j. 5*i + 3*j = x$ **by cooper**

theorem $(\forall (y::int). 3\ dvd\ y) ==> \forall (x::int). b < x \dashv\vdash a \leq x$
by cooper

theorem $!! (y::int)\ (z::int)\ (n::int). 3\ dvd\ z ==> 2\ dvd\ (y::int) ==>$
 $(\exists (x::int). 2*x = y) \ \&\ (\exists (k::int). 3*k = z)$
by cooper

theorem $!! (y::int)\ (z::int)\ n. Suc(n::nat) < 6 ==> 3\ dvd\ z ==>$
 $2\ dvd\ (y::int) ==> (\exists (x::int). 2*x = y) \ \&\ (\exists (k::int). 3*k = z)$
by cooper

theorem $\forall (x::nat). \exists (y::nat). (0::nat) \leq 5 \dashv\vdash y = 5 + x$
by cooper

theorem $\forall (x::nat). \exists (y::nat). y = 5 + x \mid x\ div\ 6 + 1 = 2$
by cooper

theorem $\exists (x::int). 0 < x$
by cooper

theorem $\forall (x::int)\ y. x < y \dashv\vdash 2 * x + 1 < 2 * y$
by cooper

theorem $\forall (x::int)\ y. 2 * x + 1 \neq 2 * y$
by cooper

theorem $\exists (x::int)\ y. 0 < x \ \&\ 0 \leq y \ \&\ 3 * x - 5 * y = 1$
by cooper

theorem $\sim (\exists (x::int)\ (y::int)\ (z::int). 4*x + (-6::int)*y = 1)$
by cooper

theorem $\sim (\exists (x::int). False)$
by cooper

theorem $\forall (x::int). (2\ dvd\ x) \dashv\vdash (\exists (y::int). x = 2*y)$
by cooper

theorem $\forall (x::int). (2\ dvd\ x) \dashv\vdash (\exists (y::int). x = 2*y)$
by cooper

theorem $\forall (x::int). (2\ dvd\ x) = (\exists (y::int). x = 2*y)$
by cooper

theorem $\forall (x::int). ((2\ dvd\ x) = (\forall (y::int). x \neq 2*y + 1))$
by cooper

```

theorem ~ (∀ (x::int).
  ((2 dvd x) = (∀ (y::int). x ≠ 2*y+1) |
  (∃ (q::int) (u::int) i. 3*i + 2*q - u < 17)
  --> 0 < x | ((~ 3 dvd x) & (x + 8 = 0))))
by cooper

theorem ~ (∀ (i::int). 4 ≤ i --> (∃ x y. 0 ≤ x & 0 ≤ y & 3 * x + 5 * y = i))
by cooper

theorem ∀ (i::int). 8 ≤ i --> (∃ x y. 0 ≤ x & 0 ≤ y & 3 * x + 5 * y = i)
by cooper

theorem ∃ (j::int). ∀ i. j ≤ i --> (∃ x y. 0 ≤ x & 0 ≤ y & 3 * x + 5 * y = i)
by cooper

theorem ~ (∀ j (i::int). j ≤ i --> (∃ x y. 0 ≤ x & 0 ≤ y & 3 * x + 5 * y = i))
by cooper

theorem (∃ m::nat. n = 2 * m) --> (n + 1) div 2 = n div 2
by cooper

end

```

45 Generic reflection and reification

```

theory Reflection
imports Main
  uses reflection-data.ML (reflection.ML)
begin

setup << Reify-Data.setup>>

lemma ext2: (∀ x. f x = g x) ==> f = g
by (blast intro: ext)

use reflection.ML

method-setup reify = <<
  fn src =>
    Method.syntax (Attrib.thms --
      Scan.option (Scan.lift (Args.$$$ () |-- Args.term --| Scan.lift (Args.$$$ ))
    ) src #>
    (fn ((eqs, to), ctxt) => Method.SIMPLE-METHOD' (Reflection.genreify-tac ctxt
      (eqs @ (fst (Reify-Data.get ctxt)))) to))
  >> 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);
    fun optional scan = Scan.optional scan [];
  in
    fn src =>
      Method.syntax (thms -- optional (keyword rulesN |-- thms) -- Scan.option
        (keyword onlyN |-- Args.term)) src #>
        (fn (((eqs,ths),to), ctxt) =>
          let
            val (ceqs,cths) = Reify-Data.get ctxt
            val corr-thms = ths@cths
            val raw-egs = eqs@ceqs
          in Method.SIMPLE-METHOD' (Reflection.reflection-tac ctxt corr-thms raw-egs
            to)
          end) end
    >> reflection method
  end

```

46 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 (term) = 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 = ref V-a

*val pairs = 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 HOL.plus}, -)) \$ x \$ y) = c \$ (rat x) \$ (rat

y)

| rat ((c as Const(@{const-name HOL.minus}, -)) \$ x \$ y) = c \$ (rat x) \$

(rat y)

| rat ((c as Const(@{const-name HOL.divide}, -)) \$ x \$ y) = c \$ (rat x) \$

(rat y)

| rat ((c as Const(@{const-name HOL.times}, -)) \$ x \$ y) = c \$ (rat x) \$

(rat y)

| rat ((c as Const(@{const-name HOL.uminus}, -)) \$ x) = c \$ (rat x)

| rat t = lit t

*(*abstraction of an integer expression: no div, mod*)*

fun int ((c as Const(@{const-name HOL.plus}, -)) \$ x \$ y) = c \$ (int x) \$ (int

```

y)
  | int ((c as Const(@{const-name HOL.minus}, -)) $ x $ y) = c $ (int x) $
(int y)
  | int ((c as Const(@{const-name HOL.times}, -)) $ x $ y) = c $ (int x) $
(int y)
  | int ((c as Const(@{const-name HOL.uminus}, -)) $ x) = c $ (int x)
  | int t = lit t
(*abstraction of a natural number expression: no minus*)
fun nat ((c as Const(@{const-name HOL.plus}, -)) $ x $ y) = c $ (nat x) $
(nat y)
  | nat ((c as Const(@{const-name HOL.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 HOL.less}, Type (fun, [T,-])) $ - $ -) = rel
(T, t)
  | fm (t as Const(@{const-name HOL.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.*)

```

fun svc-tac i st =
  let
    val (abs-goal, -) = svc-abstract (Logic.get-goal (Thm.prop-of st) i)
    val th = svc-oracle (Thm.theory-of-thm st) abs-goal

```



```

    in compose-tac (false, th, 0) i st end
    handle TERM - => no-tac st;

(*check if user has SVC installed*)
fun svc-enabled () = getenv SVC-HOME <> ;
fun if-svc-enabled f x = if svc-enabled () then f x else ();
>>

end

```

47 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

```

47.1 Examples and Test Cases

47.1.1 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
```

47.1.2 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
```

47.1.3 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⇒'b) = g
  refute
oops

```

```

lemma (f::('d⇒'d)⇒('c⇒'d)) = g
  refute
oops

```

```

lemma distinct [a,b]
  refute
  apply simp
  refute
oops

```

47.1.4 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 $(\exists x. P\ x) \longrightarrow (EX!\ x. P\ x)$
 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 $\llbracket \forall x. P\ x\ x; \forall x\ y. P\ x\ y \longrightarrow P\ y\ x \rrbracket \Longrightarrow P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z$
 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⇒'b)⇒bool) f = P g) ⟶ (f x = g x)
  refute
oops
```

47.1.5 Higher-Order Logic

```
lemma ∃ P. P
  refute
  apply auto
done
```

```
lemma ∀ P. P
  refute
oops
```

```
lemma EX! P. P
  refute
  apply auto
done
```

```
lemma EX! P. P x
  refute
oops
```

```
lemma P Q | Q x
  refute
oops
```

```
lemma x ≠ All
  refute
oops
```

```
lemma x ≠ Ex
  refute
oops
```

```
lemma x ≠ Ex1
  refute
oops
```

”The transitive closure 'T' of an arbitrary relation 'P' is non-empty.”

```
constdefs
  trans :: ('a ⇒ 'a ⇒ bool) ⇒ bool
  trans P == (ALL x y z. P x y ⟶ P y z ⟶ P x z)
  subset :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ bool
  subset P Q == (ALL x y. P x y ⟶ Q x y)
  trans-closure :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'a ⇒ bool) ⇒ bool
```

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 $(\forall x \ y. \ (P \ x \ y \mid R \ x \ y) \longrightarrow T \ x \ y) \longrightarrow trans \ T \longrightarrow (\forall Q. \ (\forall x \ y. \ (P \ x \ y \mid R \ x \ y) \longrightarrow Q \ x \ y) \longrightarrow trans \ Q \longrightarrow subset \ T \ Q)$
 $\longrightarrow trans-closure \ TP \ P$
 $\longrightarrow trans-closure \ TR \ R$
 $\longrightarrow (T \ x \ y = (TP \ x \ y \mid TR \ x \ y))$
refute
oops

”Every surjective function is invertible.”

lemma $(\forall y. \ \exists x. \ y = f \ x) \longrightarrow (\exists g. \ \forall x. \ g \ (f \ x) = x)$
refute
oops

”Every invertible function is surjective.”

lemma $(\exists g. \ \forall x. \ g \ (f \ x) = x) \longrightarrow (\forall y. \ \exists x. \ y = f \ x)$
refute
oops

Every point is a fixed point of some function.

lemma $\exists f. \ f \ x = x$
refute [*maxsize=4*]
apply (*rule-tac* $x=\lambda x. \ x$ **in** *exI*)
apply *simp*
done

Axiom of Choice: first an incorrect version ...

lemma $(\forall x. \ \exists y. \ P \ x \ y) \longrightarrow (EX!f. \ \forall x. \ P \ x \ (f \ x))$
refute
oops

... and now two correct ones

lemma $(\forall x. \ \exists y. \ P \ x \ y) \longrightarrow (\exists f. \ \forall x. \ P \ x \ (f \ x))$
refute [*maxsize=4*]
apply (*simp* *add: choice*)
done

lemma $(\forall x. \ EX!y. \ P \ x \ y) \longrightarrow (EX!f. \ \forall x. \ P \ x \ (f \ x))$
refute [*maxsize=2*]
apply *auto*
apply (*simp* *add: ex1-implies-ex choice*)

```

    apply (fast intro: ext)
done

```

47.1.6 Meta-logic

```

lemma !!x. P x
  refute
oops

```

```

lemma f x == g x
  refute
oops

```

```

lemma P ==> Q
  refute
oops

```

```

lemma [[ P; Q; R ]] ==> S
  refute
oops

```

```

lemma (x == all) ==> False
  refute
oops

```

```

lemma (x == (op ==)) ==> False
  refute
oops

```

```

lemma (x == (op ==>)) ==> False
  refute
oops

```

47.1.7 Schematic variables

```

lemma ?P
  refute
  apply auto
done

```

```

lemma x = ?y
  refute
  apply auto
done

```

47.1.8 Abstractions

```

lemma (λx. x) = (λx. y)
  refute
oops

```

```

lemma ( $\lambda f. f\ x$ ) = ( $\lambda f. True$ )
  refute
oops

```

```

lemma ( $\lambda x. x$ ) = ( $\lambda y. y$ )
  refute
  apply simp
done

```

47.1.9 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 \longrightarrow Bex A P*
 refute
 oops

47.1.10 arbitrary

lemma *arbitrary*
 refute
 oops

lemma *P arbitrary*
 refute
 oops

lemma *arbitrary x*
 refute
 oops

lemma *arbitrary arbitrary*
 refute
 oops

47.1.11 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

47.1.12 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

```

47.1.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of 'a:

```

typedef 'a myTdef = insert (arbitrary::'a) (arbitrary::'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

```

47.1.14 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  $P\ (case\ x\ of\ Inl\ a \Rightarrow l\ a \mid Inr\ b \Rightarrow r\ b)$ 
  refute
oops

Non-recursive datatypes

datatype  $T1 = A \mid B$ 

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 ⇒ a | B ⇒ b)
  refute
oops

datatype 'a T2 = C T1 | D 'a

lemma P (x::'a T2)
  refute
oops

lemma ∀ 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  $C\ u \Rightarrow c\ u \mid D\ v \Rightarrow d\ v$ )
  refute
oops

```

```

datatype ( $'a, 'b$ )  $T\mathcal{I} = E\ 'a \Rightarrow 'b$ 

```

```

lemma  $P$  ( $x::('a, 'b)\ T\mathcal{I}$ )
  refute
oops

```

```

lemma  $\forall x::('a, 'b)\ T\mathcal{I}. P\ x$ 
  refute
oops

```

```

lemma  $P\ E$ 
  refute
oops

```

```

lemma  $T\mathcal{I}\text{-rec}\ e\ (E\ x) = e\ x$ 
  refute [maxsize=2]
  apply simp
done

```

```

lemma  $P\ (T\mathcal{I}\text{-rec}\ e\ x)$ 
  refute
oops

```

```

lemma  $P$  (case  $x$  of  $E\ f \Rightarrow e\ f$ )
  refute
oops

```

Recursive datatypes

nat

```

lemma  $P\ (x::\text{nat})$ 
  refute
oops

```

```

lemma  $\forall x::\text{nat}. P\ x$ 
  refute
oops

```

```

lemma  $P\ (\text{Suc}\ 0)$ 
  refute
oops

```

```

lemma  $P\ \text{Suc}$ 
  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-rec zero suc 0 = zero
  refute
  apply simp
done

lemma nat-rec zero suc (Suc x) = suc x (nat-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  $\Rightarrow$  zero | Suc n  $\Rightarrow$  suc n)
  refute
oops

'a list

lemma P (xs::'a list)
  refute
oops

lemma  $\forall$  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  $\Rightarrow$  nil | Cons a b  $\Rightarrow$  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  $P \ (case \ x \ of \ Leaf \ a \Rightarrow \ l \ a \mid Node \ a \ b \Rightarrow \ n \ a \ b)$ 
  refute
oops

```

Mutually recursive datatypes

```

datatype  $'a \ aexp = Number \ 'a \mid ITE \ 'a \ bexp \ 'a \ aexp \ 'a \ aexp$ 
  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 $X\text{-}Y\text{-rec-1 } a b c d e f A = a$
 refute $[maxsize=3]$
 apply *simp*
 done

lemma $X\text{-}Y\text{-rec-1 } a b c d e f (B x) = b x (X\text{-}Y\text{-rec-1 } a b c d e f x)$
 refute $[maxsize=1]$
 apply *simp*
 done

lemma $X\text{-}Y\text{-rec-1 } a b c d e f (C y) = c y (X\text{-}Y\text{-rec-2 } a b c d e f y)$
 refute $[maxsize=1]$
 apply *simp*
 done

lemma $X\text{-}Y\text{-rec-2 } a b c d e f (D x) = d x (X\text{-}Y\text{-rec-1 } a b c d e f x)$

```

    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  $\Rightarrow$  'a YOpt) option

lemma P (x::'a YOpt)
  refute
oops

lemma P (CY None)
  refute
oops

lemma P (CY (Some ( $\lambda a.$  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 ( $\lambda a.$  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  $\forall x::Trie.$  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 ⇒ InfTree

lemma P (x::InfTree)
  refute
oops

lemma ∀ x::InfTree. P x
  refute
oops

lemma P (Node (λn. 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 (λn. InfTree-rec leaf node (x n))
  refute [maxsize=1]
  apply simp
done

lemma P (InfTree-rec leaf node x)
  refute
oops

datatype 'a lambda = Var 'a | App 'a lambda 'a lambda | Lam 'a ⇒ 'a lambda

lemma P (x::'a lambda)
  refute
oops

lemma ∀ x::'a lambda. P x
  refute
oops

```



```

lemma P (Lam ( $\lambda a.$  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

47.1.15 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

47.1.16 Inductively defined sets

inductive-set *arbitrarySet* :: 'a set

where

arbitrary : *arbitrarySet*

lemma *x* : *arbitrarySet*

refute

oops

inductive-set *evenCard* :: 'a set set

where

$\{\}$: *evenCard*

| $\llbracket S : \text{evenCard}; x \notin S; y \notin S; x \neq y \rrbracket \implies S \cup \{x, y\} : \text{evenCard}$

lemma *S* : *evenCard*

refute

oops

inductive-set

even :: nat set

and *odd* :: nat set

where

0 : *even*

| *n* : *even* \implies *Suc n* : *odd*

| *n* : *odd* \implies *Suc n* : *even*

lemma *n* : *odd*

oops

consts *f* :: 'a \Rightarrow 'a

inductive-set

a-even :: 'a set

and *a-odd* :: 'a set

where

arbitrary : *a-even*

| *x* : *a-even* \implies *f x* : *a-odd*

| *x* : *a-odd* \implies *f x* : *a-even*

lemma *x* : *a-odd*

refute — finds a model of size 2, as expected

oops

47.1.17 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
```

47.1.18 Axiomatic type classes and overloading

A type class without axioms:

```
axclass classA
```

```

lemma  $P (x::'a::classA)$ 
  refute
oops

```

The axiom of this type class does not contain any type variables:

```

axclass  $classB$ 
   $classB\text{-}ax: P \mid \sim P$ 

```

```

lemma  $P (x::'a::classB)$ 
  refute
oops

```

An axiom with a type variable (denoting types which have at least two elements):

```

axclass  $classC < type$ 
   $classC\text{-}ax: \exists x y. x \neq y$ 

```

```

lemma  $P (x::'a::classC)$ 
  refute
oops

```

```

lemma  $\exists x y. (x::'a::classC) \neq y$ 
  refute — no countermodel exists
oops

```

A type class for which a constant is defined:

```

consts
   $classD\text{-}const :: 'a \Rightarrow 'a$ 

```

```

axclass  $classD < type$ 
   $classD\text{-}ax: classD\text{-}const (classD\text{-}const x) = classD\text{-}const x$ 

```

```

lemma  $P (x::'a::classD)$ 
  refute
oops

```

A type class with multiple superclasses:

```

axclass  $classE < classC, classD$ 

```

```

lemma  $P (x::'a::classE)$ 
  refute
oops

```

```

lemma  $P (x::'a::\{classB, classE\})$ 
  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, classB-class)
  refute — no countermodel exists
  apply intro-classes
  apply simp
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 $\times$ 'b))
  refute
oops

refute-params [satsolver=auto]

end

```

48 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.

48.1 Lists

theorem $\text{map } g \ (\text{map } f \ xs) = \text{map } (g \ o \ f) \ xs$
quickcheck
oops

theorem $\text{map } g \ (\text{map } f \ xs) = \text{map } (f \ o \ g) \ xs$
quickcheck
oops

theorem $\text{rev } (xs \ @ \ ys) = \text{rev } ys \ @ \ \text{rev } xs$
quickcheck
oops

theorem $\text{rev } (xs \ @ \ ys) = \text{rev } xs \ @ \ \text{rev } ys$
quickcheck
oops

theorem $\text{rev } (\text{rev } xs) = xs$
quickcheck
oops

theorem $\text{rev } xs = xs$
quickcheck
oops

An example involving functions inside other data structures

consts $\text{app} :: ('a \Rightarrow 'a) \ \text{list} \Rightarrow 'a \Rightarrow 'a$

primrec
 $\text{app } [] \ x = x$
 $\text{app } (f \ \# \ fs) \ x = \text{app } fs \ (f \ x)$

lemma $\text{app } (fs \ @ \ gs) \ x = \text{app } gs \ (\text{app } fs \ x)$
quickcheck
by $(\text{induct } fs \ \text{arbitrary: } x) \ \text{simp-all}$

lemma $\text{app } (fs \ @ \ gs) \ x = \text{app } fs \ (\text{app } gs \ x)$
quickcheck
oops

consts
 $\text{occurs} :: 'a \Rightarrow 'a \ \text{list} \Rightarrow \text{nat}$

primrec
 $\text{occurs } a \ [] = 0$

$occurs\ a\ (x\#xs) = (if\ (x=a)\ then\ Suc(occurs\ a\ xs)\ else\ occurs\ a\ xs)$

consts

$del1 :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$

primrec

$del1\ a\ [] = []$

$del1\ a\ (x\#xs) = (if\ (x=a)\ then\ xs\ else\ (x\#del1\ a\ xs))$

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

apply $(induct-tac\ xs)$

apply $auto$

— Correct!

done

consts

$replace :: 'a \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$

primrec

$replace\ a\ b\ [] = []$

$replace\ a\ b\ (x\#xs) = (if\ (x=a)\ then\ (b\#(replace\ a\ b\ xs))\ else\ (x\#(replace\ a\ b\ xs)))$

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

apply $(induct-tac\ xs)$

apply $auto$

done

48.2 Trees

datatype $'a\ tree = Twig\ |\ Leaf\ 'a\ |\ Branch\ 'a\ tree\ 'a\ tree$

consts


```

    leaves :: 'a tree  $\Rightarrow$  'a list
primrec
    leaves Twig = []
    leaves (Leaf a) = [a]
    leaves (Branch l r) = (leaves l) @ (leaves r)

consts
    plant :: 'a list  $\Rightarrow$  'a tree
primrec
    plant [] = Twig
    plant (x#xs) = Branch (Leaf x) (plant xs)

consts
    mirror :: 'a tree  $\Rightarrow$  'a tree
primrec
    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

consts
    inOrder :: 'a ntree  $\Rightarrow$  'a list
primrec
    inOrder (Tip a) = [a]
    inOrder (Node f x y) = (inOrder x)@[f]@(inOrder y)

consts
    root :: 'a ntree  $\Rightarrow$  'a
primrec
    root (Tip a) = a
    root (Node f x y) = f

theorem hd(inOrder xt) = root xt
quickcheck
    — Wrong!
oops

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] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.
- [3] K. McMillan. Lecture notes on verification of digital and hybrid systems. NATO summer school, <http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial/toc.html>.
- [4] K. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.
- [5] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *LNCS*, 1998.
- [6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.
- [7] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
- [8] L. C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.
- [9] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.
- [10] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, September 2001. Submitted.
- [11] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2001. Part of the Isabelle distribution, <http://isabelle.in.tum.de/doc/isar-ref.pdf>.

- [12] M. Wenzel. Miscellaneous Isabelle/Isar examples for higher-order logic. Part of the Isabelle distribution, http://isabelle.in.tum.de/library/HOL/Isar_examples/document.pdf, 2001.