Miscellaneous HOL Examples

November 22, 2007

Contents

1 Foundations of HOL

theory *Higher-Order-Logic* **imports** *CPure* **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*

typedecl *o*
arities
  o :: *type*
  fun :: $(type, type)$ *type*

1.1.1 Basic logical connectives

judgment
  $Trueprop :: o \Rightarrow prop \quad (\text{-}\ 5)$

axiomatization
  $imp :: o \Rightarrow o \Rightarrow o \quad$ (**infixr** $\longrightarrow$ *25*) **and**
  $All :: ('a \Rightarrow o) \Rightarrow o \quad$ (**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
  $equal :: 'a \Rightarrow 'a \Rightarrow o \quad$ (**infixl** $=$ *50*)
where
  $refl\ [intro]: x = x$ **and**

subst: $x = y \Longrightarrow P\ x \Longrightarrow P\ y$

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

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

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

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

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

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

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

1.1.3 Derived connectives

definition
false :: o $(\bot)$ **where**
$\bot \equiv \forall A.\ A$

definition
true :: o $(\top)$ **where**
$\top \equiv \bot \longrightarrow \bot$

definition
not :: $o \Rightarrow o$ ($\neg$ - [*40*] *40*) **where**
$not \equiv \lambda A.\ A \longrightarrow \bot$

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

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

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

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

theorem *falseE* [*elim*]: $\bot \Longrightarrow A$
proof (*unfold false-def*)
  assume $\forall A.\ A$
  then show A **..**
qed

theorem *trueI* [*intro*]: $\top$
proof (*unfold true-def*)
  show $\bot \longrightarrow \bot$ **..**
qed

theorem *notI* [*intro*]: $(A \Longrightarrow \bot) \Longrightarrow \neg\ A$
proof (*unfold not-def*)
  assume $A \Longrightarrow \bot$
  then show $A \longrightarrow \bot$ **..**
qed

theorem *notE* [*elim*]: $\neg\ A \Longrightarrow A \Longrightarrow B$
proof (*unfold not-def*)
  assume $A \longrightarrow \bot$
  also assume A
  finally have $\bot$ **..**
  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*]: $A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
proof (*unfold conj-def*)
assume c: $\forall C. (A \longrightarrow B \longrightarrow C) \longrightarrow C$
assume $A \Longrightarrow B \Longrightarrow C$
moreover {
from c **have** $(A \longrightarrow B \longrightarrow A) \longrightarrow A$ **..**
also have $A \longrightarrow B \longrightarrow A$
proof
assume A
then show $B \longrightarrow A$ **..**
qed
finally have A **.**
} moreover {
from c **have** $(A \longrightarrow B \longrightarrow B) \longrightarrow B$ **..**
also have $A \longrightarrow B \longrightarrow B$
proof
show $B \longrightarrow B$ **..**
qed
finally have B **.**
} ultimately show C **.**
qed

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

theorem *disjI2* [*intro*]: $B \Longrightarrow A \vee B$
proof (*unfold disj-def*)
assume B
show $\forall C. (A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$
proof
fix C **show** $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$
proof
show $(B \longrightarrow C) \longrightarrow C$

    proof
      assume $B \longrightarrow C$
      also note $\langle B\rangle$
      finally show C **.**
    qed
  qed
 qed
qed

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

theorem *exI* [*intro*]: $P\ a \Longrightarrow \exists x.\ P\ x$
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*]: $\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$
proof (*unfold Ex-def*)
  assume *c*: $\forall C. (\forall x.\ P\ x \longrightarrow C) \longrightarrow C$
  assume *r*: $\bigwedge x.\ P\ x \Longrightarrow C$
  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*: $(\neg\ A \Longrightarrow A) \Longrightarrow A$

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: $\neg\ \neg\ A \Longrightarrow A$
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 $(rule\ double\text{-}negation)$
  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** $\bot$ **by** $(rule\ contradiction)$

```
    qed
    then have A ∨ ¬ A ..
    with ‹¬ (A ∨ ¬ A)› show ⊥ by (rule contradiction)
  qed
qed

theorem (in classical)
  classical-cases: (A ⟹ C) ⟹ (¬ A ⟹ C) ⟹ C
proof −
  assume r1: A ⟹ C and r2: ¬ A ⟹ C
  from tertium-non-datur show C
  proof
    assume A
    then show ?thesis by (rule r1)
  next
    assume ¬ A
    then show ?thesis by (rule r2)
  qed
qed

lemma (in classical) (¬ A ⟹ A) ⟹ A
proof −
  assume r: ¬ A ⟹ A
  show A
  proof (rule classical-cases)
    assume A then show A .
  next
    assume ¬ 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 ⇒ 'n
  assumes succ-inject [simp]: (succ m = succ n) = (m = n)
    and succ-neq-zero [simp]: succ m ≠ zero
    and induct [case-names zero succ, induct type: 'n]:
      P zero ⟹ (⋀n. P n ⟹ P (succ n)) ⟹ P n
begin
```

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

Primitive recursion as a (functional) relation – polymorphic!

inductive
  Rec :: $'a \Rightarrow ('n \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'n \Rightarrow 'a \Rightarrow$ *bool*
  for *e* :: $'a$ **and** *r* :: $'n \Rightarrow 'a \Rightarrow 'a$
where
    Rec-zero: *Rec e r zero e*
  | *Rec-succ*: *Rec e r m n* $\Longrightarrow$ *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 ‹$\exists!y.$ *?R m y*›
    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* ⇒ ′*n* ⇒ ′*n* **where**
add m n = *rec n* (λ- *k*. *succ k*) *m*

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

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

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

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

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

Example: replication (polymorphic)

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

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

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

end

Just see that our abstract specification makes sense ...

interpretation *NAT* [*0* *Suc*]
proof (*rule* *NAT.intro*)
  fix *m n*
  show (*Suc m* = *Suc n*) = (*m* = *n*) **by** *simp*
  show *Suc m* $\neq$ *0* **by** *simp*
  fix *P*
  assume *zero*: *P 0*
    and *succ*: $\bigwedge n.$ *P n* $\Longrightarrow$ *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 ⟪
```
structure Binary =
struct
  fun dest-bit (Const (False, -)) = 0
    | dest-bit (Const (True, -)) = 1
    | dest-bit t = raise TERM (dest-bit, [t]);

  fun dest-binary (Const (@{const-name HOL.zero}, Type (nat, -))) = 0
    | dest-binary (Const (@{const-name HOL.one}, Type (nat, -))) = 1
    | dest-binary (Const (Binary.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
```
⟫

4.2 Direct operations – plain normalization

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

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

lemma *binary-mult*:
$\quad$ $n * 0 = 0$
$\quad$ $0 * n = 0$
$\quad$ $n * 1 = n$
$\quad$ $1 * n = n$
$\quad$ $\mathit{bit}\ m\ \mathit{True} * n = \mathit{bit}\ (m * n)\ \mathit{False} + n$
$\quad$ $\mathit{bit}\ m\ \mathit{False} * n = \mathit{bit}\ (m * n)\ \mathit{False}$
$\quad$ $n * \mathit{bit}\ m\ \mathit{True} = \mathit{bit}\ (m * n)\ \mathit{False} + n$
$\quad$ $n * \mathit{bit}\ m\ \mathit{False} = \mathit{bit}\ (m * n)\ \mathit{False}$
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 $n :: \mathit{nat}$
shows $n \equiv m + k \Longrightarrow (m \le n) \equiv \mathit{True}$
and $m \equiv n + k + 1 \Longrightarrow (m \le n) \equiv \mathit{False}$
by *simp-all*

lemma *binary-less*:
fixes $n :: \mathit{nat}$
shows $m \equiv n + k \Longrightarrow (m < n) \equiv \mathit{False}$
and $n \equiv m + k + 1 \Longrightarrow (m < n) \equiv \mathit{True}$
by *simp-all*

lemma *binary-diff*:
fixes $n :: \mathit{nat}$

shows $m \equiv n + k \Longrightarrow m - n \equiv k$
and $n \equiv m + k \Longrightarrow m - n \equiv 0$
by *simp-all*

lemma *binary-divmod*:
fixes n :: *nat*
assumes $m \equiv n * k + l$ **and** $0 < n$ **and** $l < n$
shows m *div* $n \equiv k$
and m *mod* $n \equiv l$
proof −
from ‹$m \equiv n * k + l$› **have** $m = l + k * n$ **by** *simp*
with ‹$0 < n$› **and** ‹$l < n$› **show** m *div* $n \equiv k$ **and** m *mod* $n \equiv l$ **by** *simp-all*
qed

ML ⟪

```
local
  infix ==;
  val op == = Logic.mk-equals;
  fun plus m n = @{term plus :: nat ⇒ nat ⇒ nat} $ m $ n;
  fun mult m n = @{term times :: nat ⇒ nat ⇒ nat} $ m $ n;

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

  fun binary-proc proc ss ct =
    (case Thm.term-of ct of
      - $ t $ u =>
      (case try (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 >= 0 then
    SOME (@{thm binary-less-eq(1)} OF [prove ctxt (u == plus t (Binary.mk-binary
k))])
    else
      SOME (@{thm binary-less-eq(2)} OF
      [prove ctxt (t == plus (plus u (Binary.mk-binary (~ k − 1))) (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 >= 0 then
    SOME (@{thm binary-less(1)} OF [prove ctxt (t == plus u (Binary.mk-binary
```

k))])
  else
   SOME (@{*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* ⇒ *'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 $*1111111111222222222233333333334444444444* − $*998877665544332211* =
  $*1111111111222222222232334455668900112233*

by *binary-simp*

lemma (*1111111111222222222233333333334444444444*::*nat*) − *998877665544332211*
=
1111111111222222222232334455668900112233
by *simp*

lemma (*1111111111222222222233333333334444444444*::*int*) − *998877665544332211*
=
1111111111222222222232334455668900112233
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 $*1111111111222222222233333333334444444444 div* $*998877665544332211*
=
$*111235955067303370787 5*
by *binary-simp*

lemma $*1111111111222222222233333333334444444444 mod* $*998877665544332211*
=
$*422451743175828 19*
by *binary-simp*

lemma (*1111111111222222222233333333334444444444::int*) *div 998877665544332211*
=
1112359550673033707875
by *simp* — legacy numerals: 30 times slower

lemma (*1111111111222222222233333333334444444444::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* (λ*y*. *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 fqsort :: ('a => 'a => bool) * 'a list => 'a list
recdef (permissive) fqsort  measure (size o snd)
  fqsort (ord, []) = []
  fqsort (ord, x # rst) =
  (let (less, more) = part ((λy. ord y x), rst, ([], []))
  in fqsort (ord, less) @ [x] @ fqsort (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 $\lambda x.$ *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* ⇒ *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*
| *gcd2 (Suc x) (Suc y) = (if x < y then gcd2 (Suc x) (y − x)*
                                    else gcd2 (x − y) (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*
| *x < y* ⟹ *gcd3 (Suc x) (Suc y) = gcd3 (Suc x) (y − x)*
| ¬ *x < y* ⟹ *gcd3 (Suc x) (Suc y) = gcd3 (x − y) (Suc y)*
  apply (*case-tac x*, *case-tac a*, *auto*)
  apply (*case-tac ba*, *auto*)
  done
termination by *lexicographic-order*

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

General patterns allow even strange definitions:

function *ev* :: *nat* ⇒ *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* ⇒ *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* ⇒ *'a* ⇒ *'a*
  and *un* :: *'a*
assumes *assoc*: *opr* (*opr x y*) *z* = *opr x* (*opr y z*)
  and *lunit*: *opr un x* = *x*
  and *runit*: *opr x un* = *x*
begin

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

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

thm *foldL.simps*

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

thm *foldR-foldL*

end

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

6.7 Regression tests

The following examples mainly serve as tests for the function package

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

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

function *h* :: *nat* ⇒ *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* ⇒ *nat*
where
i 0 = *0*
| *i* (*Suc n*) = (*if n* = *0 then 0 else i n*)

fun *fa* :: *nat* ⇒ *nat* ⇒ *nat*
where
fa 0 y = *0*
| *fa* (*Suc n*) *y* = (*if fa n y* = *0 then 0 else fa n y*)

fun *j* :: *nat* ⇒ *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 $f2 :: (nat \times nat) \Rightarrow (nat \times nat)$
where
$f2\ p = (let\ (x,y) = p\ in\ f2\ (y,x))$
by *pat-completeness auto*

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

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

lemma $lem{:}x \in set\ l \Longrightarrow size\ x < Suc\ (tree\text{-}list\text{-}size\ l)$
by $(induct\ l,\ auto)$

function $treemap :: ('a \Rightarrow 'a) \Rightarrow 'a\ tree \Rightarrow 'a\ tree$
where
$treemap\ fn\ (Leaf\ n) = (Leaf\ (fn\ n))$
$|\ treemap\ fn\ (Branch\ l) = (Branch\ (map\ (treemap\ fn)\ l))$
by *pat-completeness auto*
termination by $(lexicographic\text{-}order\ simp{:}lem)$

declare $lem[simp]$

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

record $point =$
$Xcoord :: int$
$Ycoord :: int$

function $swp :: point \Rightarrow point$
where

```
  swp (| Xcoord = x, Ycoord = y |) = (| Xcoord = y, Ycoord = x |)
proof -
  fix P x
  assume ⋀xa y. x = (|Xcoord = xa, Ycoord = y|) ⟹ P
  thus P
    by (cases x)
qed auto
termination by rule auto



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



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

fun big :: DT ⇒ 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 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 $!!y.\ (y,x) \in r \Longrightarrow S\ y\ (cut\ (wfrec\ r\ F)\ r\ x\ y)$ **by** (*simp add*: *tfl-cut-apply*)
with *inv* **have** $S\ x\ (F\ (cut\ (wfrec\ r\ F)\ r\ x)\ x)$ **by** (*unfold indinv-def*, *blast*)
thus $S\ x\ (wfrec\ r\ F\ x)$ **using** *wf* **by** (*simp add*: *wfrec*)
qed

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

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

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

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

end

8 Example use if an inductive invariant to solve termination conditions

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

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

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

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

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

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

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

declare *g.simps* [*simplified*, *simp*]

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

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

thm *g′.simps*

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

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

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

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

thm *g′.simps* [*simplified*]

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

consts *g-even* :: *nat => nat*
recdef (**permissive**) *g-even less-than*
  g-even (*Suc* (*Suc 0*)) = *3*
  g-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*: ∃ *k*. *n* = *Suc* (*Suc* (*2∗k*)) ==> *g-even n* = *3*
apply (*rule-tac D*={*n*. ∃ *k*. *n* = *Suc* (*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*: ∃ *k*. *n* = *2∗k* + *4* ==> *g-even n* = *g-even* (*g-even* (*n* − *2*) − *1*)
apply (*subgoal-tac* (∃ *k*. *n* − *2* = *2∗k* + *2*) & (∃ *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 ¡ y + 11

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

Proving the termination condition using the strengthened inductive invariant.

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

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

theorem *def-ninety-one*:
ninety-one x = (*if* $100 < x$
then $x - 10$
else ninety-one (*ninety-one* $(x{+}11)$))
by (*subst ninety-one.simps*,
simp add: *ninety-one-tc*)

end

9 Using locales in Isabelle/Isar – outdated version!

theory *Locales* **imports** *Main* **begin**

9.1 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), $\Longrightarrow$ (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 ($\Longrightarrow$), 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 $\bigwedge/\Longrightarrow/\equiv$ 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 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 extend 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.

9.2 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*: $x^{-1} \cdot x = \mathbf{1}$
    and *left-one*: $\mathbf{1} \cdot x = x$

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: $x \cdot x^{-1} = \mathbf{1}$
proof −
  have $x \cdot x^{-1} = \mathbf{1} \cdot (x \cdot x^{-1})$ **by** (*simp only*: *left-one*)
  also have $\ldots = \mathbf{1} \cdot x \cdot x^{-1}$ **by** (*simp only*: *assoc*)
  also have $\ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1}$ **by** (*simp only*: *left-inv*)
  also have $\ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1}$ **by** (*simp only*: *assoc*)
  also have $\ldots = (x^{-1})^{-1} \cdot \mathbf{1} \cdot x^{-1}$ **by** (*simp only*: *left-inv*)
  also have $\ldots = (x^{-1})^{-1} \cdot (\mathbf{1} \cdot x^{-1})$ **by** (*simp only*: *assoc*)
  also have $\ldots = (x^{-1})^{-1} \cdot x^{-1}$ **by** (*simp only*: *left-one*)
  also have $\ldots = \mathbf{1}$ **by** (*simp only*: *left-inv*)
  finally show *?thesis* **.**
qed

theorem (**in** *group-context*)
  right-one: $x \cdot \mathbf{1} = x$
proof −
  have $x \cdot \mathbf{1} = x \cdot (x^{-1} \cdot x)$ **by** (*simp only*: *left-inv*)

also have $\ldots = x \cdot x^{-1} \cdot x$ **by** (*simp only*: *assoc*)
also have $\ldots = \mathbf{1} \cdot x$ **by** (*simp only*: *right-inv*)
also have $\ldots = x$ **by** (*simp only*: *left-one*)
finally show *?thesis* **.**
qed

Facts like *right-one* are available *group-context* as stated above. The exported version looses 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* $\Longrightarrow$ *?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*: $\mathbf{1} = e$
proof −
have $\mathbf{1} = x \cdot x^{-1}$ **by** (*simp only*: *right-inv*)
also have $\ldots = (e \cdot x) \cdot x^{-1}$ **by** (*simp only*: *eq*)
also have $\ldots = e \cdot (x \cdot x^{-1})$ **by** (*simp only*: *assoc*)
also have $\ldots = e \cdot \mathbf{1}$ **by** (*simp only*: *right-inv*)
also have $\ldots = e$ **by** (*simp only*: *right-one*)
finally show *?thesis* **.**
qed

theorem (**in** *group-context*)
assumes *eq*: $x' \cdot x = \mathbf{1}$
shows *inv-equality*: $x^{-1} = x'$
proof −
have $x^{-1} = \mathbf{1} \cdot x^{-1}$ **by** (*simp only*: *left-one*)
also have $\ldots = (x' \cdot x) \cdot x^{-1}$ **by** (*simp only*: *eq*)
also have $\ldots = x' \cdot (x \cdot x^{-1})$ **by** (*simp only*: *assoc*)
also have $\ldots = x' \cdot \mathbf{1}$ **by** (*simp only*: *right-inv*)
also have $\ldots = x'$ **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 $(y^{-1} \cdot x^{-1}) \cdot (x \cdot y) = \mathbf{1}$
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 $\ldots = (y^{-1} \cdot \mathbf{1}) \cdot y$ **by** (*simp only*: *left-inv*)
also have $\ldots = y^{-1} \cdot y$ **by** (*simp only*: *right-one*)
also have $\ldots = \mathbf{1}$ **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-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 $\ldots = x^{-1} \cdot y^{-1}$ **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-inv: $(x^{-1})^{-1} = x$
proof (*rule inv-equality*)
show $x \cdot x^{-1} = \mathbf{1}$ **by** (*simp only*: *right-inv*)
qed

theorem (**in** *group-context*)
assumes *eq*: $x^{-1} = y^{-1}$
shows *inv-inject*: $x = y$
proof –
have $x = x \cdot \mathbf{1}$ **by** (*simp only*: *right-one*)
also have $\ldots = x \cdot (y^{-1} \cdot y)$ **by** (*simp only*: *left-inv*)
also have $\ldots = x \cdot (x^{-1} \cdot y)$ **by** (*simp only*: *eq*)
also have $\ldots = (x \cdot x^{-1}) \cdot y$ **by** (*simp only*: *assoc*)
also have $\ldots = \mathbf{1} \cdot y$ **by** (*simp only*: *right-inv*)
also have $\ldots = y$ **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-context prod inv one $\equiv$

$(\forall x\ y\ z.\ prod\ (prod\ x\ y)\ z = prod\ x\ (prod\ y\ z)) \wedge$
$(\forall x.\ prod\ (inv\ x)\ x = one) \wedge (\forall x.\ prod\ one\ x = x)$

The corresponding introduction rule is as follows:

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

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.

9.3 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** ·ı *70*)

record *'a group* = *'a semigroup* +
  inv :: *'a* ⇒ *'a*   ((-$^{-1}$ı) [*1000*] *999*)
  one :: *'a*   (**1**ı)

The mixfix annotations above include a special "structure index indicator" ı 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 *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*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

locale *group* = *semigroup G* +
  assumes *left-inv*: $x^{-1} \cdot x = \mathbf{1}$

and *left-one*: $\mathbf{1} \cdot x = x$

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: $x \cdot x^{-1} = \mathbf{1}$
proof –
have $x \cdot x^{-1} = \mathbf{1} \cdot (x \cdot x^{-1})$ **by** (*simp only*: *left-one*)
also have $\ldots = \mathbf{1} \cdot x \cdot x^{-1}$ **by** (*simp only*: *assoc*)
also have $\ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1}$ **by** (*simp only*: *left-inv*)
also have $\ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1}$ **by** (*simp only*: *assoc*)
also have $\ldots = (x^{-1})^{-1} \cdot \mathbf{1} \cdot x^{-1}$ **by** (*simp only*: *left-inv*)
also have $\ldots = (x^{-1})^{-1} \cdot (\mathbf{1} \cdot x^{-1})$ **by** (*simp only*: *assoc*)
also have $\ldots = (x^{-1})^{-1} \cdot x^{-1}$ **by** (*simp only*: *left-one*)
also have $\ldots = \mathbf{1}$ **by** (*simp only*: *left-inv*)
finally show *?thesis* **.**
qed

theorem (**in** *group*)
right-one: $x \cdot \mathbf{1} = x$
proof –
have $x \cdot \mathbf{1} = x \cdot (x^{-1} \cdot x)$ **by** (*simp only*: *left-inv*)
also have $\ldots = x \cdot x^{-1} \cdot x$ **by** (*simp only*: *assoc*)
also have $\ldots = \mathbf{1} \cdot x$ **by** (*simp only*: *right-inv*)
also have $\ldots = x$ **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" ($\imath$).

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 $x \cdot y \cdot \mathbf{1} = prod\ G\ (prod\ G\ x\ y)\ (one\ G)$
    and $x \cdot_2 y \cdot_2 \mathbf{1}_2 = prod\ H\ (prod\ H\ x\ y)\ (one\ H)$
    and $x \cdot \mathbf{1}_2 = prod\ G\ x\ (one\ H)$
  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.

9.4 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 $\equiv$
    $(\!|prod = \lambda p\ q.\ (prod\ S\ (fst\ p)\ (fst\ q),\ prod\ T\ (snd\ p)\ (snd\ q))|\!)$

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 $prod\ (semigroup\text{-}product\ S\ T)\ (prod\ (semigroup\text{-}product\ S\ T)\ p\ q)\ r =$
      $prod\ (semigroup\text{-}product\ S\ T)\ p\ (prod\ (semigroup\text{-}product\ S\ T)\ q\ r)$
    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 $p\ q\ r :: 'a \times 'b$
  have $(fst\ p \cdot_1 fst\ q) \cdot_1 fst\ r = fst\ p \cdot_1 (fst\ q \cdot_1 fst\ r)$
    by (*rule S.assoc*)
  moreover have $(snd\ p \cdot_2 snd\ q) \cdot_2 snd\ r = snd\ p \cdot_2 (snd\ q \cdot_2 snd\ r)$
    by (*rule T.assoc*)
  ultimately show $(p \cdot_3 q) \cdot_3 r = p \cdot_3 (q \cdot_3 r)$
    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* $\Rightarrow$ $'b$ *group* $\Rightarrow$ $('a \times 'b)$ *group*
  group-product G H $\equiv$
    semigroup.extend
      (*semigroup-product* (*semigroup.truncate G*) (*semigroup.truncate H*))
      (*group.fields* ($\lambda p.$ (*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 $I \equiv$ *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 𝟏₂ ·₂ snd p = snd p by (rule H.left-one)
    ultimately show 𝟏₃ ·₃ p = p
      by (simp add: I-def group-product-def group.defs
        semigroup-product-def semigroup.defs)
  qed
qed

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

end
```

10 Test of Locale Interpretation

```
theory LocaleTest2
imports GCD
begin
```

11 Interpretation of Defined Concepts

Naming convention for global objects: prefixes D and d

11.1 Lattices

Much of the lattice proofs are from HOL/Lattice.

11.1.1 Definitions

```
locale dpo =
  fixes le :: ['a, 'a] => bool (infixl ⊑ 50)
  assumes refl [intro, simp]: x ⊑ x
    and anti-sym [intro]: [| x ⊑ y; y ⊑ x |] ==> x = y
    and trans [trans]: [| x ⊑ y; y ⊑ z |] ==> x ⊑ z

begin

theorem circular:
```

$[| \ x \sqsubseteq y; \ y \sqsubseteq z; \ z \sqsubseteq x \ |] ==> x = y \ \& \ y = z$
by (*blast intro*: *trans*)

definition
less :: $['a, \ 'a] => bool$ (**infixl** $\sqsubset$ *50*)
where $(x \sqsubset y) = (x \sqsubseteq y \ \& \ x \ \tilde{}= y)$

theorem *abs-test*:
op $\sqsubset = (\%x \ y. \ x \sqsubset y)$
by *simp*

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

definition
is-sup :: $['a, \ 'a, \ 'a] => bool$
where *is-sup* $x \ y \ s = (x \sqsubseteq s \wedge y \sqsubseteq s \wedge (\forall z. \ x \sqsubseteq z \wedge y \sqsubseteq z \longrightarrow s \sqsubseteq z))$

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** $\sqcap$ *70*)
where $x \sqcap y = $ (*THE inf. is-inf x y inf*)

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

lemma *is-infI* [*intro?*]: $i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow$
$(\bigwedge z. \ z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i) \Longrightarrow$ *is-inf* $x \ y \ i$
by (*unfold is-inf-def*) *blast*

lemma *is-inf-lower* [*elim?*]:
is-inf $x \ y \ i \Longrightarrow (i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow C) \Longrightarrow C$
by (*unfold is-inf-def*) *blast*

lemma *is-inf-greatest* [*elim?*]:
is-inf $x \ y \ i \Longrightarrow z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i$
by (*unfold is-inf-def*) *blast*

theorem *is-inf-uniq*: *is-inf* $x \ y \ i \Longrightarrow$ *is-inf* $x \ y \ i' \Longrightarrow i = i'$
proof –

assume *inf*: *is-inf* x y i
assume *inf'*: *is-inf* x y i'
show *?thesis*
proof (*rule anti-sym*)
from *inf'* **show** $i \sqsubseteq i'$
proof (*rule is-inf-greatest*)
from *inf* **show** $i \sqsubseteq x$ **..**
from *inf* **show** $i \sqsubseteq y$ **..**
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?*]: $x \sqsubseteq y \Longrightarrow$ *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?*]: *is-inf* x y $i \Longrightarrow x \sqcap y = i$
proof (*unfold meet-def*)
assume *is-inf* x y i
then show (*THE* i. *is-inf* x y i) $= i$
by (*rule the-equality*) (*rule is-inf-uniq* [*OF* - ‹*is-inf* x y i›])
qed

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

lemma *is-inf-meet* [*intro?*]: *is-inf* x y $(x \sqcap y)$
proof (*unfold meet-def*)
from *ex-inf* **obtain** i **where** *is-inf* x y i **..**
then show *is-inf* x y (*THE* i. *is-inf* x y i)
by (*rule theI*) (*rule is-inf-uniq* [*OF* - ‹*is-inf* x y i›])
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?*]:
[| $z \sqsubseteq x$; $z \sqsubseteq y$ |] ==> $z \sqsubseteq x \sqcap y$
by (*rule is-inf-greatest*) (*rule is-inf-meet*)

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

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

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

theorem *is-sup-uniq*: *is-sup* $x\ y\ s \Longrightarrow$ *is-sup* $x\ y\ s' \Longrightarrow s = s'$
proof –
  assume *sup*: *is-sup* $x\ y\ s$
  assume *sup'*: *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?*]: $x \sqsubseteq y \Longrightarrow$ *is-sup* $x\ y\ y$
proof –
  assume $x \sqsubseteq y$
  show *?thesis*
  proof
    show $x \sqsubseteq y$ **by** *fact*
    show $y \sqsubseteq y$ **..**
    fix z **assume** $x \sqsubseteq z$ **and** $y \sqsubseteq z$
    show $y \sqsubseteq z$ **by** *fact*

qed
qed

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

lemma *joinI* [*intro?*]: $x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow$
$(\bigwedge z.\ x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z) \Longrightarrow x \sqcup y = s$
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* $x\ y\ (THE\ s.\ \textit{is-sup}\ x\ y\ s)$
by (*rule theI*) (*rule is-sup-uniq* [*OF* - ‹*is-sup x y s*›])
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?*]:
$[|\ x \sqsubseteq z;\ y \sqsubseteq z\ |] ==> x \sqcup y \sqsubseteq z$
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 $\ldots \sqsubseteq y$ **..**
finally show *?thesis* **.**
qed
qed
show $x \sqcap (y \sqcap z) \sqsubseteq z$
proof −
have $x \sqcap (y \sqcap z) \sqsubseteq y \sqcap z$ **..**
also have $\ldots \sqsubseteq z$ **..**

```
      finally show ?thesis .
    qed
    fix w assume w ⊑ x ⊓ y and w ⊑ z
    show w ⊑ x ⊓ (y ⊓ z)
    proof
      show w ⊑ x
      proof −
        have w ⊑ x ⊓ y by fact
        also have . . . ⊑ x ..
        finally show ?thesis .
      qed
      show w ⊑ y ⊓ z
      proof
        show w ⊑ y
        proof −
          have w ⊑ x ⊓ y by fact
          also have . . . ⊑ y ..
          finally show ?thesis .
        qed
        show w ⊑ z by fact
      qed
    qed
  qed

theorem meet-commute: x ⊓ y = y ⊓ x
proof (rule meetI)
  show y ⊓ x ⊑ x ..
  show y ⊓ x ⊑ y ..
  fix z assume z ⊑ y and z ⊑ x
  then show z ⊑ y ⊓ x ..
qed

theorem meet-join-absorb: x ⊓ (x ⊔ y) = x
proof (rule meetI)
  show x ⊑ x ..
  show x ⊑ x ⊔ y ..
  fix z assume z ⊑ x and z ⊑ x ⊔ y
  show z ⊑ x by fact
qed

theorem join-assoc: (x ⊔ y) ⊔ z = x ⊔ (y ⊔ z)
proof (rule joinI)
  show x ⊔ y ⊑ x ⊔ (y ⊔ z)
  proof
    show x ⊑ x ⊔ (y ⊔ z) ..
    show y ⊑ x ⊔ (y ⊔ z)
    proof −
      have y ⊑ y ⊔ z ..
      also have ... ⊑ x ⊔ (y ⊔ z) ..
```

    finally show *?thesis* **.**
   qed
  qed
  show $z \sqsubseteq x \sqcup (y \sqcup z)$
  proof $-$
   have $z \sqsubseteq y \sqcup z$ **..**
   also have ... $\sqsubseteq x \sqcup (y \sqcup z)$ **..**
   finally show *?thesis* **.**
  qed
  fix w **assume** $x \sqcup y \sqsubseteq w$ **and** $z \sqsubseteq w$
  show $x \sqcup (y \sqcup z) \sqsubseteq w$
  proof
   show $x \sqsubseteq w$
   proof $-$
    have $x \sqsubseteq x \sqcup y$ **..**
    also have $\ldots \sqsubseteq w$ **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 ... $\sqsubseteq w$ **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?*]: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
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?*]: $y \sqsubseteq x \Longrightarrow x \sqcap y = y$
by (*drule meet-related*) (*simp add*: *meet-commute*)

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

Additional theorems

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

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

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

next
  have $x \sqsubseteq x \sqcup y$ **..**
  also assume $x \sqcup y = y$
  finally show $x \sqsubseteq y$ **.**
qed

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

Naming according to Jacobson I, p. 459.

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

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

end

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

begin

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

Jacobson I, p. 462

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

end

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

begin

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

end

interpretation *dlo* < *dlat*

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

interpretation *dlo* < *ddlat*
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

11.1.2 Total order <= 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 ⟦ (*x*::*int*) ≤ *y*; *y* ≤ *z*; *z* ≤ *x*⟧ ⟹ *x* = *y* ∧ *y* = *z*
apply (*rule int.circular*) **apply** *assumption* **apply** *assumption* **apply** *assumption*
done
thm *int.abs-test*
lemma (*op* < :: [*int*, *int*] => *bool*) = *op* <
apply (*rule int.abs-test*) **done**

interpretation *int*: *dlat* [*op* <= :: [*int*, *int*] => *bool*]
where *meet-eq*: *dlat.meet* (*op* <=) (*x*::*int*) *y* = *min x y*
and *join-eq*: *dlat.join* (*op* <=) (*x*::*int*) *y* = *max x y*
proof −
show *dlat* (*op* <= :: [*int*, *int*] => *bool*)
apply *unfold-locales*
apply (*unfold int.is-inf-def int.is-sup-def*)
apply *arith*+
done
then interpret *int*: *dlat* [*op* <= :: [*int*, *int*] => *bool*] **.**

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

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

interpretation *int*: *dlo* [*op* <= :: [*int*, *int*] => *bool*]
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

11.1.3 Total order <= on *nat*

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

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

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

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

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

```
interpretation nat: dlat [op <= :: [nat, nat] => bool]
  where dlat.meet (op <=) (x::nat) y = min x y
    and dlat.join (op <=) (x::nat) y = max x y
proof -
  show dlat (op <= :: [nat, nat] => bool)
    apply unfold-locales
    apply (unfold nat.is-inf-def nat.is-sup-def)
    apply arith+
    done
  then interpret nat: dlat [op <= :: [nat, nat] => bool] .
```

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

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

interpretation *nat*: *dlo* [*op* <= :: [*nat*, *nat*] => *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 ddlat

thm *nat.less-total*

from ldo

11.1.4 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
```

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

11.2.1 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: unit x
  then show x ** y = x ** z by simp
qed

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

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

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

end


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

begin
```

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

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

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

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

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

lemma *r-cancel* [*simp*]:
  $(y ** x = z ** x) = (y = z)$
proof
  assume *eq*: $y ** x = z ** x$
  then have $y ** (x ** inv\ x) = z ** (x ** inv\ x)$
    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 $... = one$ **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 ==> y ** x = one
  by (rule unit-inv-comm) auto

lemma inv-equality:
     y ** x = one ==> 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
```

11.2.2 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 o f) x = (g o f) y by simp
        with id2 show x = y by simp
      qed
    next
      show surj f
      proof (rule surjI)
        fix x
        from id1 have (f o g) x = x by simp
        then show f (g x) = x by simp
      qed
    qed
  next
    fix f
    assume bij: bij f
    then
    have inv: f o Hilbert-Choice.inv f = id & Hilbert-Choice.inv f o f = id
      by (simp add: bij-def surj-iff inj-iff)
    show EX g. f o g = id & g o f = id by rule (rule inv)
  qed
qed

thm Dmonoid.unit-def Dfun.unit-def

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

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

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

12 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* := $\lambda x\ y.$ *times M y x* |)

end

13 Binary arithmetic examples

theory *BinEx* **imports** *Main* **begin**

13.1 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**

13.2 Arithmetic Method Tests

lemma $!!a::int.$ [| $a <= b$; $c <= d$; $x+y<z$ |] ==> $a+c <= b+d$
by *arith*

lemma $!!a::int.$ [| $a < b$; $c < d$ |] ==> $a-d+ 2 <= b+(-c)$
by *arith*

lemma $!!a::int.$ [| $a < b$; $c < d$ |] ==> $a+c+ 1 < b+d$
by *arith*

lemma $!!a::int.$ [| $a <= b$; $b+b <= c$ |] ==> $a+a <= c$
by *arith*

lemma $!!a::int.$ [| $a+b <= i+j$; $a<=b$; $i<=j$ |] ==> $a+a <= j+j$
by *arith*

lemma $!!a::int.$ [| $a+b < i+j$; $a<b$; $i<j$ |] ==> $a+a - - -1 < j+j - 3$

by *arith*

lemma !!a::int. a+b+c <= i+j+k & a<=b & b<=c & i<=j & j<=k -->
a+a+a <= k+k+k
by *arith*

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

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

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

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

lemma !!a::int. [| a+b+c+d <= i+j+k+l; a<=b; b<=c; c<=d; i<=j; j<=k;
k<=l |]
 ==> 6*a <= 5*l+i
by *arith*

13.3 The Integers

Addition

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

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

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

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

Negation

lemma − (65745::int) = −65745

by *simp*

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

Multiplication

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

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

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

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

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

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

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

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

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

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

No integer overflow!

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

Quotient and Remainder

lemma $(10::int)$ *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 dividend[1]

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

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

A negative dividend *and* divisor

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

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

A few bigger examples

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

lemma $(59485::int)$ *div* $434 = 137$
  by *simp*

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

Division by shifting

lemma 10000000 *div* $2 = (5000000::int)$
  by *simp*

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

lemma 10000055 *div* $32 = (312501::int)$
  by *simp*

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

lemma 100094 *div* $144 = (695::int)$
  by *simp*

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

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

Powers

lemma $2\ \hat{}\ 10 = (1024::\mathit{int})$
  by *simp*

lemma $-3\ \hat{}\ 7 = (-2187::\mathit{int})$
  by *simp*

lemma $13\ \hat{}\ 7 = (62748517::\mathit{int})$
  by *simp*

lemma $3\ \hat{}\ 15 = (14348907::\mathit{int})$
  by *simp*

lemma $-5\ \hat{}\ 11 = (-48828125::\mathit{int})$
  by *simp*

13.4 The Natural Numbers

Successor

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

Addition

lemma $(13::\mathit{nat}) + 19 = 32$
  by *simp*

lemma $(1234::\mathit{nat}) + 5678 = 6912$
  by *simp*

lemma $(973646::\mathit{nat}) + 6475 = 980121$
  by *simp*

Subtraction

lemma $(32::\mathit{nat}) - 14 = 18$
  by *simp*

lemma $(14::\mathit{nat}) - 15 = 0$
  by *simp*

lemma $(14::\mathit{nat}) - 1576644 = 0$
  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 $(10::nat)\ div\ 3 = 3$
  by *simp*

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

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

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

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

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

Powers

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

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

lemma $12\ \hat{}\ 7 = (35831808::nat)$
  by *simp*

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

lemma $5\ \hat{}\ 11 = (48828125::nat)$
  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

14 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

15 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

16 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

17 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.

17.1 Partial equivalence

Type class *partial-equiv* models partial equivalence relations (PERs) using the polymorphic $\sim$ :: *'a => 'a => bool* relation, which is required to be symmetric and transitive, but not necessarily reflexive.

consts
  eqv :: *'a => 'a => bool*   (**infixl** $\sim$ *50*)

axclass *partial-equiv* < *type*
  partial-equiv-sym [*elim?*]: $x \sim y$ ==> $y \sim x$
  partial-equiv-trans [*trans*]: $x \sim y$ ==> $y \sim z$ ==> $x \sim z$

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

lemma *domainI* [*intro*]: $x \sim x$ ==> $x \in$ *domain*
  unfolding *domain-def* **by** *blast*

lemma *domainD* [*dest*]: $x \in$ *domain* ==> $x \sim x$
  unfolding *domain-def* **by** *blast*

theorem *domainI'* [*elim?*]: $x \sim y$ ==> $x \in$ *domain*
proof
  assume *xy*: $x \sim y$
  also from *xy* **have** $y \sim x$ **..**
  finally show $x \sim x$ **.**
qed

17.2 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: $f \sim g$ == $\forall x \in$ *domain*. $\forall y \in$ *domain*. $x \sim y$ --> $f\,x \sim g\,y$

lemma *partial-equiv-funI* [*intro?*]:
  (!!*x y*. *x* ∈ *domain* ==> *y* ∈ *domain* ==> *x* ∼ *y* ==> *f x* ∼ *g y*) ==> *f* ∼ *g*
 unfolding *eqv-fun-def* **by** *blast*

lemma *partial-equiv-funD* [*dest?*]:
  f ∼ *g* ==> *x* ∈ *domain* ==> *y* ∈ *domain* ==> *x* ∼ *y* ==> *f x* ∼ *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

17.3 Total equivalence

The class of total equivalence relations on top of PERs. It coincides with the standard notion of equivalence, i.e. ∼ :: *'a* => *'a* => *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?*]: $f \sim g$ ==> $x \sim y$ ==> $f\ x \sim g\ (y::'a::equiv)$
proof −
  assume $f \sim g$
  moreover have $x \in$ *domain* **..**
  moreover have $y \in$ *domain* **..**
  moreover assume $x \sim y$
  ultimately show *?thesis* **..**
qed

17.4 Quotient types

The quotient type *'a quot* consists of all *equivalence classes* over elements of the base type *'a*.

typedef *'a quot* = $\{\{x.\ a \sim x\}|\ a::'a.\ True\}$
  by *blast*

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in$ *quot*
  unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]: $R \in$ *quot* ==> (!!a. $R = \{x.\ a \sim x\}$ ==> 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*   ($\lfloor$-$\rfloor$) **where**
  $\lfloor a \rfloor$ = *Abs-quot* $\{x.\ a \sim x\}$

theorem *quot-rep*: $\exists\, a.\ A = \lfloor a \rfloor$
proof (*cases A*)
  fix R **assume** R: A = *Abs-quot* R
  assume $R \in$ *quot* **then have** $\exists\, a.\ R = \{x.\ a \sim x\}$ **by** *blast*
  with R **have** $\exists\, a.\ A$ = *Abs-quot* $\{x.\ a \sim x\}$ **by** *blast*
  then show *?thesis* **by** (*unfold eqv-class-def*)
qed

lemma *quot-cases* [*cases type*: *quot*]:
  obtains (*rep*) a **where** $A = \lfloor a \rfloor$
  using *quot-rep* **by** *blast*

17.5 Equality on quotients

Equality of canonical quotient elements corresponds to the original relation as follows.

theorem *eqv-class-eqI* [*intro*]: $a \sim b$ ==> $\lfloor a \rfloor = \lfloor b \rfloor$

proof –
  assume *ab*: $a \sim b$
  have $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
  proof (*rule Collect-cong*)
    fix x **show** $(a \sim x) = (b \sim x)$
    proof
      from *ab* **have** $b \sim a$ **..**
      also assume $a \sim x$
      finally show $b \sim x$ **.**
    next
      note *ab*
      also assume $b \sim x$
      finally show $a \sim x$ **.**
    qed
  qed
  then show *?thesis* **by** (*simp only*: *eqv-class-def*)
qed

theorem *eqv-class-eqD'* [*dest?*]: $\lfloor a\rfloor = \lfloor b\rfloor$ ==> $a \in$ *domain* ==> $a \sim b$
proof (*unfold eqv-class-def*)
  assume *Abs-quot* $\{x.\ a \sim x\}$ = *Abs-quot* $\{x.\ b \sim x\}$
  then have $\{x.\ a \sim x\} = \{x.\ b \sim x\}$ **by** (*simp only*: *Abs-quot-inject quotI*)
  moreover assume $a \in$ *domain* **then have** $a \sim a$ **..**
  ultimately have $a \in \{x.\ b \sim x\}$ **by** *blast*
  then have $b \sim a$ **by** *blast*
  then show $a \sim b$ **..**
qed

theorem *eqv-class-eqD* [*dest?*]: $\lfloor a\rfloor = \lfloor b\rfloor$ ==> $a \sim (b::'a::equiv)$
proof (*rule eqv-class-eqD'*)
  show $a \in$ *domain* **..**
qed

lemma *eqv-class-eq'* [*simp*]: $a \in$ *domain* ==> $(\lfloor a\rfloor = \lfloor b\rfloor) = (a \sim b)$
  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*

17.6 Picking representing elements

definition
  pick :: *'a::partial-equiv quot* => *'a* **where**
  pick A = (*SOME a*. $A = \lfloor a\rfloor$)

theorem *pick-eqv'* [*intro?*, *simp*]: $a \in$ *domain* ==> *pick* $\lfloor a\rfloor \sim a$
proof (*unfold pick-def*)
  assume *a*: $a \in$ *domain*
  show (*SOME x*. $\lfloor a\rfloor = \lfloor x\rfloor) \sim a$

proof (*rule someI2*)
  show $\lfloor a \rfloor = \lfloor a \rfloor$ **..**
  fix x **assume** $\lfloor a \rfloor = \lfloor x \rfloor$
  from *this* **and** a **have** $a \sim x$ **..**
  then show $x \sim a$ **..**
qed
qed

theorem *pick-eqv* [*intro*, *simp*]: *pick* $\lfloor a \rfloor \sim (a{::}'a{::}equiv)$
proof (*rule pick-eqv'*)
  show $a \in domain$ **..**
qed

theorem *pick-inverse*: $\lfloor pick\ A \rfloor = (A{::}'a{::}equiv\ quot)$
proof (*cases A*)
  fix a **assume** a: $A = \lfloor a \rfloor$
  then have $pick\ A \sim a$ **by** *simp*
  then have $\lfloor pick\ A \rfloor = \lfloor a \rfloor$ **by** *simp*
  with a **show** *?thesis* **by** *simp*
qed

end

18 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-distribs
  diff-mult-distrib diff-mult-distrib2 — for type nat

The sum of the first n odd numbers equals n squared.

lemma *sum-of-odds*: $(\sum i{=}0..{<}n.\ Suc\ (i + i)) = n * n$
  by (*induct n*) *auto*

The sum of the first n odd squares.

lemma *sum-of-odd-squares*:
  $3 * (\sum i{=}0..{<}n.\ Suc(2{*}i) * Suc(2{*}i)) = n * (4 * n * n - 1)$
  by (*induct n*) *auto*

The sum of the first n odd cubes

lemma *sum-of-odd-cubes*:

$(\sum i{=}0..{<}n.\ Suc\ (2{*}i) * Suc\ (2{*}i) * Suc\ (2{*}i)) =$
  $n * n * (2 * n * n - 1)$
  by $(induct\ n)\ auto$

The sum of the first n positive integers equals $n\ (n + 1)\ /\ 2$.

lemma *sum-of-naturals*:
  $2 * (\sum i{=}0..n.\ i) = n * Suc\ n$
  by $(induct\ n)\ auto$

lemma *sum-of-squares*:
  $6 * (\sum i{=}0..n.\ i * i) = n * Suc\ n * Suc\ (2 * n)$
  by $(induct\ n)\ auto$

lemma *sum-of-cubes*:
  $4 * (\sum i{=}0..n.\ i * i * i) = n * n * Suc\ n * Suc\ n$
  by $(induct\ n)\ auto$

A cute identity:

lemma *sum-squared*: $(\sum i{=}0..n.\ i)\hat{}2 = (\sum i{=}0..n{::}nat.\ i\hat{}3)$
proof$(induct\ n)$
  case 0 **show** *?case* **by** *simp*
next
  case $(Suc\ n)$
  have $(\sum i = 0..Suc\ n.\ i)\hat{}2 =$
        $(\sum i = 0..n.\ i\hat{}3) + (2{*}(\sum i = 0..n.\ i){*}(n{+}1) + (n{+}1)\hat{}2)$
    (**is** $- = ?A + ?B$)
    using *Suc* **by**(*simp add*:*nat-number*)
  also have $?B = (n{+}1)\hat{}3$
    using *sum-of-naturals* **by**(*simp add*:*nat-number*)
  also have $?A + (n{+}1)\hat{}3 = (\sum i{=}0..Suc\ n.\ i\hat{}3)$ **by** *simp*
  finally show *?case* **.**
qed

Sum of fourth powers: three versions.

lemma *sum-of-fourth-powers*:
  $30 * (\sum i{=}0..n.\ i * i * i * i) =$
    $n * Suc\ n * Suc\ (2 * n) * (3 * n * n + 3 * n - 1)$
  apply $(induct\ n)$
   apply *simp-all*
  apply $(case\text{-}tac\ n)$ — eliminates the subtraction
   apply (*simp-all* (*no-asm-simp*))
  done

Two alternative proofs, with a change of variables and much more subtraction, performed using the integers.

lemma *int-sum-of-fourth-powers*:
  $30 * int\ (\sum i{=}0..{<}m.\ i * i * i * i) =$

int m ∗ (*int m* − *1*) ∗ (*int*(*2* ∗ *m*) − *1*) ∗
(*int*(*3* ∗ *m* ∗ *m*) − *int*(*3* ∗ *m*) − *1*)
by (*induct m*) (*simp-all add*: *int-mult*)

lemma *of-nat-sum-of-fourth-powers*:
30 ∗ *of-nat* (∑ *i=0..<m. i* ∗ *i* ∗ *i* ∗ *i*) =
of-nat m ∗ (*of-nat m* − *1*) ∗ (*of-nat* (*2* ∗ *m*) − *1*) ∗
(*of-nat* (*3* ∗ *m* ∗ *m*) − *of-nat* (*3* ∗ *m*) − (*1*::*int*))
by (*induct m*) (*simp-all add*: *of-nat-mult*)

Sums of geometric series: *2*, *3* and the general case.

lemma *sum-of-2-powers*: (∑ *i=0..<n. 2ˆi*) = *2ˆn* − (*1*::*nat*)
by (*induct n*) (*auto split*: *nat-diff-split*)

lemma *sum-of-3-powers*: *2* ∗ (∑ *i=0..<n. 3ˆi*) = *3ˆn* − (*1*::*nat*)
by (*induct n*) *auto*

lemma *sum-of-powers*: *0* < *k* ==> (*k* − *1*) ∗ (∑ *i=0..<n. kˆi*) = *kˆn* − (*1*::*nat*)
by (*induct n*) *auto*

end

19 Three Divides Theorem

theory *ThreeDivides*
imports *Main LaTeXsugar*
begin

19.1 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

$$\forall n\ 3|n \Longleftrightarrow 3|\sum n_j$$

□

19.2 Formal proof

19.2.1 Miscellaneous summation lemmas

If a divides $A\ x$ for all x then a divides any sum over terms of the form $(A\ x)*(P\ x)$ for arbitrary P.

lemma *div-sum*:
  fixes $a::nat$ **and** $n::nat$
  shows $\forall x.\ a\ dvd\ A\ x \Longrightarrow a\ dvd\ (\sum x<n.\ A\ x * D\ x)$
proof (*induct n*)
  case *0* **show** *?case* **by** *simp*
next
  case (*Suc n*)
  from *Suc*
  have $a\ dvd\ (A\ n * D\ n)$ **by** (*simp add*: *dvd-mult2*)
  with *Suc*
  have $a\ dvd\ ((\sum x<n.\ A\ x * D\ x) + (A\ n * D\ n))$ **by** (*simp add*: *dvd-add*)
  thus *?case* **by** *simp*
qed

19.2.2 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 $n = (\sum x\in\{..<nd\}.\ (D\ x)*((10::nat)\hat{\ }x)) \Longrightarrow$
      $(n - (\sum x<nd.\ (D\ x))) = (\sum x<nd.\ (D\ x)*(10\hat{\ }x - 1))$
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 $(3::nat)\ dvd\ (10\hat{\ }x - 1)$
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 $?thr\ dvd\ (10*(10\hat{\ }n - 1))$ **by** (*rule dvd-mult*) (*rule Suc*)
  hence $?thr\ dvd\ (10\hat{\ }(n+1) - 10)$ **by** (*simp add*: *nat-distrib*)

ultimately
have *?thr dvd* $((10\hat{}(n+1) - 10) + 9)$
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 $3\ dvd\ (\sum x{<}nd.\ D\ x * (10\hat{}x - 1))$
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 $3\ dvd\ ((\sum x{<}nd.\ (D\ x){*}(10\hat{}x)) - (\sum x{<}nd.\ (D\ x)))$
proof –
from *three-divs-1* **have** $3\ dvd\ (\sum x{<}nd.\ D\ x * (10\ \hat{}\ x - 1))$ **.**
thus *?thesis* **by** (*simp only*: *digit-diff-split*)
qed

We now present the final theorem of this section. For any sequence of numbers (defined by a function D), we show that 3 divides the expansive sum $\sum (D\ x) * 10^x$ over x if and only if 3 divides the sum of the individual numbers $\sum D\ x$.

lemma *three-div-general*:
fixes $D :: nat \Rightarrow nat$
shows $(3\ dvd\ (\sum x{<}nd.\ D\ x * 10\hat{}x)) = (3\ dvd\ (\sum x{<}nd.\ D\ x))$
proof
have *mono*: $(\sum x{<}nd.\ D\ x) \leq (\sum x{<}nd.\ D\ x * 10\hat{}x)$
by (*rule setsum-mono*) *simp*

This lets us form the term $(\sum x{<}nd.\ D\ x * 10\ \hat{}\ x) - setsum\ D\ \{..{<}nd\}$

{
assume $3\ dvd\ (\sum x{<}nd.\ D\ x)$
with *three-divs-2 mono*
show $3\ dvd\ (\sum x{<}nd.\ D\ x * 10\hat{}x)$
by (*blast intro*: *dvd-diffD*)
}
{
assume $3\ dvd\ (\sum x{<}nd.\ D\ x * 10\hat{}x)$
with *three-divs-2 mono*
show $3\ dvd\ (\sum x{<}nd.\ D\ x)$
by (*blast intro*: *dvd-diffD1*)
}
qed

19.2.3 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* ⇒ *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* ⇒ *nat* **where**
  sumdig n = ($\sum$ *x* < *nlen n*. *n div 10*ˆ*x mod 10*)

Some properties of these functions follow.

lemma *nlen-zero*:
  0 = *nlen x* ⟹ *x* = *0*
  by (*induct x rule*: *nlen.induct*) *auto*

lemma *nlen-suc*:
  Suc m = *nlen n* ⟹ *m* = *nlen* (*n div 10*)
  by (*induct n rule*: *nlen.induct*) *simp-all*

The following lemma is the principle lemma required to prove our theorem. It states that an expansion of some natural number n into a sequence of its individual digits is always possible.

lemma *exp-exists*:
  m = ($\sum$ *x*<*nlen m*. (*m div* (*10*::*nat*)ˆ*x mod 10*) ∗ *10*ˆ*x*)
proof (*induct nd* ≡ *nlen m arbitrary*: *m*)
  case *0* **thus** *?case* **by** (*simp add*: *nlen-zero*)
next
  case (*Suc nd*)
  hence *IH*:
    nd = *nlen* (*m div 10*) ⟹
    m div 10 = ($\sum$ *x*<*nd*. *m div 10 div 10*ˆ*x mod 10* ∗ *10*ˆ*x*)
    by *blast*
  have ∃ *c*. *m* = *10*∗(*m div 10*) + *c* ∧ *c* < *10* **by** *presburger*
  then obtain *c* **where** *mexp*: *m* = *10*∗(*m div 10*) + *c* ∧ *c* < *10* **..**
  then have *cdef*: *c* = *m mod 10* **by** *arith*

show $m = (\sum x<nlen\ m.\ m\ div\ 10\hat{}x\ mod\ 10 * 10\hat{}x)$
proof −
from ‹$Suc\ nd = nlen\ m$›
have $nd = nlen\ (m\ div\ 10)$ **by** (*rule nlen-suc*)
with *IH* **have**
$m\ div\ 10 = (\sum x<nd.\ m\ div\ 10\ div\ 10\hat{}x\ mod\ 10 * 10\hat{}x)$ **by** *simp*
with *mexp* **have**
$m = 10*(\sum x<nd.\ m\ div\ 10\ div\ 10\hat{}x\ mod\ 10 * 10\hat{}x) + c$ **by** *simp*
also have
$\ldots = (\sum x<nd.\ m\ div\ 10\ div\ 10\hat{}x\ mod\ 10 * 10\hat{}(x+1)) + c$
by (*subst setsum-right-distrib*) (*simp add*: *mult-ac*)
also have
$\ldots = (\sum x<nd.\ m\ div\ 10\hat{}(Suc\ x)\ mod\ 10 * 10\hat{}(Suc\ x)) + c$
by (*simp add*: *div-mult2-eq*[*symmetric*])
also have
$\ldots = (\sum x\in\{Suc\ 0..<Suc\ nd\}.\ m\ div\ 10\hat{}x\ \ mod\ 10 * 10\hat{}x) + c$
by (*simp only*: *setsum-shift-bounds-Suc-ivl*)
(*simp add*: *atLeast0LessThan*)
also have
$\ldots = (\sum x<Suc\ nd.\ m\ div\ 10\hat{}x\ mod\ 10 * 10\hat{}x)$
by (*simp add*: *setsum-head-upt cdef*)
also note ‹$Suc\ nd = nlen\ m$›
finally
show $m = (\sum x<nlen\ m.\ m\ div\ 10\hat{}x\ mod\ 10 * 10\hat{}x)$ **.**
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 $n = (\sum x<nlen\ n.\ (n\ div\ (10::nat)\hat{}x\ mod\ 10) * 10\hat{}x)$
by (*rule exp-exists*)
moreover
have $3\ dvd\ (\sum x<nlen\ n.\ (n\ div\ (10::nat)\hat{}x\ mod\ 10) * 10\hat{}x) =$
$(3\ dvd\ (\sum x<nlen\ n.\ n\ div\ 10\hat{}x\ mod\ 10))$
by (*rule three-div-general*)
ultimately
show $3\ dvd\ n = (3\ dvd\ (\sum x<nlen\ n.\ n\ div\ 10\hat{}x\ mod\ 10))$ **by** *simp*
qed

end

20 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 $\sim\sim \ ((\sim P \ --> \ Q) \ --> (\sim P \ --> \ \sim Q) \ --> \ P)$
by *iprover*

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

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

lemma $\sim\sim((P \ --> \ Q \ | \ R) \ --> \ (P-->Q) \ | \ (P-->R))$
by *iprover*

lemma $(P{=}Q) = (Q{=}P)$
by *iprover*

lemma $((P \ --> \ (Q \ | \ (Q-->R))) \ --> \ R) \ --> \ R$
by *iprover*

lemma $(((G-->A) \ --> \ J) \ --> \ D \ --> \ E) \ --> \ (((H-->B)-->I)-->C-->J)$
$--> (A-->H) \ --> \ F \ --> \ G \ --> \ (((C-->B)-->I)-->D)-->(A-->C)$
$--> (((F-->A)-->B) \ --> \ I) \ --> \ E$
by *iprover*

lemma $P \ --> \ \sim\sim P$
by *iprover*

lemma $\sim\sim(\sim\sim P \ --> \ P)$
by *iprover*

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

lemma $((P{=}Q) \dashrightarrow P\&Q\&R)$ &
$((Q{=}R) \dashrightarrow P\&Q\&R)$ &
$((R{=}P) \dashrightarrow P\&Q\&R) \dashrightarrow P\&Q\&R$
by *iprover*

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 ~ $(EX\ x.\ ALL\ y.\ p\ y\ x = ($~ $p\ x\ x))$
by *iprover*

lemma ~~$((P{-}{-}{>}Q)\ =\ ($~$Q\ {-}{-}{>}$ ~$P))$
by *iprover*

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

lemma ~$(P{-}{-}{>}Q)\ {-}{-}{>}\ (Q{-}{-}{>}P)$
by *iprover*

lemma ~~((~$P{-}{-}{>}Q)\ =\ ($~$Q\ {-}{-}{>}\ P))$
by *iprover*

lemma ~~$((P|Q{-}{-}{>}P|R)\ {-}{-}{>}\ P|(Q{-}{-}{>}R))$

by *iprover*

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

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

lemma $\sim\sim(((P{-}{-}{>}Q) \;{-}{-}{>}\; P) \;\; {-}{-}{>} \;\; P)$
by *iprover*

lemma $((P|Q)\ \&\ (\sim P|Q)\ \&\ (P|\ \sim Q)) \;{-}{-}{>}\; \sim\ (\sim P \mid \sim Q)$
by *iprover*

lemma $(Q{-}{-}{>}R) \;{-}{-}{>}\; (R{-}{-}{>}P\&Q) \;{-}{-}{>}\; (P{-}{-}{>}(Q|R)) \;{-}{-}{>}\; (P{=}Q)$
by *iprover*

lemma $P{=}P$
by *iprover*

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

lemma $((P\ =\ Q) = R)\ \ {-}{-}{>}\ \ \sim\sim(P\ =\ (Q\ =\ R))$
by *iprover*

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

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

lemma $\sim\sim((P\ {-}{-}{>}\ Q) = (\sim P \mid Q))$
by *iprover*

lemma $\sim\sim((P{-}{-}{>}Q) \mid (Q{-}{-}{>}P))$
by *iprover*

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

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

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

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

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

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

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

lemma $\sim\sim(EX\ x.\ ALL\ y\ z.\ (P(y)-->Q(z))\ -->\ (P(x)-->Q(x)))$
  by *iprover*

lemma $(ALL\ x\ y.\ EX\ z.\ ALL\ w.\ (P(x)\&Q(y)-->R(z)\&S(w)))$
    $-->\ (EX\ x\ y.\ P(x)\ \&\ Q(y))\ -->\ (EX\ z.\ R(z))$
  by *iprover*

lemma $(EX\ x.\ P-->Q(x))\ \&\ (EX\ x.\ Q(x)-->P)\ -->\ \sim\sim(EX\ x.\ P=Q(x))$
  by *iprover*

lemma $(ALL\ x.\ P = Q(x))\ \ -->\ \ (P = (ALL\ x.\ Q(x)))$
  by *iprover*

lemma ~~ ((*ALL x. P* | *Q*(*x*)) = (*P* | (*ALL x. Q*(*x*))))
by *iprover*

lemma (*EX x. P*(*x*)) &
(*ALL x. L*(*x*) --> ~ (*M*(*x*) & *R*(*x*))) &
(*ALL x. P*(*x*) --> (*M*(*x*) & *L*(*x*))) &
((*ALL x. P*(*x*)-->*Q*(*x*)) | (*EX x. P*(*x*)&*R*(*x*)))
--> (*EX x. Q*(*x*)&*P*(*x*))
by *iprover*

lemma (*EX x. P*(*x*) & ~*Q*(*x*)) &
(*ALL x. P*(*x*) --> *R*(*x*)) &
(*ALL x. M*(*x*) & *L*(*x*) --> *P*(*x*)) &
((*EX x. R*(*x*) & ~ *Q*(*x*)) --> (*ALL x. L*(*x*) --> ~ *R*(*x*)))
--> (*ALL x. M*(*x*) --> ~*L*(*x*))
by *iprover*

lemma (*ALL x. P*(*x*) --> (*ALL x. Q*(*x*))) &
(~~(*ALL x. Q*(*x*)|*R*(*x*)) --> (*EX x. Q*(*x*)&*S*(*x*))) &
(~~(*EX x. S*(*x*)) --> (*ALL x. L*(*x*) --> *M*(*x*)))
--> (*ALL x. P*(*x*) & *L*(*x*) --> *M*(*x*))
by *iprover*

lemma (((*EX x. P*(*x*)) & (*EX y. Q*(*y*))) -->
(((*ALL x.* (*P*(*x*) --> *R*(*x*))) & (*ALL y.* (*Q*(*y*) --> *S*(*y*)))) =
(*ALL x y.* ((*P*(*x*) & *Q*(*y*)) --> (*R*(*x*) & *S*(*y*))))))
by *iprover*

lemma (*ALL x.* (*P*(*x*) | *Q*(*x*)) --> ~ *R*(*x*)) &
(*ALL x.* (*Q*(*x*) --> ~ *S*(*x*)) --> *P*(*x*) & *R*(*x*))
--> (*ALL x.* ~~*S*(*x*))
by *iprover*

lemma ~(*EX x. P*(*x*) & (*Q*(*x*) | *R*(*x*))) &
(*EX x. L*(*x*) & *P*(*x*)) &
(*ALL x.* ~ *R*(*x*) --> *M*(*x*))
--> (*EX x. L*(*x*) & *M*(*x*))
by *iprover*

lemma (*ALL x. P*(*x*) & (*Q*(*x*)|*R*(*x*))-->*S*(*x*)) &

```
      (ALL x. S(x) & R(x) --> L(x)) &
      (ALL x. M(x) --> R(x))
  --> (ALL x. P(x) & M(x) --> L(x))
  by iprover


lemma (ALL x. ~~(P(a) & (P(x)-->P(b))-->P(c))) =
      (ALL x. ~~((~P(a) | P(x) | P(c)) & (~P(a) | ~P(b) | P(c))))
  oops


lemma
     (ALL x. EX y. J x y) &
      (ALL x. EX y. G x y) &
      (ALL x y. J x y | G x y --> (ALL z. J y z | G y z --> H x z))
  --> (ALL x. EX y. H x y)
  by iprover


lemma ~ (EX x. ALL y. F y x = (~F y y))
  by iprover


lemma (EX y. ALL x. F x y = F x x) -->
          ~(ALL x. EX y. ALL z. F z y = (~ F z x))
  by iprover


lemma (ALL x. f(x) -->
          (EX y. g(y) & h x y & (EX y. g(y) & ~ h x y))) &
          (EX x. j(x) & (ALL y. g(y) --> h x y))
          --> (EX x. j(x) & ~f(x))
  by iprover


lemma (a=b | c=d) & (a=c | b=d) --> a=d | b=c
  by iprover


lemma ((EX z w. (ALL x y. (P x y = ((x = z) & (y = w))))) -->
  (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))))) -->
   (EX w. (ALL y. (EX z. ((ALL x. (P x y = (x = z))) = (y = w))))))
  by iprover
```

lemma $(ALL\ x.\ (EX\ y.\ P(y)\ \&\ x{=}f(y))\ {-}{-}{>}\ P(x)) = (ALL\ x.\ P(x)\ {-}{-}{>}\ P(f(x)))$
  by *iprover*

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
    $(ALL\ x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ {-}{-}{>}\ P\ x\ z)\ {-}{-}{>}\ P\ (f\ a\ b)\ (f\ a\ c)$
  by *iprover*

lemma $ALL\ x.\ P\ x\ (f\ x) = (EX\ y.\ (ALL\ z.\ P\ z\ y\ {-}{-}{>}\ P\ z\ (f\ x))\ \&\ P\ x\ y)$
  by *iprover*

end

21 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 $p = \mathit{lfp}\ (\lambda s.\ p \cup \text{EX}\ s)$
definition
  EG (EG - [*80*] *90*) **where** EG $p = \mathit{gfp}\ (\lambda s.\ p \cap \text{EX}\ s)$

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*

21.1 Basic fixed point properties

First of all, we use the de-Morgan property of fixed points

lemma *lfp-gfp*: $\mathit{lfp}\ f = -\ \mathit{gfp}\ (\lambda s{::}'a\ set.\ -\ (f\ (-\ s)))$
proof
  show $\mathit{lfp}\ f \subseteq -\ \mathit{gfp}\ (\lambda s.\ -\ f\ (-\ s))$
  proof
    fix x **assume** l: $x \in \mathit{lfp}\ f$
    show $x \in -\ \mathit{gfp}\ (\lambda s.\ -\ f\ (-\ s))$
    proof
      assume $x \in \mathit{gfp}\ (\lambda s.\ -\ f\ (-\ s))$
      then obtain u **where** $x \in u$ **and** $u \subseteq -\ f\ (-\ u)$
        by (*auto simp add*: *gfp-def Sup-set-def*)
      then have $f\ (-\ u) \subseteq -\ u$ **by** *auto*
      then have $\mathit{lfp}\ f \subseteq -\ u$ **by** (*rule lfp-lowerbound*)
      from l **and** *this* **have** $x \notin u$ **by** *auto*
      with ‹$x \in u$› **show** *False* **by** *contradiction*
    qed
  qed
  show $-\ \mathit{gfp}\ (\lambda s.\ -\ f\ (-\ s)) \subseteq \mathit{lfp}\ f$
  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 AF p and AG p:

lemma *AF-lfp*: AF $p = lfp\ (\lambda s.\ p \cup \mathrm{AX}\ s)$ **by** (*simp add*: *lfp-gfp*)
lemma *AG-gfp*: AG $p = gfp\ (\lambda s.\ p \cap \mathrm{AX}\ s)$ **by** (*simp add*: *lfp-gfp*)

lemma *EF-fp*: EF $p = p \cup \mathrm{EX\ EF}\ p$
proof −
have *mono* $(\lambda s.\ p \cup \mathrm{EX}\ s)$ **by** *rule* (*auto simp add*: *EX-def*)
then show *?thesis* **by** (*simp only*: *EF-def*) (*rule lfp-unfold*)
qed

lemma *AF-fp*: AF $p = p \cup \mathrm{AX\ AF}\ p$
proof −
have *mono* $(\lambda s.\ p \cup \mathrm{AX}\ s)$ **by** *rule* (*auto simp add*: *AX-def EX-def*)
then show *?thesis* **by** (*simp only*: *AF-lfp*) (*rule lfp-unfold*)
qed

lemma *EG-fp*: EG $p = p \cap \mathrm{EX\ EG}\ p$
proof −
have *mono* $(\lambda s.\ p \cap \mathrm{EX}\ s)$ **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 AG p, we derive as a consequence of the Knaster-Tarski theorem on the one hand that AG p is a fixed point of the monotonic function $\lambda s.\ p \cap \mathrm{AX}\ s$.

lemma *AG-fp*: AG $p = p \cap \mathrm{AX\ AG}\ p$
proof −
have *mono* $(\lambda s.\ p \cap \mathrm{AX}\ s)$ **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*: AG $p \subseteq p$
proof −
note *AG-fp* **also have** $p \cap \mathrm{AX\ AG}\ p \subseteq p$ **by** *auto*
finally show *?thesis* **.**

qed

lemma *AG-fp-2*: AG $p \subseteq$ AX AG p
proof $-$
  note *AG-fp* **also have** $p \cap$ AX AG $p \subseteq$ AX 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 $\lambda s.\ p \cap$ AX s is smaller than AG p. A post-fixed point is a set of states q such that $q \subseteq p \cap$ AX q. This leads to the following co-induction principle for AG p.

lemma *AG-I*: $q \subseteq p \cap$ AX $q \Longrightarrow q \subseteq$ AG p
  by (*simp only*: *AG-gfp*) (*rule gfp-upperbound*)

21.2 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*: $p \subseteq q \Longrightarrow$ AX $p \subseteq$ AX q **by** *auto*
lemma *AG-mono*: $p \subseteq q \Longrightarrow$ 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*: $p \cap \text{AG } (p \to \text{AX } p) = \text{AG } p$
proof
  show $p \cap \text{AG } (p \to \text{AX } p) \subseteq \text{AG } p$ (**is** *?lhs* $\subseteq$ -)
  proof (*rule AG-I*)
    show *?lhs* $\subseteq p \cap \text{AX}$ *?lhs*
    proof
      show *?lhs* $\subseteq p$ **..**
      show *?lhs* $\subseteq$ AX *?lhs*
      proof −
        {
          have $\text{AG } (p \to \text{AX } p) \subseteq p \to \text{AX } p$ **by** (*rule AG-fp-1*)
          also have $p \cap p \to \text{AX } p \subseteq \text{AX } p$ **..**
          finally have *?lhs* $\subseteq \text{AX } p$ **by** *auto*
        }
        moreover
        {
          have $p \cap \text{AG } (p \to \text{AX } p) \subseteq \text{AG } (p \to \text{AX } p)$ **..**
          also have $\ldots \subseteq \text{AX} \ldots$ **by** (*rule AG-fp-2*)
          finally have *?lhs* $\subseteq \text{AX AG } (p \to \text{AX } p)$ **.**
        }
        ultimately have *?lhs* $\subseteq \text{AX } p \cap \text{AX AG } (p \to \text{AX } p)$ **..**
        also have $\ldots =$ AX *?lhs* **by** (*simp only*: *AX-int*)
        finally show *?thesis* **.**
      qed
    qed
  qed
next
  show $\text{AG } p \subseteq p \cap \text{AG } (p \to \text{AX } p)$
  proof
    show $\text{AG } p \subseteq p$ **by** (*rule AG-fp-1*)
    show $\text{AG } p \subseteq \text{AG } (p \to \text{AX } p)$
    proof −
      have $\text{AG } p = \text{AG AG } p$ **by** (*simp only*: *AG-AG*)
      also have $\text{AG } p \subseteq \text{AX } p$ **by** (*rule AG-AX*) **moreover note** *AG-mono*
      also have $\text{AX } p \subseteq (p \to \text{AX } p)$ **.. moreover note** *AG-mono*
      finally show *?thesis* **.**
    qed
  qed
qed

21.3 An application of tree induction

Further interesting properties of CTL expressions may be demonstrated with the help of tree induction; here we show that AX and AG commute.

theorem *AG-AX-commute*: AG AX p = AX AG p
proof –
  have AG AX p = AX p ∩ AX AG AX p **by** (*rule AG-fp*)
  also have ... = AX (p ∩ AG AX p) **by** (*simp only*: *AX-int*)
  also have p ∩ AG AX p = AG p (**is** *?lhs* = -)
  proof
    have AX p ⊆ p → AX p **..**
    also have p ∩ AG (p → AX p) = AG p **by** (*rule AG-induct*)
    also note *Int-mono AG-mono*
    ultimately show *?lhs* ⊆ AG p **by** *fast*
  next
    have AG p ⊆ p **by** (*rule AG-fp-1*)
    moreover
    {
      have AG p = AG AG p **by** (*simp only*: *AG-AG*)
      also have AG p ⊆ AX p **by** (*rule AG-AX*)
      also note *AG-mono*
      ultimately have AG p ⊆ AG AX p **.**
    }
    ultimately show AG p ⊆ *?lhs* **..**
  qed
  finally show *?thesis* **.**
qed

end

22 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.)

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

lemma (*i*::*nat*) <= *max i j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma (*i*::*int*) <= *max i j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma *min i j* <= (*i*::*nat*)
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma *min i j* <= (*i*::*int*)
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma *min* (*i*::*nat*) *j* <= *max i j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma *min* (*i*::*int*) *j* <= *max i j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma *min* (*i*::*nat*) *j* + *max i j* = *i* + *j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma *min* (*i*::*int*) *j* + *max i j* = *i* + *j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma (*i*::*nat*) < *j* ==> *min i j* < *max i j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma (*i*::*int*) < *j* ==> *min i j* < *max i j*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

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

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

lemma *abs* (*abs* (*i*::*int*)) = *abs i*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

Also testing subgoals with bound variables.

lemma !!*x*. (*x*::*nat*) <= *y* ==> *x* − *y* = *0*
  by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma !!*x*. (*x*::*nat*) − *y* = *0* ==> *x* <= *y*

by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma !!*x*. ((*x*::*nat*) <= *y*) = (*x* − *y* = *0*)
by (*tactic* ⟪ *simple-arith-tac* @{*context*} *1* ⟫)

lemma [| (*x*::*nat*) < *y*; *d* < *1* |] ==> *x* − *y* = *d*
by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

lemma [| (*x*::*nat*) < *y*; *d* < *1* |] ==> *x* − *y* − *x* = *d* − *x*
by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

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* ⟫)

22.2 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* ⟪ *simple-arith-tac* @{*context*} *1* ⟫)

22.3 Various Other Examples

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

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

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

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* ⟪ *simple-arith-tac* @{*context*} *1* ⟫)

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

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

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

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

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

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

lemma $[| (x::nat) \sim= y;\ x <= y |] ==> x < y$
  by $(tactic \langle\langle\ fast\text{-}arith\text{-}tac\ @\{context\}\ 1\ \rangle\rangle)$

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

lemma $(x - y) - (x::nat) = (x - x) - y$
  by $(tactic \langle\langle\ fast\text{-}arith\text{-}tac\ @\{context\}\ 1\ \rangle\rangle)$

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

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

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 $(tactic \langle\langle\ fast\text{-}arith\text{-}tac\ @\{context\}\ 1\ \rangle\rangle)$

lemma $(0::int) < 1$
  by $(tactic \langle\langle\ fast\text{-}arith\text{-}tac\ @\{context\}\ 1\ \rangle\rangle)$

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

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

Splitting of inequalities of different type.

```
lemma [| (a::nat) ~= b; (i::int) ~= j; a < 2; b < 2 |] ==>
  a + b <= nat (max (abs i) (abs j))
  by (tactic ⟪ fast-arith-tac @{context} 1 ⟫)
```

Again, but different order.

```
lemma [| (i::int) ~= j; (a::nat) ~= b; a < 2; b < 2 |] ==>
  a + b <= nat (max (abs i) (abs j))
  by (tactic ⟪ fast-arith-tac @{context} 1 ⟫)
```

```
end
```

23 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

24 Sorting: Basic Theory

theory *Sorting*
imports *Main Multiset*
begin

consts
  sorted1:: ($'a \Rightarrow 'a \Rightarrow bool$) ⇒ $'a$ *list* ⇒ *bool*
  sorted :: ($'a \Rightarrow 'a \Rightarrow bool$) ⇒ $'a$ *list* ⇒ *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*) = ((∀ *y* ∈ *set xs*. *le x y*) & *sorted le xs*)

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

```
definition
  transf :: ('a ⇒ 'a ⇒ bool) => bool where
   transf f = (∀ x y z. f x y & f y z --> f x z)
```

```
lemma sorted1-is-sorted: transf(le) ==> 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 & (∀ x ∈ set xs. ∀ y ∈ set ys. le x y))
  by (induct xs) auto

end
```

25 Merge Sort

```
theory MergeSort
imports Sorting
begin

consts merge :: ('a::linorder)list * 'a list ⇒ 'a list

recdef merge measure(%(xs,ys). size xs + size ys)
    merge(x#xs, y#ys) =
         (if x ≤ y then x # merge(xs, y#ys) else y # merge(x#xs, ys))

    merge(xs,[]) = xs

    merge([],ys) = ys

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

lemma set-merge[simp]: set(merge(xs,ys)) = set xs ∪ 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* $\leq$) (*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

26 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*]: $\forall n.\ k{=}f(n) \dashrightarrow n <= f(n)$
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

27 A lemma for Lagrange's theorem

theory *Lagrange* **imports** *Main* **begin**

This theory only contains a single theorem, which is a lemma in Lagrange's proof that every natural number is the sum of 4 squares. Its sole purpose is to demonstrate ordered rewriting for commutative rings.

The enterprising reader might consider proving all of Lagrange's theorem.

definition *sq* :: *'a*::*times* => *'a* **where** *sq x* == *x***x*

The following lemma essentially shows that every natural number is the sum of four squares, provided all prime numbers are. However, this is an abstract theorem about commutative rings. It has, a priori, nothing to do with nat.

ML-setup ⟪
Delsimprocs [*ab-group-add-cancel.sum-conv*, *ab-group-add-cancel.rel-conv*]
⟫

lemma *Lagrange-lemma*: **fixes** *x1* :: *'a*::*comm-ring* **shows**
(*sq x1* + *sq x2* + *sq x3* + *sq x4*) * (*sq y1* + *sq y2* + *sq y3* + *sq y4*) =
sq (*x1***y1* − *x2***y2* − *x3***y3* − *x4***y4*) +
sq (*x1***y2* + *x2***y1* + *x3***y4* − *x4***y3*) +
sq (*x1***y3* − *x2***y4* + *x3***y1* + *x4***y2*) +
sq (*x1***y4* + *x2***y3* − *x3***y2* + *x4***y1*)
by (*simp add*: *sq-def ring-simps*)

A challenge by John Harrison. Takes about 17s 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

28 Groebner Basis Examples

theory *Groebner-Examples*
imports *Groebner-Basis*
begin

28.1 Basic examples

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

lemma (*x* − (−*2*))ˆ*5* == *?X*::*int*
  by *sring-norm*

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

lemma ((−*3*) ˆ (*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*)ˆ*3* − *1* = (*x* − *z*)ˆ*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 $a * x\hat{}2 + b * x + c = (0::int)$ **and** $d * x\hat{}2 + e * x + f = 0$
  shows $d\hat{}2*c\hat{}2 - 2*d*c*a*f + a\hat{}2*f\hat{}2 - e*d*b*c - e*b*a*f + a*e\hat{}2*c + f*d*b\hat{}2 = 0$
  using *assms* **by** *algebra*

lemma $(x::int)\hat{}3 - x\hat{}2 - 5*x - 3 = 0 \longleftrightarrow (x = 3 \vee x = -1)$
  by *algebra*

theorem $x* (x^2 - x - 5) - 3 = (0::int) \longleftrightarrow (x = 3 \vee x = -1)$
  by *algebra*

lemma
  fixes $x::'a::\{idom,recpower,number\text{-}ring\}$
  shows $x\hat{}2*y = x\hat{}2$ & $x*y\hat{}2 = y\hat{}2 \longleftrightarrow x=1$ & $y=1 \mid x=0$ & $y=0$
  by *algebra*

28.2 Lemmas for Lagrange's theorem

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

lemma
  fixes $x1 :: 'a::\{idom,recpower,number\text{-}ring\}$
  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 $p1 :: 'a::\{idom,recpower,number\text{-}ring\}$
  shows
  $(sq\ p1 + sq\ q1 + sq\ r1 + sq\ s1 + sq\ t1 + sq\ u1 + sq\ v1 + sq\ w1) *$
   $(sq\ p2 + sq\ q2 + sq\ r2 + sq\ s2 + sq\ t2 + sq\ u2 + sq\ v2 + sq\ w2)$
   $= sq\ (p1*p2 - q1*q2 - r1*r2 - s1*s2 - t1*t2 - u1*u2 - v1*v2 - w1*w2)$
$+$
     $sq\ (p1*q2 + q1*p2 + r1*s2 - s1*r2 + t1*u2 - u1*t2 - v1*w2 + w1*v2)$
$+$

$sq\ (p1*r2 - q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 - v1*t2 - w1*u2)$
$+$
$sq\ (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)$
$+$
$sq\ (p1*t2 - q1*u2 - r1*v2 - s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)$
$+$
$sq\ (p1*u2 + q1*t2 - r1*w2 + s1*v2 - t1*q2 + u1*p2 - v1*s2 + w1*r2)$
$+$
$sq\ (p1*v2 + q1*w2 + r1*t2 - s1*u2 - t1*r2 + u1*s2 + v1*p2 - w1*q2)$
$+$
$sq\ (p1*w2 - q1*v2 + r1*u2 + s1*t2 - t1*s2 - u1*r2 + v1*q2 + w1*p2)$
by (*algebra add*: *sq-def*)

28.3 Colinearity is invariant by rotation

types *point* = *int* × *int*

definition *collinear* ::*point* ⇒ *point* ⇒ *point* ⇒ *bool* **where**
collinear ≡ λ(*Ax*,*Ay*) (*Bx*,*By*) (*Cx*,*Cy*).
((*Ax* − *Bx*) ∗ (*By* − *Cy*) = (*Ay* − *By*) ∗ (*Bx* − *Cx*))

lemma *collinear-inv-rotation*:
assumes *collinear* (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) **and** $c^2 + s^2 = 1$
shows *collinear* (*Ax* ∗ *c* − *Ay* ∗ *s*, *Ay* ∗ *c* + *Ax* ∗ *s*)
(*Bx* ∗ *c* − *By* ∗ *s*, *By* ∗ *c* + *Bx* ∗ *s*) (*Cx* ∗ *c* − *Cy* ∗ *s*, *Cy* ∗ *c* + *Cx* ∗ *s*)
using *assms*
by (*algebra add*: *collinear-def split-def fst-conv snd-conv*)

lemma *EX* (*d*::*int*). *a*∗*y* − *a*∗*x* = *n*∗*d* ⟹ *EX u v*. *a*∗*u* + *n*∗*v* = *1* ⟹ *EX e*. *y* − *x* = *n*∗*e*
apply *algebra*
done

end

29 Milner-Tofte: Co-induction in Relational Semantics

theory *MT*
imports *Main*
begin

typedecl *Const*

typedecl *ExVar*
typedecl *Ex*

```
typedecl TyConst
typedecl Ty

typedecl Clos
typedecl Val

typedecl ValEnv
typedecl TyEnv

consts
  c-app :: [Const, Const] => Const

  e-const :: Const => Ex
  e-var :: ExVar => Ex
  e-fn :: [ExVar, Ex] => Ex (fn - => - [0,51] 1000)
  e-fix :: [ExVar, ExVar, Ex] => Ex (fix - ( - ) = - [0,51,51] 1000)
  e-app :: [Ex, Ex] => Ex (- @@ - [51,51] 1000)
  e-const-fst :: Ex => Const

  t-const :: TyConst => Ty
  t-fun :: [Ty, Ty] => Ty (- -> - [51,51] 1000)

  v-const :: Const => Val
  v-clos :: Clos => Val

  ve-emp :: ValEnv
  ve-owr :: [ValEnv, ExVar, Val] => ValEnv (- + { - |-> - } [36,0,0] 50)
  ve-dom :: ValEnv => ExVar set
  ve-app :: [ValEnv, ExVar] => Val

  clos-mk :: [ExVar, Ex, ValEnv] => Clos (<| - , - , - |> [0,0,0] 1000)

  te-emp :: TyEnv
  te-owr :: [TyEnv, ExVar, Ty] => TyEnv (- + { - |=> - } [36,0,0] 50)
  te-app :: [TyEnv, ExVar] => Ty
  te-dom :: TyEnv => ExVar set

  eval-fun :: ((ValEnv * Ex) * Val) set => ((ValEnv * Ex) * Val) set
  eval-rel :: ((ValEnv * Ex) * Val) set
  eval :: [ValEnv, Ex, Val] => bool (- |- - ---> - [36,0,36] 50)

  elab-fun :: ((TyEnv * Ex) * Ty) set => ((TyEnv * Ex) * Ty) set
  elab-rel :: ((TyEnv * Ex) * Ty) set
  elab :: [TyEnv, Ex, Ty] => bool (- |- - ===> - [36,0,36] 50)

  isof :: [Const, Ty] => bool (- isof - [36,36] 50)
  isof-env :: [ValEnv,TyEnv] => bool (- isofenv -)

  hasty-fun :: (Val * Ty) set => (Val * Ty) set
```

hasty-rel :: (*Val* $*$ *Ty*) *set*
hasty :: [*Val*, *Ty*] => *bool* (- *hasty* - [*36*,*36*] *50*)
hasty-env :: [*ValEnv*,*TyEnv*] => *bool* (- *hastyenv* - [*36*,*36*] *35*)

axioms

e-const-inj: *e-const*(*c1*) = *e-const*(*c2*) ==> *c1* = *c2*
e-var-inj: *e-var*(*ev1*) = *e-var*(*ev2*) ==> *ev1* = *ev2*
e-fn-inj: *fn ev1* => *e1* = *fn ev2* => *e2* ==> *ev1* = *ev2* & *e1* = *e2*
e-fix-inj:
  fix ev11e(*v12*) = *e1* = *fix ev21*(*ev22*) = *e2* ==>
  ev11 = *ev21* & *ev12* = *ev22* & *e1* = *e2*

e-app-inj: *e11* @@ *e12* = *e21* @@ *e22* ==> *e11* = *e21* & *e12* = *e22*

e-disj-const-var: ~*e-const*(*c*) = *e-var*(*ev*)
e-disj-const-fn: ~*e-const*(*c*) = *fn ev* => *e*
e-disj-const-fix: ~*e-const*(*c*) = *fix ev1*(*ev2*) = *e*
e-disj-const-app: ~*e-const*(*c*) = *e1* @@ *e2*
e-disj-var-fn: ~*e-var*(*ev1*) = *fn ev2* => *e*
e-disj-var-fix: ~*e-var*(*ev*) = *fix ev1*(*ev2*) = *e*
e-disj-var-app: ~*e-var*(*ev*) = *e1* @@ *e2*
e-disj-fn-fix: ~*fn ev1* => *e1* = *fix ev21*(*ev22*) = *e2*
e-disj-fn-app: ~*fn ev1* => *e1* = *e21* @@ *e22*
e-disj-fix-app: ~*fix ev11*(*ev12*) = *e1* = *e21* @@ *e22*

e-ind:
  [| !!*ev*. *P*(*e-var*(*ev*));
     !!*c*. *P*(*e-const*(*c*));
     !!*ev e*. *P*(*e*) ==> *P*(*fn ev* => *e*);
     !!*ev1 ev2 e*. *P*(*e*) ==> *P*(*fix ev1*(*ev2*) = *e*);
     !!*e1 e2*. *P*(*e1*) ==> *P*(*e2*) ==> *P*(*e1* @@ *e2*)
  |] ==>
 P(*e*)

t-const-inj: *t-const*(*c1*) = *t-const*(*c2*) ==> *c1* = *c2* *t-fun-inj*: *t11* −> *t12* = *t21* −> *t22* ==> *t11* = *t21* & *t12* = *t22*

t-ind:
  [| !!*p*. *P*(*t-const p*); !!*t1 t2*. *P*(*t1*) ==> *P*(*t2*) ==> *P*(*t-fun t1 t2*) |]
  ==> *P*(*t*)

v-const-inj: *v-const*(*c1*) = *v-const*(*c2*) ==> *c1* = *c2*
v-clos-inj:
  v-clos(<|*ev1*,*e1*,*ve1*|>) = *v-clos*(<|*ev2*,*e2*,*ve2*|>) ==>
  ev1 = *ev2* & *e1* = *e2* & *ve1* = *ve2*

v-disj-const-clos: ~*v-const*(*c*) = *v-clos*(*cl*)

ve-dom-owr: *ve-dom*(*ve* + {*ev* |−> *v*}) = *ve-dom*(*ve*) *Un* {*ev*}

ve-app-owr1: *ve-app* (*ve* + {*ev* |−> *v*}) *ev*=*v*
ve-app-owr2: ~*ev1*=*ev2* ==> *ve-app* (*ve*+{*ev1* |−> *v*}) *ev2*=*ve-app ve ev2*

te-dom-owr: *te-dom*(*te* + {*ev* |=> *t*}) = *te-dom*(*te*) *Un* {*ev*}

te-app-owr1: *te-app* (*te* + {*ev* |=> *t*}) *ev*=*t*
te-app-owr2: ~*ev1*=*ev2* ==> *te-app* (*te*+{*ev1* |=> *t*}) *ev2*=*te-app te ev2*

defs
eval-fun-def:
  eval-fun(*s*) ==
  { *pp*.

```
    (? ve c. pp=((ve,e-const(c)),v-const(c))) |
    (? ve x. pp=((ve,e-var(x)),ve-app ve x) & x:ve-dom(ve)) |
    (? ve e x. pp=((ve,fn x => e),v-clos(<|x,e,ve|>)))|
    ( ? ve e x f cl.
        pp=((ve,fix f(x) = e),v-clos(cl)) &
        cl=<|x, e, ve+{f |-> v-clos(cl)} |>
    ) |
    ( ? ve e1 e2 c1 c2.
        pp=((ve,e1 @@ e2),v-const(c-app c1 c2)) &
        ((ve,e1),v-const(c1)):s & ((ve,e2),v-const(c2)):s
    ) |
    ( ? ve vem e1 e2 em xm v v2.
        pp=((ve,e1 @@ e2),v) &
        ((ve,e1),v-clos(<|xm,em,vem|>)):s &
        ((ve,e2),v2):s &
        ((vem+{xm |-> v2},em),v):s
    )
  }

eval-rel-def: eval-rel == lfp(eval-fun)
eval-def: ve |- e ---> v == ((ve,e),v):eval-rel


elab-fun-def:
elab-fun(s) ==
{ pp.
  (? te c t. pp=((te,e-const(c)),t) & c isof t) |
  (? te x. pp=((te,e-var(x)),te-app te x) & x:te-dom(te)) |
  (? te x e t1 t2. pp=((te,fn x => e),t1->t2) & ((te+{x |=> t1},e),t2):s) |
  (? te f x e t1 t2.
     pp=((te,fix f(x)=e),t1->t2) & ((te+{f |=> t1->t2}+{x |=> t1},e),t2):s
  ) |
  (? te e1 e2 t1 t2.
     pp=((te,e1 @@ e2),t2) & ((te,e1),t1->t2):s & ((te,e2),t1):s
  )
}

elab-rel-def: elab-rel == lfp(elab-fun)
elab-def: te |- e ===> t == ((te,e),t):elab-rel


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

)

axioms
isof-app: [| c1 isof t1->t2; c2 isof t1 |] ==> 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 *gfph*: *x*:*gfp*(*f*)
    and *monoh*: *mono*(*f*)
    and *caseh*: !!*y*. *y*:*f*(*gfp*(*f*)) ==> *P*(*y*)
  shows *P*(*x*)
apply (*rule caseh*)
apply (*rule subsetD*)
apply (*rule gfp-lemma2*)
apply (*rule monoh*)
apply (*rule gfph*)
done

lemmas *e-injs* = *e-const-inj e-var-inj e-fn-inj e-fix-inj e-app-inj*

lemmas *e-disjs* =
  e-disj-const-var
  e-disj-const-fn
  e-disj-const-fix
  e-disj-const-app
  e-disj-var-fn
  e-disj-var-fix
  e-disj-var-app
  e-disj-fn-fix
  e-disj-fn-app
  e-disj-fix-app

lemmas *e-disj-si* = *e-disjs* *e-disjs* [*symmetric*]

lemmas *e-disj-se* = *e-disj-si* [*THEN notE*]

lemmas *v-disjs* = *v-disj-const-clos*
lemmas *v-disj-si* = *v-disjs* *v-disjs* [*symmetric*]
lemmas *v-disj-se* = *v-disj-si* [*THEN notE*]

lemmas *v-injs* = *v-const-inj v-clos-inj*

lemma *eval-fun-mono*: *mono*(*eval-fun*)
unfolding *mono-def eval-fun-def*
apply (*tactic infsys-mono-tac*)
done

lemma *eval-const*: *ve* |− *e-const*(*c*) −−−> *v-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-dom*(*ve*) ==> *ve* |− *e-var*(*ev*) −−−> *ve-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 |− *fn ev* => *e* −−−> *v-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* |−> *v-clos*(*cl*)} |> ==>
ve |− *fix ev2*(*ev1*) = *e* −−−> *v-clos*(*cl*)
apply (*unfold eval-def eval-rel-def*)
apply (*rule lfp-intro2*)
apply (*rule eval-fun-mono*)
apply (*unfold eval-fun-def*)

```
apply (blast intro!: exI)
done

lemma eval-app1:
  [| ve |- e1 ---> v-const(c1); ve |- e2 ---> v-const(c2) |] ==>
   ve |- e1 @@ e2 ---> v-const(c-app c1 c2)
apply (unfold eval-def eval-rel-def)
apply (rule lfp-intro2)
apply (rule eval-fun-mono)
apply (unfold eval-fun-def)
apply (blast intro!: exI)
done

lemma eval-app2:
  [|  ve |- e1 ---> v-clos(<|xm,em,vem|>);
      ve |- e2 ---> v2;
      vem + {xm |-> v2} |- em ---> v
   |] ==>
   ve |- e1 @@ e2 ---> v
apply (unfold eval-def eval-rel-def)
apply (rule lfp-intro2)
apply (rule eval-fun-mono)
apply (unfold eval-fun-def)
apply (blast intro!: disjI2)
done


lemma eval-ind0:
  [| ve |- e ---> v;
     !!ve c. P(((ve,e-const(c)),v-const(c)));
     !!ev ve. ev:ve-dom(ve) ==> P(((ve,e-var(ev)),ve-app ve ev));
     !!ev ve e. P(((ve,fn ev => e),v-clos(<|ev,e,ve|>)));
     !!ev1 ev2 ve cl e.
       cl = <| ev1, e, ve + {ev2 |-> v-clos(cl)} |> ==>
       P(((ve,fix ev2(ev1) = e),v-clos(cl)));
     !!ve c1 c2 e1 e2.
       [| P(((ve,e1),v-const(c1))); P(((ve,e2),v-const(c2))) |] ==>
       P(((ve,e1 @@ e2),v-const(c-app c1 c2)));
     !!ve vem xm e1 e2 em v v2.
       [|  P(((ve,e1),v-clos(<|xm,em,vem|>)));
           P(((ve,e2),v2));
           P(((vem + {xm |-> v2},em),v))
       |] ==>
       P(((ve,e1 @@ e2),v))
   |] ==>
   P(((ve,e),v))
unfolding eval-def eval-rel-def
apply (erule lfp-ind2)
```

```
apply (rule eval-fun-mono)
apply (unfold eval-fun-def)
apply (drule CollectD)
apply safe
apply auto
done

lemma eval-ind:
  [| ve |- e ---> v;
     !!ve c. P ve (e-const c) (v-const c);
     !!ev ve. ev:ve-dom(ve) ==> P ve (e-var ev) (ve-app ve ev);
     !!ev ve e. P ve (fn ev => e) (v-clos <|ev,e,ve|>);
     !!ev1 ev2 ve cl e.
       cl = <| ev1, e, ve + {ev2 |-> v-clos(cl)} |> ==>
       P ve (fix ev2(ev1) = e) (v-clos cl);
     !!ve c1 c2 e1 e2.
       [| P ve e1 (v-const c1); P ve e2 (v-const c2) |] ==>
       P ve (e1 @@ e2) (v-const(c-app c1 c2));
     !!ve vem evm e1 e2 em v v2.
       [|  P ve e1 (v-clos <|evm,em,vem|>);
           P ve e2 v2;
           P (vem + {evm |-> v2}) em v
       |] ==> P ve (e1 @@ e2) v
   |] ==> P ve e v
apply (rule-tac P = P in infsys-pp2)
apply (rule eval-ind0)
apply (rule infsys-pp1)
apply auto
done

lemma elab-fun-mono: mono(elab-fun)
unfolding mono-def elab-fun-def
apply (tactic infsys-mono-tac)
done

lemma elab-const:
  c isof ty ==> te |- e-const(c) ===> ty
apply (unfold elab-def elab-rel-def)
apply (rule lfp-intro2)
apply (rule elab-fun-mono)
apply (unfold elab-fun-def)
apply (blast intro!: exI)
done
```

lemma *elab-var*:
  x:*te-dom*(*te*) ==> *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* |− *e-var*(*ev*) ===> *t* ==> *t*=*te-app te ev* & *ev* : *te-dom*(*te*)
apply (*drule elab-var-elim-lem*)
apply *blast*
done

lemma *elab-fn-elim-lem*:
te |− *e* ===> *t* ==>
(*e* = *fn x1* => *e1* −−>
(*? t1 t2*. *t*=*t-fun t1 t2* & *te* + {*x1* |=> *t1*} |− *e1* ===> *t2*)
)
apply (*erule elab-elim*)
apply (*fast intro*!: *e-disj-si elim*!: *e-disj-se dest*!: *e-injs*)+
done

lemma *elab-fn-elim*: *te* |− *fn x1* => *e1* ===> *t* ==>
(*? t1 t2*. *t*=*t1*−>*t2* & *te* + {*x1* |=> *t1*} |− *e1* ===> *t2*)
apply (*drule elab-fn-elim-lem*)
apply *blast*
done

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

lemma *elab-fix-elim*: *te* |− *fix ev1*(*ev2*) = *e1* ===> *t* ==>
(*? t1 t2*. *t*=*t1*−>*t2* & *te* + {*ev1* |=> *t1*−>*t2*} + {*ev2* |=> *t1*} |− *e1* ===> *t2*)
apply (*drule elab-fix-elim-lem*)
apply *blast*
done

```
lemma elab-app-elim-lem:
  te |- e ===> t2 ==>
   (e = e1 @@ e2 --> (? t1 . te |- e1 ===> t1->t2 & te |- e2 ===> t1))
apply (erule elab-elim)
apply (fast intro!: e-disj-si elim!: e-disj-se dest!: e-injs)+
done

lemma elab-app-elim: te |- e1 @@ e2 ===> t2 ==> (? t1 . te |- e1 ===>
t1->t2 & te |- e2 ===> t1)
apply (drule elab-app-elim-lem)
apply blast
done

lemma mono-hasty-fun: mono(hasty-fun)
unfolding mono-def hasty-fun-def
apply (tactic infsys-mono-tac)
apply blast
done

lemma hasty-rel-const-coind: c isof t ==> (v-const(c),t) : hasty-rel
apply (unfold hasty-rel-def)
apply (rule gfp-coind2)
apply (unfold hasty-fun-def)
apply (rule CollectI)
apply (rule disjI1)
apply blast
apply (rule mono-hasty-fun)
done

lemma hasty-rel-clos-coind:
  [|  te |- fn ev => e ===> t;
      ve-dom(ve) = te-dom(te);
      ! ev1.
        ev1:ve-dom(ve) -->
        (ve-app ve ev1,te-app te ev1) : {(v-clos(<|ev,e,ve|>),t)} Un hasty-rel
   |] ==>
```

```
  (v-clos(<|ev,e,ve|>),t) : hasty-rel
apply (unfold hasty-rel-def)
apply (rule gfp-coind2)
apply (unfold hasty-fun-def)
apply (rule CollectI)
apply (rule disjI2)
apply blast
apply (rule mono-hasty-fun)
done
```

```
lemma hasty-rel-elim0:
  [| !! c t. c isof t ==> P((v-const(c),t));
     !! te ev e t ve.
       [| te |- fn ev => e ===> t;
          ve-dom(ve) = te-dom(te);
          !ev1. ev1:ve-dom(ve) --> (ve-app ve ev1,te-app te ev1) : hasty-rel
       |] ==> P((v-clos(<|ev,e,ve|>),t));
     (v,t) : hasty-rel
  |] ==> P(v,t)
unfolding hasty-rel-def
apply (erule gfp-elim2)
apply (rule mono-hasty-fun)
apply (unfold hasty-fun-def)
apply (drule CollectD)
apply (fold hasty-fun-def)
apply auto
done
```

```
lemma hasty-rel-elim:
  [| (v,t) : hasty-rel;
     !! c t. c isof t ==> P (v-const c) t;
     !! te ev e t ve.
       [| te |- fn ev => e ===> t;
          ve-dom(ve) = te-dom(te);
          !ev1. ev1:ve-dom(ve) --> (ve-app ve ev1,te-app te ev1) : hasty-rel
       |] ==> P (v-clos <|ev,e,ve|>) t
  |] ==> P v t
apply (rule-tac P = P in infsys-p2)
apply (rule hasty-rel-elim0)
apply auto
done
```

```
lemma hasty-const: c 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 hastyenv 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 hasty t ==> (!c.(v = v-const(c) --> c 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-const(c) hasty t ==> c isof t
apply (drule hasty-elim-const-lem)
apply blast
done



lemma hasty-elim-clos-lem:
   v hasty t ==>
    ! x e ve.
      v=v-clos(<|x,e,ve|>) --> (? te. te |- fn x => e ===> t & ve 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-clos(<|ev,e,ve|>) hasty t ==>
        ? te. te |- fn ev => e ===> t & ve hastyenv te
apply (drule hasty-elim-clos-lem)
apply blast
done




lemma hasty-env1: [| ve hastyenv te; v hasty t |] ==>
         ve + {ev |-> v} hastyenv te + {ev |=> t}
apply (unfold hasty-env-def)
apply (simp del: mem-simps add: ve-dom-owr te-dom-owr)
```

apply (*tactic* ⟪ *safe-tac HOL-cs* ⟫)
apply (*case-tac ev=x*)
apply (*simp* (*no-asm-simp*) *add*: *ve-app-owr1 te-app-owr1*)
apply (*simp add*: *ve-app-owr2 te-app-owr2*)
done

lemma *consistency-const*: [| *ve hastyenv te* ; *te* |− *e-const*(*c*) ===> *t* |] ==> *v-const*(*c*) *hasty t*
apply (*drule elab-const-elim*)
apply (*erule hasty-const*)
done

lemma *consistency-var*:
[| *ev* : *ve-dom*(*ve*); *ve hastyenv te* ; *te* |− *e-var*(*ev*) ===> *t* |] ==>
ve-app ve ev hasty t
apply (*unfold hasty-env-def*)
apply (*drule elab-var-elim*)
apply *blast*
done

lemma *consistency-fn*: [| *ve hastyenv te* ; *te* |− *fn ev* => *e* ===> *t* |] ==>
v-clos(<| *ev*, *e*, *ve* |>) *hasty t*
apply (*rule hasty-clos*)
apply *blast*
done

lemma *consistency-fix*:
[| *cl* = <| *ev1*, *e*, *ve* + { *ev2* |−> *v-clos*(*cl*) } |>;
ve hastyenv te ;
te |− *fix ev2 ev1* = *e* ===> *t*
|] ==>
v-clos(*cl*) *hasty t*
apply (*unfold hasty-env-def hasty-def*)
apply (*drule elab-fix-elim*)
apply (*tactic* ⟪ *safe-tac HOL-cs* ⟫)

apply (*frule ssubst*) **prefer** *2* **apply** *assumption*
apply (*rule hasty-rel-clos-coind*)
apply (*erule elab-fn*)
apply (*simp* (*no-asm-simp*) *add*: *ve-dom-owr te-dom-owr*)

apply (*simp* (*no-asm-simp*) *del*: *mem-simps add*: *ve-dom-owr*)
apply (*tactic* ⟪ *safe-tac HOL-cs* ⟫)
apply (*case-tac ev2=ev1a*)
apply (*simp* (*no-asm-simp*) *del*: *mem-simps add*: *ve-app-owr1 te-app-owr1*)

```
apply blast
apply (simp add: ve-app-owr2 te-app-owr2)
done

lemma consistency-app1: [| ! t te. ve hastyenv te --> te |- e1 ===> t -->
v-const(c1) hasty t;
      ! t te. ve hastyenv te  --> te |- e2 ===> t --> v-const(c2) hasty t;
      ve hastyenv te ; te |- e1 @@ e2 ===> t
   |] ==>
   v-const(c-app c1 c2) hasty t
apply (drule elab-app-elim)
apply safe
apply (rule hasty-const)
apply (rule isof-app)
apply (rule hasty-elim-const)
apply blast
apply (rule hasty-elim-const)
apply blast
done

lemma consistency-app2: [| ! t te.
         ve hastyenv te  -->
         te |- e1 ===> t --> v-clos(<|evm, em, vem|>) hasty t;
      ! t te. ve hastyenv te  --> te |- e2 ===> t --> v2 hasty t;
      ! t te.
        vem + { evm |-> v2 } hastyenv te  --> te |- em ===> t --> v hasty
t;
      ve hastyenv te ;
      te |- e1 @@ e2 ===> t
   |] ==>
   v hasty t
apply (drule elab-app-elim)
apply safe
apply (erule allE, erule allE, erule impE)
apply assumption
apply (erule impE)
apply assumption
apply (erule allE, erule allE, erule impE)
apply assumption
apply (erule impE)
apply assumption
apply (drule hasty-elim-clos)
apply safe
apply (drule elab-fn-elim)
apply (blast intro: hasty-env1 dest!: t-fun-inj)
done

lemma consistency: ve |- 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 (*drule 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 (*drule consistency*)
apply (*blast intro*!: *basic-consistency-lem*)
done

end

30 Case study: Unification Algorithm

theory *Unification*
imports *Main*
begin

This is a formalization of a first-order unification algorithm. It uses the new "function" package to define recursive functions, which allows a better treatment of nested recursion.

This is basically a modernized version of a previous formalization by Konrad Slind (see: HOL/Subst/Unify.thy), which itself builds on previous work by Paulson and Manna & Waldinger (for details, see there).

Unlike that formalization, where the proofs of termination and some partial correctness properties are intertwined, we can prove partial correctness and termination separately.

30.1 Basic definitions

datatype $'a\ trm =$
  $Var\ 'a$
  $|\ Const\ 'a$
  $|\ App\ 'a\ trm\ 'a\ trm$ (**infix** $\cdot$ 60)

types
  $'a\ subst = ('a \times 'a\ trm)\ list$

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 $apply\text{-}subst :: 'a\ trm \Rightarrow 'a\ subst \Rightarrow 'a\ trm$ (**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** $\bullet$ 80)
where
  $[] \bullet bl = bl$
$|\ ((a,b)\ \#\ al) \bullet bl = (a,\ b \triangleleft bl)\ \#\ (al \bullet bl)$

Equivalence of substitutions:

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

30.2 Basic lemmas

lemma $apply\text{-}empty[simp]$: $t \triangleleft [] = t$
by $(induct\ t)\ auto$

lemma $compose\text{-}empty[simp]$: $\sigma \bullet [] = \sigma$
by $(induct\ \sigma)\ auto$

lemma $apply\text{-}compose[simp]$: $t \triangleleft (s1 \bullet s2) = t \triangleleft s1 \triangleleft s2$

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*]: $[\![s1 =_s s2; s2 =_s s3]\!] \Longrightarrow s1 =_s s3$
  by (*auto simp*:*eqv-def*)

lemma *eqv-sym*[*sym*]: $[\![s1 =_s s2]\!] \Longrightarrow s2 =_s s1$
  by (*auto simp*:*eqv-def*)

lemma *eqv-intro*[*intro*]: $(\bigwedge t.\ t \lhd \sigma = t \lhd \vartheta) \Longrightarrow \sigma =_s \vartheta$
  by (*auto simp*:*eqv-def*)

lemma *eqv-dest*[*dest*]: $s1 =_s s2 \Longrightarrow t \lhd s1 = t \lhd s2$
  by (*auto simp*:*eqv-def*)

lemma *compose-eqv*: $[\![\sigma =_s \sigma'; \vartheta =_s \vartheta']\!] \Longrightarrow (\sigma \bullet \vartheta) =_s (\sigma' \bullet \vartheta')$
  by (*auto simp*:*eqv-def*)

lemma *compose-assoc*: $(a \bullet b) \bullet c =_s a \bullet (b \bullet c)$
  by *auto*

30.3 Specification: Most general unifiers

definition
  $Unifier\ \sigma\ t\ u \equiv (t \lhd \sigma = u \lhd \sigma)$

definition
  $MGU\ \sigma\ t\ u \equiv Unifier\ \sigma\ t\ u \wedge (\forall \vartheta.\ Unifier\ \vartheta\ t\ u$
  $\longrightarrow (\exists \gamma.\ \vartheta =_s \sigma \bullet \gamma))$

lemma *MGUI*[*intro*]:
  $[\![t \lhd \sigma = u \lhd \sigma; \bigwedge\vartheta.\ t \lhd \vartheta = u \lhd \vartheta \Longrightarrow \exists \gamma.\ \vartheta =_s \sigma \bullet \gamma]\!]$
  $\Longrightarrow MGU\ \sigma\ t\ u$
  by (*simp only*:*Unifier-def MGU-def*, *auto*)

lemma *MGU-sym*[*sym*]:

```
  MGU σ s t ⟹ MGU σ t s
  by (auto simp:MGU-def Unifier-def)
```

30.4 The unification algorithm

Occurs check: Proper subterm relation

```
fun occ :: 'a trm ⇒ 'a trm ⇒ bool
where
  occ u (Var v) = False
| occ u (Const c) = False
| occ u (M · N) = (u = M ∨ u = N ∨ occ u M ∨ occ u N)
```

The unification algorithm:

```
function unify :: 'a trm ⇒ 'a trm ⇒ 'a subst option
where
  unify (Const c) (M · N)   = None
| unify (M · N)   (Const c) = None
| unify (Const c) (Var v)   = Some [(v, Const c)]
| unify (M · N)   (Var v)   = (if (occ (Var v) (M · N))
                                then None
                                else Some [(v, M · 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 · N) (M' · N') = (case unify M M' of
                              None ⇒ None |
                              Some ϑ ⇒ (case unify (N ◃ ϑ) (N' ◃ ϑ)
                                of None ⇒ None |
                                  Some σ ⇒ Some (ϑ • σ)))
  by pat-completeness auto
```

30.5 Partial correctness

Some lemmas about occ and MGU:

```
lemma subst-no-occ: ¬occ (Var v) t ⟹ Var v ≠ t
  ⟹ t ◃ [(v,s)] = t
by (induct t) auto

lemma MGU-Var[intro]:
  assumes no-occ: ¬occ (Var v) t
  shows MGU [(v,t)] (Var v) t
proof (intro MGUI exI)
  show Var v ◃ [(v,t)] = t ◃ [(v,t)] using no-occ
    by (cases Var v = t, auto simp:subst-no-occ)
next
  fix ϑ assume th: Var v ◃ ϑ = t ◃ ϑ
  show ϑ =s [(v,t)] • ϑ
```

proof
fix s **show** $s \lhd \vartheta = s \lhd [(v,t)] \bullet \vartheta$ **using** th
by $(induct\ s)\ auto$
qed
qed

declare $MGU\text{-}Var[symmetric,\ intro]$

lemma $MGU\text{-}Const[simp]$: $MGU\ []\ (Const\ c)\ (Const\ d) = (c = d)$
unfolding $MGU\text{-}def\ Unifier\text{-}def$
by $auto$

If unification terminates, then it computes most general unifiers:

lemma *unify-partial-correctness*:
assumes $unify\text{-}dom\ (M,\ N)$
assumes $unify\ M\ N = Some\ \sigma$
shows $MGU\ \sigma\ M\ N$
using $assms$
proof $(induct\ M\ N\ arbitrary:\ \sigma)$
case $(7\ M\ N\ M'\ N'\ \sigma)$ — The interesting case

then obtain $\vartheta 1\ \vartheta 2$
where $unify\ M\ M' = Some\ \vartheta 1$
and $unify\ (N \lhd \vartheta 1)\ (N' \lhd \vartheta 1) = Some\ \vartheta 2$
and σ: $\sigma = \vartheta 1 \bullet \vartheta 2$
and $MGU\text{-}inner$: $MGU\ \vartheta 1\ M\ M'$
and $MGU\text{-}outer$: $MGU\ \vartheta 2\ (N \lhd \vartheta 1)\ (N' \lhd \vartheta 1)$
by $(auto\ split{:}option.split\text{-}asm)$

show $?case$
proof
from $MGU\text{-}inner$ **and** $MGU\text{-}outer$
have $M \lhd \vartheta 1 = M' \lhd \vartheta 1$
and $N \lhd \vartheta 1 \lhd \vartheta 2 = N' \lhd \vartheta 1 \lhd \vartheta 2$
unfolding $MGU\text{-}def\ Unifier\text{-}def$
by $auto$
thus $M \cdot N \lhd \sigma = M' \cdot N' \lhd \sigma$ **unfolding** σ
by $simp$
next
fix σ' **assume** $M \cdot N \lhd \sigma' = M' \cdot N' \lhd \sigma'$
hence $M \lhd \sigma' = M' \lhd \sigma'$
and Ns: $N \lhd \sigma' = N' \lhd \sigma'$ **by** $auto$

with $MGU\text{-}inner$ **obtain** δ
where eqv: $\sigma' =_s \vartheta 1 \bullet \delta$
unfolding $MGU\text{-}def\ Unifier\text{-}def$
by $auto$

from Ns **have** $N \lhd \vartheta 1 \lhd \delta = N' \lhd \vartheta 1 \lhd \delta$

by $(simp\ add{:}eqv\text{-}dest[OF\ eqv])$

with *MGU-outer* **obtain** ϱ
where $eqv2{:}\ \delta =_s \vartheta 2 \cdot \varrho$
unfolding *MGU-def Unifier-def*
by *auto*

have $\sigma' =_s \sigma \cdot \varrho$ **unfolding** σ
by $(rule\ eqv\text{-}intro,\ auto\ simp{:}eqv\text{-}dest[OF\ eqv]$
$eqv\text{-}dest[OF\ eqv2])$
thus $\exists \gamma.\ \sigma' =_s \sigma \cdot \gamma$ **..**
qed
qed $(auto\ split{:}split\text{-}if\text{-}asm)$ — Solve the remaining cases automatically

30.6 Properties used in termination proof

The variables of a term:

fun $vars\text{-}of :: 'a\ trm \Rightarrow 'a\ set$
where
$vars\text{-}of\ (Var\ v) = \{\ v\ \}$
$|\ vars\text{-}of\ (Const\ c) = \{\}$
$|\ vars\text{-}of\ (M \cdot N) = vars\text{-}of\ M \cup vars\text{-}of\ N$

lemma $vars\text{-}of\text{-}finite[intro]{:}\ finite\ (vars\text{-}of\ t)$
by $(induct\ t)\ simp\text{-}all$

Elimination of variables by a substitution:

definition
$elim\ \sigma\ v \equiv \forall t.\ v \notin vars\text{-}of\ (t \triangleleft \sigma)$

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

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

lemma $elim\text{-}eqv{:}\ \sigma =_s \vartheta \Longrightarrow elim\ \sigma\ x = elim\ \vartheta\ x$
by $(auto\ simp{:}elim\text{-}def\ eqv\text{-}def)$

Replacing a variable by itself yields an identity subtitution:

lemma $var\text{-}self[intro]{:}\ [(v,\ Var\ v)] =_s []$
proof
fix t **show** $t \triangleleft [(v,\ Var\ v)] = t \triangleleft []$
by $(induct\ t)\ simp\text{-}all$
qed

lemma $var\text{-}same{:}\ (t = Var\ v) = ([(v,\ t)] =_s [])$
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 ◁ [(v, t)] by simp
    also from id have ... = Var v ◁ [] ..
    finally show ?thesis by simp
  qed
qed
```

A lemma about occ and elim

```
lemma remove-var:
  assumes [simp]: v ∉ vars-of s
  shows v ∉ vars-of (t ◁ [(v, s)])
  by (induct t) simp-all

lemma occ-elim: ¬occ (Var v) t
  ⟹ elim [(v,t)] v ∨ [(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 neq: v ≠ x
    have elim [(v, Var x)] v
      by (auto intro!:remove-var simp:neq)
    thus ?thesis ..
  qed
next
  case (Const c)
  have elim [(v, Const c)] v
    by (auto intro!:remove-var)
  thus ?case ..
next
  case (App M N)

  hence ih1: elim [(v, M)] v ∨ [(v, M)] =_s []
    and ih2: elim [(v, N)] v ∨ [(v, N)] =_s []
    and nonocc: Var v ≠ M Var v ≠ N
    by auto

  from nonocc have ¬ [(v,M)] =_s []
    by (simp add:var-same[symmetric])
```

```
  with ih1 have elim [(v, M)] v by blast
  hence v ∉ vars-of (Var v ◁ [(v,M)]) ..
  hence not-in-M: v ∉ vars-of M by simp

  from nonocc have ¬ [(v,N)] =s []
    by (simp add:var-same[symmetric])
  with ih2 have elim [(v, N)] v by blast
  hence v ∉ vars-of (Var v ◁ [(v,N)]) ..
  hence not-in-N: v ∉ vars-of N by simp

  have elim [(v, M · N)] v
  proof
    fix t
    show v ∉ vars-of (t ◁ [(v, M · 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 σ
  shows vars-of (t ◁ σ) ⊆ vars-of M ∪ vars-of N ∪ vars-of t
  (is ?P M N σ t)
using assms
proof (induct M N arbitrary:σ t)
  case (3 c v)
  hence σ = [(v, Const c)] by simp
  thus ?case by (induct t) auto
next
  case (4 M N v)
  hence ¬occ (Var v) (M·N) by (cases occ (Var v) (M·N), auto)
  with 4 have σ = [(v, M·N)] by simp
  thus ?case by (induct t) auto
next
  case (5 v M)
  hence ¬occ (Var v) M by (cases occ (Var v) M, auto)
  with 5 have σ = [(v, M)] by simp
  thus ?case by (induct t) auto
next
  case (7 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: ⋀t. ?P M M' ϑ1 t
```

```
    and ih2: ⋀t. ?P (N◁ϑ1) (N'◁ϑ1) ϑ2 t
    by (auto split:option.split-asm)

  show ?case
  proof
    fix v assume a: v ∈ vars-of (t ◁ σ)

    show v ∈ vars-of (M · N) ∪ vars-of (M' · N') ∪ vars-of t
    proof (cases v ∉ vars-of M ∧ v ∉ vars-of M'
            ∧ v ∉ vars-of N ∧ v ∉ vars-of N')
      case True
      with ih1 have l:⋀t. v ∈ vars-of (t ◁ ϑ1) ⟹ v ∈ vars-of t
            by auto

      from a and ih2[where t=t ◁ ϑ1]
      have v ∈ vars-of (N ◁ ϑ1) ∪ vars-of (N' ◁ ϑ1)
        ∨ v ∈ vars-of (t ◁ ϑ1) unfolding σ
            by auto
      hence v ∈ vars-of t
      proof
            assume v ∈ vars-of (N ◁ ϑ1) ∪ vars-of (N' ◁ ϑ1)
            with True show ?thesis by (auto dest:l)
      next
            assume v ∈ vars-of (t ◁ ϑ1)
            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 = Some σ
  shows (∃ v∈vars-of M ∪ vars-of N. elim σ v) ∨ σ =s []
  (is ?P M N σ)
using assms
proof (induct M N arbitrary:σ)
  case 1 thus ?case by simp
next
  case 2 thus ?case by simp
next
  case (3 c v)
  have no-occ: ¬ occ (Var v) (Const c) by simp
  with 3 have σ = [(v, Const c)] by simp
  with occ-elim[OF no-occ]
```

```
  show ?case by auto
next
  case (4 M N v)
  hence no-occ: ¬occ (Var v) (M·N) by (cases occ (Var v) (M·N), auto)
  with 4 have σ = [(v, M·N)] by simp
  with occ-elim[OF no-occ]
  show ?case by auto
next
  case (5 v M)
  hence no-occ: ¬occ (Var v) M by (cases occ (Var v) M, auto)
  with 5 have σ = [(v, M)] by simp
  with occ-elim[OF no-occ]
  show ?case by auto
next
  case (6 c d) thus ?case
    by (cases c = d) auto
next
  case (7 M N M′ N′ 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-eqv) 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-eqv[OF ‹σ =s ϑ1›]
      show ?thesis by auto
    next
      note ‹σ =s ϑ1›
      also assume ϑ1 =s []
      finally show ?thesis ..
    qed
  qed
qed
```

30.7 Termination proof

```
termination unify
proof
  let ?R = measures [λ(M,N). card (vars-of M ∪ vars-of N),
                      λ(M, N). size M]
  show wf ?R by simp

  fix M N M' N'
  show ((M, M'), (M · N, M' · N')) ∈ ?R — Inner call
    by (rule measures-lesseq) (auto intro: card-mono)

  fix ϑ                                   — Outer call
  assume inner: unify-dom (M, M')
    unify M M' = Some ϑ

  from unify-eliminates[OF inner]
  show ((N ◁ ϑ, N' ◁ ϑ), (M · N, M' · N')) ∈?R
  proof
    — Either a variable is eliminated ...
    assume (∃ v∈vars-of M ∪ vars-of M'. elim ϑ v)
    then obtain v
          where elim ϑ v
          and v∈vars-of M ∪ vars-of M' by auto
    with unify-vars[OF inner]
    have vars-of (N◁ϑ) ∪ vars-of (N'◁ϑ)
          ⊂ vars-of (M·N) ∪ vars-of (M'·N')
          by auto

    thus ?thesis
      by (auto intro!: measures-less intro: psubset-card-mono)
  next
    — Or the substitution is empty
    assume ϑ =s []
    hence N ◁ ϑ = N
          and N' ◁ ϑ = N' by auto
```

```
    thus ?thesis
      by (auto intro!: measures-less intro: psubset-card-mono)
  qed
qed

end
```

31 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 $(a::int)\hat{}4 - b\hat{}4 = (a - b) * (a + b)*(a\hat{}2 + b\hat{}2)$
by *comm-ring*

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

end

32 Small examples for evaluation mechanisms

theory *Eval-Examples*
imports *Eval* ~~/*src*/*HOL*/*Real*/*Rational*
begin

evaluation oracle

lemma *True* ∨ *False* **by** *eval*
lemma ¬ (*Suc 0* = *Suc 1*) **by** *eval*
lemma [] = ([]:: *int list*) **by** *eval*
lemma [()] = [()] **by** *eval*
lemma *fst* ([]::*nat list*, *Suc 0*) = [] **by** *eval*

SML evaluation oracle

lemma *True* ∨ *False* **by** *evaluation*
lemma ¬ (*Suc 0* = *Suc 1*) **by** *evaluation*
lemma [] = ([]:: *int list*) **by** *evaluation*
lemma [()] = [()] **by** *evaluation*
lemma *fst* ([]::*nat list*, *Suc 0*) = [] **by** *evaluation*

normalization

lemma *True* ∨ *False* **by** *normalization*
lemma ¬ (*Suc 0* = *Suc 1*) **by** *normalization*
lemma [] = ([]:: *int list*) **by** *normalization*
lemma [()] = [()] **by** *normalization*
lemma *fst* ([]::*nat list*, *Suc 0*) = [] **by** *normalization*

term evaluation

value (*Suc 2* + *1*) ∗ *4*
value (*code*) (*Suc 2* + *1*) ∗ *4*
value (*SML*) (*Suc 2* + *1*) ∗ *4*
value (*normal-form*) (*Suc 2* + *1*) ∗ *4*

value (*Suc 2* + *Suc 0*) ∗ *Suc 3*
value (*code*) (*Suc 2* + *Suc 0*) ∗ *Suc 3*
value (*SML*) (*Suc 2* + *Suc 0*) ∗ *Suc 3*
value (*normal-form*) (*Suc 2* + *Suc 0*) ∗ *Suc 3*

value *nat 100*
value (*code*) *nat 100*
value (*SML*) *nat 100*
value (*normal-form*) *nat 100*

value (*10*::*int*) ≤ *12*
value (*code*) (*10*::*int*) ≤ *12*
value (*SML*) (*10*::*int*) ≤ *12*
value (*normal-form*) (*10*::*int*) ≤ *12*

value *max* (*2*::*int*) *4*
value (*code*) *max* (*2*::*int*) *4*
value (*SML*) *max* (*2*::*int*) *4*
value (*normal-form*) *max* (*2*::*int*) *4*

value *of-int 2* / *of-int 4* ∗ (*1*::*rat*)

value (*SML*) *of-int 2* / *of-int 4* ∗ (*1*::*rat*)
value (*normal-form*) *of-int 2* / *of-int 4* ∗ (*1*::*rat*)

value []::*nat list*
value (*code*) []::*nat list*
value (*SML*) []::*nat list*
value (*normal-form*) []::*nat list*

value [(*nat 100*, ())]
value (*code*) [(*nat 100*, ())]
value (*SML*) [(*nat 100*, ())]
value (*normal-form*) [(*nat 100*, ())]

a fancy datatype

datatype (*'a*, *'b*) *bair* =
  Bair 'a::*order 'b*
  | *Shift* (*'a*, *'b*) *cair*
  | *Dummy unit*
and (*'a*, *'b*) *cair* =
  Cair 'a 'b

value *Shift* (*Cair* (*4*::*nat*) [*Suc 0*])
value (*code*) *Shift* (*Cair* (*4*::*nat*) [*Suc 0*])
value (*SML*) *Shift* (*Cair* (*4*::*nat*) [*Suc 0*])
value (*normal-form*) *Shift* (*Cair* (*4*::*nat*) [*Suc 0*])

end

33 A simple random engine

theory *Random*

```
imports State-Monad Code-Integer
begin

fun
  pick :: (nat × 'a) list ⇒ nat ⇒ 'a
where
  pick-undef: pick [] n = undefined
  | pick-simp: pick ((k, v)#xs) n = (if n < k then v else pick xs (n − k))
lemmas [code func del] = pick-undef

typedecl randseed

axiomatization
  random-shift :: randseed ⇒ randseed

axiomatization
  random-seed :: randseed ⇒ nat

definition
  random :: nat ⇒ randseed ⇒ nat × randseed where
  random n s = (random-seed s mod n, random-shift s)

lemma random-bound:
  assumes 0 < n
  shows fst (random n s) < n
proof −
  from prems mod-less-divisor have !!m .m mod n < n by auto
  then show ?thesis unfolding random-def by simp
qed

lemma random-random-seed [simp]:
  snd (random n s) = random-shift s unfolding random-def by simp

definition
  select :: 'a list ⇒ randseed ⇒ 'a × randseed where
  [simp]: select xs = (do
      n ← random (length xs);
      return (nth xs n)
    done)
definition
  select-weight :: (nat × 'a) list ⇒ randseed ⇒ 'a × randseed where
  [simp]: select-weight xs = (do
      n ← random (foldl (op +) 0 (map fst xs));
      return (pick xs n)
    done)

lemma
  select (x#xs) s = select-weight (map (Pair 1) (x#xs)) s
proof (induct xs)
```

```
  case Nil show ?case by (simp add: monad-collapse random-def)
next
  have map-fst-Pair: !!xs y. map fst (map (Pair y) xs) = replicate (length xs) y
  proof -
    fix xs
    fix y
    show map fst (map (Pair y) xs) = replicate (length xs) y
      by (induct xs) simp-all
  qed
  have pick-nth: !!xs n. n < length xs ==> pick (map (Pair 1) xs) n = nth xs n
  proof -
    fix xs
    fix n
    assume n < length xs
    then show pick (map (Pair 1) xs) n = nth xs n
    proof (induct xs arbitrary: n)
      case Nil then show ?case by simp
    next
      case (Cons x xs) show ?case
      proof (cases n)
        case 0 then show ?thesis by simp
      next
        case (Suc -)
    from Cons have n < length (x # xs) by auto
        then have n < Suc (length xs) by simp
        with Suc have n - 1 < Suc (length xs) - 1 by auto
       with Cons have pick (map (Pair (1::nat)) xs) (n - 1) = xs ! (n - 1) by
auto
        with Suc show ?thesis by auto
      qed
    qed
  qed
  have sum-length: !!xs. foldl (op +) 0 (map fst (map (Pair 1) xs)) = length xs
  proof -
    have replicate-append:
      !!x xs y. replicate (length (x # xs)) y = replicate (length xs) y @ [y]
      by (simp add: replicate-app-Cons-same)
    fix xs
    show foldl (op +) 0 (map fst (map (Pair 1) xs)) = length xs
    unfolding map-fst-Pair proof (induct xs)
      case Nil show ?case by simp
    next
      case (Cons x xs) then show ?case unfolding replicate-append by simp
    qed
  qed
  have pick-nth-random:
    !!x xs s. pick (map (Pair 1) (x#xs)) (fst (random (length (x#xs)) s)) = nth
(x#xs) (fst (random (length (x#xs)) s))
  proof -
```

fix *s*
fix *x*
fix *xs*
have *bound*: *fst* (*random* (*length* (*x#xs*)) *s*) < *length* (*x#xs*) **by** (*rule random-bound*) *simp*
from *pick-nth* [*OF bound*] **show**
pick (*map* (*Pair 1*) (*x#xs*)) (*fst* (*random* (*length* (*x#xs*)) *s*)) = *nth* (*x#xs*) (*fst* (*random* (*length* (*x#xs*)) *s*)) **.**
qed
have *pick-nth-random-do*:
!!*x xs s*. (*do n* ← *random* (*length* (*x#xs*)); *return* (*pick* (*map* (*Pair 1*) (*x#xs*)) *n*) *done*) *s* =
(*do n* ← *random* (*length* (*x#xs*)); *return* (*nth* (*x#xs*) *n*) *done*) *s*
unfolding *monad-collapse split-def* **unfolding** *pick-nth-random* **..**
case (*Cons x xs*) **then show** *?case*
unfolding *select-weight-def sum-length pick-nth-random-do*
by *simp*
qed

definition
random-int :: *int* ⇒ *randseed* ⇒ *int* ∗ *randseed* **where**
random-int k = (*do n* ← *random* (*nat k*); *return* (*int n*) *done*)

lemma *random-nat* [*code*]:
random n = (*do k* ← *random-int* (*int n*); *return* (*nat k*) *done*)
unfolding *random-int-def* **by** *simp*

axiomatization
run-random :: (*randseed* ⇒ *'a* ∗ *randseed*) ⇒ *'a*

ML ⟪
signature RANDOM =
sig
type seed = *int*;
val seed: *unit* −> *seed*;
val value: *int* −> *seed* −> *int* ∗ *seed*;
end;

structure Random : *RANDOM* =
struct

exception RANDOM;

type seed = *int*;

local
val a = *16807*;
val m = *2147483647*;
in

```
  fun next s = (a * s) mod m;
end;

local
  val seed-ref = ref 1;
in
  fun seed () = CRITICAL (fn () =>
    let
      val r = next (!seed-ref)
    in
      (seed-ref := r; r)
    end);
end;

fun value h s =
  if h < 1 then raise RANDOM
  else (s mod (h - 1), seed ());

end;
»
```

code-reserved *SML Random*

code-type *randseed*
  (*SML Random.seed*)
types-code *randseed* (*Random.seed*)

code-const *random-int*
  (*SML Random.value*)
consts-code *random-int* (*Random.value*)

code-const *run-random*
  (*SML case* (*Random.seed* ()) *of* (*x*, '-) => - *x*)
consts-code *run-random* (*case* (*Random.seed* ()) *of* (*x*, '-) => - *x*)

end

34 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 *nat list* ⇒ *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* (λ*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′*) (λ*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* ==> ∀ *f* ∈ *set fs*. *PRIMREC f* ==> *PRIMREC* (*COMP g 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 ≤

lemma *ack-le-mono2*: *j* ≤ *k* ==> *ack* (*i*, *j*) ≤ *ack* (*i*, *k*)
  apply (*simp add*: *order-le-less*)

apply (*blast intro*: *ack-less-mono2*)
done

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: *ack* (*i*, *Suc j*) $\leq$ *ack* (*Suc i*, *j*)
apply (*induct j*)
apply *simp-all*
apply (*metis Suc-leI Suc-lessI ack-le-mono2 le-def less-ack2*)
done

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: *ack* (*i*, *j*) < *ack* (*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* < *ack* (*i*, *j*)
apply (*induct i*)
apply *simp-all*
apply (*blast intro*: *Suc-leI le-less-trans*)
done

PROPERTY A 8

lemma *ack-1* [*simp*]: *ack* (*Suc 0*, *j*) = *j* + *2*
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*]: *ack* (*Suc* (*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 $\exists\, k'.\ \forall\, i\ j.$ ...

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*]: ∀ *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*: ∀ *f* ∈ *set fs*. *PRIMREC f* ∧ (∃ *kf*. ∀ *l*. *f l* < *ack* (*kf*, *list-add l*))
==> ∃ *k*. ∀ *l*. *list-add* (*map* (λ*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*:
∀ *l*. *g l* < *ack* (*kg*, *list-add l*) ==>
∀ *f* ∈ *set fs*. *PRIMREC f* ∧ (∃ *kf*. ∀ *l*. *f l* < *ack*(*kf*, *list-add l*))
==> ∃ *k*. ∀ *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*:
 $\forall$ *l*. *f l* < *ack* (*kf*, *list-add l*) ==>
  $\forall$ *l*. *g l* < *ack* (*kg*, *list-add l*) ==>
  $\exists$ *k*. $\forall$ *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*: *PRIMREC f* ==> $\exists$ *k*. $\forall$ *l*. *f l* < *ack* (*k*, *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*: ¬ *PRIMREC* (λ*l*. *case l of* [] => *0* | *x* # *l′* => *ack* (*x*, *x*))
 apply (*rule notI*)
 apply (*erule ack-bounds-PRIMREC* [*THEN exE*])
 apply (*rule Nat.less-irrefl*)
 apply (*drule-tac x* = [*x*] **in** *spec*)
 apply *simp*
 done

end

35 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 = (∀ *x*∈*A*. ∀ *y*∈*A*. (*x*, *y*): *r* −−> ((*f x*), (*f y*)) : *r*)

definition
  least :: [*'a* => *bool*, *'a potype*] => *'a* **where**
  least P po = (*SOME x*. *x*: *pset po* & *P x* &
            (∀ *y* ∈ *pset po*. *P y* −−> (*x*,*y*): *order po*))

definition
  greatest :: [*'a* => *bool*, *'a potype*] => *'a* **where**
  greatest P po = (*SOME x*. *x*: *pset po* & *P x* &
            (∀ *y* ∈ *pset po*. *P y* −−> (*y*,*x*): *order po*))

definition
  lub :: [*'a set*, *'a potype*] => *'a* **where**
  lub S po = *least* (%*x*. ∀ *y*∈*S*. (*y*,*x*): *order po*) *po*

definition
  glb :: [*'a set*, *'a potype*] => *'a* **where**
  glb S po = *greatest* (%*x*. ∀ *y*∈*S*. (*x*,*y*): *order po*) *po*

definition
  isLub :: [*'a set*, *'a potype*, *'a*] => *bool* **where**
  isLub S po = (%*L*. (*L*: *pset po* & (∀ *y*∈*S*. (*y*,*L*): *order po*) &
          (∀ *z*∈*pset po*. (∀ *y*∈*S*. (*y*,*z*): *order po*) −−> (*L*,*z*): *order po*)))

definition
  isGlb :: [*'a set*, *'a potype*, *'a*] => *bool* **where**
  isGlb S po = (%*G*. (*G*: *pset po* & (∀ *y*∈*S*. (*G*,*y*): *order po*) &
          (∀ *z* ∈ *pset po*. (∀ *y*∈*S*. (*z*,*y*): *order po*) −−> (*z*,*G*): *order po*)))

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

definition
  interval :: [(*'a*∗*'a*) *set*,*'a*, *'a*] => *'a set* **where**
  interval r a b = {*x*. (*a*,*x*): *r* & (*x*,*b*): *r*}

```
definition
  Bot :: 'a potype => 'a where
  Bot po = least (%x. True) po

definition
  Top :: 'a potype => 'a where
  Top po = greatest (%x. True) po

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

definition
  CompleteLattice :: ('a potype) set where
  CompleteLattice = {cl. cl: PartialOrder &
                     (∀S. S ⊆ pset cl --> (∃L. isLub S cl L)) &
                     (∀S. S ⊆ pset cl --> (∃G. isGlb S cl G))}

definition
  CLF :: ('a potype * ('a => 'a)) set where
  CLF = (SIGMA cl: CompleteLattice.
          {f. f: pset cl -> pset cl & monotone f (pset cl) (order cl)})

definition
  induced :: ['a set, ('a * 'a) set] => ('a *'a)set where
  induced A r = {(a,b). a : A & b: A & (a,b): r}


definition
  sublattice :: ('a potype * 'a set)set where
  sublattice =
     (SIGMA cl: CompleteLattice.
         {S. S ⊆ pset cl &
          (| pset = S, order = induced S (order cl) |): CompleteLattice})

abbreviation
  sublat :: ['a set, 'a potype] => bool  (- <<= - [51,50]50) where
  S <<= cl == S : sublattice `` {cl}

definition
  dual :: 'a potype => 'a potype where
  dual po = (| 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|)
```

35.1 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 ∈ A ==> (x, x) ∈ r
apply (insert cl-po)
apply (simp add: PartialOrder-def refl-def A-def r-def)
done

lemma (in PO) antisymE: [| (a, b) ∈ r; (b, a) ∈ r |] ==> a = b
```

apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def antisym-def r-def*)
done

lemma (**in** *PO*) *transE*: $[\mid (a,\ b) \in r;\ (b,\ c) \in r\mid] ==> (a,c) \in r$
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def r-def*)
apply (*unfold trans-def*, *fast*)
done

lemma (**in** *PO*) *monotoneE*:
$[\mid$ *monotone f A r*; $x \in A;\ y \in A;\ (x,\ y) \in r \mid] ==> (f\,x,\ f\,y) \in r$
by (*simp add*: *monotone-def*)

lemma (**in** *PO*) *po-subset-po*:
$S \subseteq A ==> (\mid pset = S,\ order = induced\ S\ r \mid) \in PartialOrder$
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*: $[\mid (x,\ y) \in induced\ S\ r;\ S \subseteq A \mid] ==> (x,\ y) \in r$
by (*simp add*: *add*: *induced-def*)

lemma (**in** *PO*) *indI*: $[\mid (x,\ y) \in r;\ x \in S;\ y \in S \mid] ==> (x,\ y) \in induced\ S\ r$
by (*simp add*: *add*: *induced-def*)

lemma (**in** *CL*) *CL-imp-ex-isLub*: $S \subseteq A ==> \exists\,L.\ isLub\ S\ cl\ L$
apply (*insert cl-co*)
apply (*simp add*: *CompleteLattice-def A-def*)
done

declare (**in** *CL*) *cl-co* [*simp*]

lemma *isLub-lub*: $(\exists\,L.\ isLub\ S\ cl\ L) = isLub\ S\ cl\ (lub\ S\ cl)$
by (*simp add*: *lub-def least-def isLub-def some-eq-ex* [*symmetric*])

lemma *isGlb-glb*: $(\exists\,G.\ isGlb\ S\ cl\ G) = isGlb\ S\ cl\ (glb\ S\ cl)$
by (*simp add*: *glb-def greatest-def isGlb-def some-eq-ex* [*symmetric*])

lemma *isGlb-dual-isLub*: $isGlb\ S\ cl = isLub\ S\ (dual\ cl)$

by (*simp add*: *isLub-def isGlb-def dual-def converse-def*)

lemma *isLub-dual-isGlb*: *isLub S cl* = *isGlb S* (*dual cl*)
by (*simp add*: *isLub-def isGlb-def dual-def converse-def*)

lemma (**in** *PO*) *dualPO*: *dual cl* ∈ *PartialOrder*
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def dual-def refl-converse*
trans-converse antisym-converse)
done

lemma *Rdual*:
∀ *S*. (*S* ⊆ *A* −−>(∃ *L*. *isLub S* (| *pset* = *A*, *order* = *r*|) *L*))
==> ∀ *S*. (*S* ⊆ *A* −−> (∃ *G*. *isGlb S* (| *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*: *lub S cl* = *glb S* (*dual cl*)
by (*simp add*: *lub-def glb-def least-def greatest-def dual-def converse-def*)

lemma *glb-dual-lub*: *glb S cl* = *lub S* (*dual cl*)
by (*simp add*: *lub-def glb-def least-def greatest-def dual-def converse-def*)

lemma *CL-subset-PO*: *CompleteLattice* ⊆ *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*: *refl A 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*));

$(\forall S.\ S \subseteq pset\ po \ {-}{-}{>}\ (\exists G.\ isGlb\ S\ po\ G))|]$
$==> po \in CompleteLattice$
apply (*unfold CompleteLattice-def*, *blast*)
done

lemma (**in** *CL*) *CL-dualCL*: *dual cl* $\in$ *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*: $((x,\ y) \in (order(dual\ cl))) = ((y,\ x) \in order\ cl)$
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*:
$[|\ x \in A;\ y \in A|] ==> interval\ r\ x\ y = interval\ (order(dual\ cl))\ y\ x$
apply (*simp add*: *interval-def dualr-iff*)
apply (*fold r-def*, *fast*)
done

lemma (**in** *PO*) *interval-not-empty*:
$[|\ trans\ r;\ interval\ r\ a\ b \neq \{\}\ |] ==> (a,\ b) \in r$
apply (*simp add*: *interval-def*)
apply (*unfold trans-def*, *blast*)
done

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

lemma (**in** *PO*) *left-in-interval*:
$[|\ a \in A;\ b \in A;\ interval\ r\ a\ b \neq \{\}\ |] ==> a \in interval\ r\ a\ b$
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*:
$[|\ a \in A;\ b \in A;\ interval\ r\ a\ b \neq \{\}\ |] ==> b \in interval\ r\ a\ b$
apply (*simp* (*no-asm-simp*) *add*: *interval-def*)

apply (*simp add*: *PO-imp-trans interval-not-empty*)
apply (*simp add*: *reflE*)
done

35.2 sublattice

lemma (**in** *PO*) *sublattice-imp-CL*:
  $S <<= cl ==> (| pset = S, order = induced\ S\ r |) \in CompleteLattice$
by (*simp add*: *sublattice-def CompleteLattice-def r-def*)

lemma (**in** *CL*) *sublatticeI*:
  $[| S \subseteq A; (| pset = S, order = induced\ S\ r |) \in CompleteLattice |]$
  $==> S <<= cl$
by (*simp add*: *sublattice-def A-def r-def*)

35.3 lub

lemma (**in** *CL*) *lub-unique*: $[| S \subseteq A; isLub\ S\ cl\ x; isLub\ S\ cl\ L|] ==> x = L$
apply (*rule antisymE*)
apply (*auto simp add*: *isLub-def r-def*)
done

lemma (**in** *CL*) *lub-upper*: $[|S \subseteq A; x \in S|] ==> (x, lub\ S\ cl) \in r$
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*:
  $[| S \subseteq A; L \in A; \forall x \in S. (x,L) \in r |] ==> (lub\ S\ cl, L) \in r$
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*: $S \subseteq A ==> lub\ S\ cl \in A$
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*:
  [| $S \subseteq A$; $L \in A$; $\forall x \in S$. $(x,L) \in r$;
    $\forall z \in A$. $(\forall y \in S.\ (y,z) \in r)$ $-->$ $(L,z) \in r$ |] $==>$ $L = lub\ S\ cl$
apply (*rule lub-unique*, *assumption*)
apply (*simp add*: *isLub-def A-def r-def*)
apply (*unfold isLub-def*)
apply (*rule conjI*)
apply (*fold A-def r-def*)
apply (*rule lub-in-lattice*, *assumption*)
apply (*simp add*: *lub-upper lub-least*)
done

lemma (**in** *CL*) *lubIa*: [| $S \subseteq A$; *isLub S cl L* |] $==>$ $L = lub\ S\ cl$
by (*simp add*: *lubI isLub-def A-def r-def*)

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

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

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

35.4 glb

lemma (**in** *CL*) *glb-in-lattice*: $S \subseteq A$ $==>$ $glb\ S\ cl \in A$
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*: [|$S \subseteq A$; $x \in S$|] $==>$ $(glb\ S\ cl,\ x) \in r$
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* $->$ *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*: $f \in A -> A$
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

35.5 fixed points

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

lemma *fix-imp-eq*: $x \in$ *fix f A* $==> f\ x = x$
by (*simp add*: *fix-def*)

lemma *fixf-subset*:
  [| $A \subseteq B$; $x \in$ *fix* (%y: A. $f\ y$) A |] $==> x \in$ *fix f B*
by (*simp add*: *fix-def*, *auto*)

35.6 lemmas for Tarski, lub

lemma (**in** *CLF*) *lubH-le-flubH*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} ==>$ (*lub H cl*, f (*lub H 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 (*lub H r*)) $\in r$
apply (*rule ballI*)
apply (*rule transE*)
— instantiates (x, *???z*) $\in$ *order cl to* (x, $f\ x$),
— because of the def of H

apply *fast*
— so it remains to show $(f\ x,\ f\ (lub\ H\ cl)) \in r$
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*:
  $[\mid\ H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}\ \mid] ==> (f\ (lub\ H\ cl),\ lub\ H\ cl) \in r$
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*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} ==> lub\ H\ cl \in fix\ f\ A$
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*:
  $[\mid H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\};\ \ x \in P\ \mid] ==> x \in H$
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*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} ==> \forall\, x \in fix\ f\ A.\ (x,\ lub\ H\ cl) \in r$
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*:

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

35.7 Tarski fixpoint theorem 1, first part

lemma (**in** *CLF*) *T-thm-1-lub*: $lub\ P\ cl = lub\ \{x.\ (x, f\ x) \in r\ \&\ x \in A\}\ cl$
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*: $H = \{x.\ (f\ x, x) \in r\ \&\ x \in A\} ==> glb\ H\ cl \in P$
— 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*: $glb\ P\ cl = glb\ \{x.\ (f\ x, x) \in r\ \&\ x \in A\}\ cl$
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

35.8 interval

lemma (**in** *CLF*) *rel-imp-elem*: $(x,\ y) \in r ==> x \in A$
apply (*insert CO-refl*)
apply (*simp add*: *refl-def*, *blast*)
done

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

lemma (**in** *CLF*) *intervalI*:
  [| (a, x) ∈ r; (x, b) ∈ r |] ==> x ∈ *interval* r a b
by (*simp add*: *interval-def*)

lemma (**in** *CLF*) *interval-lemma1*:
  [| S ⊆ *interval* r a b; x ∈ S |] ==> (a, x) ∈ r
by (*unfold interval-def*, *fast*)

lemma (**in** *CLF*) *interval-lemma2*:
  [| S ⊆ *interval* r a b; x ∈ S |] ==> (x, b) ∈ r
by (*unfold interval-def*, *fast*)

lemma (**in** *CLF*) *a-less-lub*:
  [| S ⊆ A; S ≠ {};
    ∀ x ∈ S. (a,x) ∈ r; ∀ y ∈ S. (y, L) ∈ r |] ==> (a,L) ∈ r
by (*blast intro*: *transE*)

lemma (**in** *CLF*) *glb-less-b*:
  [| S ⊆ A; S ≠ {};
    ∀ x ∈ S. (x,b) ∈ r; ∀ y ∈ S. (G, y) ∈ r |] ==> (G,b) ∈ r
by (*blast intro*: *transE*)

lemma (**in** *CLF*) *S-intv-cl*:
  [| a ∈ A; b ∈ A; S ⊆ *interval* r a b |]==> S ⊆ A
by (*simp add*: *subset-trans* [*OF - interval-subset*])

lemma (**in** *CLF*) *L-in-interval*:
  [| a ∈ A; b ∈ A; S ⊆ *interval* r a b;
    S ≠ {}; *isLub* S *cl* L; *interval* r a b ≠ {} |] ==> L ∈ *interval* r a b
apply (*rule intervalI*)
apply (*rule a-less-lub*)
prefer *2* **apply** *assumption*
apply (*simp add*: *S-intv-cl*)
apply (*rule ballI*)
apply (*simp add*: *interval-lemma1*)
apply (*simp add*: *isLub-upper*)
— (L, b) ∈ r
apply (*simp add*: *isLub-least interval-lemma2*)
done

lemma (**in** *CLF*) *G-in-interval*:
  [| a ∈ A; b ∈ A; *interval* r a b ≠ {}; S ⊆ *interval* r a b; *isGlb* S *cl* G;
    S ≠ {} |] ==> G ∈ *interval* r a b
apply (*simp add*: *interval-dual*)
apply (*simp add*: *CLF.L-in-interval* [*of - f*]
      dualA-iff A-def dualPO CL-dualCL CLF-dual isGlb-dual-isLub)
done

lemma (**in** *CLF*) *intervalPO*:

$[| \ a \in A;\ b \in A;\ interval\ r\ a\ b \neq \{\}\ |]$
$==> (| \ pset = interval\ r\ a\ b,\ order = induced\ (interval\ r\ a\ b)\ r\ |)$
$\in PartialOrder$
apply (*rule po-subset-po*)
apply (*simp add*: *interval-subset*)
done

lemma (**in** *CLF*) *intv-CL-lub*:
$[| \ a \in A;\ b \in A;\ interval\ r\ a\ b \neq \{\}\ |]$
$==> \forall S.\ S \subseteq interval\ r\ a\ b\ -->$
$(\exists L.\ isLub\ S\ (| \ pset = interval\ r\ a\ b,$
$order = induced\ (interval\ r\ a\ b)\ r\ |)\ \ L)$
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*)
— $(if\ S = \{\}\ then\ a\ else\ L) \in interval\ r\ a\ b$
apply (*simp add*: *CL-imp-PO L-in-interval*)
apply (*simp add*: *left-in-interval*)
— lub prop 1
apply (*case-tac* $S = \{\}$)
— $S = \{\},\ y \in S = False => everything$
apply *fast*
— $S \neq \{\}$
apply *simp*
— $\forall y{:}S.\ (y,\ L) \in induced\ (interval\ r\ a\ b)\ r$
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*)
— $\forall z{:}interval\ r\ a\ b.\ (\forall y{:}S.\ (y,\ z) \in induced\ (interval\ r\ a\ b)\ r \longrightarrow (if\ S = \{\}\ then\ a\ else\ L,\ z) \in induced\ (interval\ r\ a\ b)\ r$
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*]

35.9 Top and Bottom

lemma (**in** *CLF*) *Top-dual-Bot*: *Top cl* = *Bot* (*dual cl*)
by (*simp add*: *Top-def Bot-def least-def greatest-def dualA-iff dualr-iff*)

lemma (**in** *CLF*) *Bot-dual-Top*: *Bot cl* = *Top* (*dual cl*)
by (*simp add*: *Top-def Bot-def least-def greatest-def dualA-iff dualr-iff*)

lemma (**in** *CLF*) *Bot-in-lattice*: *Bot cl* $\in A$
apply (*simp add*: *Bot-def least-def*)
apply (*rule-tac a*=*glb A cl* **in** *someI2*)
apply (*simp-all add*: *glb-in-lattice glb-lower*
        r-def [*symmetric*] *A-def* [*symmetric*])
done

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

lemma (**in** *CLF*) *Top-prop*: $x \in A$ ==> (*x*, *Top cl*) $\in r$
apply (*simp add*: *Top-def greatest-def*)

apply (*rule-tac a=lub A cl* **in** *someI2*)
apply (*rule someI2*)
apply (*simp-all add*: *lub-in-lattice lub-upper*
r-def [*symmetric*] *A-def* [*symmetric*])
done

lemma (**in** *CLF*) *Bot-prop*: *x* ∈ *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

35.10 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*: *lub Y cl* ∈ *A*
by (*rule Y-subset-A* [*THEN lub-in-lattice*])

lemma (**in** *Tarski*) *lubY-le-flubY*: (*lub Y cl*, *f* (*lub Y cl*)) ∈ *r*
apply (*rule lub-least*)
apply (*rule Y-subset-A*)
apply (*rule f-in-funcset* [*THEN funcset-mem*])
apply (*rule lubY-in-A*)

— $Y \subseteq P$ ==> $f\ x = x$
apply (*rule ballI*)
apply (*rule-tac t = x* **in** *fix-imp-eq* [*THEN subst*])
apply (*erule Y-ss* [*simplified P-def*, *THEN subsetD*])
— *reduce* $(f\ x,\ f\ (lub\ Y\ cl)) \in r$ *to* $(x,\ lub\ Y\ cl) \in r$ 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*: *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*: $x \in$ *intY1* $\Longrightarrow$ $f\ x \in$ *intY1*
apply (*simp add*: *intY1-def interval-def*)
apply (*rule conjI*)
apply (*rule transE*)
apply (*rule lubY-le-flubY*)
— $(f\ (lub\ Y\ cl),\ f\ x) \in r$
apply (*rule-tac f=f* **in** *monotoneE*)
apply (*rule monotone-f*)
apply (*rule lubY-in-A*)
apply (*simp add*: *intY1-def interval-def intY1-elem*)
apply (*simp add*: *intY1-def interval-def*)
— $(f\ x,\ Top\ cl) \in r$
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*: *intY1*. *f x*) $\in$ *intY1* −> *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** *fixf-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* ∈ *P*
apply (*simp add*: *v-in-P*)
apply (*rule conjI*)
— *v* is lub
— *1*. ∀ *y*:*Y*. (*y*, *v*) ∈ *induced P r*
apply (*rule ballI*)
apply (*simp add*: *induced-def subsetD v-in-P*)
apply (*rule conjI*)

apply (*erule Y-ss* [*THEN subsetD*])
apply (*rule-tac b* = *lub Y cl* **in** *transE*)
apply (*rule lub-upper*)
apply (*rule Y-subset-A*, *assumption*)
apply (*rule-tac b* = *Top cl* **in** *interval-imp-mem*)
apply (*simp add*: *v-def*)
apply (*fold intY1-def*)
apply (*rule CL.glb-in-lattice* [*OF - 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*:
  [| (| *pset* = *A*, *order* = *r* |) ∈ *PartialOrder*;
    ∀ *S*. *S* ⊆ *A* −−> (∃ *L*. *isLub S* (| *pset* = *A*, *order* = *r* |) *L*) |]
  ==> (| *pset* = *A*, *order* = *r* |) ∈ *CompleteLattice*
by (*simp add*: *CompleteLatticeI Rdual*)

theorem (**in** *CLF*) *Tarski-full*:
  (| *pset* = *P*, *order* = *induced P r*|) ∈ *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

36 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′*:
  $bv \neq [] ==> bv\text{-}to\text{-}nat\ bv = bitval\ (hd\ bv) * 2\ \hat{}\ (length\ bv - 1) + bv\text{-}to\text{-}nat\ (tl\ bv)$
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-to-nat (*full-adder a b c*) = *bitval a* + *bitval b* + *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*)

36.1 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-nonnull*: *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-nonnull*]
    ring-distribs 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-nonnull*: *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.prems* **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-distribs bv-to-nat-helper cca-length)
    done
  qed
qed

end
```

37 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).

37.1 Proof text

```
theorem tnd: A ∨ ¬ A
proof −
  let ?P = λX. X = False ∨ X = True ∧ A
  let ?Q = λX. X = False ∧ A ∨ X = True

  have a: ?P (Eps ?P)
  proof (rule someI)
    have False = False ..
    thus ?P False ..
  qed
  have b: ?Q (Eps ?Q)
  proof (rule someI)
    have True = True ..
    thus ?Q True ..
  qed

  from a show ?thesis
  proof
    assume Eps ?P = True ∧ A
    hence A ..
    thus ?thesis ..
  next
    assume P: Eps ?P = False
    from b show ?thesis
    proof
      assume Eps ?Q = False ∧ A
      hence A ..
      thus ?thesis ..
    next
      assume Q: Eps ?Q = True
```

```
      have neq: ?P ≠ ?Q
      proof
        assume ?P = ?Q
        hence Eps ?P = Eps ?Q by (rule arg-cong)
        also note P
        also note Q
        finally show False by (rule False-neq-True)
      qed
      have ¬ A
      proof
        assume a: A
        have ?P = ?Q
        proof
          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
        with neq show False by contradiction
      qed
      thus ?thesis ..
    qed
  qed
qed
```

37.2 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)))
```

37.3 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
  (λx. (x = False) ∨ (x = True) ∧ A) ≠
   (λx. (x = False) ∧ A ∨ (x = True)))
prefer 2
apply (rule notI)
apply (drule-tac f = λy. SOME x. y x in arg-cong)
apply (drule trans, assumption)
apply (drule sym)
apply (drule trans, assumption)
apply (erule False-neq-True)
apply (rule disjI2)
apply (rule notI)
apply (erule notE)
apply (rule ext)
apply (rule iffI)
apply (erule disjE)
apply (rule disjI1)
apply (erule conjI)
apply assumption
apply (erule conjE)
apply (erule disjI2)
apply (erule disjE)
apply (erule conjE)
apply (erule disjI1)
apply (rule disjI2)
apply (erule conjI)
apply assumption
done
```

37.4 Proof term of script

```
conjE · - · - · - •
 (conjI · - · - •
   (someI · (λx. x = False ∨ x = True ∧ ?A) · - •
     (disjI1 · - · - • (HOL.refl · -))) •
   (someI · (λx. x = False ∧ ?A ∨ x = True) · - •
     (disjI2 · - · - • (HOL.refl · -)))) •
 (λ(H: -) Ha: -.
   disjE · - · - · - • H •
    (λH: -.
      disjE · - · - · - • Ha •
       (λH: -. conjE · - · - · - • H • (λH: -. disjI1 · - · -)) •
       (λHa: -.
         disjI2 · - · - •
          (notI · - •
            (λHb: -.
              notE · - · - •
               (notI · - •
                 (λHb: -.
                   False-neq-True · - •
                    (HOL.trans · - · - · - • (HOL.sym · - · - • H) •
                      (HOL.trans · - · - · - •
                        (arg-cong · (λx. x = False ∨ x = True ∧ ?A) ·
                          (λx. x = False ∧ ?A ∨ x = True) ·
                          Eps •
                          Hb) •
                        Ha)))) •
               (ext · - · - •
                 (λx. iffI · - · - •
                        (λH: -.
                          disjE · - · - · - • H •
                           (λH: -. disjI1 · - · - • (conjI · - · - • H • Hb)) •
                           (λH: -.
                             conjE · - · - · - • H •
                              (λ(H: -) Ha: -. disjI2 · - · - • H))) •
                        (λH: -.
                          disjE · - · - · - • H •
                           (λH: -.
                             conjE · - · - · - • H •
                              (λ(H: -) Ha: -. disjI1 · - · - • H)) •
                           (λH: -.
                             disjI2 · - · - • (conjI · - · - • H • Hb))))))))) •
    (λH: -. conjE · - · - · - • H • (λH: -. disjI1 · - · -)))
```

end

38 Classical Predicate Calculus Problems

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

38.1 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 ↔.

lemma $(P \text{ --> } Q \mid R) \text{ --> } (P\text{-->}Q) \mid (P\text{-->}R)$
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.

38.1.1 Pelletier's examples

1

lemma $(P\text{-->}Q) \ = \ (\sim Q \text{ --> } \sim P)$
by *blast*

2

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

3

lemma $\sim(P\text{-->}Q) \text{ --> } (Q\text{-->}P)$
by *blast*

4

lemma $(\sim P\text{-->}Q) \ = \ (\sim Q \text{ --> } P)$
by *blast*

5

lemma $((P|Q)\text{-->}(P|R)) \text{ --> } (P|(Q\text{-->}R))$

by *blast*

6

lemma $P \mid \sim P$
by *blast*

7

lemma $P \mid \sim \sim \sim P$
by *blast*

8. Peirce's law

lemma $((P{-}{-}{>}Q) \;{-}{-}{>}\; P) \;\; {-}{-}{>} \;\; P$
by *blast*

9

lemma $((P|Q) \ \& \ ({\sim}P|Q) \ \& \ (P| \sim Q)) \ {-}{-}{>} \sim ({\sim}P \mid {\sim}Q)$
by *blast*

10

lemma $(Q{-}{-}{>}R) \ \& \ (R{-}{-}{>}P\&Q) \ \& \ (P{-}{-}{>}Q|R) \ {-}{-}{>} \ (P{=}Q)$
by *blast*

11. Proved in each direction (incorrectly, says Pelletier!!)

lemma $P{=}(P{::}bool)$
by *blast*

12. "Dijkstra's law"

lemma $((P = Q) = R) = (P = (Q = R))$
by *blast*

13. Distributive law

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

14

lemma $(P = Q) = ((Q \mid {\sim}P) \ \& \ ({\sim}Q|P))$
by *blast*

15

lemma $(P \ {-}{-}{>} \ Q) = ({\sim}P \mid Q)$
by *blast*

16

lemma $(P{-}{-}{>}Q) \mid (Q{-}{-}{>}P)$
by *blast*

17

lemma $((P \ \& \ (Q{-}{-}{>}R)){-}{-}{>}S) \ = \ (({\sim}P \mid Q \mid S) \ \& \ ({\sim}P \mid {\sim}R \mid S))$
by *blast*

38.1.2 Classical Logic: examples with quantifiers

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

lemma $(\exists x.\ P{-}{-}{>}Q(x))\ =\ (P\ {-}{-}{>}\ (\exists x.\ Q(x)))$
by *blast*

lemma $(\exists x.\ P(x){-}{-}{>}Q) = ((\forall x.\ P(x))\ {-}{-}{>}\ Q)$
by *blast*

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

From Wishnu Prasetya

lemma $(\forall s.\ q(s)\ {-}{-}{>}\ r(s))\ \&\ {\sim}r(s)\ \&\ (\forall s.\ {\sim}r(s)\ \&\ {\sim}q(s)\ {-}{-}{>}\ p(t)\ |\ q(t))$
$\quad {-}{-}{>}\ p(t)\ |\ r(t)$
by *blast*

38.1.3 Problems requiring quantifier duplication

Theorem B of Peter Andrews, Theorem Proving via General Matings, JACM 28 (1981).

lemma $(\exists x.\ \forall y.\ P(x) = P(y))\ {-}{-}{>}\ ((\exists x.\ P(x)) = (\forall y.\ P(y)))$
by *blast*

Needs multiple instantiation of the quantifier.

lemma $(\forall x.\ P(x){-}{-}{>}P(f(x)))\ \ \&\ \ P(d){-}{-}{>}P(f(f(f(d))))$
by *blast*

Needs double instantiation of the quantifier

lemma $\exists x.\ P(x)\ {-}{-}{>}\ P(a)\ \&\ P(b)$
by *blast*

lemma $\exists z.\ P(z)\ {-}{-}{>}\ (\forall x.\ P(x))$
by *blast*

lemma $\exists x.\ (\exists y.\ P(y))\ {-}{-}{>}\ P(x)$
by *blast*

38.1.4 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P(y){-}{-}{>}P(x)$
by *blast*

Problem 19

lemma $\exists x.\ \forall y\ z.\ (P(y){-}{-}{>}Q(z))\ {-}{-}{>}\ (P(x){-}{-}{>}Q(x))$

by *blast*

Problem 20

lemma $(\forall x\ y.\ \exists z.\ \forall w.\ (P(x)\&Q(y){-}{-}{>}R(z)\&S(w)))$
$\quad {-}{-}{>}\ (\exists x\ y.\ P(x)\ \&\ Q(y))\ {-}{-}{>}\ (\exists z.\ R(z))$
by *blast*

Problem 21

lemma $(\exists x.\ P{-}{-}{>}Q(x))\ \&\ (\exists x.\ Q(x){-}{-}{>}P)\ {-}{-}{>}\ (\exists x.\ P{=}Q(x))$
by *blast*

Problem 22

lemma $(\forall x.\ P\ =\ Q(x))\ \ {-}{-}{>}\ \ (P\ =\ (\forall x.\ Q(x)))$
by *blast*

Problem 23

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

Problem 24

lemma $\sim(\exists x.\ S(x)\&Q(x))\ \&\ (\forall x.\ P(x)\ {-}{-}{>}\ Q(x)|R(x))\ \&$
$\quad (\sim(\exists x.\ P(x))\ {-}{-}{>}\ (\exists x.\ Q(x)))\ \&\ (\forall x.\ Q(x)|R(x)\ {-}{-}{>}\ S(x))$
$\quad {-}{-}{>}\ (\exists x.\ P(x)\&R(x))$
by *blast*

Problem 25

lemma $(\exists x.\ P(x))\ \&$
$\qquad (\forall x.\ L(x)\ {-}{-}{>}\ \sim\ (M(x)\ \&\ R(x)))\ \&$
$\qquad (\forall x.\ P(x)\ {-}{-}{>}\ (M(x)\ \&\ L(x)))\ \&$
$\qquad ((\forall x.\ P(x){-}{-}{>}Q(x))\ |\ (\exists x.\ P(x)\&R(x)))$
$\quad {-}{-}{>}\ (\exists x.\ Q(x)\&P(x))$
by *blast*

Problem 26

lemma $((\exists x.\ p(x))\ =\ (\exists x.\ q(x)))\ \&$
$\quad (\forall x.\ \forall y.\ p(x)\ \&\ q(y)\ {-}{-}{>}\ (r(x)\ =\ s(y)))$
$\ {-}{-}{>}\ ((\forall x.\ p(x){-}{-}{>}r(x))\ =\ (\forall x.\ q(x){-}{-}{>}s(x)))$
by *blast*

Problem 27

lemma $(\exists x.\ P(x)\ \&\ \sim Q(x))\ \&$
$\qquad\qquad (\forall x.\ P(x)\ {-}{-}{>}\ R(x))\ \&$
$\qquad\qquad (\forall x.\ M(x)\ \&\ L(x)\ {-}{-}{>}\ P(x))\ \&$
$\qquad\qquad ((\exists x.\ R(x)\ \&\ \sim\ Q(x))\ {-}{-}{>}\ (\forall x.\ L(x)\ {-}{-}{>}\ \sim\ R(x)))$
$\qquad {-}{-}{>}\ (\forall x.\ M(x)\ {-}{-}{>}\ \sim L(x))$
by *blast*

Problem 28. AMENDED

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

Problem 29. Essentially the same as Principia Mathematica *11.71

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

Problem 30

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

Problem 31

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

Problem 32

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

Problem 33

lemma $(\forall x.\ P(a)\ \&\ (P(x) \dashrightarrow P(b)) \dashrightarrow P(c))\ =$
$(\forall x.\ (\sim P(a)\ |\ P(x)\ |\ P(c))\ \&\ (\sim P(a)\ |\ \sim P(b)\ |\ P(c)))$
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 $\exists x\ y.\ P\ x\ y \dashrightarrow (\forall u\ v.\ P\ u\ v)$

by *blast*

Problem 36

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

Problem 37

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

Problem 38

lemma $(\forall x. p(a)$ & $(p(x)$ −−> $(\exists y. p(y)$ & $r\ x\ y))$ −−>
$(\exists z. \exists w. p(z)$ & $r\ x\ w$ & $r\ w\ z))$ =
$(\forall x.$ (~$p(a) \mid p(x) \mid (\exists z. \exists w. p(z)$ & $r\ x\ w$ & $r\ w\ z))$ &
(~$p(a)$ | ~$(\exists y. p(y)$ & $r\ x\ y)$ |
$(\exists z. \exists w. p(z)$ & $r\ x\ w$ & $r\ w\ z)))$
by *blast*

Problem 39

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

Problem 40. AMENDED

lemma $(\exists y. \forall x. F\ x\ y = F\ x\ x)$
−−> ~ $(\forall x. \exists y. \forall z. F\ z\ y =$ (~ $F\ z\ x))$
by *blast*

Problem 41

lemma $(\forall z. \exists y. \forall x. f\ x\ y = (f\ x\ z$ & ~ $f\ x\ x))$
−−> ~ $(\exists z. \forall x. f\ x\ z)$
by *blast*

Problem 42

lemma ~ $(\exists y. \forall x. p\ x\ y =$ (~ $(\exists z. p\ x\ z$ & $p\ z\ x)))$
by *blast*

Problem 43!!

lemma $(\forall x{::}'a. \forall y{::}'a. q\ x\ y = (\forall z. p\ z\ x = (p\ z\ y{::}bool)))$
−−> $(\forall x. (\forall y. q\ x\ y = (q\ y\ x{::}bool)))$

by *blast*

Problem 44

lemma $(\forall x.\ f(x) --\!>$
$(\exists y.\ g(y)\ \&\ h\ x\ y\ \&\ (\exists y.\ g(y)\ \&\ {\sim}\ h\ x\ y)))\ \ \&$
$(\exists x.\ j(x)\ \&\ (\forall y.\ g(y)\ --\!>\ h\ x\ y))$
$--\!>\ (\exists x.\ j(x)\ \&\ {\sim}f(x))$
by *blast*

Problem 45

lemma $(\forall x.\ f(x)\ \&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ --\!>\ j\ x\ y)$
$--\!>\ (\forall y.\ g(y)\ \&\ h\ x\ y\ --\!>\ k(y)))\ \&$
${\sim}\ (\exists y.\ l(y)\ \&\ k(y))\ \&$
$(\exists x.\ f(x)\ \&\ (\forall y.\ h\ x\ y\ --\!>\ l(y))$
$\&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ --\!>\ j\ x\ y))$
$--\!>\ (\exists x.\ f(x)\ \&\ {\sim}\ (\exists y.\ g(y)\ \&\ h\ x\ y))$
by *blast*

38.1.5 Problems (mainly) involving equality or functions

Problem 48

lemma $(a{=}b\ |\ c{=}d)\ \&\ (a{=}c\ |\ b{=}d)\ --\!>\ a{=}d\ |\ b{=}c$
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 $(\exists x\ y{::}'a.\ \forall z.\ z{=}x\ |\ z{=}y)\ \&\ P(a)\ \&\ P(b)\ \&\ ({\sim}a{=}b)$
$--\!>\ (\forall u{::}'a.\ P(u))$
by *metis*

Problem 50. (What has this to do with equality?)

lemma $(\forall x.\ P\ a\ x\ |\ (\forall y.\ P\ x\ y))\ --\!>\ (\exists x.\ \forall y.\ P\ x\ y)$
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 $(\exists z\ w.\ \forall x\ y.\ P\ x\ y = (x{=}z\ \&\ y{=}w))\ --\!>$
$(\exists w.\ \forall y.\ \exists z.\ (\forall x.\ P\ x\ y = (x{=}z)) = (y{=}w))$
by *blast*

Problem 55

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

lemma *lives(agatha) & lives(butler) & lives(charles) &*
  (killed agatha agatha | killed butler agatha | killed charles agatha) &
  (∀ x y. killed x y −−> hates x y & ~richer x y) &
  (∀ x. hates agatha x −−> ~hates charles x) &
  (hates agatha agatha & hates agatha charles) &
  (∀ x. lives(x) & ~richer x agatha −−> hates butler x) &
  (∀ x. hates agatha x −−> hates butler x) &
  (∀ x. ~hates x agatha | ~hates x butler | ~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 $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
  $(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ {-}{-}{>}\ P\ x\ z)\quad {-}{-}{>}\quad P\ (f\ a\ b)\ (f\ a\ c)$
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 $(\forall x.\ P(x) = ({\sim}P(f(x))))\ {-}{-}{>}\ (\exists x.\ P(x)\ \&\ {\sim}P(f(x)))$
by *blast*

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y\ {-}{-}{>}\ P\ z\ (f\ x))\ \&\ P\ x\ y)$
by *blast*

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ {-}{-}{>}\ p(f\ x))\ {-}{-}{>}\ p(f(f\ x)))\ =$
  $(\forall x.\ ({\sim}\ p\ a\ |\ p\ x\ |\ p(f(f\ x)))\ \&$
    $({\sim}\ p\ a\ |\ {\sim}\ p(f\ x)\ |\ p(f(f\ x))))$
by *blast*

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

lemma $(\forall x.\ F(x)\ \&\ {\sim}G(x)\ {-}{-}{>}\ (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
  $(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ {-}{-}{>}\ K(y)))\ \&$
  $(\forall x.\ K(x)\ {-}{-}{>}\ {\sim}G(x))\ \ {-}{-}{>}\ \ (\exists x.\ K(x)\ \&\ J(x))$
by *fast*

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

lemma $(\forall x.\ F(x)\ \&\ {\sim}G(x)\ {-}{-}{>}\ (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ {-}{-}{>}\ K(y)))\ \&$
$(\forall x.\ K(x)\ {-}{-}{>}\ {\sim}G(x))\quad {-}{-}{>}\quad (\exists x.\ K(x)\ {-}{-}{>}\ {\sim}G(x))$
by *fast*

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

lemma $(\forall x.\ \mathit{honest}(x)\ \&\ \mathit{industrious}(x)\ {-}{-}{>}\ \mathit{healthy}(x))\ \&$
${\sim}\ (\exists x.\ \mathit{grocer}(x)\ \&\ \mathit{healthy}(x))\ \&$
$(\forall x.\ \mathit{industrious}(x)\ \&\ \mathit{grocer}(x)\ {-}{-}{>}\ \mathit{honest}(x))\ \&$
$(\forall x.\ \mathit{cyclist}(x)\ {-}{-}{>}\ \mathit{industrious}(x))\ \&$
$(\forall x.\ {\sim}\mathit{healthy}(x)\ \&\ \mathit{cyclist}(x)\ {-}{-}{>}\ {\sim}\mathit{honest}(x))$
${-}{-}{>}\ (\forall x.\ \mathit{grocer}(x)\ {-}{-}{>}\ {\sim}\mathit{cyclist}(x))$
by *blast*

lemma $(\forall x\ y.\ R(x,y)\ |\ R(y,x))\ \&$
$(\forall x\ y.\ S(x,y)\ \&\ S(y,x)\ {-}{-}{>}\ x{=}y)\ \&$
$(\forall x\ y.\ R(x,y)\ {-}{-}{>}\ S(x,y))\quad {-}{-}{>}\quad (\forall x\ y.\ S(x,y)\ {-}{-}{>}\ R(x,y))$
by *blast*

38.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y\ \&\ y < z\ {-}{-}{>}\ {\sim}\ (z < (x{::}\mathit{nat}))$
by (*meson order-less-irrefl order-less-trans*)

The "small example" from Bezem, Hendriks and de Nivelle, Automatic Proof Construction in Type Theory Using Resolution, JAR 29: 3-4 (2002), pages 253-275

lemma $(\forall x\ y\ z.\ R(x,y)\ \&\ R(y,z)\ {-}{-}{>}\ R(x,z))\ \&$
$(\forall x.\ \exists y.\ R(x,y))\ {-}{-}{>}$
${\sim}\ (\forall x.\ P\ x = (\forall y.\ R(x,y)\ {-}{-}{>}\ {\sim}\ P\ y))$
by (*tactic*⟨⟨*Meson.safe-best-meson-tac 1*⟩⟩)
— In contrast, *meson* is SLOW: 7.6s on griffon

38.2.1 Pelletier's examples

1

lemma $(P\ {-}{-}{>}\ Q)\ =\ ({\sim}Q\ {-}{-}{>}\ {\sim}P)$
by *blast*

2

lemma $({\sim}\ {\sim}\ P) =\ P$
by *blast*

3

lemma $\sim(P{-}{-}{>}Q)\ {-}{-}{>}\ (Q{-}{-}{>}P)$
by *blast*

4

lemma $(\sim P{-}{-}{>}Q)\ =\ (\sim Q\ {-}{-}{>}\ P)$
by *blast*

5

lemma $((P|Q){-}{-}{>}(P|R))\ {-}{-}{>}\ (P|(Q{-}{-}{>}R))$
by *blast*

6

lemma $P\ |\ \sim\ P$
by *blast*

7

lemma $P\ |\ \sim\ \sim\ \sim\ P$
by *blast*

8. Peirce's law

lemma $((P{-}{-}{>}Q)\ {-}{-}{>}\ P)\ \ {-}{-}{>}\ \ P$
by *blast*

9

lemma $((P|Q)\ \&\ (\sim P|Q)\ \&\ (P|\ \sim Q))\ {-}{-}{>}\ \sim\ (\sim P\ |\ \sim Q)$
by *blast*

10

lemma $(Q{-}{-}{>}R)\ \&\ (R{-}{-}{>}P\&Q)\ \&\ (P{-}{-}{>}Q|R)\ {-}{-}{>}\ (P{=}Q)$
by *blast*

11. Proved in each direction (incorrectly, says Pelletier!!)

lemma $P{=}(P{::}bool)$
by *blast*

12. "Dijkstra's law"

lemma $((P\ =\ Q)\ =\ R)\ =\ (P\ =\ (Q\ =\ R))$
by *blast*

13. Distributive law

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

14

lemma $(P\ =\ Q)\ =\ ((Q\ |\ \sim P)\ \&\ (\sim Q|P))$

by *blast*

15

lemma $(P \text{ --> } Q) = (\sim P \mid Q)$
by *blast*

16

lemma $(P\text{-->}Q) \mid (Q\text{-->}P)$
by *blast*

17

lemma $((P \,\&\, (Q\text{-->}R))\text{-->}S) \;=\; ((\sim P \mid Q \mid S) \,\&\, (\sim P \mid \sim R \mid S))$
by *blast*

38.2.2 Classical Logic: examples with quantifiers

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

lemma $(\exists x.\ P \text{ --> } Q\ x) \;=\; (P \text{ --> } (\exists x.\ Q\ x))$
by *blast*

lemma $(\exists x.\ P\ x \text{ --> } Q) = ((\forall x.\ P\ x) \text{ --> } Q)$
by *blast*

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

lemma $(\forall x.\ P\ x \text{ --> } P(f\ x)) \;\&\;\ P\ d \text{ --> } P(f(f(f\ d)))$
by *blast*

Needs double instantiation of EXISTS

lemma $\exists x.\ P\ x \text{ --> } P\ a \,\&\, P\ b$
by *blast*

lemma $\exists z.\ P\ z \text{ --> } (\forall x.\ P\ x)$
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*

38.2.3 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P\ y \text{ --> } P\ x$
by *blast*

Problem 19

lemma $\exists x. \forall y\ z. (P\ y \text{ --> } Q\ z) \text{ --> } (P\ x \text{ --> } Q\ x)$
by *blast*

Problem 20

lemma $(\forall x\ y. \exists z. \forall w. (P\ x\ \&\ Q\ y \text{ --> } R\ z\ \&\ S\ w))$
$\quad \text{ --> } (\exists x\ y. P\ x\ \&\ Q\ y) \text{ --> } (\exists z. R\ z)$
by *blast*

Problem 21

lemma $(\exists x. P \text{ --> } Q\ x)\ \&\ (\exists x. Q\ x \text{ --> } P) \text{ --> } (\exists x. P{=}Q\ x)$
by *blast*

Problem 22

lemma $(\forall x. P = Q\ x) \text{ --> } (P = (\forall x. Q\ x))$
by *blast*

Problem 23

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

Problem 24

lemma $\sim(\exists x. S\ x\ \&\ Q\ x)\ \&\ (\forall x. P\ x \text{ --> } Q\ x \mid R\ x)\ \&$
$\quad (\sim(\exists x. P\ x) \text{ --> } (\exists x. Q\ x))\ \&\ (\forall x. Q\ x \mid R\ x \text{ --> } S\ x)$
$\quad \text{ --> } (\exists x. P\ x\ \&\ R\ x)$
by *blast*

Problem 25

lemma $(\exists x. P\ x)\ \&$
$\quad (\forall x. L\ x \text{ --> } \sim (M\ x\ \&\ R\ x))\ \&$
$\quad (\forall x. P\ x \text{ --> } (M\ x\ \&\ L\ x))\ \&$
$\quad ((\forall x. P\ x \text{ --> } Q\ x) \mid (\exists x. P\ x\ \&\ R\ x))$
$\quad \text{ --> } (\exists x. Q\ x\ \&\ P\ x)$
by *blast*

Problem 26; has 24 Horn clauses

lemma $((\exists x. p\ x) = (\exists x. q\ x))\ \&$
$\quad (\forall x. \forall y. p\ x\ \&\ q\ y \text{ --> } (r\ x = s\ y))$
$\quad \text{ --> } ((\forall x. p\ x \text{ --> } r\ x) = (\forall x. q\ x \text{ --> } s\ x))$
by *blast*

Problem 27; has 13 Horn clauses

lemma $(\exists x. P\ x\ \&\ \sim Q\ x)\ \&$
$\quad (\forall x. P\ x \text{ --> } R\ x)\ \&$
$\quad (\forall x. M\ x\ \&\ L\ x \text{ --> } P\ x)\ \&$
$\quad ((\exists x. R\ x\ \&\ \sim Q\ x) \text{ --> } (\forall x. L\ x \text{ --> } \sim R\ x))$
$\quad \text{ --> } (\forall x. M\ x \text{ --> } \sim L\ x)$

by *blast*

Problem 28. AMENDED; has 14 Horn clauses

lemma $(\forall x.\ P\ x\ -\!-\!>\ (\forall x.\ Q\ x))\ \&$
$((\forall x.\ Q\ x\ |\ R\ x)\ -\!-\!>\ (\exists x.\ Q\ x\ \&\ S\ x))\ \&$
$((\exists x.\ S\ x)\ -\!-\!>\ (\forall x.\ L\ x\ -\!-\!>\ M\ x))$
$-\!-\!>\ (\forall x.\ P\ x\ \&\ L\ x\ -\!-\!>\ M\ x)$
by *blast*

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

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

Problem 30

lemma $(\forall x.\ P\ x\ |\ Q\ x\ -\!-\!>\ {}^{\sim}\ R\ x)\ \&\ (\forall x.\ (Q\ x\ -\!-\!>\ {}^{\sim}\ S\ x)\ -\!-\!>\ P\ x\ \&\ R\ x)$
$-\!-\!>\ (\forall x.\ S\ x)$
by *blast*

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

lemma ${}^{\sim}(\exists x.\ P\ x\ \&\ (Q\ x\ |\ R\ x))\ \&$
$(\exists x.\ L\ x\ \&\ P\ x)\ \&$
$(\forall x.\ {}^{\sim}\ R\ x\ -\!-\!>\ M\ x)$
$-\!-\!>\ (\exists x.\ L\ x\ \&\ M\ x)$
by *blast*

Problem 32

lemma $(\forall x.\ P\ x\ \&\ (Q\ x\ |\ R\ x)-\!-\!>S\ x)\ \&$
$(\forall x.\ S\ x\ \&\ R\ x\ -\!-\!>\ L\ x)\ \&$
$(\forall x.\ M\ x\ -\!-\!>\ R\ x)$
$-\!-\!>\ (\forall x.\ P\ x\ \&\ M\ x\ -\!-\!>\ L\ x)$
by *blast*

Problem 33; has 55 Horn clauses

lemma $(\forall x.\ P\ a\ \&\ (P\ x\ -\!-\!>\ P\ b)-\!-\!>P\ c)\ =$
$(\forall x.\ ({}^{\sim}P\ a\ |\ P\ x\ |\ P\ c)\ \&\ ({}^{\sim}P\ a\ |\ {}^{\sim}P\ b\ |\ P\ c))$
by *blast*

Problem 34: Andrews's challenge has 924 Horn clauses

lemma $((\exists x.\ \forall y.\ p\ x = p\ y)\ = ((\exists x.\ q\ x) = (\forall y.\ p\ y)))\quad =$
$((\exists x.\ \forall y.\ q\ x = q\ y)\ = ((\exists x.\ p\ x) = (\forall y.\ q\ y)))$
by *blast*

Problem 35

lemma $\exists x\ y.\ P\ x\ y\ -\!-\!>\ (\forall u\ v.\ P\ u\ v)$

by *blast*

Problem 36; has 15 Horn clauses

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&\ (\forall x.\ \exists y.\ G\ x\ y)\ \&$
$\quad (\forall x\ y.\ J\ x\ y\ |\ G\ x\ y\ \text{-->}\ (\forall z.\ J\ y\ z\ |\ G\ y\ z\ \text{-->}\ H\ x\ z))$
$\quad \text{-->}\ (\forall x.\ \exists y.\ H\ x\ y)$
by *blast*

Problem 37; has 10 Horn clauses

lemma $(\forall z.\ \exists w.\ \forall x.\ \exists y.$
$\qquad (P\ x\ z\ \text{-->}\ P\ y\ w)\ \&\ P\ y\ z\ \&\ (P\ y\ w\ \text{-->}\ (\exists u.\ Q\ u\ w)))\ \&$
$\quad (\forall x\ z.\ {\sim}P\ x\ z\ \text{-->}\ (\exists y.\ Q\ y\ z))\ \&$
$\quad ((\exists x\ y.\ Q\ x\ y)\ \text{-->}\ (\forall x.\ R\ x\ x))$
$\quad \text{-->}\ (\forall x.\ \exists y.\ R\ x\ y)$
by *blast* — causes unification tracing messages

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ \text{-->}\ (\exists y.\ p\ y\ \&\ r\ x\ y))\ \text{-->}$
$\qquad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$\quad (\forall x.\ ({\sim}p\ a\ |\ p\ x\ |\ (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$\qquad ({\sim}p\ a\ |\ {\sim}(\exists y.\ p\ y\ \&\ r\ x\ y)\ |$
$\qquad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
by *blast*

Problem 39

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

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
$\quad \text{-->}\ {\sim}\ (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = ({\sim}F\ z\ x))$
by *blast*

Problem 41

lemma $(\forall z.\ (\exists y.\ (\forall x.\ f\ x\ y = (f\ x\ z\ \&\ {\sim}\ f\ x\ x))))$
$\quad \text{-->}\ {\sim}\ (\exists z.\ \forall x.\ f\ x\ z)$
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 $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}\mathit{bool})))$
$\quad \text{-->}\ (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}\mathit{bool})))$

by *blast*

Problem 44: 13 Horn clauses; 7-step proof

lemma $(\forall x.\ f\ x \dashrightarrow (\exists y.\ g\ y\ \&\ h\ x\ y\ \&\ (\exists y.\ g\ y\ \&\ {\sim}\ h\ x\ y)))\ \ \&$
$\quad(\exists x.\ j\ x\ \&\ (\forall y.\ g\ y \dashrightarrow h\ x\ y))$
$\quad\dashrightarrow (\exists x.\ j\ x\ \&\ {\sim} f\ x)$
by *blast*

Problem 45; has 27 Horn clauses; 54-step proof

lemma $(\forall x.\ f\ x\ \&\ (\forall y.\ g\ y\ \&\ h\ x\ y \dashrightarrow j\ x\ y)$
$\qquad\dashrightarrow (\forall y.\ g\ y\ \&\ h\ x\ y \dashrightarrow k\ y))\ \&$
$\quad{\sim}\ (\exists y.\ l\ y\ \&\ k\ y)\ \&$
$\quad(\exists x.\ f\ x\ \&\ (\forall y.\ h\ x\ y \dashrightarrow l\ y)$
$\qquad\ \&\ (\forall y.\ g\ y\ \&\ h\ x\ y \dashrightarrow j\ x\ y))$
$\quad\dashrightarrow (\exists x.\ f\ x\ \&\ {\sim}\ (\exists y.\ g\ y\ \&\ h\ x\ y))$
by *blast*

Problem 46; has 26 Horn clauses; 21-step proof

lemma $(\forall x.\ f\ x\ \&\ (\forall y.\ f\ y\ \&\ h\ y\ x \dashrightarrow g\ y) \dashrightarrow g\ x)\ \&$
$\quad((\exists x.\ f\ x\ \&\ {\sim} g\ x) \dashrightarrow$
$\quad(\exists x.\ f\ x\ \&\ {\sim} g\ x\ \&\ (\forall y.\ f\ y\ \&\ {\sim} g\ y \dashrightarrow j\ x\ y)))\ \&$
$\quad(\forall x\ y.\ f\ x\ \&\ f\ y\ \&\ h\ x\ y \dashrightarrow {\sim} j\ y\ x)$
$\quad\dashrightarrow (\forall x.\ f\ x \dashrightarrow g\ x)$
by *blast*

Problem 47. Schubert's Steamroller. 26 clauses; 63 Horn clauses. 87094 inferences so far. Searching to depth 36

lemma $(\forall x.\ \mathit{wolf}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{wolf}\ x)\ \&$
$\quad(\forall x.\ \mathit{fox}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{fox}\ x)\ \&$
$\quad(\forall x.\ \mathit{bird}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{bird}\ x)\ \&$
$\quad(\forall x.\ \mathit{caterpillar}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{caterpillar}\ x)\ \&$
$\quad(\forall x.\ \mathit{snail}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{snail}\ x)\ \&$
$\quad(\forall x.\ \mathit{grain}\ x \longrightarrow \mathit{plant}\ x)\ \&\ (\exists x.\ \mathit{grain}\ x)\ \&$
$\quad(\forall x.\ \mathit{animal}\ x \longrightarrow$
$\qquad((\forall y.\ \mathit{plant}\ y \longrightarrow \mathit{eats}\ x\ y)\ \ \vee$
$\qquad\ (\forall y.\ \mathit{animal}\ y\ \&\ \mathit{smaller\text{-}than}\ y\ x\ \&$
$\qquad\qquad(\exists z.\ \mathit{plant}\ z\ \&\ \mathit{eats}\ y\ z) \longrightarrow \mathit{eats}\ x\ y)))\ \&$
$\quad(\forall x\ y.\ \mathit{bird}\ y\ \&\ (\mathit{snail}\ x \vee \mathit{caterpillar}\ x) \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$\quad(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{fox}\ y \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$\quad(\forall x\ y.\ \mathit{fox}\ x\ \&\ \mathit{wolf}\ y \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$\quad(\forall x\ y.\ \mathit{wolf}\ x\ \&\ (\mathit{fox}\ y \vee \mathit{grain}\ y) \longrightarrow {\sim}\mathit{eats}\ x\ y)\ \&$
$\quad(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{caterpillar}\ y \longrightarrow \mathit{eats}\ x\ y)\ \&$
$\quad(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{snail}\ y \longrightarrow {\sim}\mathit{eats}\ x\ y)\ \&$
$\quad(\forall x.\ (\mathit{caterpillar}\ x \vee \mathit{snail}\ x) \longrightarrow (\exists y.\ \mathit{plant}\ y\ \&\ \mathit{eats}\ x\ y))$
$\quad\longrightarrow (\exists x\ y.\ \mathit{animal}\ x\ \&\ \mathit{animal}\ y\ \&\ (\exists z.\ \mathit{grain}\ z\ \&\ \mathit{eats}\ y\ z\ \&\ \mathit{eats}\ x\ y))$
by $(\mathit{tactic}\langle\langle \mathit{Meson.safe\text{-}best\text{-}meson\text{-}tac}\ 1\rangle\rangle)$

— Nearly twice as fast as *meson*, which performs iterative deepening rather than best-first search

The Los problem. Circulated by John Harrison

lemma $(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ {-}{-}{>}\ P\ x\ z)\ \&$
$(\forall x\ y\ z.\ Q\ x\ y\ \&\ Q\ y\ z\ {-}{-}{>}\ Q\ x\ z)\ \&$
$(\forall x\ y.\ P\ x\ y\ {-}{-}{>}\ P\ y\ x)\ \&$
$(\forall x\ y.\ P\ x\ y\ |\ Q\ x\ y)$
${-}{-}{>}\ (\forall x\ y.\ P\ x\ y)\ |\ (\forall x\ y.\ Q\ x\ y)$
by *meson*

A similar example, suggested by Johannes Schumann and credited to Pelletier

lemma $(\forall x\ y\ z.\ P\ x\ y\ {-}{-}{>}\ P\ y\ z\ {-}{-}{>}\ P\ x\ z)\ {-}{-}{>}$
$(\forall x\ y\ z.\ Q\ x\ y\ {-}{-}{>}\ Q\ y\ z\ {-}{-}{>}\ Q\ x\ z)\ {-}{-}{>}$
$(\forall x\ y.\ Q\ x\ y\ {-}{-}{>}\ Q\ y\ x)\ {-}{-}{>}\ \ (\forall x\ y.\ P\ x\ y\ |\ Q\ x\ y)\ {-}{-}{>}$
$(\forall x\ y.\ P\ x\ y)\ |\ (\forall x\ y.\ Q\ x\ y)$
by *meson*

Problem 50. What has this to do with equality?

lemma $(\forall x.\ P\ a\ x\ |\ (\forall y.\ P\ x\ y))\ {-}{-}{>}\ (\exists x.\ \forall y.\ P\ x\ y)$
by *blast*

Problem 54: NOT PROVED

lemma $(\forall y{::}'a.\ \exists z.\ \forall x.\ F\ x\ z = (x{=}y))\ {-}{-}{>}$
$\sim (\exists w.\ \forall x.\ F\ x\ w = (\forall u.\ F\ x\ u\ {-}{-}{>}\ (\exists y.\ F\ y\ u\ \&\ \sim (\exists z.\ F\ z\ u\ \&\ F\ z\ y))))$
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*) &
(∀ *x y*. *killed x y* --> *hates x y* & ~*richer x y*) &
(∀ *x*. *hates agatha x* --> ~*hates charles x*) &
(*hates agatha agatha* & *hates agatha charles*) &
(∀ *x*. *lives x* & ~*richer x agatha* --> *hates butler x*) &
(∀ *x*. *hates agatha x* --> *hates butler x*) &
(∀ *x*. ~*hates x agatha* | ~*hates x butler* | ~*hates x charles*) -->
(∃ *x*. *killed x agatha*)
by *meson*

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ {-}{-}{>}\ P\ x\ z)\quad {-}{-}{>}\quad P\ (f\ a\ b)\ (f\ a\ c)$
by *blast*

Problem 58: Challenge found on info-hol

lemma $\forall P\ Q\ R\ x.\ \exists v\ w.\ \forall y\ z.\ P\ x\ \&\ Q\ y\ {-}{-}{>}\ (P\ v\ |\ R\ w)\ \&\ (R\ z\ {-}{-}{>}\ Q\ v)$

by *blast*

Problem 59

lemma $(\forall x.\ P\ x = (\sim P(f\ x))) \ {-}{-}{>}\ (\exists x.\ P\ x\ \&\ \sim P(f\ x))$
by *blast*

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y\ {-}{-}{>}\ P\ z\ (f\ x))\ \&\ P\ x\ y)$
by *blast*

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ {-}{-}{>}\ p(f\ x))\ {-}{-}{>}\ p(f(f\ x))) =$
$(\forall x.\ (\sim\ p\ a \mid p\ x \mid p(f(f\ x)))\ \&$
$(\sim\ p\ a \mid \sim\ p(f\ x) \mid p(f(f\ x))))$
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: $\forall x\ y.\ \ \ T(i\ x\ y)\ \&\ T\ x\ {-}{-}{>}\ T\ y$
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

39 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 $a \neq b \Longrightarrow a \in ?X \wedge b \notin ?X$
by *blast*

39.1 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*:
$\bigwedge S{::}'a\ set\ set.\ \forall x \in S.\ \forall y \in S.\ x \subseteq y \Longrightarrow \exists z.\ S \subseteq \{z\}$
by *blast*

lemma *singleton-example-2*:
$\forall x \in S.\ \bigcup S \subseteq x \Longrightarrow \exists z.\ S \subseteq \{z\}$
— Variant of the problem above.
by *blast*

lemma $\exists !x.\ f\ (g\ x) = x \Longrightarrow \exists !y.\ g\ (f\ y) = y$
— A unique fixpoint theorem — *fast*/*best*/*meson* all fail.
by *metis*

39.2 Cantor's Theorem: There is no surjection from a set to its powerset

lemma *cantor1*: $\neg\ (\exists f{::}\ 'a \Rightarrow 'a\ set.\ \forall S.\ \exists x.\ f\ x = S)$
— Requires best-first search because it is undirectional.
by *best*

lemma $\forall f{::}\ 'a \Rightarrow 'a\ set.\ \forall x.\ f\ x \neq\ ?S\ f$
— This form displays the diagonal term.
by *best*

lemma $?S \notin range\ (f\ ::\ 'a \Rightarrow 'a\ set)$

— This form exploits the set constructs.
by (*rule notI*, *erule rangeE*, *best*)

lemma $?S \notin range\ (f :: {'a} \Rightarrow {'a}\ set)$
— Or just this!
by *best*

39.3 The Schröder-Berstein Theorem

lemma *disj-lemma*: $- (f\ `\ X) = g\ `\ (-X) \Longrightarrow f\ a = g\ b \Longrightarrow a \in X \Longrightarrow b \in X$
by *blast*

lemma *surj-if-then-else*:
$-(f\ `\ X) = g\ `\ (-X) \Longrightarrow surj\ (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g\ z)$
by (*simp add*: *surj-def*) *blast*

lemma *bij-if-then-else*:
$inj\text{-}on\ f\ X \Longrightarrow inj\text{-}on\ g\ (-X) \Longrightarrow -(f\ `\ X) = g\ `\ (-X) \Longrightarrow$
$h = (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g\ z) \Longrightarrow inj\ h \wedge surj\ h$
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*:
$inj\ (f :: {'a} \Rightarrow {'b}) \Longrightarrow inj\ (g :: {'b} \Rightarrow {'a})$
$\Longrightarrow \exists h:: {'a} \Rightarrow {'b}.\ inj\ h \wedge surj\ h$
apply (*rule decomposition* [**where** *f=f* **and** *g=g*, *THEN exE*])
apply (*rule-tac* $x = (\lambda z.\ if\ z \in x\ then\ f\ z\ else\ inv\ g\ z)$ **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 ssubst*, *subst double-complement*, *erule inv-image-comp* [*symmetric*])
done

39.4 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* ∉ *R* `` {*a*} **using** ‹*a*≠*b*› **by** *blast*
    hence *a* ∉ *R* `` {*b*} **by** (*metis Image-singleton-iff assms*(*2*) *sym-def*)
    hence *3*: *R* `` {*b*} − {*b*} <= *A* − {*a*,*b*} **using** *0 ab* **by** *blast*
    have *4*: *finite* (*A* − {*a*,*b*}) **using** ‹*finite A*› **by** *simp*
    have *?N b* <= *?n* − *2* **using** *ab* ‹*a*≠*b*› ‹*finite A*› *card-mono*[*OF 4 3*] **by** *simp*
    then show *False* **using** *Nb* ‹*card A* ≥ *2*› **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 ∃ *A*. (∀ *x* ∈ *A*. *x* ≤ (*0*::*int*))
  — Example 1, page 295.
  by *force*

lemma *D* ∈ *F* ⟹ ∃ *G*. ∀ *A* ∈ *G*. ∃ *B* ∈ *F*. *A* ⊆ *B*

— Example 2.
by *force*

lemma $P\ a \Longrightarrow \exists A.\ (\forall x \in A.\ P\ x) \wedge (\exists y.\ y \in A)$
— Example 3.
by *force*

lemma $a < b \wedge b < (c{::}int) \Longrightarrow \exists A.\ a \notin A \wedge b \in A \wedge c \notin A$
— Example 4.
by *force*

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 5, page 298.
by *force*

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 6.
by *force*

lemma $\exists A.\ a \notin A$
— Example 7.
by *force*

lemma $(\forall u\ v.\ u < (0{::}int) \longrightarrow u \neq abs\ v)$
$\longrightarrow (\exists A{::}int\ set.\ (\forall y.\ abs\ y \notin A) \wedge -2 \in A)$
— 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 $(\forall A.\ 0 \in A \wedge (\forall x \in A.\ Suc\ x \in A) \longrightarrow n \in A) \wedge$
$P\ 0 \wedge (\forall x.\ P\ x \longrightarrow P\ (Suc\ x)) \longrightarrow P\ n$
— Example 11: needs a hint.
apply *clarify*
apply (*drule-tac* $x = \{x.\ P\ x\}$ **in** *spec*)
apply *force*
done

lemma
$(\forall A.\ (0,\ 0) \in A \wedge (\forall x\ y.\ (x,\ y) \in A \longrightarrow (Suc\ x,\ Suc\ y) \in A) \longrightarrow (n,\ m) \in A)$
$\wedge\ P\ n \longrightarrow P\ m$
— Example 12.
by *auto*

lemma
$(\forall x.\ (\exists u.\ x = 2 * u) = (\neg\ (\exists v.\ Suc\ x = 2 * v))) \longrightarrow$
$(\exists A.\ \forall x.\ (x \in A) = (Suc\ x \notin A))$

— 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* $x = Suc\ v$ **and** $P = \lambda x.\ ?a\ x \neq ?b\ x$ **in** *spec*, *force*)
done

end

40 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 between*

Test data for the MESON proof procedure (Excludes the equality problems 51, 52, 56, 58)

40.1 Interactive examples

ML ⟪ *Logic.auto-rename* := *true*; *Logic.set-rename-prefix a* ⟫

ML ⟪
writelnProblem 25;
Goal $(\exists x.\ P\ x)$ & $(\forall x.\ L\ x$ --> ~ $(M\ x$ & $R\ x))$ & $(\forall x.\ P\ x$ --> $(M\ x$ & $L\ x))$ & $((\forall x.\ P\ x$ --> $Q\ x)$ | $(\exists x.\ P\ x$ & $R\ x))$ --> $(\exists x.\ Q\ x$ & $P\ x)$;
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 $((\exists x.\ p\ x) = (\exists x.\ q\ x))$ & $(\forall x.\ \forall y.\ p\ x$ & $q\ y$ −−> $(r\ x = s\ y))$ −−> $((\forall x.\ p\ x$ −−> $r\ x) = (\forall x.\ q\ x$ −−> $s\ x))$;
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 $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}bool)))$ −−> $(\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}bool)))$;
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*∗)
⟫

ML ⟪ *Logic.auto-rename* := *false*; ⟫

MORE and MUCH HARDER test data for the MESON proof procedure (courtesy John Harrison).

abbreviation *EQU001-0-ax equal* ≡ $(\forall X.\ equal(X{::}'a,X))$ &

$(\forall\, Y\, X.\ equal(X{::}'a,Y)\ {-}{-}{>}\ equal(Y{::}'a,X))$ &
$(\forall\, Y\, X\, Z.\ equal(X{::}'a,Y)$ & $equal(Y{::}'a,Z)\ {-}{-}{>}\ equal(X{::}'a,Z))$

abbreviation *BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum* $\equiv$
$(\forall\, X\ Y.\ sum(X{::}'a,Y,add(X{::}'a,Y)))$ &
$(\forall\, X\ Y.\ product(X{::}'a,Y,multiply(X{::}'a,Y)))$ &
$(\forall\, Y\, X\, Z.\ sum(X{::}'a,Y,Z)\ {-}{-}{>}\ sum(Y{::}'a,X,Z))$ &
$(\forall\, Y\, X\, Z.\ product(X{::}'a,Y,Z)\ {-}{-}{>}\ product(Y{::}'a,X,Z))$ &
$(\forall\, X.\ sum(additive\text{-}identity{::}'a,X,X))$ &
$(\forall\, X.\ sum(X{::}'a,additive\text{-}identity,X))$ &
$(\forall\, X.\ product(multiplicative\text{-}identity{::}'a,X,X))$ &
$(\forall\, X.\ product(X{::}'a,multiplicative\text{-}identity,X))$ &
$(\forall\, Y\, Z\, X\, V3\ V1\ V2\ V4.\ product(X{::}'a,Y,V1)$ & $product(X{::}'a,Z,V2)$ & $sum(Y{::}'a,Z,V3)$ & $product(X{::}'a,V3,V4)\ {-}{-}{>}\ sum(V1{::}'a,V2,V4))$ &
$(\forall\, Y\, Z\, V1\ V2\ X\ V3\ V4.\ product(X{::}'a,Y,V1)$ & $product(X{::}'a,Z,V2)$ & $sum(Y{::}'a,Z,V3)$ & $sum(V1{::}'a,V2,V4)\ {-}{-}{>}\ product(X{::}'a,V3,V4))$ &
$(\forall\, Y\, Z\, V3\ X\ V1\ V2\ V4.\ product(Y{::}'a,X,V1)$ & $product(Z{::}'a,X,V2)$ & $sum(Y{::}'a,Z,V3)$ & $product(V3{::}'a,X,V4)\ {-}{-}{>}\ sum(V1{::}'a,V2,V4))$ &
$(\forall\, Y\, Z\, V1\ V2\ V3\ X\ V4.\ product(Y{::}'a,X,V1)$ & $product(Z{::}'a,X,V2)$ & $sum(Y{::}'a,Z,V3)$ & $sum(V1{::}'a,V2,V4)\ {-}{-}{>}\ product(V3{::}'a,X,V4))$ &
$(\forall\, Y\, Z\, X\, V3\ V1\ V2\ V4.\ sum(X{::}'a,Y,V1)$ & $sum(X{::}'a,Z,V2)$ & $product(Y{::}'a,Z,V3)$ & $sum(X{::}'a,V3,V4)\ {-}{-}{>}\ product(V1{::}'a,V2,V4))$ &
$(\forall\, Y\, Z\, V1\ V2\ X\ V3\ V4.\ sum(X{::}'a,Y,V1)$ & $sum(X{::}'a,Z,V2)$ & $product(Y{::}'a,Z,V3)$ & $product(V1{::}'a,V2,V4)\ {-}{-}{>}\ sum(X{::}'a,V3,V4))$ &
$(\forall\, Y\, Z\, V3\ X\ V1\ V2\ V4.\ sum(Y{::}'a,X,V1)$ & $sum(Z{::}'a,X,V2)$ & $product(Y{::}'a,Z,V3)$ & $sum(V3{::}'a,X,V4)\ {-}{-}{>}\ product(V1{::}'a,V2,V4))$ &
$(\forall\, Y\, Z\, V1\ V2\ V3\ X\ V4.\ sum(Y{::}'a,X,V1)$ & $sum(Z{::}'a,X,V2)$ & $product(Y{::}'a,Z,V3)$ & $product(V1{::}'a,V2,V4)\ {-}{-}{>}\ sum(V3{::}'a,X,V4))$ &
$(\forall\, X.\ sum(INVERSE(X),X,multiplicative\text{-}identity))$ &
$(\forall\, X.\ sum(X{::}'a,INVERSE(X),multiplicative\text{-}identity))$ &
$(\forall\, X.\ product(INVERSE(X),X,additive\text{-}identity))$ &
$(\forall\, X.\ product(X{::}'a,INVERSE(X),additive\text{-}identity))$ &
$(\forall\, X\ Y\ U\ V.\ sum(X{::}'a,Y,U)$ & $sum(X{::}'a,Y,V)\ {-}{-}{>}\ equal(U{::}'a,V))$ &
$(\forall\, X\ Y\ U\ V.\ product(X{::}'a,Y,U)$ & $product(X{::}'a,Y,V)\ {-}{-}{>}\ equal(U{::}'a,V))$

abbreviation *BOO002-0-eq INVERSE multiply add product sum equal* $\equiv$
$(\forall\, X\ Y\ W\ Z.\ equal(X{::}'a,Y)$ & $sum(X{::}'a,W,Z)\ {-}{-}{>}\ sum(Y{::}'a,W,Z))$ &
$(\forall\, X\ W\ Y\ Z.\ equal(X{::}'a,Y)$ & $sum(W{::}'a,X,Z)\ {-}{-}{>}\ sum(W{::}'a,Y,Z))$ &
$(\forall\, X\ W\ Z\ Y.\ equal(X{::}'a,Y)$ & $sum(W{::}'a,Z,X)\ {-}{-}{>}\ sum(W{::}'a,Z,Y))$ &
$(\forall\, X\ Y\ W\ Z.\ equal(X{::}'a,Y)$ & $product(X{::}'a,W,Z)\ {-}{-}{>}\ product(Y{::}'a,W,Z))$
&
$(\forall\, X\ W\ Y\ Z.\ equal(X{::}'a,Y)$ & $product(W{::}'a,X,Z)\ {-}{-}{>}\ product(W{::}'a,Y,Z))$
&
$(\forall\, X\ W\ Z\ Y.\ equal(X{::}'a,Y)$ & $product(W{::}'a,Z,X)\ {-}{-}{>}\ product(W{::}'a,Z,Y))$
&
$(\forall\, X\ Y\ W.\ equal(X{::}'a,Y)\ {-}{-}{>}\ equal(add(X{::}'a,W),add(Y{::}'a,W)))$ &
$(\forall\, X\ W\ Y.\ equal(X{::}'a,Y)\ {-}{-}{>}\ equal(add(W{::}'a,X),add(W{::}'a,Y)))$ &
$(\forall\, X\ Y\ W.\ equal(X{::}'a,Y)\ {-}{-}{>}\ equal(multiply(X{::}'a,W),multiply(Y{::}'a,W)))$

&
 $(\forall X\ W\ Y.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(multiply(W::'a,X),multiply(W::'a,Y)))$
&
 $(\forall X\ Y.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(INVERSE(X),INVERSE(Y)))$

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 &
 $(\sim product(x::'a,x,x)) \ {-}{-}{>}\ False$
 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 &
 $(\sim sum(x::'a,x,x)) \ {-}{-}{>}\ False$
 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 &
 $(\sim sum(x::'a,multiplicative\text{-}identity,multiplicative\text{-}identity)) \ {-}{-}{>}\ False$
 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 &
 $(\sim product(x::'a,additive\text{-}identity,additive\text{-}identity)) \ {-}{-}{>}\ False$
 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 &
 $(\sim equal(INVERSE(additive\text{-}identity),multiplicative\text{-}identity)) \ {-}{-}{>}\ False$
 by *meson*

abbreviation *CAT003-0-ax f1 compos codomain domain equal there-exists equivalent* $\equiv$
$(\forall\, Y\, X.\ \mathit{equivalent}(X{::}'a,Y) \dashrightarrow \mathit{there\text{-}exists}(X))$ &
$(\forall\, X\ Y.\ \mathit{equivalent}(X{::}'a,Y) \dashrightarrow \mathit{equal}(X{::}'a,Y))$ &
$(\forall\, X\ Y.\ \mathit{there\text{-}exists}(X)\ \&\ \mathit{equal}(X{::}'a,Y) \dashrightarrow \mathit{equivalent}(X{::}'a,Y))$ &
$(\forall\, X.\ \mathit{there\text{-}exists}(\mathit{domain}(X)) \dashrightarrow \mathit{there\text{-}exists}(X))$ &
$(\forall\, X.\ \mathit{there\text{-}exists}(\mathit{codomain}(X)) \dashrightarrow \mathit{there\text{-}exists}(X))$ &
$(\forall\, Y\, X.\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y)) \dashrightarrow \mathit{there\text{-}exists}(\mathit{domain}(X)))$ &
$(\forall\, X\ Y.\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y)) \dashrightarrow \mathit{equal}(\mathit{domain}(X),\mathit{codomain}(Y)))$
&
$(\forall\, X\, Y.\ \mathit{there\text{-}exists}(\mathit{domain}(X))\ \&\ \mathit{equal}(\mathit{domain}(X),\mathit{codomain}(Y)) \dashrightarrow \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y)))$
&
$(\forall\, X\ Y\ Z.\ \mathit{equal}(\mathit{compos}(X{::}'a,\mathit{compos}(Y{::}'a,Z)),\mathit{compos}(\mathit{compos}(X{::}'a,Y),Z)))$
&
$(\forall\, X.\ \mathit{equal}(\mathit{compos}(X{::}'a,\mathit{domain}(X)),X))$ &
$(\forall\, X.\ \mathit{equal}(\mathit{compos}(\mathit{codomain}(X),X),X))$ &
$(\forall\, X\ Y.\ \mathit{equivalent}(X{::}'a,Y) \dashrightarrow \mathit{there\text{-}exists}(Y))$ &
$(\forall\, X\, Y.\ \mathit{there\text{-}exists}(X)\ \&\ \mathit{there\text{-}exists}(Y)\ \&\ \mathit{equal}(X{::}'a,Y) \dashrightarrow \mathit{equivalent}(X{::}'a,Y))$
&
$(\forall\, Y\, X.\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y)) \dashrightarrow \mathit{there\text{-}exists}(\mathit{codomain}(X)))$ &
$(\forall\, X\ Y.\ \mathit{there\text{-}exists}(\mathit{f1}(X{::}'a,Y))\ |\ \mathit{equal}(X{::}'a,Y))$ &
$(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,\mathit{f1}(X{::}'a,Y))\ |\ \mathit{equal}(Y{::}'a,\mathit{f1}(X{::}'a,Y))\ |\ \mathit{equal}(X{::}'a,Y))$
&
$(\forall\, X\, Y.\ \mathit{equal}(X{::}'a,\mathit{f1}(X{::}'a,Y))\ \&\ \mathit{equal}(Y{::}'a,\mathit{f1}(X{::}'a,Y)) \dashrightarrow \mathit{equal}(X{::}'a,Y))$

abbreviation *CAT003-0-eq f1 compos codomain domain equivalent there-exists equal* $\equiv$
$(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,Y)\ \&\ \mathit{there\text{-}exists}(X) \dashrightarrow \mathit{there\text{-}exists}(Y))$ &
$(\forall\, X\ Y\, Z.\ \mathit{equal}(X{::}'a,Y)\ \&\ \mathit{equivalent}(X{::}'a,Z) \dashrightarrow \mathit{equivalent}(Y{::}'a,Z))$ &
$(\forall\, X\ Z\ Y.\ \mathit{equal}(X{::}'a,Y)\ \&\ \mathit{equivalent}(Z{::}'a,X) \dashrightarrow \mathit{equivalent}(Z{::}'a,Y))$ &
$(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,Y) \dashrightarrow \mathit{equal}(\mathit{domain}(X),\mathit{domain}(Y)))$ &
$(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,Y) \dashrightarrow \mathit{equal}(\mathit{codomain}(X),\mathit{codomain}(Y)))$ &
$(\forall\, X\ Y\ Z.\ \mathit{equal}(X{::}'a,Y) \dashrightarrow \mathit{equal}(\mathit{compos}(X{::}'a,Z),\mathit{compos}(Y{::}'a,Z)))$ &
$(\forall\, X\ Z\ Y.\ \mathit{equal}(X{::}'a,Y) \dashrightarrow \mathit{equal}(\mathit{compos}(Z{::}'a,X),\mathit{compos}(Z{::}'a,Y)))$ &
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B) \dashrightarrow \mathit{equal}(\mathit{f1}(A{::}'a,C),\mathit{f1}(B{::}'a,C)))$ &
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E) \dashrightarrow \mathit{equal}(\mathit{f1}(F'{::}'a,D),\mathit{f1}(F'{::}'a,E)))$

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 &
$(\mathit{there\text{-}exists}(\mathit{compos}(a{::}'a,b)))$ &
$(\forall\, Y\, X\, Z.\ \mathit{equal}(\mathit{compos}(\mathit{compos}(a{::}'a,b),X),Y)\ \&\ \mathit{equal}(\mathit{compos}(\mathit{compos}(a{::}'a,b),Z),Y)$
$\dashrightarrow \mathit{equal}(X{::}'a,Z))$ &
$(\mathit{there\text{-}exists}(\mathit{compos}(b{::}'a,h)))$ &
$(\mathit{equal}(\mathit{compos}(b{::}'a,h),\mathit{compos}(b{::}'a,g)))$ &
$(\sim \mathit{equal}(h{::}'a,g)) \dashrightarrow \mathit{False}$

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 &
$(there\text{-}exists(compos(a::'a,b)))$ &
$(\forall\, Y\, X\, Z.\ equal(compos(X::'a,compos(a::'a,b)),Y)$ & $equal(compos(Z::'a,compos(a::'a,b)),Y)$ $-->$ $equal(X::'a,Z))$ &
$(there\text{-}exists(h))$ &
$(equal(compos(h::'a,a),compos(g::'a,a)))$ &
$(\sim equal(g::'a,h))$ $-->$ *False*
by *meson*

abbreviation *CAT001-0-ax equal codomain domain identity-map compos product defined* $\equiv$
$(\forall\, X\ Y.\ defined(X::'a,Y)$ $-->$ $product(X::'a,Y,compos(X::'a,Y)))$ &
$(\forall\, Z\ X\ Y.\ product(X::'a,Y,Z)$ $-->$ $defined(X::'a,Y))$ &
$(\forall\, X\ Xy\ Y\ Z.\ product(X::'a,Y,Xy)$ & $defined(Xy::'a,Z)$ $-->$ $defined(Y::'a,Z))$ &
$(\forall\, Y\ Xy\ Z\ X\ Yz.\ product(X::'a,Y,Xy)$ & $product(Y::'a,Z,Yz)$ & $defined(Xy::'a,Z)$ $-->$ $defined(X::'a,Yz))$ &
$(\forall\, Xy\ Y\ Z\ X\ Yz\ Xyz.\ product(X::'a,Y,Xy)$ & $product(Xy::'a,Z,Xyz)$ & $product(Y::'a,Z,Yz)$ $-->$ $product(X::'a,Yz,Xyz))$ &
$(\forall\, Z\ Yz\ X\ Y.\ product(Y::'a,Z,Yz)$ & $defined(X::'a,Yz)$ $-->$ $defined(X::'a,Y))$ &
$(\forall\, Y\ X\ Yz\ Xy\ Z.\ product(Y::'a,Z,Yz)$ & $product(X::'a,Y,Xy)$ & $defined(X::'a,Yz)$ $-->$ $defined(Xy::'a,Z))$ &
$(\forall\, Yz\ X\ Y\ Xy\ Z\ Xyz.\ product(Y::'a,Z,Yz)$ & $product(X::'a,Yz,Xyz)$ & $product(X::'a,Y,Xy)$ $-->$ $product(Xy::'a,Z,Xyz))$ &
$(\forall\, Y\ X\ Z.\ defined(X::'a,Y)$ & $defined(Y::'a,Z)$ & $identity\text{-}map(Y)$ $-->$ $defined(X::'a,Z))$ &
$(\forall\, X.\ identity\text{-}map(domain(X)))$ &
$(\forall\, X.\ identity\text{-}map(codomain(X)))$ &
$(\forall\, X.\ defined(X::'a,domain(X)))$ &
$(\forall\, X.\ defined(codomain(X),X))$ &
$(\forall\, X.\ product(X::'a,domain(X),X))$ &
$(\forall\, X.\ product(codomain(X),X,X))$ &
$(\forall\, X\ Y.\ defined(X::'a,Y)$ & $identity\text{-}map(X)$ $-->$ $product(X::'a,Y,Y))$ &
$(\forall\, Y\ X.\ defined(X::'a,Y)$ & $identity\text{-}map(Y)$ $-->$ $product(X::'a,Y,X))$ &
$(\forall\, X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W)$ $-->$ $equal(Z::'a,W))$

abbreviation *CAT001-0-eq compos defined identity-map codomain domain product equal* $\equiv$
$(\forall\, X\ Y\ Z\ W.\ equal(X::'a,Y)$ & $product(X::'a,Z,W)$ $-->$ $product(Y::'a,Z,W))$ &
$(\forall\, X\ Z\ Y\ W.\ equal(X::'a,Y)$ & $product(Z::'a,X,W)$ $-->$ $product(Z::'a,Y,W))$ &

(∀ X Z W Y. *equal*(X::'a,Y) & *product*(Z::'a,W,X) --> *product*(Z::'a,W,Y))
&
(∀ X Y. *equal*(X::'a,Y) --> *equal*(*domain*(X),*domain*(Y))) &
(∀ X Y. *equal*(X::'a,Y) --> *equal*(*codomain*(X),*codomain*(Y))) &
(∀ X Y. *equal*(X::'a,Y) & *identity-map*(X) --> *identity-map*(Y)) &
(∀ X Y Z. *equal*(X::'a,Y) & *defined*(X::'a,Z) --> *defined*(Y::'a,Z)) &
(∀ X Z Y. *equal*(X::'a,Y) & *defined*(Z::'a,X) --> *defined*(Z::'a,Y)) &
(∀ X Z Y. *equal*(X::'a,Y) --> *equal*(*compos*(Z::'a,X),*compos*(Z::'a,Y))) &
(∀ X Y Z. *equal*(X::'a,Y) --> *equal*(*compos*(X::'a,Z),*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 &
(*defined*(a::'a,d)) &
(*identity-map*(d)) &
(~*equal*(*domain*(a),d)) --> *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 &
(*equal*(*domain*(a),*codomain*(b))) &
(~*defined*(a::'a,b)) --> *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 &
(*defined*(a::'a,b)) &
(*defined*(b::'a,c)) &
(~*defined*(a::'a,*compos*(b::'a,c))) --> *False*
oops

lemma *COL001-2*:
EQU001-0-ax equal &
(∀ X Y Z. *equal*(*apply*(*apply*(*apply*(s::'a,X),Y),Z),*apply*(*apply*(X::'a,Z),*apply*(Y::'a,Z))))
&
(∀ Y X. *equal*(*apply*(*apply*(k::'a,X),Y),X)) &
(∀ X Y Z. *equal*(*apply*(*apply*(*apply*(b::'a,X),Y),Z),*apply*(X::'a,*apply*(Y::'a,Z))))
&
(∀ X. *equal*(*apply*(i::'a,X),X)) &
(∀ A B C. *equal*(A::'a,B) --> *equal*(*apply*(A::'a,C),*apply*(B::'a,C))) &

$(\forall D\ F'\ E.\ equal(D::'a,E)\ -->\ equal(apply(F'::'a,D),apply(F'::'a,E)))$ &
$(\forall X.\ equal(apply(apply(apply(s::'a,apply(b::'a,X)),i),apply(apply(s::'a,apply(b::'a,X)),i)),apply(x::'a,apply($
&
$(\forall Y.\ \sim equal(Y::'a,apply(combinator::'a,Y)))\ -->\ False$
by *meson*

lemma *COL023-1*:
EQU001-0-ax equal &
$(\forall X\ Y\ Z.\ equal(apply(apply(apply(b::'a,X),Y),Z),apply(X::'a,apply(Y::'a,Z))))$
&
$(\forall X\ Y\ Z.\ equal(apply(apply(apply(n::'a,X),Y),Z),apply(apply(apply(X::'a,Z),Y),Z)))$
&
$(\forall A\ B\ C.\ equal(A::'a,B)\ -->\ equal(apply(A::'a,C),apply(B::'a,C)))$ &
$(\forall D\ F'\ E.\ equal(D::'a,E)\ -->\ equal(apply(F'::'a,D),apply(F'::'a,E)))$ &
$(\forall Y.\ \sim equal(Y::'a,apply(combinator::'a,Y)))\ -->\ False$
by *meson*

lemma *COL032-1*:
EQU001-0-ax equal &
$(\forall X.\ equal(apply(m::'a,X),apply(X::'a,X)))$ &
$(\forall Y\ X\ Z.\ equal(apply(apply(apply(q::'a,X),Y),Z),apply(Y::'a,apply(X::'a,Z))))$
&
$(\forall A\ B\ C.\ equal(A::'a,B)\ -->\ equal(apply(A::'a,C),apply(B::'a,C)))$ &
$(\forall D\ F'\ E.\ equal(D::'a,E)\ -->\ equal(apply(F'::'a,D),apply(F'::'a,E)))$ &
$(\forall G\ H.\ equal(G::'a,H)\ -->\ equal(f(G),f(H)))$ &
$(\forall Y.\ \sim equal(apply(Y::'a,f(Y)),apply(f(Y),apply(Y::'a,f(Y)))))\ -->\ False$
by *meson*

lemma *COL052-2*:
EQU001-0-ax equal &
$(\forall X\ Y\ W.\ equal(response(compos(X::'a,Y),W),response(X::'a,response(Y::'a,W))))$
&
$(\forall X\ Y.\ agreeable(X)\ -->\ equal(response(X::'a,common\text{-}bird(Y)),response(Y::'a,common\text{-}bird(Y))))$
&
$(\forall Z\ X.\ equal(response(X::'a,Z),response(compatible(X),Z))\ -->\ agreeable(X))$
&
$(\forall A\ B.\ equal(A::'a,B)\ -->\ equal(common\text{-}bird(A),common\text{-}bird(B)))$ &
$(\forall C\ D.\ equal(C::'a,D)\ -->\ equal(compatible(C),compatible(D)))$ &
$(\forall Q\ R.\ equal(Q::'a,R)\ \&\ agreeable(Q)\ -->\ agreeable(R))$ &
$(\forall A\ B\ C.\ equal(A::'a,B)\ -->\ equal(compos(A::'a,C),compos(B::'a,C)))$ &
$(\forall D\ F'\ E.\ equal(D::'a,E)\ -->\ equal(compos(F'::'a,D),compos(F'::'a,E)))$ &
$(\forall G\ H\ I'.\ equal(G::'a,H)\ -->\ equal(response(G::'a,I'),response(H::'a,I')))$ &
$(\forall J\ L\ K'.\ equal(J::'a,K')\ -->\ equal(response(L::'a,J),response(L::'a,K')))$ &
$(agreeable(c))$ &
$(\sim agreeable(a))$ &
$(equal(c::'a,compos(a::'a,b)))\ -->\ False$

oops

lemma *COL075-2*:
EQU001-0-ax equal &
(∀ Y X. equal(apply(apply(k::'a,X),Y),X)) &
(∀ X Y Z. equal(apply(apply(apply(abstraction::'a,X),Y),Z),apply(apply(X::'a,apply(k::'a,Z)),apply(Y::'a,Z
&
(∀ D E F'. equal(D::'a,E) --> equal(apply(D::'a,F'),apply(E::'a,F'))) &
(∀ G I' H. equal(G::'a,H) --> equal(apply(I'::'a,G),apply(I'::'a,H))) &
(∀ A B. equal(A::'a,B) --> equal(b(A),b(B))) &
(∀ C D. equal(C::'a,D) --> equal(c(C),c(D))) &
(∀ Y. ~equal(apply(apply(Y::'a,b(Y)),c(Y)),apply(b(Y),b(Y)))) --> False
oops

lemma *COM001-1*:
(∀ Goal-state Start-state. follows(Goal-state::'a,Start-state) --> succeeds(Goal-state::'a,Start-state))
&
(∀ Goal-state Intermediate-state Start-state. succeeds(Goal-state::'a,Intermediate-state)
& succeeds(Intermediate-state::'a,Start-state) --> succeeds(Goal-state::'a,Start-state))
&
(∀ Start-state Label Goal-state. has(Start-state::'a,goto(Label)) & labels(Label::'a,Goal-state)
--> succeeds(Goal-state::'a,Start-state)) &
(∀ Start-state Condition Goal-state. has(Start-state::'a,ifthen(Condition::'a,Goal-state))
--> succeeds(Goal-state::'a,Start-state)) &
(labels(loop::'a,p3)) &
(has(p3::'a,ifthen(equal(register-j::'a,n),p4))) &
(has(p4::'a,goto(out))) &
(follows(p5::'a,p4)) &
(follows(p8::'a,p3)) &
(has(p8::'a,goto(loop))) &
(~succeeds(p3::'a,p3)) --> False
by *meson*

lemma *COM002-1*:
(∀ Goal-state Start-state. follows(Goal-state::'a,Start-state) --> succeeds(Goal-state::'a,Start-state))
&
(∀ Goal-state Intermediate-state Start-state. succeeds(Goal-state::'a,Intermediate-state)
& succeeds(Intermediate-state::'a,Start-state) --> succeeds(Goal-state::'a,Start-state))
&
(∀ Start-state Label Goal-state. has(Start-state::'a,goto(Label)) & labels(Label::'a,Goal-state)
--> succeeds(Goal-state::'a,Start-state)) &
(∀ Start-state Condition Goal-state. has(Start-state::'a,ifthen(Condition::'a,Goal-state))
--> succeeds(Goal-state::'a,Start-state)) &
(has(p1::'a,assign(register-j::'a,num0))) &
(follows(p2::'a,p1)) &
(has(p2::'a,assign(register-k::'a,num1))) &

(*labels*(*loop*::′*a*,*p3*)) &
(*follows*(*p3*::′*a*,*p2*)) &
(*has*(*p3*::′*a*,*ifthen*(*equal*(*register-j*::′*a*,*n*),*p4*))) &
(*has*(*p4*::′*a*,*goto*(*out*))) &
(*follows*(*p5*::′*a*,*p4*)) &
(*follows*(*p6*::′*a*,*p3*)) &
(*has*(*p6*::′*a*,*assign*(*register-k*::′*a*,*mtimes*(*num2*::′*a*,*register-k*)))) &
(*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*))) &
(~*succeeds*(*p3*::′*a*,*p3*)) --> *False*
by *meson*

lemma *COM002-2*:
(∀ *Goal-state Start-state*. ~(*fails*(*Goal-state*::′*a*,*Start-state*) & *follows*(*Goal-state*::′*a*,*Start-state*)))
&
(∀ *Goal-state Intermediate-state Start-state*. *fails*(*Goal-state*::′*a*,*Start-state*) -->
fails(*Goal-state*::′*a*,*Intermediate-state*) | *fails*(*Intermediate-state*::′*a*,*Start-state*)) &
(∀ *Start-state Label Goal-state*. ~(*fails*(*Goal-state*::′*a*,*Start-state*) & *has*(*Start-state*::′*a*,*goto*(*Label*))
& *labels*(*Label*::′*a*,*Goal-state*))) &
(∀ *Start-state Condition Goal-state*. ~(*fails*(*Goal-state*::′*a*,*Start-state*) & *has*(*Start-state*::′*a*,*ifthen*(*Condition*::
&
(*has*(*p1*::′*a*,*assign*(*register-j*::′*a*,*num0*))) &
(*follows*(*p2*::′*a*,*p1*)) &
(*has*(*p2*::′*a*,*assign*(*register-k*::′*a*,*num1*))) &
(*labels*(*loop*::′*a*,*p3*)) &
(*follows*(*p3*::′*a*,*p2*)) &
(*has*(*p3*::′*a*,*ifthen*(*equal*(*register-j*::′*a*,*n*),*p4*))) &
(*has*(*p4*::′*a*,*goto*(*out*))) &
(*follows*(*p5*::′*a*,*p4*)) &
(*follows*(*p6*::′*a*,*p3*)) &
(*has*(*p6*::′*a*,*assign*(*register-k*::′*a*,*mtimes*(*num2*::′*a*,*register-k*)))) &
(*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*)) --> *False*
by *meson*

lemma *COM003-2*:
(∀ *X Y Z*. *program-decides*(*X*) & *program*(*Y*) --> *decides*(*X*::′*a*,*Y*,*Z*)) &
(∀ *X*. *program-decides*(*X*) | *program*(*f2*(*X*))) &
(∀ *X*. *decides*(*X*::′*a*,*f2*(*X*),*f1*(*X*)) --> *program-decides*(*X*)) &
(∀ *X*. *program-program-decides*(*X*) --> *program*(*X*)) &
(∀ *X*. *program-program-decides*(*X*) --> *program-decides*(*X*)) &
(∀ *X*. *program*(*X*) & *program-decides*(*X*) --> *program-program-decides*(*X*)) &

(∀ X. algorithm-program-decides(X) --> algorithm(X)) &
(∀ X. algorithm-program-decides(X) --> program-decides(X)) &
(∀ X. algorithm(X) & program-decides(X) --> algorithm-program-decides(X))
&
(∀ Y X. program-halts2(X::'a, Y) --> program(X)) &
(∀ X Y. program-halts2(X::'a, Y) --> halts2(X::'a, Y)) &
(∀ X Y. program(X) & halts2(X::'a, Y) --> program-halts2(X::'a, Y)) &
(∀ W X Y Z. halts3-outputs(X::'a, Y, Z, W) --> halts3(X::'a, Y, Z)) &
(∀ Y Z X W. halts3-outputs(X::'a, Y, Z, W) --> outputs(X::'a, W)) &
(∀ Y Z X W. halts3(X::'a, Y, Z) & outputs(X::'a, W) --> halts3-outputs(X::'a, Y, Z, W))
&
(∀ Y X. program-not-halts2(X::'a, Y) --> program(X)) &
(∀ X Y. ~(program-not-halts2(X::'a, Y) & halts2(X::'a, Y))) &
(∀ X Y. program(X) --> program-not-halts2(X::'a, Y) | halts2(X::'a, Y)) &
(∀ W X Y. halts2-outputs(X::'a, Y, W) --> halts2(X::'a, Y)) &
(∀ Y X W. halts2-outputs(X::'a, Y, W) --> outputs(X::'a, W)) &
(∀ Y X W. halts2(X::'a, Y) & outputs(X::'a, W) --> halts2-outputs(X::'a, Y, W))
&
(∀ X W Y Z. program-halts2-halts3-outputs(X::'a, Y, Z, W) --> program-halts2(Y::'a, Z))
&
(∀ X Y Z W. program-halts2-halts3-outputs(X::'a, Y, Z, W) --> halts3-outputs(X::'a, Y, Z, W))
&
(∀ X Y Z W. program-halts2(Y::'a, Z) & halts3-outputs(X::'a, Y, Z, W) -->
program-halts2-halts3-outputs(X::'a, Y, Z, W)) &
(∀ X W Y Z. program-not-halts2-halts3-outputs(X::'a, Y, Z, W) --> program-not-halts2(Y::'a, Z))
&
(∀ X Y Z W. program-not-halts2-halts3-outputs(X::'a, Y, Z, W) --> halts3-outputs(X::'a, Y, Z, W))
&
(∀ X Y Z W. program-not-halts2(Y::'a, Z) & halts3-outputs(X::'a, Y, Z, W) -->
program-not-halts2-halts3-outputs(X::'a, Y, Z, W)) &
(∀ X W Y. program-halts2-halts2-outputs(X::'a, Y, W) --> program-halts2(Y::'a, Y))
&
(∀ X Y W. program-halts2-halts2-outputs(X::'a, Y, W) --> halts2-outputs(X::'a, Y, W))
&
(∀ X Y W. program-halts2(Y::'a, Y) & halts2-outputs(X::'a, Y, W) --> program-halts2-halts2-outputs(X::'a
&
(∀ X W Y. program-not-halts2-halts2-outputs(X::'a, Y, W) --> program-not-halts2(Y::'a, Y))
&
(∀ X Y W. program-not-halts2-halts2-outputs(X::'a, Y, W) --> halts2-outputs(X::'a, Y, W))
&
(∀ X Y W. program-not-halts2(Y::'a, Y) & halts2-outputs(X::'a, Y, W) -->
program-not-halts2-halts2-outputs(X::'a, Y, W)) &
(∀ X. algorithm-program-decides(X) --> program-program-decides(c1)) &
(∀ W Y Z. program-program-decides(W) --> program-halts2-halts3-outputs(W::'a, Y, Z, good))
&
(∀ W Y Z. program-program-decides(W) --> program-not-halts2-halts3-outputs(W::'a, Y, Z, bad))
&
(∀ W. program(W) & program-halts2-halts3-outputs(W::'a, f3(W), f3(W), good)
& program-not-halts2-halts3-outputs(W::'a, f3(W), f3(W), bad) --> program(c2))

&
(∀ W Y. program(W) & program-halts2-halts3-outputs(W::'a,f3(W),f3(W),good)
& program-not-halts2-halts3-outputs(W::'a,f3(W),f3(W),bad) --> program-halts2-halts2-outputs(c2::'a,Y,g
&
(∀ W Y. program(W) & program-halts2-halts3-outputs(W::'a,f3(W),f3(W),good)
& program-not-halts2-halts3-outputs(W::'a,f3(W),f3(W),bad) --> program-not-halts2-halts2-outputs(c2::'a,
&
(∀ V. program(V) & program-halts2-halts2-outputs(V::'a,f4(V),good) & program-not-halts2-halts2-outputs(V
--> program(c3)) &
(∀ V Y. program(V) & program-halts2-halts2-outputs(V::'a,f4(V),good) & program-not-halts2-halts2-outputs
& program-halts2(Y::'a,Y) --> halts2(c3::'a,Y)) &
(∀ V Y. program(V) & program-halts2-halts2-outputs(V::'a,f4(V),good) & program-not-halts2-halts2-outputs
--> program-not-halts2-halts2-outputs(c3::'a,Y,bad)) &
(algorithm-program-decides(c4)) --> False
by *meson*

lemma *COM004-1*:
EQU001-0-ax equal &
(∀ C D P Q X Y. failure-node(X::'a,or(C::'a,P)) & failure-node(Y::'a,or(D::'a,Q))
& contradictory(P::'a,Q) & siblings(X::'a,Y) --> failure-node(parent-of(X::'a,Y),or(C::'a,D)))
&
(∀ X. contradictory(negate(X),X)) &
(∀ X. contradictory(X::'a,negate(X))) &
(∀ X. siblings(left-child-of(X),right-child-of(X))) &
(∀ D E. equal(D::'a,E) --> equal(left-child-of(D),left-child-of(E))) &
(∀ F' G. equal(F'::'a,G) --> equal(negate(F'),negate(G))) &
(∀ H I' J. equal(H::'a,I') --> equal(or(H::'a,J),or(I'::'a,J))) &
(∀ K' M L. equal(K'::'a,L) --> equal(or(M::'a,K'),or(M::'a,L))) &
(∀ N O' P. equal(N::'a,O') --> equal(parent-of(N::'a,P),parent-of(O'::'a,P)))
&
(∀ Q S' R. equal(Q::'a,R) --> equal(parent-of(S'::'a,Q),parent-of(S'::'a,R)))
&
(∀ T' U. equal(T'::'a,U) --> equal(right-child-of(T'),right-child-of(U))) &
(∀ V W X. equal(V::'a,W) & contradictory(V::'a,X) --> contradictory(W::'a,X))
&
(∀ Y A1 Z. equal(Y::'a,Z) & contradictory(A1::'a,Y) --> contradictory(A1::'a,Z))
&
(∀ B1 C1 D1. equal(B1::'a,C1) & failure-node(B1::'a,D1) --> failure-node(C1::'a,D1))
&
(∀ E1 G1 F1. equal(E1::'a,F1) & failure-node(G1::'a,E1) --> failure-node(G1::'a,F1))
&
(∀ H1 I1 J1. equal(H1::'a,I1) & siblings(H1::'a,J1) --> siblings(I1::'a,J1)) &
(∀ K1 M1 L1. equal(K1::'a,L1) & siblings(M1::'a,K1) --> siblings(M1::'a,L1))
&
(failure-node(n-left::'a,or(EMPTY::'a,atom))) &
(failure-node(n-right::'a,or(EMPTY::'a,negate(atom)))) &
(equal(n-left::'a,left-child-of(n))) &
(equal(n-right::'a,right-child-of(n))) &

(∀ Z. ~failure-node(Z::'a,or(EMPTY::'a,EMPTY))) --> 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* ≡
(∀ X Y. between(X::'a,Y,X) --> equal(X::'a,Y)) &
(∀ V X Y Z. between(X::'a,Y,V) & between(Y::'a,Z,V) --> between(X::'a,Y,Z))
&
(∀ Y X V Z. between(X::'a,Y,Z) & between(X::'a,Y,V) --> equal(X::'a,Y) | between(X::'a,Z,V) | between(X::'a,V,Z)) &
(∀ Y X. equidistant(X::'a,Y,Y,X)) &
(∀ Z X Y. equidistant(X::'a,Y,Z,Z) --> equal(X::'a,Y)) &
(∀ X Y Z V V2 W. equidistant(X::'a,Y,Z,V) & equidistant(X::'a,Y,V2,W) --> equidistant(Z::'a,V,V2,W)) &
(∀ W X Z V Y. between(X::'a,W,V) & between(Y::'a,V,Z) --> between(X::'a,outer-pasch(W::'a,X,Y,Z,V
&
(∀ W X Y Z V. between(X::'a,W,V) & between(Y::'a,V,Z) --> between(Z::'a,W,outer-pasch(W::'a,X,Y,Z
&
(∀ W X Y Z V. between(X::'a,V,W) & between(Y::'a,V,Z) --> equal(X::'a,V) | between(X::'a,Z,euclid1(W::'a,X,Y,Z,V))) &
(∀ W X Y Z V. between(X::'a,V,W) & between(Y::'a,V,Z) --> equal(X::'a,V) | between(X::'a,Y,euclid2(W::'a,X,Y,Z,V))) &
(∀ W X Y Z V. between(X::'a,V,W) & between(Y::'a,V,Z) --> equal(X::'a,V) | between(euclid1(W::'a,X,Y,Z,V),W,euclid2(W::'a,X,Y,Z,V))) &
(∀ X1 Y1 X Y Z V Z1 V1. equidistant(X::'a,Y,X1,Y1) & equidistant(Y::'a,Z,Y1,Z1) & equidistant(X::'a,V,X1,V1) & equidistant(Y::'a,V,Y1,V1) & between(X::'a,Y,Z) & between(X1::'a,Y1,Z1) --> equal(X::'a,Y) | equidistant(Z::'a,V,Z1,V1)) &
(∀ X Y W V. between(X::'a,Y,extension(X::'a,Y,W,V))) &
(∀ X Y W V. equidistant(Y::'a,extension(X::'a,Y,W,V),W,V)) &
(~between(lower-dimension-point-1::'a,lower-dimension-point-2,lower-dimension-point-3))
&
(~between(lower-dimension-point-2::'a,lower-dimension-point-3,lower-dimension-point-1))
&
(~between(lower-dimension-point-3::'a,lower-dimension-point-1,lower-dimension-point-2))
&
(∀ Z X Y W V. equidistant(X::'a,W,X,V) & equidistant(Y::'a,W,Y,V) & equidistant(Z::'a,W,Z,V) --> between(X::'a,Y,Z) | between(Y::'a,Z,X) | between(Z::'a,X,Y) | equal(W::'a,V)) &
(∀ X Y Z X1 Z1 V. equidistant(V::'a,X,V,X1) & equidistant(V::'a,Z,V,Z1) & between(V::'a,X,Z) & between(X::'a,Y,Z) --> equidistant(V::'a,Y,Z,continuous(X::'a,Y,Z,X1,Z1,V)))
&
(∀ X Y Z X1 V Z1. equidistant(V::'a,X,V,X1) & equidistant(V::'a,Z,V,Z1) & between(V::'a,X,Z) & between(X::'a,Y,Z) --> between(X1::'a,continuous(X::'a,Y,Z,X1,Z1,V),Z1))

abbreviation *GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between equal* ≡
(∀ X Y W Z. equal(X::'a,Y) & between(X::'a,W,Z) --> between(Y::'a,W,Z))
&

(∀ X W Y Z. equal(X::'a,Y) & between(W::'a,X,Z) --> between(W::'a,Y,Z))
&
(∀ X W Z Y. equal(X::'a,Y) & between(W::'a,Z,X) --> between(W::'a,Z,Y))
&
(∀ X Y V W Z. equal(X::'a,Y) & equidistant(X::'a,V,W,Z) --> equidistant(Y::'a,V,W,Z)) &
(∀ X V Y W Z. equal(X::'a,Y) & equidistant(V::'a,X,W,Z) --> equidistant(V::'a,Y,W,Z)) &
(∀ X V W Y Z. equal(X::'a,Y) & equidistant(V::'a,W,X,Z) --> equidistant(V::'a,W,Y,Z)) &
(∀ X V W Z Y. equal(X::'a,Y) & equidistant(V::'a,W,Z,X) --> equidistant(V::'a,W,Z,Y)) &
(∀ X Y V1 V2 V3 V4. equal(X::'a,Y) --> equal(outer-pasch(X::'a,V1,V2,V3,V4),outer-pasch(Y::'a,V1,V
&
(∀ X V1 Y V2 V3 V4. equal(X::'a,Y) --> equal(outer-pasch(V1::'a,X,V2,V3,V4),outer-pasch(V1::'a,Y,V
&
(∀ X V1 V2 Y V3 V4. equal(X::'a,Y) --> equal(outer-pasch(V1::'a,V2,X,V3,V4),outer-pasch(V1::'a,V2,Y
&
(∀ X V1 V2 V3 Y V4. equal(X::'a,Y) --> equal(outer-pasch(V1::'a,V2,V3,X,V4),outer-pasch(V1::'a,V2,V
&
(∀ X V1 V2 V3 V4 Y. equal(X::'a,Y) --> equal(outer-pasch(V1::'a,V2,V3,V4,X),outer-pasch(V1::'a,V2,V
&
(∀ A B C D E F'. equal(A::'a,B) --> equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F')))
&
(∀ G I' H J K' L. equal(G::'a,H) --> equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L)))
&
(∀ M O' P N Q R. equal(M::'a,N) --> equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R)))
&
(∀ S' U V W T' X. equal(S'::'a,T') --> equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X)))
&
(∀ Y A1 B1 C1 D1 Z. equal(Y::'a,Z) --> equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z)
&
(∀ E1 F1 G1 H1 I1 J1. equal(E1::'a,F1) --> equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1,J
&
(∀ K1 M1 L1 N1 O1 P1. equal(K1::'a,L1) --> equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1,
&
(∀ Q1 S1 T1 R1 U1 V1. equal(Q1::'a,R1) --> equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U
&
(∀ W1 Y1 Z1 A2 X1 B2. equal(W1::'a,X1) --> equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2
&
(∀ C2 E2 F2 G2 H2 D2. equal(C2::'a,D2) --> equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H
&
(∀ X Y V1 V2 V3. equal(X::'a,Y) --> equal(extension(X::'a,V1,V2,V3),extension(Y::'a,V1,V2,V3)))
&
(∀ X V1 Y V2 V3. equal(X::'a,Y) --> equal(extension(V1::'a,X,V2,V3),extension(V1::'a,Y,V2,V3)))
&
(∀ X V1 V2 Y V3. equal(X::'a,Y) --> equal(extension(V1::'a,V2,X,V3),extension(V1::'a,V2,Y,V3)))
&
(∀ X V1 V2 V3 Y. equal(X::'a,Y) --> equal(extension(V1::'a,V2,V3,X),extension(V1::'a,V2,V3,Y)))

&
 $(\forall X\ Y\ V1\ V2\ V3\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(X::'a,V1,V2,V3,V4,V5),continuous(Y::'a,$
&
 $(\forall X\ V1\ Y\ V2\ V3\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,X,V2,V3,V4,V5),continuous(V1::'a$
&
 $(\forall X\ V1\ V2\ Y\ V3\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,V2,X,V3,V4,V5),continuous(V1::'a$
&
 $(\forall X\ V1\ V2\ V3\ Y\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,X,V4,V5),continuous(V1::'a$
&
 $(\forall X\ V1\ V2\ V3\ V4\ Y\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,V4,X,V5),continuous(V1::'a$
&
 $(\forall X\ V1\ V2\ V3\ V4\ V5\ Y.\ equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,V4,V5,X),continuous(V1::'a$

lemma *GEO003-1*:
 EQU001-0-ax equal &
 GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2 lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal between &
 GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between equal &
 $(\sim between(a::'a,b,b)) --> False$
 by *meson*

abbreviation *GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3 lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension between equal equidistant* $\equiv$
 $(\forall Y\ X.\ equidistant(X::'a,Y,Y,X))$ &
 $(\forall X\ Y\ Z\ V\ V2\ W.\ equidistant(X::'a,Y,Z,V)$ & $equidistant(X::'a,Y,V2,W) --> equidistant(Z::'a,V,V2,W))$ &
 $(\forall Z\ X\ Y.\ equidistant(X::'a,Y,Z,Z) --> equal(X::'a,Y))$ &
 $(\forall X\ Y\ W\ V.\ between(X::'a,Y,extension(X::'a,Y,W,V)))$ &
 $(\forall X\ Y\ W\ V.\ equidistant(Y::'a,extension(X::'a,Y,W,V),W,V))$ &
 $(\forall X1\ Y1\ X\ Y\ Z\ V\ Z1\ V1.\ equidistant(X::'a,Y,X1,Y1)$ & $equidistant(Y::'a,Z,Y1,Z1)$ & $equidistant(X::'a,V,X1,V1)$ & $equidistant(Y::'a,V,Y1,V1)$ & $between(X::'a,Y,Z)$ & $between(X1::'a,Y1,Z1) --> equal(X::'a,Y) \mid equidistant(Z::'a,V,Z1,V1))$ &
 $(\forall X\ Y.\ between(X::'a,Y,X) --> equal(X::'a,Y))$ &
 $(\forall U\ V\ W\ X\ Y.\ between(U::'a,V,W)$ & $between(Y::'a,X,W) --> between(V::'a,inner\text{-}pasch(U::'a,V,W,X$
&
 $(\forall V\ W\ X\ Y\ U.\ between(U::'a,V,W)$ & $between(Y::'a,X,W) --> between(X::'a,inner\text{-}pasch(U::'a,V,W,X,$
&
 $(\sim between(lower\text{-}dimension\text{-}point\text{-}1::'a,lower\text{-}dimension\text{-}point\text{-}2,lower\text{-}dimension\text{-}point\text{-}3))$
&
 $(\sim between(lower\text{-}dimension\text{-}point\text{-}2::'a,lower\text{-}dimension\text{-}point\text{-}3,lower\text{-}dimension\text{-}point\text{-}1))$
&
 $(\sim between(lower\text{-}dimension\text{-}point\text{-}3::'a,lower\text{-}dimension\text{-}point\text{-}1,lower\text{-}dimension\text{-}point\text{-}2))$
&
 $(\forall Z\ X\ Y\ W\ V.\ equidistant(X::'a,W,X,V)$ & $equidistant(Y::'a,W,Y,V)$ & *equidis-*

tant(Z::'a,W,Z,V) --> between(X::'a,Y,Z) | between(Y::'a,Z,X) | between(Z::'a,X,Y)
| equal(W::'a,V)) &
(∀ U V W X Y. between(U::'a,W,Y) & between(V::'a,W,X) --> equal(U::'a,W)
| between(U::'a,V,euclid1(U::'a,V,W,X,Y))) &
(∀ U V W X Y. between(U::'a,W,Y) & between(V::'a,W,X) --> equal(U::'a,W)
| between(U::'a,X,euclid2(U::'a,V,W,X,Y))) &
(∀ U V W X Y. between(U::'a,W,Y) & between(V::'a,W,X) --> equal(U::'a,W)
| between(euclid1(U::'a,V,W,X,Y),Y,euclid2(U::'a,V,W,X,Y))) &
(∀ U V V1 W X X1. equidistant(U::'a,V,U,V1) & equidistant(U::'a,X,U,X1) &
between(U::'a,V,X) & between(V::'a,W,X) --> between(V1::'a,continuous(U::'a,V,V1,W,X,X1),X1))
&
(∀ U V V1 W X X1. equidistant(U::'a,V,U,V1) & equidistant(U::'a,X,U,X1) &
between(U::'a,V,X) & between(V::'a,W,X) --> equidistant(U::'a,W,U,continuous(U::'a,V,V1,W,X,X1))
&
(∀ X Y W Z. equal(X::'a,Y) & between(X::'a,W,Z) --> between(Y::'a,W,Z))
&
(∀ X W Y Z. equal(X::'a,Y) & between(W::'a,X,Z) --> between(W::'a,Y,Z))
&
(∀ X W Z Y. equal(X::'a,Y) & between(W::'a,Z,X) --> between(W::'a,Z,Y))
&
(∀ X Y V W Z. equal(X::'a,Y) & equidistant(X::'a,V,W,Z) --> equidistant(Y::'a,V,W,Z)) &
(∀ X V Y W Z. equal(X::'a,Y) & equidistant(V::'a,X,W,Z) --> equidistant(V::'a,Y,W,Z)) &
(∀ X V W Y Z. equal(X::'a,Y) & equidistant(V::'a,W,X,Z) --> equidistant(V::'a,W,Y,Z)) &
(∀ X V W Z Y. equal(X::'a,Y) & equidistant(V::'a,W,Z,X) --> equidistant(V::'a,W,Z,Y)) &
(∀ X Y V1 V2 V3 V4. equal(X::'a,Y) --> equal(inner-pasch(X::'a,V1,V2,V3,V4),inner-pasch(Y::'a,V1,V
&
(∀ X V1 Y V2 V3 V4. equal(X::'a,Y) --> equal(inner-pasch(V1::'a,X,V2,V3,V4),inner-pasch(V1::'a,Y,V
&
(∀ X V1 V2 Y V3 V4. equal(X::'a,Y) --> equal(inner-pasch(V1::'a,V2,X,V3,V4),inner-pasch(V1::'a,V2,
&
(∀ X V1 V2 V3 Y V4. equal(X::'a,Y) --> equal(inner-pasch(V1::'a,V2,V3,X,V4),inner-pasch(V1::'a,V2,
&
(∀ X V1 V2 V3 V4 Y. equal(X::'a,Y) --> equal(inner-pasch(V1::'a,V2,V3,V4,X),inner-pasch(V1::'a,V2,
&
(∀ A B C D E F'. equal(A::'a,B) --> equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F')))
&
(∀ G I' H J K' L. equal(G::'a,H) --> equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L)))
&
(∀ M O' P N Q R. equal(M::'a,N) --> equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R)))
&
(∀ S' U V W T' X. equal(S'::'a,T') --> equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X)))
&
(∀ Y A1 B1 C1 D1 Z. equal(Y::'a,Z) --> equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z)
&
(∀ E1 F1 G1 H1 I1 J1. equal(E1::'a,F1) --> equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1,

&
(∀ K1 M1 L1 N1 O1 P1. equal(K1::'a,L1) --> equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1,
&
(∀ Q1 S1 T1 R1 U1 V1. equal(Q1::'a,R1) --> equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U
&
(∀ W1 Y1 Z1 A2 X1 B2. equal(W1::'a,X1) --> equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2
&
(∀ C2 E2 F2 G2 H2 D2. equal(C2::'a,D2) --> equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H
&
(∀ X Y V1 V2 V3. equal(X::'a,Y) --> equal(extension(X::'a,V1,V2,V3),extension(Y::'a,V1,V2,V3)))
&
(∀ X V1 Y V2 V3. equal(X::'a,Y) --> equal(extension(V1::'a,X,V2,V3),extension(V1::'a,Y,V2,V3)))
&
(∀ X V1 V2 Y V3. equal(X::'a,Y) --> equal(extension(V1::'a,V2,X,V3),extension(V1::'a,V2,Y,V3)))
&
(∀ X V1 V2 V3 Y. equal(X::'a,Y) --> equal(extension(V1::'a,V2,V3,X),extension(V1::'a,V2,V3,Y)))
&
(∀ X Y V1 V2 V3 V4 V5. equal(X::'a,Y) --> equal(continuous(X::'a,V1,V2,V3,V4,V5),continuous(Y::'a,
&
(∀ X V1 Y V2 V3 V4 V5. equal(X::'a,Y) --> equal(continuous(V1::'a,X,V2,V3,V4,V5),continuous(V1::'a
&
(∀ X V1 V2 Y V3 V4 V5. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,X,V3,V4,V5),continuous(V1::'a
&
(∀ X V1 V2 V3 Y V4 V5. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,X,V4,V5),continuous(V1::'a
&
(∀ X V1 V2 V3 V4 Y V5. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,V4,X,V5),continuous(V1::'a
&
(∀ X V1 V2 V3 V4 V5 Y. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,V4,V5,X),continuous(V1::'a

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 &
(equidistant(u::'a,v,w,x)) &
(~equidistant(u::'a,v,x,w)) --> 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 &
(∀ U V. equal(reflection(U::'a,V),extension(U::'a,V,U,V))) &
(∀ X Y Z. equal(X::'a,Y) --> equal(reflection(X::'a,Z),reflection(Y::'a,Z))) &
(∀ A1 C1 B1. equal(A1::'a,B1) --> equal(reflection(C1::'a,A1),reflection(C1::'a,B1)))
&

(∀ U V. *equidistant*(U::'a,V,U,V)) &
(∀ W X U V. *equidistant*(U::'a,V,W,X) --> *equidistant*(W::'a,X,U,V)) &
(∀ V U W X. *equidistant*(U::'a,V,W,X) --> *equidistant*(V::'a,U,W,X)) &
(∀ U V X W. *equidistant*(U::'a,V,W,X) --> *equidistant*(U::'a,V,X,W)) &
(∀ V U X W. *equidistant*(U::'a,V,W,X) --> *equidistant*(V::'a,U,X,W)) &
(∀ W X V U. *equidistant*(U::'a,V,W,X) --> *equidistant*(W::'a,X,V,U)) &
(∀ X W U V. *equidistant*(U::'a,V,W,X) --> *equidistant*(X::'a,W,U,V)) &
(∀ X W V U. *equidistant*(U::'a,V,W,X) --> *equidistant*(X::'a,W,V,U)) &
(∀ W X U V Y Z. *equidistant*(U::'a,V,W,X) & *equidistant*(W::'a,X,Y,Z) -->
equidistant(U::'a,V,Y,Z)) &
(∀ U V W. *equal*(V::'a,*extension*(U::'a,V,W,W))) &
(∀ W X U V Y. *equal*(Y::'a,*extension*(U::'a,V,W,X)) --> *between*(U::'a,V,Y))
&
(∀ U V. *between*(U::'a,V,*reflection*(U::'a,V))) &
(∀ U V. *equidistant*(V::'a,*reflection*(U::'a,V),U,V)) &
(∀ U V. *equal*(U::'a,V) --> *equal*(V::'a,*reflection*(U::'a,V))) &
(∀ U. *equal*(U::'a,*reflection*(U::'a,U))) &
(∀ U V. *equal*(V::'a,*reflection*(U::'a,V)) --> *equal*(U::'a,V)) &
(∀ U V. *equidistant*(U::'a,U,V,V)) &
(∀ V V1 U W U1 W1. *equidistant*(U::'a,V,U1,V1) & *equidistant*(V::'a,W,V1,W1)
& *between*(U::'a,V,W) & *between*(U1::'a,V1,W1) --> *equidistant*(U::'a,W,U1,W1))
&
(∀ U V W X. *between*(U::'a,V,W) & *between*(U::'a,V,X) & *equidistant*(V::'a,W,V,X)
--> *equal*(U::'a,V) | *equal*(W::'a,X)) &
(*between*(u::'a,v,w)) &
(~*equal*(u::'a,v)) &
(~*equal*(w::'a,*extension*(u::'a,v,v,w))) --> *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 &
(∀ U V. *equal*(*reflection*(U::'a,V),*extension*(U::'a,V,U,V))) &
(∀ X Y Z. *equal*(X::'a,Y) --> *equal*(*reflection*(X::'a,Z),*reflection*(Y::'a,Z))) &
(∀ A1 C1 B1. *equal*(A1::'a,B1) --> *equal*(*reflection*(C1::'a,A1),*reflection*(C1::'a,B1)))
&
(*equal*(v::'a,*reflection*(u::'a,v))) &
(~*equal*(u::'a,v)) --> *False*
oops

lemma *GEO079-1*:
(∀ U V W X Y Z. *right-angle*(U::'a,V,W) & *right-angle*(X::'a,Y,Z) --> *eq*(U::'a,V,W,X,Y,Z))
&
(∀ U V W X Y Z. *CONGRUENT*(U::'a,V,W,X,Y,Z) --> *eq*(U::'a,V,W,X,Y,Z))
&

$(\forall\ V\ W\ U\ X.\ trapezoid(U::'a,V,W,X) \ {-}{-}{>}\ parallel(V::'a,W,U,X))$ &
$(\forall\ U\ V\ X\ Y.\ parallel(U::'a,V,X,Y) \ {-}{-}{>}\ eq(X::'a,V,U,V,X,Y))$ &
$(trapezoid(a::'a,b,c,d))$ &
$(\sim eq(a::'a,c,b,c,a,d)) \ {-}{-}{>}\ False$
by *meson*

abbreviation *GRP003-0-ax equal multiply INVERSE identity product* $\equiv$
$(\forall\ X.\ product(identity::'a,X,X))$ &
$(\forall\ X.\ product(X::'a,identity,X))$ &
$(\forall\ X.\ product(INVERSE(X),X,identity))$ &
$(\forall\ X.\ product(X::'a,INVERSE(X),identity))$ &
$(\forall\ X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall\ X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W) \ {-}{-}{>}\ equal(Z::'a,W))$
&
$(\forall\ Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
${-}{-}{>}\ product(X::'a,V,W))$ &
$(\forall\ Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
${-}{-}{>}\ product(U::'a,Z,W))$

abbreviation *GRP003-0-eq product multiply INVERSE equal* $\equiv$
$(\forall\ X\ Y.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(INVERSE(X),INVERSE(Y)))$ &
$(\forall\ X\ Y\ W.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall\ X\ W\ Y.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(multiply(W::'a,X),multiply(W::'a,Y)))$
&
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z) \ {-}{-}{>}\ product(Y::'a,W,Z))$
&
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z) \ {-}{-}{>}\ product(W::'a,Y,Z))$
&
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X) \ {-}{-}{>}\ product(W::'a,Z,Y))$

lemma *GRP001-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall\ X.\ product(X::'a,X,identity))$ &
$(product(a::'a,b,c))$ &
$(\sim product(b::'a,a,c)) \ {-}{-}{>}\ False$
oops

lemma *GRP008-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall\ A\ B.\ equal(A::'a,B) \ {-}{-}{>}\ equal(h(A),h(B)))$ &
$(\forall\ C\ D.\ equal(C::'a,D) \ {-}{-}{>}\ equal(j(C),j(D)))$ &
$(\forall\ A\ B.\ equal(A::'a,B)$ & $q(A) \ {-}{-}{>}\ q(B))$ &

(∀ B A C. q(A) & product(A::'a,B,C) --> product(B::'a,A,C)) &
(∀ A. product(j(A),A,h(A)) | product(A::'a,j(A),h(A)) | q(A)) &
(∀ A. product(j(A),A,h(A)) & product(A::'a,j(A),h(A)) --> q(A)) &
(~q(identity)) --> 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 &
(∀ A. product(A::'a,A,identity)) &
(product(a::'a,b,c)) &
(product(INVERSE(a),INVERSE(b),d)) &
(∀ A C B. product(INVERSE(A),INVERSE(B),C) --> product(A::'a,C,B)) &
(~product(c::'a,d,identity)) --> False
oops

lemma *GRP037-3*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
(∀ A B C. subgroup-member(A) & subgroup-member(B) & product(A::'a,INVERSE(B),C)
--> subgroup-member(C)) &
(∀ A B. equal(A::'a,B) & subgroup-member(A) --> subgroup-member(B)) &
(∀ A. subgroup-member(A) --> product(Gidentity::'a,A,A)) &
(∀ A. subgroup-member(A) --> product(A::'a,Gidentity,A)) &
(∀ A. subgroup-member(A) --> product(A::'a,Ginverse(A),Gidentity)) &
(∀ A. subgroup-member(A) --> product(Ginverse(A),A,Gidentity)) &
(∀ A. subgroup-member(A) --> subgroup-member(Ginverse(A))) &
(∀ A B. equal(A::'a,B) --> equal(Ginverse(A),Ginverse(B))) &
(∀ A C D B. product(A::'a,B,C) & product(A::'a,D,C) --> equal(D::'a,B)) &
(∀ B C D A. product(A::'a,B,C) & product(D::'a,B,C) --> equal(D::'a,A)) &
(subgroup-member(a)) &
(subgroup-member(Gidentity)) &
(~equal(INVERSE(a),Ginverse(a))) --> False
by *meson*

lemma *GRP031-2*:
(∀ X Y. product(X::'a,Y,multiply(X::'a,Y))) &
(∀ X Y Z W. product(X::'a,Y,Z) & product(X::'a,Y,W) --> equal(Z::'a,W))
&
(∀ Y U Z X V W. product(X::'a,Y,U) & product(Y::'a,Z,V) & product(U::'a,Z,W)
--> product(X::'a,V,W)) &
(∀ Y X V U Z W. product(X::'a,Y,U) & product(Y::'a,Z,V) & product(X::'a,V,W)
--> product(U::'a,Z,W)) &
(∀ A. product(A::'a,INVERSE(A),identity)) &

($\forall$ *A*. *product*(*A*::'*a*,*identity*,*A*)) &
($\forall$ *A*. ~*product*(*A*::'*a*,*a*,*identity*)) --> *False*
by *meson*

lemma *GRP034-4*:
($\forall$ *X Y*. *product*(*X*::'*a*, *Y*,*multiply*(*X*::'*a*, *Y*))) &
($\forall$ *X*. *product*(*identity*::'*a*,*X*,*X*)) &
($\forall$ *X*. *product*(*X*::'*a*,*identity*,*X*)) &
($\forall$ *X*. *product*(*X*::'*a*,*INVERSE*(*X*),*identity*)) &
($\forall$ *Y U Z X V W*. *product*(*X*::'*a*, *Y*, *U*) & *product*(*Y*::'*a*,*Z*, *V*) & *product*(*U*::'*a*,*Z*, *W*)
--> *product*(*X*::'*a*, *V*, *W*)) &
($\forall$ *Y X V U Z W*. *product*(*X*::'*a*, *Y*, *U*) & *product*(*Y*::'*a*,*Z*, *V*) & *product*(*X*::'*a*, *V*, *W*)
--> *product*(*U*::'*a*,*Z*, *W*)) &
($\forall$ *B A C*. *subgroup-member*(*A*) & *subgroup-member*(*B*) & *product*(*B*::'*a*,*INVERSE*(*A*),*C*)
--> *subgroup-member*(*C*)) &
(*subgroup-member*(*a*)) &
(~*subgroup-member*(*INVERSE*(*a*))) --> *False*
by *meson*

lemma *GRP047-2*:
($\forall$ *X*. *product*(*identity*::'*a*,*X*,*X*)) &
($\forall$ *X*. *product*(*INVERSE*(*X*),*X*,*identity*)) &
($\forall$ *X Y*. *product*(*X*::'*a*, *Y*,*multiply*(*X*::'*a*, *Y*))) &
($\forall$ *X Y Z W*. *product*(*X*::'*a*, *Y*,*Z*) & *product*(*X*::'*a*, *Y*, *W*) --> *equal*(*Z*::'*a*, *W*))
&
($\forall$ *Y U Z X V W*. *product*(*X*::'*a*, *Y*, *U*) & *product*(*Y*::'*a*,*Z*, *V*) & *product*(*U*::'*a*,*Z*, *W*)
--> *product*(*X*::'*a*, *V*, *W*)) &
($\forall$ *Y X V U Z W*. *product*(*X*::'*a*, *Y*, *U*) & *product*(*Y*::'*a*,*Z*, *V*) & *product*(*X*::'*a*, *V*, *W*)
--> *product*(*U*::'*a*,*Z*, *W*)) &
($\forall$ *X W Z Y*. *equal*(*X*::'*a*, *Y*) & *product*(*W*::'*a*,*Z*,*X*) --> *product*(*W*::'*a*,*Z*, *Y*))
&
(*equal*(*a*::'*a*,*b*)) &
(~*equal*(*multiply*(*c*::'*a*,*a*),*multiply*(*c*::'*a*,*b*))) --> *False*
by *meson*

lemma *GRP130-1-002*:
(*group-element*(*e-1*)) &
(*group-element*(*e-2*)) &
(~*equal*(*e-1*::'*a*,*e-2*)) &
(~*equal*(*e-2*::'*a*,*e-1*)) &
($\forall$ *X Y*. *group-element*(*X*) & *group-element*(*Y*) --> *product*(*X*::'*a*, *Y*,*e-1*) |
product(*X*::'*a*, *Y*,*e-2*)) &
($\forall$ *X Y W Z*. *product*(*X*::'*a*, *Y*, *W*) & *product*(*X*::'*a*, *Y*,*Z*) --> *equal*(*W*::'*a*,*Z*))
&
($\forall$ *X Y W Z*. *product*(*X*::'*a*, *W*, *Y*) & *product*(*X*::'*a*,*Z*, *Y*) --> *equal*(*W*::'*a*,*Z*))
&

(∀ Y X W Z. product(W::′a,Y,X) & product(Z::′a,Y,X) --> equal(W::′a,Z))
&
(∀ Z1 Z2 Y X. product(X::′a,Y,Z1) & product(X::′a,Z1,Z2) --> product(Z2::′a,Y,X))
--> False
oops

abbreviation *GRP004-0-ax INVERSE identity multiply equal* ≡
(∀ X. equal(multiply(identity::′a,X),X)) &
(∀ X. equal(multiply(INVERSE(X),X),identity)) &
(∀ X Y Z. equal(multiply(multiply(X::′a,Y),Z),multiply(X::′a,multiply(Y::′a,Z))))
&
(∀ A B. equal(A::′a,B) --> equal(INVERSE(A),INVERSE(B))) &
(∀ C D E. equal(C::′a,D) --> equal(multiply(C::′a,E),multiply(D::′a,E))) &
(∀ F′ H G. equal(F′::′a,G) --> equal(multiply(H::′a,F′),multiply(H::′a,G)))

abbreviation *GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal*
≡
(∀ Y X. equal(greatest-lower-bound(X::′a,Y),greatest-lower-bound(Y::′a,X))) &
(∀ Y X. equal(least-upper-bound(X::′a,Y),least-upper-bound(Y::′a,X))) &
(∀ X Y Z. equal(greatest-lower-bound(X::′a,greatest-lower-bound(Y::′a,Z)),greatest-lower-bound(greatest-lowe
&
(∀ X Y Z. equal(least-upper-bound(X::′a,least-upper-bound(Y::′a,Z)),least-upper-bound(least-upper-bound(X:
&
(∀ X. equal(least-upper-bound(X::′a,X),X)) &
(∀ X. equal(greatest-lower-bound(X::′a,X),X)) &
(∀ Y X. equal(least-upper-bound(X::′a,greatest-lower-bound(X::′a,Y)),X)) &
(∀ Y X. equal(greatest-lower-bound(X::′a,least-upper-bound(X::′a,Y)),X)) &
(∀ Y X Z. equal(multiply(X::′a,least-upper-bound(Y::′a,Z)),least-upper-bound(multiply(X::′a,Y),multiply(X:
&
(∀ Y X Z. equal(multiply(X::′a,greatest-lower-bound(Y::′a,Z)),greatest-lower-bound(multiply(X::′a,Y),multip
&
(∀ Y Z X. equal(multiply(least-upper-bound(Y::′a,Z),X),least-upper-bound(multiply(Y::′a,X),multiply(Z::′a,
&
(∀ Y Z X. equal(multiply(greatest-lower-bound(Y::′a,Z),X),greatest-lower-bound(multiply(Y::′a,X),multiply(
&
(∀ A B C. equal(A::′a,B) --> equal(greatest-lower-bound(A::′a,C),greatest-lower-bound(B::′a,C)))
&
(∀ A C B. equal(A::′a,B) --> equal(greatest-lower-bound(C::′a,A),greatest-lower-bound(C::′a,B)))
&
(∀ A B C. equal(A::′a,B) --> equal(least-upper-bound(A::′a,C),least-upper-bound(B::′a,C)))
&
(∀ A C B. equal(A::′a,B) --> equal(least-upper-bound(C::′a,A),least-upper-bound(C::′a,B)))
&
(∀ A B C. equal(A::′a,B) --> equal(multiply(A::′a,C),multiply(B::′a,C))) &
(∀ A C B. equal(A::′a,B) --> equal(multiply(C::′a,A),multiply(C::′a,B)))

lemma *GRP156-1*:
EQU001-0-ax equal &

GRP004-0-ax INVERSE identity multiply equal &
GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal &
(*equal*(*least-upper-bound*(*a*::′*a*,*b*),*b*)) &
(~*equal*(*greatest-lower-bound*(*multiply*(*a*::′*a*,*c*),*multiply*(*b*::′*a*,*c*)),*multiply*(*a*::′*a*,*c*)))
−−> *False*
by *meson*

lemma *GRP168-1*:
EQU001-0-ax equal &
GRP004-0-ax INVERSE identity multiply equal &
GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal &
(*equal*(*least-upper-bound*(*a*::′*a*,*b*),*b*)) &
(~*equal*(*least-upper-bound*(*multiply*(*INVERSE*(*c*),*multiply*(*a*::′*a*,*c*)),*multiply*(*INVERSE*(*c*),*multiply*(*b*::′*a*,*c*))
−−> *False*
by *meson*

abbreviation *HEN002-0-ax identity Zero Divide equal mless-equal* ≡
(∀ *X Y*. *mless-equal*(*X*::′*a*,*Y*) −−> *equal*(*Divide*(*X*::′*a*,*Y*),*Zero*)) &
(∀ *X Y*. *equal*(*Divide*(*X*::′*a*,*Y*),*Zero*) −−> *mless-equal*(*X*::′*a*,*Y*)) &
(∀ *Y X*. *mless-equal*(*Divide*(*X*::′*a*,*Y*),*X*)) &
(∀ *X Y Z*. *mless-equal*(*Divide*(*Divide*(*X*::′*a*,*Z*),*Divide*(*Y*::′*a*,*Z*)),*Divide*(*Divide*(*X*::′*a*,*Y*),*Z*)))
&
(∀ *X*. *mless-equal*(*Zero*::′*a*,*X*)) &
(∀ *X Y*. *mless-equal*(*X*::′*a*,*Y*) & *mless-equal*(*Y*::′*a*,*X*) −−> *equal*(*X*::′*a*,*Y*)) &
(∀ *X*. *mless-equal*(*X*::′*a*,*identity*))

abbreviation *HEN002-0-eq mless-equal Divide equal* ≡
(∀ *A B C*. *equal*(*A*::′*a*,*B*) −−> *equal*(*Divide*(*A*::′*a*,*C*),*Divide*(*B*::′*a*,*C*))) &
(∀ *D F′ E*. *equal*(*D*::′*a*,*E*) −−> *equal*(*Divide*(*F′*::′*a*,*D*),*Divide*(*F′*::′*a*,*E*))) &
(∀ *G H I′*. *equal*(*G*::′*a*,*H*) & *mless-equal*(*G*::′*a*,*I′*) −−> *mless-equal*(*H*::′*a*,*I′*)) &
(∀ *J L K′*. *equal*(*J*::′*a*,*K′*) & *mless-equal*(*L*::′*a*,*J*) −−> *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 &
(~*equal*(*Divide*(*a*::′*a*,*a*),*Zero*)) −−> *False*
oops

lemma *HEN007-2*:
EQU001-0-ax equal &
(∀ *X Y*. *mless-equal*(*X*::′*a*,*Y*) −−> *quotient*(*X*::′*a*,*Y*,*Zero*)) &
(∀ *X Y*. *quotient*(*X*::′*a*,*Y*,*Zero*) −−> *mless-equal*(*X*::′*a*,*Y*)) &
(∀ *Y Z X*. *quotient*(*X*::′*a*,*Y*,*Z*) −−> *mless-equal*(*Z*::′*a*,*X*)) &
(∀ *Y X V3 V2 V1 Z V4 V5*. *quotient*(*X*::′*a*,*Y*,*V1*) & *quotient*(*Y*::′*a*,*Z*,*V2*) &
quotient(*X*::′*a*,*Z*,*V3*) & *quotient*(*V3*::′*a*,*V2*,*V4*) & *quotient*(*V1*::′*a*,*Z*,*V5*) −−>

$mless\text{-}equal(V4::'a, V5))$ &
 $(\forall X.\ mless\text{-}equal(Zero::'a,X))$ &
 $(\forall X\ Y.\ mless\text{-}equal(X::'a,Y)$ & $mless\text{-}equal(Y::'a,X) \dashrightarrow equal(X::'a,Y))$ &
 $(\forall X.\ mless\text{-}equal(X::'a,identity))$ &
 $(\forall X\ Y.\ quotient(X::'a,Y,Divide(X::'a,Y)))$ &
 $(\forall X\ Y\ Z\ W.\ quotient(X::'a,Y,Z)$ & $quotient(X::'a,Y,W) \dashrightarrow equal(Z::'a,W))$
&
 $(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $quotient(X::'a,W,Z) \dashrightarrow quotient(Y::'a,W,Z))$
&
 $(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $quotient(W::'a,X,Z) \dashrightarrow quotient(W::'a,Y,Z))$
&
 $(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $quotient(W::'a,Z,X) \dashrightarrow quotient(W::'a,Z,Y))$
&
 $(\forall X\ Z\ Y.\ equal(X::'a,Y)$ & $mless\text{-}equal(Z::'a,X) \dashrightarrow mless\text{-}equal(Z::'a,Y))$ &
 $(\forall X\ Y\ Z.\ equal(X::'a,Y)$ & $mless\text{-}equal(X::'a,Z) \dashrightarrow mless\text{-}equal(Y::'a,Z))$ &
 $(\forall X\ Y\ W.\ equal(X::'a,Y) \dashrightarrow equal(Divide(X::'a,W),Divide(Y::'a,W)))$ &
 $(\forall X\ W\ Y.\ equal(X::'a,Y) \dashrightarrow equal(Divide(W::'a,X),Divide(W::'a,Y)))$ &
 $(\forall X.\ quotient(X::'a,identity,Zero))$ &
 $(\forall X.\ quotient(Zero::'a,X,Zero))$ &
 $(\forall X.\ quotient(X::'a,X,Zero))$ &
 $(\forall X.\ quotient(X::'a,Zero,X))$ &
 $(\forall Y\ X\ Z.\ mless\text{-}equal(X::'a,Y)$ & $mless\text{-}equal(Y::'a,Z) \dashrightarrow mless\text{-}equal(X::'a,Z))$
&
 $(\forall W1\ X\ Z\ W2\ Y.\ quotient(X::'a,Y,W1)$ & $mless\text{-}equal(W1::'a,Z)$ & $quotient(X::'a,Z,W2)$
$\dashrightarrow mless\text{-}equal(W2::'a,Y))$ &
 $(mless\text{-}equal(x::'a,y))$ &
 $(quotient(z::'a,y,zQy))$ &
 $(quotient(z::'a,x,zQx))$ &
 $(\sim mless\text{-}equal(zQy::'a,zQx)) \dashrightarrow False$
 oops

lemma *HEN008-4*:
 EQU001-0-ax equal &
 HEN002-0-ax identity Zero Divide equal mless-equal &
 HEN002-0-eq mless-equal Divide equal &
 $(\forall X.\ equal(Divide(X::'a,identity),Zero))$ &
 $(\forall X.\ equal(Divide(Zero::'a,X),Zero))$ &
 $(\forall X.\ equal(Divide(X::'a,X),Zero))$ &
 $(equal(Divide(a::'a,Zero),a))$ &
 $(\forall Y\ X\ Z.\ mless\text{-}equal(X::'a,Y)$ & $mless\text{-}equal(Y::'a,Z) \dashrightarrow mless\text{-}equal(X::'a,Z))$
&
 $(\forall X\ Z\ Y.\ mless\text{-}equal(Divide(X::'a,Y),Z) \dashrightarrow mless\text{-}equal(Divide(X::'a,Z),Y))$
&
 $(\forall Y\ Z\ X.\ mless\text{-}equal(X::'a,Y) \dashrightarrow mless\text{-}equal(Divide(Z::'a,Y),Divide(Z::'a,X)))$
&
 $(mless\text{-}equal(a::'a,b))$ &
 $(\sim mless\text{-}equal(Divide(a::'a,c),Divide(b::'a,c))) \dashrightarrow False$
 oops

lemma *HEN009-5*:
EQU001-0-ax equal &
$(\forall\, Y\, X.\ equal(Divide(Divide(X::'a,Y),X),Zero))$ &
$(\forall\, X\, Y\, Z.\ equal(Divide(Divide(Divide(X::'a,Z),Divide(Y::'a,Z)),Divide(Divide(X::'a,Y),Z)),Zero))$
&
$(\forall\, X.\ equal(Divide(Zero::'a,X),Zero))$ &
$(\forall\, X\, Y.\ equal(Divide(X::'a,Y),Zero)\ \&\ equal(Divide(Y::'a,X),Zero) \dashrightarrow equal(X::'a,Y))$
&
$(\forall\, X.\ equal(Divide(X::'a,identity),Zero))$ &
$(\forall\, A\ B\ C.\ equal(A::'a,B) \dashrightarrow equal(Divide(A::'a,C),Divide(B::'a,C)))$ &
$(\forall\, D\ F'\ E.\ equal(D::'a,E) \dashrightarrow equal(Divide(F'::'a,D),Divide(F'::'a,E)))$ &
$(\forall\, Y\ X\ Z.\ equal(Divide(X::'a,Y),Zero)\ \&\ equal(Divide(Y::'a,Z),Zero) \dashrightarrow$
$equal(Divide(X::'a,Z),Zero))$ &
$(\forall\, X\, Z\, Y.\ equal(Divide(Divide(X::'a,Y),Z),Zero) \dashrightarrow equal(Divide(Divide(X::'a,Z),Y),Zero))$
&
$(\forall\, Y\, Z\, X.\ equal(Divide(X::'a,Y),Zero) \dashrightarrow equal(Divide(Divide(Z::'a,Y),Divide(Z::'a,X)),Zero))$
&
$(\sim equal(Divide(identity::'a,a),Divide(identity::'a,Divide(identity::'a,Divide(identity::'a,a)))))$
&
$(equal(Divide(identity::'a,a),b))$ &
$(equal(Divide(identity::'a,b),c))$ &
$(equal(Divide(identity::'a,c),d))$ &
$(\sim equal(b::'a,d)) \dashrightarrow False$
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 &
$(\sim mless\text{-}equal(a::'a,a)) \dashrightarrow False$
oops

lemma *LCL010-1*:
$(\forall\, X\, Y.\ is\text{-}a\text{-}theorem(equivalent(X::'a,Y))\ \&\ is\text{-}a\text{-}theorem(X) \dashrightarrow is\text{-}a\text{-}theorem(Y))$
&
$(\forall\, X\, Z\, Y.\ is\text{-}a\text{-}theorem(equivalent(equivalent(X::'a,Y),equivalent(equivalent(X::'a,Z),equivalent(Z::'a,Y)))))$
&
$(\sim is\text{-}a\text{-}theorem(equivalent(equivalent(a::'a,b),equivalent(equivalent(c::'a,b),equivalent(a::'a,c)))))$
$\dashrightarrow False$
by *meson*

lemma *LCL077-2*:
$(\forall\, X\ Y.\ is\text{-}a\text{-}theorem(implies(X,Y))\ \&\ is\text{-}a\text{-}theorem(X) \dashrightarrow is\text{-}a\text{-}theorem(Y))$

&
(∀ Y X. is-a-theorem(implies(X,implies(Y,X)))) &
(∀ Y X Z. is-a-theorem(implies(implies(X,implies(Y,Z)),implies(implies(X,Y),implies(X,Z)))))
&
(∀ Y X. is-a-theorem(implies(implies(not(X),not(Y)),implies(Y,X)))) &
(∀ X2 X1 X3. is-a-theorem(implies(X1,X2)) & is-a-theorem(implies(X2,X3))
--> is-a-theorem(implies(X1,X3))) &
(~is-a-theorem(implies(not(not(a)),a))) --> False
by *meson*

lemma *LCL082-1*:
(∀ X Y. is-a-theorem(implies(X::'a,Y)) & is-a-theorem(X) --> is-a-theorem(Y))
&
(∀ Y Z U X. is-a-theorem(implies(implies(implies(X::'a,Y),Z),implies(implies(Z::'a,X),implies(U::'a,X)))))
&
(~is-a-theorem(implies(a::'a,implies(b::'a,a)))) --> False
by *meson*

lemma *LCL111-1*:
(∀ X Y. is-a-theorem(implies(X,Y)) & is-a-theorem(X) --> is-a-theorem(Y))
&
(∀ Y X. is-a-theorem(implies(X,implies(Y,X)))) &
(∀ Y X Z. is-a-theorem(implies(implies(X,Y),implies(implies(Y,Z),implies(X,Z)))))
&
(∀ Y X. is-a-theorem(implies(implies(implies(X,Y),Y),implies(implies(Y,X),X))))
&
(∀ Y X. is-a-theorem(implies(implies(not(X),not(Y)),implies(Y,X)))) &
(~is-a-theorem(implies(implies(a,b),implies(implies(c,a),implies(c,b))))) --> False
by *meson*

lemma *LCL143-1*:
(∀ X. equal(X,X)) &
(∀ Y X. equal(X,Y) --> equal(Y,X)) &
(∀ Y X Z. equal(X,Y) & equal(Y,Z) --> equal(X,Z)) &
(∀ X. equal(implies(true,X),X)) &
(∀ Y X Z. equal(implies(implies(X,Y),implies(implies(Y,Z),implies(X,Z))),true))
&
(∀ Y X. equal(implies(implies(X,Y),Y),implies(implies(Y,X),X))) &
(∀ Y X. equal(implies(implies(not(X),not(Y)),implies(Y,X)),true)) &
(∀ A B C. equal(A,B) --> equal(implies(A,C),implies(B,C))) &
(∀ D F' E. equal(D,E) --> equal(implies(F',D),implies(F',E))) &
(∀ G H. equal(G,H) --> equal(not(G),not(H))) &
(∀ X Y. equal(big-V(X,Y),implies(implies(X,Y),Y))) &
(∀ X Y. equal(big-hat(X,Y),not(big-V(not(X),not(Y))))) &
(∀ X Y. ordered(X,Y) --> equal(implies(X,Y),true)) &
(∀ X Y. equal(implies(X,Y),true) --> ordered(X,Y)) &

(∀ A B C. *equal*(A,B) --> *equal*(*big-V*(A,C),*big-V*(B,C))) &
(∀ D F′ E. *equal*(D,E) --> *equal*(*big-V*(F′,D),*big-V*(F′,E))) &
(∀ G H I′. *equal*(G,H) --> *equal*(*big-hat*(G,I′),*big-hat*(H,I′))) &
(∀ J L K′. *equal*(J,K′) --> *equal*(*big-hat*(L,J),*big-hat*(L,K′))) &
(∀ M N O′. *equal*(M,N) & *ordered*(M,O′) --> *ordered*(N,O′)) &
(∀ P R Q. *equal*(P,Q) & *ordered*(R,P) --> *ordered*(R,Q)) &
(*ordered*(x,y)) &
(~*ordered*(*implies*(z,x),*implies*(z,y))) --> *False*
by *meson*

lemma *LCL182-1*:
(∀ A. *axiom*(*or*(*not*(*or*(A,A)),A))) &
(∀ B A. *axiom*(*or*(*not*(A),*or*(B,A)))) &
(∀ B A. *axiom*(*or*(*not*(*or*(A,B)),*or*(B,A)))) &
(∀ B A C. *axiom*(*or*(*not*(*or*(A,*or*(B,C))),*or*(B,*or*(A,C))))) &
(∀ A C B. *axiom*(*or*(*not*(*or*(*not*(A),B)),*or*(*not*(*or*(C,A)),*or*(C,B))))) &
(∀ X. *axiom*(X) --> *theorem*(X)) &
(∀ X Y. *axiom*(*or*(*not*(Y),X)) & *theorem*(Y) --> *theorem*(X)) &
(∀ X Y Z. *axiom*(*or*(*not*(X),Y)) & *theorem*(*or*(*not*(Y),Z)) --> *theorem*(*or*(*not*(X),Z)))
&
(~*theorem*(*or*(*not*(*or*(*not*(p),q)),*or*(*not*(*not*(q)),*not*(p))))) --> *False*
by *meson*

lemma *LCL200-1*:
(∀ A. *axiom*(*or*(*not*(*or*(A,A)),A))) &
(∀ B A. *axiom*(*or*(*not*(A),*or*(B,A)))) &
(∀ B A. *axiom*(*or*(*not*(*or*(A,B)),*or*(B,A)))) &
(∀ B A C. *axiom*(*or*(*not*(*or*(A,*or*(B,C))),*or*(B,*or*(A,C))))) &
(∀ A C B. *axiom*(*or*(*not*(*or*(*not*(A),B)),*or*(*not*(*or*(C,A)),*or*(C,B))))) &
(∀ X. *axiom*(X) --> *theorem*(X)) &
(∀ X Y. *axiom*(*or*(*not*(Y),X)) & *theorem*(Y) --> *theorem*(X)) &
(∀ X Y Z. *axiom*(*or*(*not*(X),Y)) & *theorem*(*or*(*not*(Y),Z)) --> *theorem*(*or*(*not*(X),Z)))
&
(~*theorem*(*or*(*not*(*not*(*or*(p,q))),*not*(q)))) --> *False*
by *meson*

lemma *LCL215-1*:
(∀ A. *axiom*(*or*(*not*(*or*(A,A)),A))) &
(∀ B A. *axiom*(*or*(*not*(A),*or*(B,A)))) &
(∀ B A. *axiom*(*or*(*not*(*or*(A,B)),*or*(B,A)))) &
(∀ B A C. *axiom*(*or*(*not*(*or*(A,*or*(B,C))),*or*(B,*or*(A,C))))) &
(∀ A C B. *axiom*(*or*(*not*(*or*(*not*(A),B)),*or*(*not*(*or*(C,A)),*or*(C,B))))) &
(∀ X. *axiom*(X) --> *theorem*(X)) &
(∀ X Y. *axiom*(*or*(*not*(Y),X)) & *theorem*(Y) --> *theorem*(X)) &
(∀ X Y Z. *axiom*(*or*(*not*(X),Y)) & *theorem*(*or*(*not*(Y),Z)) --> *theorem*(*or*(*not*(X),Z)))
&

$(\sim theorem(or(not(or(not(p),q)),or(not(or(p,q)),q)))) \; --> False$
by *meson*

lemma *LCL230-2*:
$(q \; --> p \mid r)$ &
$(\sim p)$ &
(q) &
$(\sim r) \; --> False$
by *meson*

lemma *LDA003-1*:
EQU001-0-ax equal &
$(\forall\, Y\ X\ Z.\ equal(f(X::'a,f(Y::'a,Z)),f(f(X::'a,Y),f(X::'a,Z))))$ &
$(\forall\, X\ Y.\ left(X::'a,f(X::'a,Y)))$ &
$(\forall\, Y\ X\ Z.\ left(X::'a,Y)$ & $left(Y::'a,Z) \; --> left(X::'a,Z))$ &
$(equal(num2::'a,f(num1::'a,num1)))$ &
$(equal(num3::'a,f(num2::'a,num1)))$ &
$(equal(u::'a,f(num2::'a,num2)))$ &
$(\forall\, A\ B\ C.\ equal(A::'a,B) \; --> equal(f(A::'a,C),f(B::'a,C)))$ &
$(\forall\, D\ F'\ E.\ equal(D::'a,E) \; --> equal(f(F'::'a,D),f(F'::'a,E)))$ &
$(\forall\, G\ H\ I'.\ equal(G::'a,H)$ & $left(G::'a,I') \; --> left(H::'a,I'))$ &
$(\forall\, J\ L\ K'.\ equal(J::'a,K')$ & $left(L::'a,J) \; --> left(L::'a,K'))$ &
$(\sim left(num3::'a,u)) \; --> False$
oops

lemma *MSC002-1*:
$(at(something::'a,here,now))$ &
$(\forall\, Place\ Situation.\ hand\text{-}at(Place::'a,Situation) \; --> hand\text{-}at(Place::'a,let\text{-}go(Situation)))$
&
$(\forall\, Place\ Another\text{-}place\ Situation.\ hand\text{-}at(Place::'a,Situation) \; --> hand\text{-}at(Another\text{-}place::'a,go(Another\text{-}pl$
&
$(\forall\, Thing\ Situation.\ \sim held(Thing::'a,let\text{-}go(Situation)))$ &
$(\forall\, Situation\ Thing.\ at(Thing::'a,here,Situation) \; --> red(Thing))$ &
$(\forall\, Thing\ Place\ Situation.\ at(Thing::'a,Place,Situation) \; --> at(Thing::'a,Place,let\text{-}go(Situation)))$
&
$(\forall\, Thing\ Place\ Situation.\ at(Thing::'a,Place,Situation) \; --> at(Thing::'a,Place,pick\text{-}up(Situation)))$
&
$(\forall\, Thing\ Place\ Situation.\ at(Thing::'a,Place,Situation) \; --> grabbed(Thing::'a,pick\text{-}up(go(Place::'a,let\text{-}go(Si$
&
$(\forall\, Thing\ Situation.\ red(Thing)$ & $put(Thing::'a,there,Situation) \; --> answer(Situation))$
&
$(\forall\, Place\ Thing\ Another\text{-}place\ Situation.\ at(Thing::'a,Place,Situation)$ & $grabbed(Thing::'a,Situation)$
$--> put(Thing::'a,Another\text{-}place,go(Another\text{-}place::'a,Situation)))$ &
$(\forall\, Thing\ Place\ Another\text{-}place\ Situation.\ at(Thing::'a,Place,Situation) \; --> held(Thing::'a,Situation)$
$\mid at(Thing::'a,Place,go(Another\text{-}place::'a,Situation)))$ &

(∀ *One-place Thing Place Situation. hand-at*(*One-place*::′*a*,*Situation*) & *held*(*Thing*::′*a*,*Situation*)
−−> *at*(*Thing*::′*a*,*Place*,*go*(*Place*::′*a*,*Situation*))) &
(∀ *Place Thing Situation. hand-at*(*Place*::′*a*,*Situation*) & *at*(*Thing*::′*a*,*Place*,*Situation*)
−−> *held*(*Thing*::′*a*,*pick-up*(*Situation*))) &
(∀ *Situation.* ~*answer*(*Situation*)) −−> *False*
by *meson*

lemma *MSC003-1*:
(∀ *Number-of-small-parts Small-part Big-part Number-of-mid-parts Mid-part. has-parts*(*Big-part*::′*a*,*Number-o*
−−> *in*′(*object-in*(*Big-part*::′*a*,*Mid-part*,*Small-part*,*Number-of-mid-parts*,*Number-of-small-parts*),*Mid-part*)
| *has-parts*(*Big-part*::′*a*,*mtimes*(*Number-of-mid-parts*::′*a*,*Number-of-small-parts*),*Small-part*))
&
(∀ *Big-part Mid-part Number-of-mid-parts Number-of-small-parts Small-part. has-parts*(*Big-part*::′*a*,*Number-o*
& *has-parts*(*object-in*(*Big-part*::′*a*,*Mid-part*,*Small-part*,*Number-of-mid-parts*,*Number-of-small-parts*),*Number-o*
−−> *has-parts*(*Big-part*::′*a*,*mtimes*(*Number-of-mid-parts*::′*a*,*Number-of-small-parts*),*Small-part*))
&
(*in*′(*john*::′*a*,*boy*)) &
(∀ *X*. *in*′(*X*::′*a*,*boy*) −−> *in*′(*X*::′*a*,*human*)) &
(∀ *X*. *in*′(*X*::′*a*,*hand*) −−> *has-parts*(*X*::′*a*,*num5*,*fingers*)) &
(∀ *X*. *in*′(*X*::′*a*,*human*) −−> *has-parts*(*X*::′*a*,*num2*,*arm*)) &
(∀ *X*. *in*′(*X*::′*a*,*arm*) −−> *has-parts*(*X*::′*a*,*num1*,*hand*)) &
(~*has-parts*(*john*::′*a*,*mtimes*(*num2*::′*a*,*num1*),*hand*)) −−> *False*
by *meson*

lemma *MSC004-1*:
(∀ *Number-of-small-parts Small-part Big-part Number-of-mid-parts Mid-part. has-parts*(*Big-part*::′*a*,*Number-o*
−−> *in*′(*object-in*(*Big-part*::′*a*,*Mid-part*,*Small-part*,*Number-of-mid-parts*,*Number-of-small-parts*),*Mid-part*)
| *has-parts*(*Big-part*::′*a*,*mtimes*(*Number-of-mid-parts*::′*a*,*Number-of-small-parts*),*Small-part*))
&
(∀ *Big-part Mid-part Number-of-mid-parts Number-of-small-parts Small-part. has-parts*(*Big-part*::′*a*,*Number-o*
& *has-parts*(*object-in*(*Big-part*::′*a*,*Mid-part*,*Small-part*,*Number-of-mid-parts*,*Number-of-small-parts*),*Number-o*
−−> *has-parts*(*Big-part*::′*a*,*mtimes*(*Number-of-mid-parts*::′*a*,*Number-of-small-parts*),*Small-part*))
&
(*in*′(*john*::′*a*,*boy*)) &
(∀ *X*. *in*′(*X*::′*a*,*boy*) −−> *in*′(*X*::′*a*,*human*)) &
(∀ *X*. *in*′(*X*::′*a*,*hand*) −−> *has-parts*(*X*::′*a*,*num5*,*fingers*)) &
(∀ *X*. *in*′(*X*::′*a*,*human*) −−> *has-parts*(*X*::′*a*,*num2*,*arm*)) &
(∀ *X*. *in*′(*X*::′*a*,*arm*) −−> *has-parts*(*X*::′*a*,*num1*,*hand*)) &
(~*has-parts*(*john*::′*a*,*mtimes*(*mtimes*(*num2*::′*a*,*num1*),*num5*),*fingers*)) −−> *False*
by *meson*

lemma *MSC005-1*:
(*value*(*truth*::′*a*,*truth*)) &
(*value*(*falsity*::′*a*,*falsity*)) &
(∀ *X Y*. *value*(*X*::′*a*,*truth*) & *value*(*Y*::′*a*,*truth*) −−> *value*(*xor*(*X*::′*a*,*Y*),*falsity*))
&

$(\forall X\ Y.\ value(X::'a,truth)\ \&\ value(Y::'a,falsity) \dashrightarrow value(xor(X::'a,Y),truth))$
&
$(\forall X\ Y.\ value(X::'a,falsity)\ \&\ value(Y::'a,truth) \dashrightarrow value(xor(X::'a,Y),truth))$
&
$(\forall X\ Y.\ value(X::'a,falsity)\ \&\ value(Y::'a,falsity) \dashrightarrow value(xor(X::'a,Y),falsity))$
&
$(\forall\ Value.\ {\sim}value(xor(xor(xor(xor(truth::'a,falsity),falsity),truth),falsity),Value))$
$\dashrightarrow False$
by *meson*

lemma *MSC006-1*:
$(\forall Y\ X\ Z.\ p(X::'a,Y)\ \&\ p(Y::'a,Z) \dashrightarrow p(X::'a,Z))\ \&$
$(\forall Y\ X\ Z.\ q(X::'a,Y)\ \&\ q(Y::'a,Z) \dashrightarrow q(X::'a,Z))\ \&$
$(\forall Y\ X.\ q(X::'a,Y) \dashrightarrow q(Y::'a,X))\ \&$
$(\forall X\ Y.\ p(X::'a,Y)\ |\ q(X::'a,Y))\ \&$
$({\sim}p(a::'a,b))\ \&$
$({\sim}q(c::'a,d)) \dashrightarrow False$
by *meson*

lemma *NUM001-1*:
$(\forall A.\ equal(A::'a,A))\ \&$
$(\forall B\ A\ C.\ equal(A::'a,B)\ \&\ equal(B::'a,C) \dashrightarrow equal(A::'a,C))\ \&$
$(\forall B\ A.\ equal(add(A::'a,B),add(B::'a,A)))\ \&$
$(\forall A\ B\ C.\ equal(add(A::'a,add(B::'a,C)),add(add(A::'a,B),C)))\ \&$
$(\forall B\ A.\ equal(subtract(add(A::'a,B),B),A))\ \&$
$(\forall A\ B.\ equal(A::'a,subtract(add(A::'a,B),B)))\ \&$
$(\forall A\ C\ B.\ equal(add(subtract(A::'a,B),C),subtract(add(A::'a,C),B)))\ \&$
$(\forall A\ C\ B.\ equal(subtract(add(A::'a,B),C),add(subtract(A::'a,C),B)))\ \&$
$(\forall A\ C\ B\ D.\ equal(A::'a,B)\ \&\ equal(C::'a,add(A::'a,D)) \dashrightarrow equal(C::'a,add(B::'a,D)))$
&
$(\forall A\ C\ D\ B.\ equal(A::'a,B)\ \&\ equal(C::'a,add(D::'a,A)) \dashrightarrow equal(C::'a,add(D::'a,B)))$
&
$(\forall A\ C\ B\ D.\ equal(A::'a,B)\ \&\ equal(C::'a,subtract(A::'a,D)) \dashrightarrow equal(C::'a,subtract(B::'a,D)))$
&
$(\forall A\ C\ D\ B.\ equal(A::'a,B)\ \&\ equal(C::'a,subtract(D::'a,A)) \dashrightarrow equal(C::'a,subtract(D::'a,B)))$
&
$({\sim}equal(add(add(a::'a,b),c),add(a::'a,add(b::'a,c)))) \dashrightarrow False$
by *meson*

abbreviation *NUM001-0-ax multiply successor num0 add equal* $\equiv$
$(\forall A.\ equal(add(A::'a,num0),A))\ \&$
$(\forall A\ B.\ equal(add(A::'a,successor(B)),successor(add(A::'a,B))))\ \&$
$(\forall A.\ equal(multiply(A::'a,num0),num0))\ \&$
$(\forall B\ A.\ equal(multiply(A::'a,successor(B)),add(multiply(A::'a,B),A)))\ \&$
$(\forall A\ B.\ equal(successor(A),successor(B)) \dashrightarrow equal(A::'a,B))\ \&$
$(\forall A\ B.\ equal(A::'a,B) \dashrightarrow equal(successor(A),successor(B)))$

abbreviation *NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless*
≡
(∀ A C B. mless(A::'a,B) & mless(C::'a,A) --> mless(C::'a,B)) &
(∀ A B C. equal(add(successor(A),B),C) --> mless(B::'a,C)) &
(∀ A B. mless(A::'a,B) --> equal(add(successor(predecessor-of-1st-minus-2nd(B::'a,A)),A),B))

abbreviation *NUM001-2-ax equal mless divides* ≡
(∀ A B. divides(A::'a,B) --> mless(A::'a,B) | equal(A::'a,B)) &
(∀ A B. mless(A::'a,B) --> divides(A::'a,B)) &
(∀ A B. equal(A::'a,B) --> 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 &
(mless(b::'a,c)) &
(~mless(b::'a,a)) &
(divides(c::'a,a)) &
(∀ A. ~equal(successor(A),num0)) --> 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 &
(∀ B A. equal(add(A::'a,B),add(B::'a,A))) &
(∀ B A C. equal(add(A::'a,B),add(C::'a,B)) --> equal(A::'a,C)) &
(mless(a::'a,a)) &
(∀ A. ~equal(successor(A),num0)) --> 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* ≡
(∀ X U Y. subclass(X::'a,Y) & member(U::'a,X) --> member(U::'a,Y)) &
(∀ X Y. member(not-subclass-element(X::'a,Y),X) | subclass(X::'a,Y)) &
(∀ X Y. member(not-subclass-element(X::'a,Y),Y) --> subclass(X::'a,Y)) &
(∀ X. subclass(X::'a,universal-class)) &
(∀ X Y. equal(X::'a,Y) --> subclass(X::'a,Y)) &
(∀ Y X. equal(X::'a,Y) --> subclass(Y::'a,X)) &

(∀ X Y. subclass(X::'a,Y) & subclass(Y::'a,X) --> equal(X::'a,Y)) &
(∀ X U Y. member(U::'a,unordered-pair(X::'a,Y)) --> equal(U::'a,X) | equal(U::'a,Y))
&
(∀ X Y. member(X::'a,universal-class) --> member(X::'a,unordered-pair(X::'a,Y)))
&
(∀ X Y. member(Y::'a,universal-class) --> member(Y::'a,unordered-pair(X::'a,Y)))
&
(∀ X Y. member(unordered-pair(X::'a,Y),universal-class)) &
(∀ X. equal(unordered-pair(X::'a,X),singleton(X))) &
(∀ X Y. equal(unordered-pair(singleton(X),unordered-pair(X::'a,singleton(Y))),ordered-pair(X::'a,Y)))
&
(∀ V Y U X. member(ordered-pair(U::'a,V),cross-product(X::'a,Y)) --> member(U::'a,X)) &
(∀ U X V Y. member(ordered-pair(U::'a,V),cross-product(X::'a,Y)) --> member(V::'a,Y)) &
(∀ U V X Y. member(U::'a,X) & member(V::'a,Y) --> member(ordered-pair(U::'a,V),cross-product(X::'a
&
(∀ X Y Z. member(Z::'a,cross-product(X::'a,Y)) --> equal(ordered-pair(first(Z),second(Z)),Z))
&
(subclass(element-relation::'a,cross-product(universal-class::'a,universal-class)))
&
(∀ X Y. member(ordered-pair(X::'a,Y),element-relation) --> member(X::'a,Y))
&
(∀ X Y. member(ordered-pair(X::'a,Y),cross-product(universal-class::'a,universal-class))
& member(X::'a,Y) --> member(ordered-pair(X::'a,Y),element-relation)) &
(∀ Y Z X. member(Z::'a,intersection(X::'a,Y)) --> member(Z::'a,X)) &
(∀ X Z Y. member(Z::'a,intersection(X::'a,Y)) --> member(Z::'a,Y)) &
(∀ Z X Y. member(Z::'a,X) & member(Z::'a,Y) --> member(Z::'a,intersection(X::'a,Y)))
&
(∀ Z X. ~(member(Z::'a,complement(X)) & member(Z::'a,X))) &
(∀ Z X. member(Z::'a,universal-class) --> member(Z::'a,complement(X)) | member(Z::'a,X)) &
(∀ X Y. equal(complement(intersection(complement(X),complement(Y))),union(X::'a,Y)))
&
(∀ X Y. equal(intersection(complement(intersection(X::'a,Y)),complement(intersection(complement(X),com
&
(∀ Xr X Y. equal(intersection(Xr::'a,cross-product(X::'a,Y)),restrct(Xr::'a,X,Y)))
&
(∀ Xr X Y. equal(intersection(cross-product(X::'a,Y),Xr),restrct(Xr::'a,X,Y)))
&
(∀ Z X. ~(equal(restrct(X::'a,singleton(Z),universal-class),null-class) & member(Z::'a,domain-of(X)))) &
(∀ Z X. member(Z::'a,universal-class) --> equal(restrct(X::'a,singleton(Z),universal-class),null-class)
| member(Z::'a,domain-of(X))) &
(∀ X. subclass(rot(X),cross-product(cross-product(universal-class::'a,universal-class),universal-class)))
&
(∀ V W U X. member(ordered-pair(ordered-pair(U::'a,V),W),rot(X)) --> member(ordered-pair(ordered-pair(V::'a,W),U),X)) &
(∀ U V W X. member(ordered-pair(ordered-pair(V::'a,W),U),X) & member(ordered-pair(ordered-pair(U::'a

--> member(ordered-pair(ordered-pair(U::'a,V),W),rot(X))) &
(∀ X. subclass(flip(X),cross-product(cross-product(universal-class::'a,universal-class),universal-class)))
&
(∀ V U W X. member(ordered-pair(ordered-pair(U::'a,V),W),flip(X)) --> member(ordered-pair(ordered-pair(V::'a,U),W),X)) &
(∀ U V W X. member(ordered-pair(ordered-pair(V::'a,U),W),X) & member(ordered-pair(ordered-pair(U::'a
--> member(ordered-pair(ordered-pair(U::'a,V),W),flip(X))) &
(∀ Y. equal(domain-of(flip(cross-product(Y::'a,universal-class))),INVERSE(Y)))
&
(∀ Z. equal(domain-of(INVERSE(Z)),range-of(Z))) &
(∀ Z X Y. equal(first(not-subclass-element(restrct(Z::'a,X,singleton(Y)),null-class)),domain(Z::'a,X,Y)))
&
(∀ Z X Y. equal(second(not-subclass-element(restrct(Z::'a,singleton(X),Y),null-class)),rng(Z::'a,X,Y)))
&
(∀ Xr X. equal(range-of(restrct(Xr::'a,X,universal-class)),image'(Xr::'a,X))) &
(∀ X. equal(union(X::'a,singleton(X)),successor(X))) &
(subclass(successor-relation::'a,cross-product(universal-class::'a,universal-class)))
&
(∀ X Y. member(ordered-pair(X::'a,Y),successor-relation) --> equal(successor(X),Y))
&
(∀ X Y. equal(successor(X),Y) & member(ordered-pair(X::'a,Y),cross-product(universal-class::'a,universal-c
--> member(ordered-pair(X::'a,Y),successor-relation)) &
(∀ X. inductive(X) --> member(null-class::'a,X)) &
(∀ X. inductive(X) --> subclass(image'(successor-relation::'a,X),X)) &
(∀ X. member(null-class::'a,X) & subclass(image'(successor-relation::'a,X),X)
--> inductive(X)) &
(inductive(omega)) &
(∀ Y. inductive(Y) --> subclass(omega::'a,Y)) &
(member(omega::'a,universal-class)) &
(∀ X. equal(domain-of(restrct(element-relation::'a,universal-class,X)),sum-class(X)))
&
(∀ X. member(X::'a,universal-class) --> member(sum-class(X),universal-class))
&
(∀ X. equal(complement(image'(element-relation::'a,complement(X))),powerClass(X)))
&
(∀ U. member(U::'a,universal-class) --> member(powerClass(U),universal-class))
&
(∀ Yr Xr. subclass(compos(Yr::'a,Xr),cross-product(universal-class::'a,universal-class)))
&
(∀ Z Yr Xr Y. member(ordered-pair(Y::'a,Z),compos(Yr::'a,Xr)) --> member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y))))) &
(∀ Y Z Yr Xr. member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y)))) & member(ordered-pair(Y::'a,Z),cross-product(universal-class::'a,universal-class)) --> member(ordered-pair(Y::'a,Z),compos(Yr::'a,Xr))) &
(∀ X. single-valued-class(X) --> subclass(compos(X::'a,INVERSE(X)),identity-relation))
&
(∀ X. subclass(compos(X::'a,INVERSE(X)),identity-relation) --> single-valued-class(X))
&
(∀ Xf. function(Xf) --> subclass(Xf::'a,cross-product(universal-class::'a,universal-class)))

&
$(\forall\, Xf.\ function(Xf) \dashrightarrow subclass(compos(Xf::'a,INVERSE(Xf)),identity\text{-}relation))$
&
$(\forall\, Xf.\ subclass(Xf::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))\ \&\ subclass(compos(Xf::'a,INVERSE(Xf)),identity\text{-}relation) \dashrightarrow function(Xf))$ &
$(\forall\, Xf\, X.\ function(Xf)\ \&\ member(X::'a,universal\text{-}class) \dashrightarrow member(image'(Xf::'a,X),universal\text{-}class))$
&
$(\forall\, X.\ equal(X::'a,null\text{-}class) \mid member(regular(X),X))$ &
$(\forall\, X.\ equal(X::'a,null\text{-}class) \mid equal(intersection(X::'a,regular(X)),null\text{-}class))$ &
$(\forall\, Xf\ Y.\ equal(sum\text{-}class(image'(Xf::'a,singleton(Y))),apply(Xf::'a,Y)))$ &
$(function(choice))$ &
$(\forall\, Y.\ member(Y::'a,universal\text{-}class) \dashrightarrow equal(Y::'a,null\text{-}class) \mid member(apply(choice::'a,Y),Y))$
&
$(\forall\, Xf.\ one\text{-}to\text{-}one(Xf) \dashrightarrow function(Xf))$ &
$(\forall\, Xf.\ one\text{-}to\text{-}one(Xf) \dashrightarrow function(INVERSE(Xf)))$ &
$(\forall\, Xf.\ function(INVERSE(Xf))\ \&\ function(Xf) \dashrightarrow one\text{-}to\text{-}one(Xf))$ &
$(equal(intersection(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),intersection(cross\text{-}product(universal\text{-}clas$
&
$(equal(intersection(INVERSE(subset\text{-}relation),subset\text{-}relation),identity\text{-}relation))$
&
$(\forall\, Xr.\ equal(complement(domain\text{-}of(intersection(Xr::'a,identity\text{-}relation))),diagonalise(Xr)))$
&
$(\forall\, X.\ equal(intersection(domain\text{-}of(X),diagonalise(compos(INVERSE(element\text{-}relation),X))),cantor(X)))$
&
$(\forall\, Xf.\ operation(Xf) \dashrightarrow function(Xf))$ &
$(\forall\, Xf.\ operation(Xf) \dashrightarrow equal(cross\text{-}product(domain\text{-}of(domain\text{-}of(Xf)),domain\text{-}of(domain\text{-}of(Xf))),dom$
&
$(\forall\, Xf.\ operation(Xf) \dashrightarrow subclass(range\text{-}of(Xf),domain\text{-}of(domain\text{-}of(Xf))))$
&
$(\forall\, Xf.\ function(Xf)\ \&\ equal(cross\text{-}product(domain\text{-}of(domain\text{-}of(Xf)),domain\text{-}of(domain\text{-}of(Xf))),domain\text{-}$
$\&\ subclass(range\text{-}of(Xf),domain\text{-}of(domain\text{-}of(Xf))) \dashrightarrow operation(Xf))$ &
$(\forall\, Xf1\ Xf2\ Xh.\ compatible(Xh::'a,Xf1,Xf2) \dashrightarrow function(Xh))$ &
$(\forall\, Xf2\ Xf1\ Xh.\ compatible(Xh::'a,Xf1,Xf2) \dashrightarrow equal(domain\text{-}of(domain\text{-}of(Xf1)),domain\text{-}of(Xh)))$
&
$(\forall\, Xf1\ Xh\ Xf2.\ compatible(Xh::'a,Xf1,Xf2) \dashrightarrow subclass(range\text{-}of(Xh),domain\text{-}of(domain\text{-}of(Xf2))))$
&
$(\forall\, Xh\ Xh1\ Xf1\ Xf2.\ function(Xh)\ \&\ equal(domain\text{-}of(domain\text{-}of(Xf1)),domain\text{-}of(Xh))$
$\&\ subclass(range\text{-}of(Xh),domain\text{-}of(domain\text{-}of(Xf2))) \dashrightarrow compatible(Xh1::'a,Xf1,Xf2))$
&
$(\forall\, Xh\ Xf2\ Xf1.\ homomorphism(Xh::'a,Xf1,Xf2) \dashrightarrow operation(Xf1))$ &
$(\forall\, Xh\ Xf1\ Xf2.\ homomorphism(Xh::'a,Xf1,Xf2) \dashrightarrow operation(Xf2))$ &
$(\forall\, Xh\ Xf1\ Xf2.\ homomorphism(Xh::'a,Xf1,Xf2) \dashrightarrow compatible(Xh::'a,Xf1,Xf2))$
&
$(\forall\, Xf2\ Xh\ Xf1\ X\ Y.\ homomorphism(Xh::'a,Xf1,Xf2)\ \&\ member(ordered\text{-}pair(X::'a,Y),domain\text{-}of(Xf1))$
$\dashrightarrow equal(apply(Xf2::'a,ordered\text{-}pair(apply(Xh::'a,X),apply(Xh::'a,Y))),apply(Xh::'a,apply(Xf1::'a,ordered\text{-}$
&
$(\forall\, Xh\ Xf1\ Xf2.\ operation(Xf1)\ \&\ operation(Xf2)\ \&\ compatible(Xh::'a,Xf1,Xf2)$
$\dashrightarrow member(ordered\text{-}pair(not\text{-}homomorphism1(Xh::'a,Xf1,Xf2),not\text{-}homomorphism2(Xh::'a,Xf1,Xf2)),dom$
$\mid homomorphism(Xh::'a,Xf1,Xf2))$ &

$(\forall Xh\ Xf1\ Xf2.\ operation(Xf1)\ \&\ operation(Xf2)\ \&\ compatible(Xh{::}'a,Xf1,Xf2)$
$\&\ equal(apply(Xf2{::}'a,ordered\text{-}pair(apply(Xh{::}'a,not\text{-}homomorphism1(Xh{::}'a,Xf1,Xf2)),apply(Xh{::}'a,not\text{-}hom$
$\text{-->}\ homomorphism(Xh{::}'a,Xf1,Xf2))$

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 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* ≡

$(\forall D\ E\ F'.\ equal(D{::}'a,E)\ \text{-->}\ equal(apply(D{::}'a,F'),apply(E{::}'a,F')))\ \&$
$(\forall G\ I'\ H.\ equal(G{::}'a,H)\ \text{-->}\ equal(apply(I'{::}'a,G),apply(I'{::}'a,H)))\ \&$
$(\forall J\ K'.\ equal(J{::}'a,K')\ \text{-->}\ equal(cantor(J),cantor(K')))\ \&$
$(\forall L\ M.\ equal(L{::}'a,M)\ \text{-->}\ equal(complement(L),complement(M)))\ \&$
$(\forall N\ O'\ P.\ equal(N{::}'a,O')\ \text{-->}\ equal(compos(N{::}'a,P),compos(O'{::}'a,P)))\ \&$
$(\forall Q\ S'\ R.\ equal(Q{::}'a,R)\ \text{-->}\ equal(compos(S'{::}'a,Q),compos(S'{::}'a,R)))\ \&$
$(\forall T'\ U\ V.\ equal(T'{::}'a,U)\ \text{-->}\ equal(cross\text{-}product(T'{::}'a,V),cross\text{-}product(U{::}'a,V)))$
$\&$
$(\forall W\ Y\ X.\ equal(W{::}'a,X)\ \text{-->}\ equal(cross\text{-}product(Y{::}'a,W),cross\text{-}product(Y{::}'a,X)))$
$\&$
$(\forall Z\ A1.\ equal(Z{::}'a,A1)\ \text{-->}\ equal(diagonalise(Z),diagonalise(A1)))\ \&$
$(\forall B1\ C1\ D1.\ equal(B1{::}'a,C1)\ \text{-->}\ equal(difference(B1{::}'a,D1),difference(C1{::}'a,D1)))$
$\&$
$(\forall E1\ G1\ F1.\ equal(E1{::}'a,F1)\ \text{-->}\ equal(difference(G1{::}'a,E1),difference(G1{::}'a,F1)))$
$\&$
$(\forall H1\ I1\ J1\ K1.\ equal(H1{::}'a,I1)\ \text{-->}\ equal(domain(H1{::}'a,J1,K1),domain(I1{::}'a,J1,K1)))$
$\&$
$(\forall L1\ N1\ M1\ O1.\ equal(L1{::}'a,M1)\ \text{-->}\ equal(domain(N1{::}'a,L1,O1),domain(N1{::}'a,M1,O1)))$
$\&$
$(\forall P1\ R1\ S1\ Q1.\ equal(P1{::}'a,Q1)\ \text{-->}\ equal(domain(R1{::}'a,S1,P1),domain(R1{::}'a,S1,Q1)))$
$\&$
$(\forall T1\ U1.\ equal(T1{::}'a,U1)\ \text{-->}\ equal(domain\text{-}of(T1),domain\text{-}of(U1)))\ \&$
$(\forall V1\ W1.\ equal(V1{::}'a,W1)\ \text{-->}\ equal(first(V1),first(W1)))\ \&$
$(\forall X1\ Y1.\ equal(X1{::}'a,Y1)\ \text{-->}\ equal(flip(X1),flip(Y1)))\ \&$
$(\forall Z1\ A2\ B2.\ equal(Z1{::}'a,A2)\ \text{-->}\ equal(image'(Z1{::}'a,B2),image'(A2{::}'a,B2)))$
$\&$
$(\forall C2\ E2\ D2.\ equal(C2{::}'a,D2)\ \text{-->}\ equal(image'(E2{::}'a,C2),image'(E2{::}'a,D2)))$
$\&$
$(\forall F2\ G2\ H2.\ equal(F2{::}'a,G2)\ \text{-->}\ equal(intersection(F2{::}'a,H2),intersection(G2{::}'a,H2)))$
$\&$
$(\forall I2\ K2\ J2.\ equal(I2{::}'a,J2)\ \text{-->}\ equal(intersection(K2{::}'a,I2),intersection(K2{::}'a,J2)))$
$\&$
$(\forall L2\ M2.\ equal(L2{::}'a,M2)\ \text{-->}\ equal(INVERSE(L2),INVERSE(M2)))\ \&$
$(\forall N2\ O2\ P2\ Q2.\ equal(N2{::}'a,O2)\ \text{-->}\ equal(not\text{-}homomorphism1(N2{::}'a,P2,Q2),not\text{-}homomorphism1(O2$
$\&$
$(\forall R2\ T2\ S2\ U2.\ equal(R2{::}'a,S2)\ \text{-->}\ equal(not\text{-}homomorphism1(T2{::}'a,R2,U2),not\text{-}homomorphism1(T2$
$\&$
$(\forall V2\ X2\ Y2\ W2.\ equal(V2{::}'a,W2)\ \text{-->}\ equal(not\text{-}homomorphism1(X2{::}'a,Y2,V2),not\text{-}homomorphism1(X$

&
(∀ Z2 A3 B3 C3. equal(Z2::'a,A3) --> equal(not-homomorphism2(Z2::'a,B3,C3),not-homomorphism2(A3:
&
(∀ D3 F3 E3 G3. equal(D3::'a,E3) --> equal(not-homomorphism2(F3::'a,D3,G3),not-homomorphism2(F3
&
(∀ H3 J3 K3 I3. equal(H3::'a,I3) --> equal(not-homomorphism2(J3::'a,K3,H3),not-homomorphism2(J3::'
&
(∀ L3 M3 N3. equal(L3::'a,M3) --> equal(not-subclass-element(L3::'a,N3),not-subclass-element(M3::'a,N3
&
(∀ O3 Q3 P3. equal(O3::'a,P3) --> equal(not-subclass-element(Q3::'a,O3),not-subclass-element(Q3::'a,P3
&
(∀ R3 S3 T3. equal(R3::'a,S3) --> equal(ordered-pair(R3::'a,T3),ordered-pair(S3::'a,T3)))
&
(∀ U3 W3 V3. equal(U3::'a,V3) --> equal(ordered-pair(W3::'a,U3),ordered-pair(W3::'a,V3)))
&
(∀ X3 Y3. equal(X3::'a,Y3) --> equal(powerClass(X3),powerClass(Y3))) &
(∀ Z3 A4 B4 C4. equal(Z3::'a,A4) --> equal(rng(Z3::'a,B4,C4),rng(A4::'a,B4,C4)))
&
(∀ D4 F4 E4 G4. equal(D4::'a,E4) --> equal(rng(F4::'a,D4,G4),rng(F4::'a,E4,G4)))
&
(∀ H4 J4 K4 I4. equal(H4::'a,I4) --> equal(rng(J4::'a,K4,H4),rng(J4::'a,K4,I4)))
&
(∀ L4 M4. equal(L4::'a,M4) --> equal(range-of(L4),range-of(M4))) &
(∀ N4 O4. equal(N4::'a,O4) --> equal(regular(N4),regular(O4))) &
(∀ P4 Q4 R4 S4. equal(P4::'a,Q4) --> equal(restrct(P4::'a,R4,S4),restrct(Q4::'a,R4,S4)))
&
(∀ T4 V4 U4 W4. equal(T4::'a,U4) --> equal(restrct(V4::'a,T4,W4),restrct(V4::'a,U4,W4)))
&
(∀ X4 Z4 A5 Y4. equal(X4::'a,Y4) --> equal(restrct(Z4::'a,A5,X4),restrct(Z4::'a,A5,Y4)))
&
(∀ B5 C5. equal(B5::'a,C5) --> equal(rot(B5),rot(C5))) &
(∀ D5 E5. equal(D5::'a,E5) --> equal(second(D5),second(E5))) &
(∀ F5 G5. equal(F5::'a,G5) --> equal(singleton(F5),singleton(G5))) &
(∀ H5 I5. equal(H5::'a,I5) --> equal(successor(H5),successor(I5))) &
(∀ J5 K5. equal(J5::'a,K5) --> equal(sum-class(J5),sum-class(K5))) &
(∀ L5 M5 N5. equal(L5::'a,M5) --> equal(union(L5::'a,N5),union(M5::'a,N5)))
&
(∀ O5 Q5 P5. equal(O5::'a,P5) --> equal(union(Q5::'a,O5),union(Q5::'a,P5)))
&
(∀ R5 S5 T5. equal(R5::'a,S5) --> equal(unordered-pair(R5::'a,T5),unordered-pair(S5::'a,T5)))
&
(∀ U5 W5 V5. equal(U5::'a,V5) --> equal(unordered-pair(W5::'a,U5),unordered-pair(W5::'a,V5)))
&
(∀ X5 Y5 Z5 A6. equal(X5::'a,Y5) & compatible(X5::'a,Z5,A6) --> compatible(Y5::'a,Z5,A6)) &
(∀ B6 D6 C6 E6. equal(B6::'a,C6) & compatible(D6::'a,B6,E6) --> compatible(D6::'a,C6,E6)) &
(∀ F6 H6 I6 G6. equal(F6::'a,G6) & compatible(H6::'a,I6,F6) --> compatible(H6::'a,I6,G6)) &

(∀ J6 K6. equal(J6::'a,K6) & function(J6) --> function(K6)) &
(∀ L6 M6 N6 O6. equal(L6::'a,M6) & homomorphism(L6::'a,N6,O6) --> homomorphism(M6::'a,N6,O6)) &
(∀ P6 R6 Q6 S6. equal(P6::'a,Q6) & homomorphism(R6::'a,P6,S6) --> homomorphism(R6::'a,Q6,S6)) &
(∀ T6 V6 W6 U6. equal(T6::'a,U6) & homomorphism(V6::'a,W6,T6) --> homomorphism(V6::'a,W6,U6)) &
(∀ X6 Y6. equal(X6::'a,Y6) & inductive(X6) --> inductive(Y6)) &
(∀ Z6 A7 B7. equal(Z6::'a,A7) & member(Z6::'a,B7) --> member(A7::'a,B7))
&
(∀ C7 E7 D7. equal(C7::'a,D7) & member(E7::'a,C7) --> member(E7::'a,D7))
&
(∀ F7 G7. equal(F7::'a,G7) & one-to-one(F7) --> one-to-one(G7)) &
(∀ H7 I7. equal(H7::'a,I7) & operation(H7) --> operation(I7)) &
(∀ J7 K7. equal(J7::'a,K7) & single-valued-class(J7) --> single-valued-class(K7))
&
(∀ L7 M7 N7. equal(L7::'a,M7) & subclass(L7::'a,N7) --> subclass(M7::'a,N7))
&
(∀ O7 Q7 P7. equal(O7::'a,P7) & subclass(Q7::'a,O7) --> 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* ≡
(∀ X. subclass(compose-class(X),cross-product(universal-class::'a,universal-class)))
&
(∀ X Y Z. member(ordered-pair(Y::'a,Z),compose-class(X)) --> equal(compos(X::'a,Y),Z))
&
(∀ Y Z X. member(ordered-pair(Y::'a,Z),cross-product(universal-class::'a,universal-class))
& equal(compos(X::'a,Y),Z) --> member(ordered-pair(Y::'a,Z),compose-class(X)))
&
(subclass(composition-function::'a,cross-product(universal-class::'a,cross-product(universal-class::'a,universa
&
(∀ X Y Z. member(ordered-pair(X::'a,ordered-pair(Y::'a,Z)),composition-function)
--> equal(compos(X::'a,Y),Z)) &
(∀ X Y. member(ordered-pair(X::'a,Y),cross-product(universal-class::'a,universal-class))
--> member(ordered-pair(X::'a,ordered-pair(Y::'a,compos(X::'a,Y))),composition-function))
&
(subclass(domain-relation::'a,cross-product(universal-class::'a,universal-class))) &
(∀ X Y. member(ordered-pair(X::'a,Y),domain-relation) --> equal(domain-of(X),Y))
&
(∀ X. member(X::'a,universal-class) --> member(ordered-pair(X::'a,domain-of(X)),domain-relation))
&
(∀ X. equal(first(not-subclass-element(compos(X::'a,INVERSE(X)),identity-relation)),single-valued1(X)))
&
(∀ X. equal(second(not-subclass-element(compos(X::'a,INVERSE(X)),identity-relation)),single-valued2(X))

&
(∀ *X*. *equal*(*domain*(*X*::′*a*,*image*′(*INVERSE*(*X*),*singleton*(*single-valued1*(*X*))),*single-valued2*(*X*)),*single-valu*
&
(*equal*(*intersection*(*complement*(*compos*(*element-relation*::′*a*,*complement*(*identity-relation*))),*element-relation*
&
(*subclass*(*application-function*::′*a*,*cross-product*(*universal-class*::′*a*,*cross-product*(*universal-class*::′*a*,*universal-*
&
(∀ *Z Y X*. *member*(*ordered-pair*(*X*::′*a*,*ordered-pair*(*Y*::′*a*,*Z*)),*application-function*)
−−> *member*(*Y*::′*a*,*domain-of*(*X*))) &
(∀ *X Y Z*. *member*(*ordered-pair*(*X*::′*a*,*ordered-pair*(*Y*::′*a*,*Z*)),*application-function*)
−−> *equal*(*apply*(*X*::′*a*,*Y*),*Z*)) &
(∀ *Z X Y*. *member*(*ordered-pair*(*X*::′*a*,*ordered-pair*(*Y*::′*a*,*Z*)),*cross-product*(*universal-class*::′*a*,*cross-product*(
& *member*(*Y*::′*a*,*domain-of*(*X*)) −−> *member*(*ordered-pair*(*X*::′*a*,*ordered-pair*(*Y*::′*a*,*apply*(*X*::′*a*,*Y*))),*applic*
&
(∀ *X Y Xf*. *maps*(*Xf*::′*a*,*X*,*Y*) −−> *function*(*Xf*)) &
(∀ *Y Xf X*. *maps*(*Xf*::′*a*,*X*,*Y*) −−> *equal*(*domain-of*(*Xf*),*X*)) &
(∀ *X Xf Y*. *maps*(*Xf*::′*a*,*X*,*Y*) −−> *subclass*(*range-of*(*Xf*),*Y*)) &
(∀ *Xf Y*. *function*(*Xf*) & *subclass*(*range-of*(*Xf*),*Y*) −−> *maps*(*Xf*::′*a*,*domain-of*(*Xf*),*Y*))

abbreviation *SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal* ≡
(∀ *L M*. *equal*(*L*::′*a*,*M*) −−> *equal*(*compose-class*(*L*),*compose-class*(*M*))) &
(∀ *N2 O2*. *equal*(*N2*::′*a*,*O2*) −−> *equal*(*single-valued1*(*N2*),*single-valued1*(*O2*)))
&
(∀ *P2 Q2*. *equal*(*P2*::′*a*,*Q2*) −−> *equal*(*single-valued2*(*P2*),*single-valued2*(*Q2*)))
&
(∀ *R2 S2*. *equal*(*R2*::′*a*,*S2*) −−> *equal*(*single-valued3*(*R2*),*single-valued3*(*S2*)))
&
(∀ *X2 Y2 Z2 A3*. *equal*(*X2*::′*a*,*Y2*) & *maps*(*X2*::′*a*,*Z2*,*A3*) −−> *maps*(*Y2*::′*a*,*Z2*,*A3*))
&
(∀ *B3 D3 C3 E3*. *equal*(*B3*::′*a*,*C3*) & *maps*(*D3*::′*a*,*B3*,*E3*) −−> *maps*(*D3*::′*a*,*C3*,*E3*))
&
(∀ *F3 H3 I3 G3*. *equal*(*F3*::′*a*,*G3*) & *maps*(*H3*::′*a*,*I3*,*F3*) −−> *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 restrct subclass irreflexive symmetrization-of INVERSE union equal* ≡
(∀ *X*. *equal*(*union*(*X*::′*a*,*INVERSE*(*X*)),*symmetrization-of*(*X*))) &
(∀ *X Y*. *irreflexive*(*X*::′*a*,*Y*) −−> *subclass*(*restrct*(*X*::′*a*,*Y*,*Y*),*complement*(*identity-relation*)))
&
(∀ *X Y*. *subclass*(*restrct*(*X*::′*a*,*Y*,*Y*),*complement*(*identity-relation*)) −−> *irreflexive*(*X*::′*a*,*Y*)) &
(∀ *Y X*. *connected*(*X*::′*a*,*Y*) −−> *subclass*(*cross-product*(*Y*::′*a*,*Y*),*union*(*identity-relation*::′*a*,*symmetrization*

&
 (∀ X Y. subclass(cross-product(Y::'a,Y),union(identity-relation::'a,symmetrization-of(X)))
--> connected(X::'a,Y)) &
 (∀ Xr Y. transitive(Xr::'a,Y) --> subclass(compos(restrct(Xr::'a,Y,Y),restrct(Xr::'a,Y,Y)),restrct(Xr::'a
&
 (∀ Xr Y. subclass(compos(restrct(Xr::'a,Y,Y),restrct(Xr::'a,Y,Y)),restrct(Xr::'a,Y,Y))
--> transitive(Xr::'a,Y)) &
 (∀ Xr Y. asymmetric(Xr::'a,Y) --> equal(restrct(intersection(Xr::'a,INVERSE(Xr)),Y,Y),null-class))
&
 (∀ Xr Y. equal(restrct(intersection(Xr::'a,INVERSE(Xr)),Y,Y),null-class) -->
asymmetric(Xr::'a,Y)) &
 (∀ Xr Y Z. equal(segment(Xr::'a,Y,Z),domain-of(restrct(Xr::'a,Y,singleton(Z)))))
&
 (∀ X Y. well-ordering(X::'a,Y) --> connected(X::'a,Y)) &
 (∀ Y Xr U. well-ordering(Xr::'a,Y) & subclass(U::'a,Y) --> equal(U::'a,null-class)
| member(least(Xr::'a,U),U)) &
 (∀ Y V Xr U. well-ordering(Xr::'a,Y) & subclass(U::'a,Y) & member(V::'a,U)
--> member(least(Xr::'a,U),U)) &
 (∀ Y Xr U. well-ordering(Xr::'a,Y) & subclass(U::'a,Y) --> equal(segment(Xr::'a,U,least(Xr::'a,U)),null-
&
 (∀ Y V U Xr. ~(well-ordering(Xr::'a,Y) & subclass(U::'a,Y) & member(V::'a,U)
& member(ordered-pair(V::'a,least(Xr::'a,U)),Xr))) &
 (∀ Xr Y. connected(Xr::'a,Y) & equal(not-well-ordering(Xr::'a,Y),null-class)
--> well-ordering(Xr::'a,Y)) &
 (∀ Xr Y. connected(Xr::'a,Y) --> subclass(not-well-ordering(Xr::'a,Y),Y) |
well-ordering(Xr::'a,Y)) &
 (∀ V Xr Y. member(V::'a,not-well-ordering(Xr::'a,Y)) & equal(segment(Xr::'a,not-well-ordering(Xr::'a,Y),
& connected(Xr::'a,Y) --> well-ordering(Xr::'a,Y)) &
 (∀ Xr Y Z. section(Xr::'a,Y,Z) --> subclass(Y::'a,Z)) &
 (∀ Xr Z Y. section(Xr::'a,Y,Z) --> subclass(domain-of(restrct(Xr::'a,Z,Y)),Y))
&
 (∀ Xr Y Z. subclass(Y::'a,Z) & subclass(domain-of(restrct(Xr::'a,Z,Y)),Y) -->
section(Xr::'a,Y,Z)) &
 (∀ X. member(X::'a,ordinal-numbers) --> well-ordering(element-relation::'a,X))
&
 (∀ X. member(X::'a,ordinal-numbers) --> subclass(sum-class(X),X)) &
 (∀ X. well-ordering(element-relation::'a,X) & subclass(sum-class(X),X) & member(X::'a,universal-class) --> member(X::'a,ordinal-numbers)) &
 (∀ X. well-ordering(element-relation::'a,X) & subclass(sum-class(X),X) -->
member(X::'a,ordinal-numbers) | equal(X::'a,ordinal-numbers)) &
 (equal(union(singleton(null-class),image'(successor-relation::'a,ordinal-numbers)),kind-1-ordinals))
&
 (equal(intersection(complement(kind-1-ordinals),ordinal-numbers),limit-ordinals))
&
 (∀ X. subclass(rest-of(X),cross-product(universal-class::'a,universal-class))) &
 (∀ V U X. member(ordered-pair(U::'a,V),rest-of(X)) --> member(U::'a,domain-of(X)))
&
 (∀ X U V. member(ordered-pair(U::'a,V),rest-of(X)) --> equal(restrct(X::'a,U,universal-class),V))
&

(∀ U V X. member(U::'a,domain-of(X)) & equal(restrct(X::'a,U,universal-class),V)
--> member(ordered-pair(U::'a,V),rest-of(X))) &
(subclass(rest-relation::'a,cross-product(universal-class::'a,universal-class))) &
(∀ X Y. member(ordered-pair(X::'a,Y),rest-relation) --> equal(rest-of(X),Y))
&
(∀ X. member(X::'a,universal-class) --> member(ordered-pair(X::'a,rest-of(X)),rest-relation))
&
(∀ X Z. member(X::'a,recursion-equation-functions(Z)) --> function(Z)) &
(∀ Z X. member(X::'a,recursion-equation-functions(Z)) --> function(X)) &
(∀ Z X. member(X::'a,recursion-equation-functions(Z)) --> member(domain-of(X),ordinal-numbers))
&
(∀ Z X. member(X::'a,recursion-equation-functions(Z)) --> equal(compos(Z::'a,rest-of(X)),X))
&
(∀ X Z. function(Z) & function(X) & member(domain-of(X),ordinal-numbers) &
equal(compos(Z::'a,rest-of(X)),X) --> member(X::'a,recursion-equation-functions(Z)))
&
(subclass(union-of-range-map::'a,cross-product(universal-class::'a,universal-class)))
&
(∀ X Y. member(ordered-pair(X::'a,Y),union-of-range-map) --> equal(sum-class(range-of(X)),Y))
&
(∀ X Y. member(ordered-pair(X::'a,Y),cross-product(universal-class::'a,universal-class))
& equal(sum-class(range-of(X)),Y) --> member(ordered-pair(X::'a,Y),union-of-range-map))
&
(∀ X Y. equal(apply(recursion(X::'a,successor-relation,union-of-range-map),Y),ordinal-add(X::'a,Y)))
&
(∀ X Y. equal(recursion(null-class::'a,apply(add-relation::'a,X),union-of-range-map),ordinal-multiply(X::'a,
&
(∀ X. member(X::'a,omega) --> equal(integer-of(X),X)) &
(∀ X. member(X::'a,omega) | equal(integer-of(X),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* ≡
(∀ D E. equal(D::'a,E) --> equal(integer-of(D),integer-of(E))) &
(∀ F' G H. equal(F'::'a,G) --> equal(least(F'::'a,H),least(G::'a,H))) &
(∀ I' K' J. equal(I'::'a,J) --> equal(least(K'::'a,I'),least(K'::'a,J))) &
(∀ L M N. equal(L::'a,M) --> equal(not-well-ordering(L::'a,N),not-well-ordering(M::'a,N)))
&
(∀ O' Q P. equal(O'::'a,P) --> equal(not-well-ordering(Q::'a,O'),not-well-ordering(Q::'a,P)))
&
(∀ R S' T'. equal(R::'a,S') --> equal(ordinal-add(R::'a,T'),ordinal-add(S'::'a,T')))
&
(∀ U W V. equal(U::'a,V) --> equal(ordinal-add(W::'a,U),ordinal-add(W::'a,V)))
&
(∀ X Y Z. equal(X::'a,Y) --> equal(ordinal-multiply(X::'a,Z),ordinal-multiply(Y::'a,Z)))
&
(∀ A1 C1 B1. equal(A1::'a,B1) --> equal(ordinal-multiply(C1::'a,A1),ordinal-multiply(C1::'a,B1)))
&

$(\forall$ *F1 G1 H1 I1*. *equal*(*F1*::'*a*,*G1*) --> *equal*(*recursion*(*F1*::'*a*,*H1*,*I1*),*recursion*(*G1*::'*a*,*H1*,*I1*)))
&
(∀ *J1 L1 K1 M1*. *equal*(*J1*::'*a*,*K1*) --> *equal*(*recursion*(*L1*::'*a*,*J1*,*M1*),*recursion*(*L1*::'*a*,*K1*,*M1*)))
&
(∀ *N1 P1 Q1 O1*. *equal*(*N1*::'*a*,*O1*) --> *equal*(*recursion*(*P1*::'*a*,*Q1*,*N1*),*recursion*(*P1*::'*a*,*Q1*,*O1*)))
&
(∀ *R1 S1*. *equal*(*R1*::'*a*,*S1*) --> *equal*(*recursion-equation-functions*(*R1*),*recursion-equation-functions*(*S1*)))
&
(∀ *T1 U1*. *equal*(*T1*::'*a*,*U1*) --> *equal*(*rest-of*(*T1*),*rest-of*(*U1*))) &
(∀ *V1 W1 X1 Y1*. *equal*(*V1*::'*a*,*W1*) --> *equal*(*segment*(*V1*::'*a*,*X1*,*Y1*),*segment*(*W1*::'*a*,*X1*,*Y1*)))
&
(∀ *Z1 B2 A2 C2*. *equal*(*Z1*::'*a*,*A2*) --> *equal*(*segment*(*B2*::'*a*,*Z1*,*C2*),*segment*(*B2*::'*a*,*A2*,*C2*)))
&
(∀ *D2 F2 G2 E2*. *equal*(*D2*::'*a*,*E2*) --> *equal*(*segment*(*F2*::'*a*,*G2*,*D2*),*segment*(*F2*::'*a*,*G2*,*E2*)))
&
(∀ *H2 I2*. *equal*(*H2*::'*a*,*I2*) --> *equal*(*symmetrization-of*(*H2*),*symmetrization-of*(*I2*)))
&
(∀ *J2 K2 L2*. *equal*(*J2*::'*a*,*K2*) & *asymmetric*(*J2*::'*a*,*L2*) --> *asymmetric*(*K2*::'*a*,*L2*))
&
(∀ *M2 O2 N2*. *equal*(*M2*::'*a*,*N2*) & *asymmetric*(*O2*::'*a*,*M2*) --> *asymmetric*(*O2*::'*a*,*N2*))
&
(∀ *P2 Q2 R2*. *equal*(*P2*::'*a*,*Q2*) & *connected*(*P2*::'*a*,*R2*) --> *connected*(*Q2*::'*a*,*R2*))
&
(∀ *S2 U2 T2*. *equal*(*S2*::'*a*,*T2*) & *connected*(*U2*::'*a*,*S2*) --> *connected*(*U2*::'*a*,*T2*))
&
(∀ *V2 W2 X2*. *equal*(*V2*::'*a*,*W2*) & *irreflexive*(*V2*::'*a*,*X2*) --> *irreflexive*(*W2*::'*a*,*X2*))
&
(∀ *Y2 A3 Z2*. *equal*(*Y2*::'*a*,*Z2*) & *irreflexive*(*A3*::'*a*,*Y2*) --> *irreflexive*(*A3*::'*a*,*Z2*))
&
(∀ *B3 C3 D3 E3*. *equal*(*B3*::'*a*,*C3*) & *section*(*B3*::'*a*,*D3*,*E3*) --> *section*(*C3*::'*a*,*D3*,*E3*))
&
(∀ *F3 H3 G3 I3*. *equal*(*F3*::'*a*,*G3*) & *section*(*H3*::'*a*,*F3*,*I3*) --> *section*(*H3*::'*a*,*G3*,*I3*))
&
(∀ *J3 L3 M3 K3*. *equal*(*J3*::'*a*,*K3*) & *section*(*L3*::'*a*,*M3*,*J3*) --> *section*(*L3*::'*a*,*M3*,*K3*))
&
(∀ *N3 O3 P3*. *equal*(*N3*::'*a*,*O3*) & *transitive*(*N3*::'*a*,*P3*) --> *transitive*(*O3*::'*a*,*P3*))
&
(∀ *Q3 S3 R3*. *equal*(*Q3*::'*a*,*R3*) & *transitive*(*S3*::'*a*,*Q3*) --> *transitive*(*S3*::'*a*,*R3*))
&
(∀ *T3 U3 V3*. *equal*(*T3*::'*a*,*U3*) & *well-ordering*(*T3*::'*a*,*V3*) --> *well-ordering*(*U3*::'*a*,*V3*))
&
(∀ *W3 Y3 X3*. *equal*(*W3*::'*a*,*X3*) & *well-ordering*(*Y3*::'*a*,*W3*) --> *well-ordering*(*Y3*::'*a*,*X3*))

lemma *NUM180-1*:
EQU001-0-ax equal &
SET004-0-ax not-homomorphism2 not-homomorphism1
homomorphism compatible operation cantor diagonalise subset-relation
one-to-one choice apply regular function identity-relation

single-valued-class compos powerClass sum-class omega inductive successor-relation successor image′ rng domain range-of INVERSE flip rot domain-of null-class restrct difference union complement intersection element-relation second first cross-product ordered-pair singleton unordered-pair equal universal-class not-subclass-element member subclass &
SET004-0-eq subclass single-valued-class operation one-to-one member inductive homomorphism function compatible unordered-pair union sum-class successor singleton second rot restrct regular range-of rng powerClass ordered-pair not-subclass-element not-homomorphism2 not-homomorphism1 INVERSE intersection image′ flip first domain-of domain difference diagonalise cross-product compos complement cantor apply equal &
SET004-1-ax range-of function maps apply application-function singleton-relation element-relation complement intersection single-valued3 singleton image′ domain single-valued2 second single-valued1 identity-relation INVERSE not-subclass-element first domain-of domain-relation composition-function compos equal ordered-pair member universal-class cross-product compose-class subclass &
SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal
&
NUM004-0-ax integer-of omega ordinal-multiply add-relation ordinal-add recursion apply range-of union-of-range-map function recursion-equation-functions rest-relation rest-of limit-ordinals kind-1-ordinals successor-relation image′ universal-class sum-class element-relation ordinal-numbers section not-well-ordering ordered-pair least member well-ordering singleton domain-of segment null-class intersection asymmetric compos transitive cross-product connected identity-relation complement restrct subclass irreflexive symmetrization-of INVERSE union equal &
NUM004-0-eq well-ordering transitive section irreflexive connected asymmetric symmetrization-of segment rest-of recursion-equation-functions recursion ordinal-multiply ordinal-add not-well-ordering least integer-of equal &
(~*subclass*(*limit-ordinals*::′*a*,*ordinal-numbers*)) −−> *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 &
(~*function*(*z*)) &
(~*equal*(*recursion-equation-functions*(*z*),*null-class*)) −−> *False*
by *meson*

lemma *PLA002-1*:
(∀ *Situation1 Situation2*. *warm*(*Situation1*) | *cold*(*Situation2*)) &
(∀ *Situation*. *at*(*a*::′*a*,*Situation*) −−> *at*(*b*::′*a*,*walk*(*b*::′*a*,*Situation*))) &
(∀ *Situation*. *at*(*a*::′*a*,*Situation*) −−> *at*(*b*::′*a*,*drive*(*b*::′*a*,*Situation*))) &
(∀ *Situation*. *at*(*b*::′*a*,*Situation*) −−> *at*(*a*::′*a*,*walk*(*a*::′*a*,*Situation*))) &
(∀ *Situation*. *at*(*b*::′*a*,*Situation*) −−> *at*(*a*::′*a*,*drive*(*a*::′*a*,*Situation*))) &
(∀ *Situation*. *cold*(*Situation*) & *at*(*b*::′*a*,*Situation*) −−> *at*(*c*::′*a*,*skate*(*c*::′*a*,*Situation*)))
&
(∀ *Situation*. *cold*(*Situation*) & *at*(*c*::′*a*,*Situation*) −−> *at*(*b*::′*a*,*skate*(*b*::′*a*,*Situation*)))
&
(∀ *Situation*. *warm*(*Situation*) & *at*(*b*::′*a*,*Situation*) −−> *at*(*d*::′*a*,*climb*(*d*::′*a*,*Situation*)))
&

(∀ Situation. warm(Situation) & at(d::'a,Situation) --> at(b::'a,climb(b::'a,Situation)))
&
(∀ Situation. at(c::'a,Situation) --> at(d::'a,go(d::'a,Situation))) &
(∀ Situation. at(d::'a,Situation) --> at(c::'a,go(c::'a,Situation))) &
(∀ Situation. at(c::'a,Situation) --> at(e::'a,go(e::'a,Situation))) &
(∀ Situation. at(e::'a,Situation) --> at(c::'a,go(c::'a,Situation))) &
(∀ Situation. at(d::'a,Situation) --> at(f::'a,go(f::'a,Situation))) &
(∀ Situation. at(f::'a,Situation) --> at(d::'a,go(d::'a,Situation))) &
(at(f::'a,s0)) &
(∀ S'. ~at(a::'a,S')) --> False
by *meson*

abbreviation *PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds* ≡
(∀ X Y State. holds(X::'a,State) & holds(Y::'a,State) --> holds(and'(X::'a,Y),State))
&
(∀ State X. holds(EMPTY::'a,State) & holds(clear(X),State) & differ(X::'a,table)
--> holds(holding(X),do(pickup(X),State))) &
(∀ Y X State. holds(on(X::'a,Y),State) & holds(clear(X),State) & holds(EMPTY::'a,State)
--> holds(clear(Y),do(pickup(X),State))) &
(∀ Y State X Z. holds(on(X::'a,Y),State) & differ(X::'a,Z) --> holds(on(X::'a,Y),do(pickup(Z),State)))
&
(∀ State X Z. holds(clear(X),State) & differ(X::'a,Z) --> holds(clear(X),do(pickup(Z),State)))
&
(∀ X Y State. holds(holding(X),State) & holds(clear(Y),State) --> holds(EMPTY::'a,do(putdown(X::'a,Y
&
(∀ X Y State. holds(holding(X),State) & holds(clear(Y),State) --> holds(on(X::'a,Y),do(putdown(X::'a,Y
&
(∀ X Y State. holds(holding(X),State) & holds(clear(Y),State) --> holds(clear(X),do(putdown(X::'a,Y),St
&
(∀ Z W X Y State. holds(on(X::'a,Y),State) --> holds(on(X::'a,Y),do(putdown(Z::'a,W),State)))
&
(∀ X State Z Y. holds(clear(Z),State) & differ(Z::'a,Y) --> holds(clear(Z),do(putdown(X::'a,Y),State)))

abbreviation *PLA001-1-ax EMPTY clear s0 on holds table d c b a differ* ≡
(∀ Y X. differ(Y::'a,X) --> differ(X::'a,Y)) &
(differ(a::'a,b)) &
(differ(a::'a,c)) &
(differ(a::'a,d)) &
(differ(a::'a,table)) &
(differ(b::'a,c)) &
(differ(b::'a,d)) &
(differ(b::'a,table)) &
(differ(c::'a,d)) &
(differ(c::'a,table)) &
(differ(d::'a,table)) &
(holds(on(a::'a,table),s0)) &
(holds(on(b::'a,table),s0)) &
(holds(on(c::'a,d),s0)) &

(*holds*(*on*(*d*::′*a*,*table*),*s0*)) &
(*holds*(*clear*(*a*),*s0*)) &
(*holds*(*clear*(*b*),*s0*)) &
(*holds*(*clear*(*c*),*s0*)) &
(*holds*(*EMPTY*::′*a*,*s0*)) &
(∀ *State*. *holds*(*clear*(*table*),*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 &
(∀ *State*. ~*holds*(*on*(*c*::′*a*,*table*),*State*)) −−> *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 &
(∀ *State*. ~*holds*(*on*(*a*::′*a*,*c*),*State*)) −−> *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 &
(∀ *State*. ~*holds*(*and′*(*on*(*c*::′*a*,*d*),*on*(*a*::′*a*,*c*)),*State*)) −−> *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 &
(∀ *State*. ~*holds*(*and′*(*on*(*a*::′*a*,*c*),*on*(*c*::′*a*,*d*)),*State*)) −−> *False*
by *meson*

lemma *PRV001-1*:
(∀ *X Y Z*. *q1*(*X*::′*a*,*Y*,*Z*) & *mless-or-equal*(*X*::′*a*,*Y*) −−> *q2*(*X*::′*a*,*Y*,*Z*)) &
(∀ *X Y Z*. *q1*(*X*::′*a*,*Y*,*Z*) −−> *mless-or-equal*(*X*::′*a*,*Y*) | *q3*(*X*::′*a*,*Y*,*Z*)) &
(∀ *Z X Y*. *q2*(*X*::′*a*,*Y*,*Z*) −−> *q4*(*X*::′*a*,*Y*,*Y*)) &
(∀ *Z Y X*. *q3*(*X*::′*a*,*Y*,*Z*) −−> *q4*(*X*::′*a*,*Y*,*X*)) &
(∀ *X*. *mless-or-equal*(*X*::′*a*,*X*)) &
(∀ *X Y*. *mless-or-equal*(*X*::′*a*,*Y*) & *mless-or-equal*(*Y*::′*a*,*X*) −−> *equal*(*X*::′*a*,*Y*)) &
(∀ *Y X Z*. *mless-or-equal*(*X*::′*a*,*Y*) & *mless-or-equal*(*Y*::′*a*,*Z*) −−> *mless-or-equal*(*X*::′*a*,*Z*))

&
 $(\forall Y X. \ \textit{mless-or-equal}(X::'a,Y) \mid \textit{mless-or-equal}(Y::'a,X))$ &
 $(\forall X Y. \ \textit{equal}(X::'a,Y) \ --> \ \textit{mless-or-equal}(X::'a,Y))$ &
 $(\forall X Y Z. \ \textit{equal}(X::'a,Y) \ \& \ \textit{mless-or-equal}(X::'a,Z) \ --> \ \textit{mless-or-equal}(Y::'a,Z))$
&
 $(\forall X Z Y. \ \textit{equal}(X::'a,Y) \ \& \ \textit{mless-or-equal}(Z::'a,X) \ --> \ \textit{mless-or-equal}(Z::'a,Y))$
&
 $(\textit{q1}(a::'a,b,c))$ &
 $(\forall W. \ \sim(\textit{q4}(a::'a,b,W) \ \& \ \textit{mless-or-equal}(a::'a,W) \ \& \ \textit{mless-or-equal}(b::'a,W) \ \&$
$\textit{mless-or-equal}(W::'a,a)))$ &
 $(\forall W. \ \sim(\textit{q4}(a::'a,b,W) \ \& \ \textit{mless-or-equal}(a::'a,W) \ \& \ \textit{mless-or-equal}(b::'a,W) \ \&$
$\textit{mless-or-equal}(W::'a,b)))$ $-->$ *False*
 by *meson*

abbreviation *SWV001-1-ax mless-THAN successor predecessor equal* $\equiv$
 $(\forall X. \ \textit{equal}(\textit{predecessor}(\textit{successor}(X)),X))$ &
 $(\forall X. \ \textit{equal}(\textit{successor}(\textit{predecessor}(X)),X))$ &
 $(\forall X Y. \ \textit{equal}(\textit{predecessor}(X),\textit{predecessor}(Y)) \ --> \ \textit{equal}(X::'a,Y))$ &
 $(\forall X Y. \ \textit{equal}(\textit{successor}(X),\textit{successor}(Y)) \ --> \ \textit{equal}(X::'a,Y))$ &
 $(\forall X. \ \textit{mless-THAN}(\textit{predecessor}(X),X))$ &
 $(\forall X. \ \textit{mless-THAN}(X::'a,\textit{successor}(X)))$ &
 $(\forall X Y Z. \ \textit{mless-THAN}(X::'a,Y) \ \& \ \textit{mless-THAN}(Y::'a,Z) \ --> \ \textit{mless-THAN}(X::'a,Z))$
&
 $(\forall X Y. \ \textit{mless-THAN}(X::'a,Y) \mid \textit{mless-THAN}(Y::'a,X) \mid \textit{equal}(X::'a,Y))$ &
 $(\forall X. \ \sim\textit{mless-THAN}(X::'a,X))$ &
 $(\forall Y X. \ \sim(\textit{mless-THAN}(X::'a,Y) \ \& \ \textit{mless-THAN}(Y::'a,X)))$ &
 $(\forall Y X Z. \ \textit{equal}(X::'a,Y) \ \& \ \textit{mless-THAN}(X::'a,Z) \ --> \ \textit{mless-THAN}(Y::'a,Z))$
&
 $(\forall Y Z X. \ \textit{equal}(X::'a,Y) \ \& \ \textit{mless-THAN}(Z::'a,X) \ --> \ \textit{mless-THAN}(Z::'a,Y))$

abbreviation *SWV001-0-eq a successor predecessor equal* $\equiv$
 $(\forall X Y. \ \textit{equal}(X::'a,Y) \ --> \ \textit{equal}(\textit{predecessor}(X),\textit{predecessor}(Y)))$ &
 $(\forall X Y. \ \textit{equal}(X::'a,Y) \ --> \ \textit{equal}(\textit{successor}(X),\textit{successor}(Y)))$ &
 $(\forall X Y. \ \textit{equal}(X::'a,Y) \ --> \ \textit{equal}(a(X),a(Y)))$

lemma *PRV003-1*:
 EQU001-0-ax equal &
 SWV001-1-ax mless-THAN successor predecessor equal &
 SWV001-0-eq a successor predecessor equal &
 $(\sim\textit{mless-THAN}(n::'a,j))$ &
 $(\textit{mless-THAN}(k::'a,j))$ &
 $(\sim\textit{mless-THAN}(k::'a,i))$ &
 $(\textit{mless-THAN}(i::'a,n))$ &
 $(\textit{mless-THAN}(a(j),a(k)))$ &
 $(\forall X. \ \textit{mless-THAN}(X::'a,j) \ \& \ \textit{mless-THAN}(a(X),a(k)) \ --> \ \textit{mless-THAN}(X::'a,i))$
&
 $(\forall X. \ \textit{mless-THAN}(\textit{One}::'a,i) \ \& \ \textit{mless-THAN}(a(X),a(\textit{predecessor}(i))) \ --> \ \textit{mless-THAN}(X::'a,i)$

| *mless-THAN(n::'a,X))* &
 (∀ *X*. ~(*mless-THAN(One::'a,X)* & *mless-THAN(X::'a,i)* & *mless-THAN(a(X),a(predecessor(X)))))*
&
 (*mless-THAN(j::'a,i)*) *--> 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 &
 (~*mless-THAN(n::'a,k)*) &
 (~*mless-THAN(k::'a,l)*) &
 (~*mless-THAN(k::'a,i)*) &
 (*mless-THAN(l::'a,n)*) &
 (*mless-THAN(One::'a,l)*) &
 (*mless-THAN(a(k),a(predecessor(l)))*) &
 (∀ *X*. *mless-THAN(X::'a,successor(n))* & *mless-THAN(a(X),a(k))* *-->* *mless-THAN(X::'a,l)*)
&
 (∀ *X*. *mless-THAN(One::'a,l)* & *mless-THAN(a(X),a(predecessor(l)))* *-->* *mless-THAN(X::'a,l)*
| *mless-THAN(n::'a,X)*) &
 (∀ *X*. ~(*mless-THAN(One::'a,X)* & *mless-THAN(X::'a,l)* & *mless-THAN(a(X),a(predecessor(X)))))*
--> 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 &
 (~*mless-THAN(n::'a,m)*) &
 (*mless-THAN(i::'a,m)*) &
 (*mless-THAN(i::'a,n)*) &
 (~*mless-THAN(i::'a,One)*) &
 (*mless-THAN(a(i),a(m))*) &
 (∀ *X*. *mless-THAN(X::'a,successor(n))* & *mless-THAN(a(X),a(m))* *-->* *mless-THAN(X::'a,i)*)
&
 (∀ *X*. *mless-THAN(One::'a,i)* & *mless-THAN(a(X),a(predecessor(i)))* *-->* *mless-THAN(X::'a,i)*
| *mless-THAN(n::'a,X)*) &
 (∀ *X*. ~(*mless-THAN(One::'a,X)* & *mless-THAN(X::'a,i)* & *mless-THAN(a(X),a(predecessor(X)))))*
--> False
 by *meson*

lemma *PRV009-1*:
 (∀ *Y X*. *mless-or-equal(X::'a,Y)* | *mless(Y::'a,X)*) &
 (*mless(j::'a,i)*) &
 (*mless-or-equal(m::'a,p)*) &
 (*mless-or-equal(p::'a,q)*) &

(*mless-or-equal*(*q*::′*a*,*n*)) &
(∀ *X Y*. *mless-or-equal*(*m*::′*a*,*X*) & *mless*(*X*::′*a*,*i*) & *mless*(*j*::′*a*,*Y*) & *mless-or-equal*(*Y*::′*a*,*n*)
−−> *mless-or-equal*(*a*(*X*),*a*(*Y*))) &
(∀ *X Y*. *mless-or-equal*(*m*::′*a*,*X*) & *mless-or-equal*(*X*::′*a*,*Y*) & *mless-or-equal*(*Y*::′*a*,*j*)
−−> *mless-or-equal*(*a*(*X*),*a*(*Y*))) &
(∀ *X Y*. *mless-or-equal*(*i*::′*a*,*X*) & *mless-or-equal*(*X*::′*a*,*Y*) & *mless-or-equal*(*Y*::′*a*,*n*)
−−> *mless-or-equal*(*a*(*X*),*a*(*Y*))) &
(~*mless-or-equal*(*a*(*p*),*a*(*q*))) −−> *False*
by *meson*

lemma *PUZ012-1*:
(∀ *X*. *equal-fruits*(*X*::′*a*,*X*)) &
(∀ *X*. *equal-boxes*(*X*::′*a*,*X*)) &
(∀ *X Y*. ~(*label*(*X*::′*a*,*Y*) & *contains*(*X*::′*a*,*Y*))) &
(∀ *X*. *contains*(*boxa*::′*a*,*X*) | *contains*(*boxb*::′*a*,*X*) | *contains*(*boxc*::′*a*,*X*)) &
(∀ *X*. *contains*(*X*::′*a*,*apples*) | *contains*(*X*::′*a*,*bananas*) | *contains*(*X*::′*a*,*oranges*))
&
(∀ *X Y Z*. *contains*(*X*::′*a*,*Y*) & *contains*(*X*::′*a*,*Z*) −−> *equal-fruits*(*Y*::′*a*,*Z*)) &
(∀ *Y X Z*. *contains*(*X*::′*a*,*Y*) & *contains*(*Z*::′*a*,*Y*) −−> *equal-boxes*(*X*::′*a*,*Z*)) &
(~*equal-boxes*(*boxa*::′*a*,*boxb*)) &
(~*equal-boxes*(*boxb*::′*a*,*boxc*)) &
(~*equal-boxes*(*boxa*::′*a*,*boxc*)) &
(~*equal-fruits*(*apples*::′*a*,*bananas*)) &
(~*equal-fruits*(*bananas*::′*a*,*oranges*)) &
(~*equal-fruits*(*apples*::′*a*,*oranges*)) &
(*label*(*boxa*::′*a*,*apples*)) &
(*label*(*boxb*::′*a*,*oranges*)) &
(*label*(*boxc*::′*a*,*bananas*)) &
(*contains*(*boxb*::′*a*,*apples*)) &
(~(*contains*(*boxa*::′*a*,*bananas*) & *contains*(*boxc*::′*a*,*oranges*))) −−> *False*
by *meson*

lemma *PUZ020-1*:
EQU001-0-ax equal &
(∀ *A B*. *equal*(*A*::′*a*,*B*) −−> *equal*(*statement-by*(*A*),*statement-by*(*B*))) &
(∀ *X*. *person*(*X*) −−> *knight*(*X*) | *knave*(*X*)) &
(∀ *X*. ~(*person*(*X*) & *knight*(*X*) & *knave*(*X*))) &
(∀ *X Y*. *says*(*X*::′*a*,*Y*) & *a-truth*(*Y*) −−> *a-truth*(*Y*)) &
(∀ *X Y*. ~(*says*(*X*::′*a*,*Y*) & *equal*(*X*::′*a*,*Y*))) &
(∀ *Y X*. *says*(*X*::′*a*,*Y*) −−> *equal*(*Y*::′*a*,*statement-by*(*X*))) &
(∀ *X Y*. ~(*person*(*X*) & *equal*(*X*::′*a*,*statement-by*(*Y*)))) &
(∀ *X*. *person*(*X*) & *a-truth*(*statement-by*(*X*)) −−> *knight*(*X*)) &
(∀ *X*. *person*(*X*) −−> *a-truth*(*statement-by*(*X*)) | *knave*(*X*)) &
(∀ *X Y*. *equal*(*X*::′*a*,*Y*) & *knight*(*X*) −−> *knight*(*Y*)) &
(∀ *X Y*. *equal*(*X*::′*a*,*Y*) & *knave*(*X*) −−> *knave*(*Y*)) &
(∀ *X Y*. *equal*(*X*::′*a*,*Y*) & *person*(*X*) −−> *person*(*Y*)) &
(∀ *X Y Z*. *equal*(*X*::′*a*,*Y*) & *says*(*X*::′*a*,*Z*) −−> *says*(*Y*::′*a*,*Z*)) &

(∀ X Z Y. equal(X::'a,Y) & says(Z::'a,X) --> says(Z::'a,Y)) &
(∀ X Y. equal(X::'a,Y) & a-truth(X) --> a-truth(Y)) &
(∀ X Y. knight(X) & says(X::'a,Y) --> a-truth(Y)) &
(∀ X Y. ~(knave(X) & says(X::'a,Y) & a-truth(Y))) &
(person(husband)) &
(person(wife)) &
(~equal(husband::'a,wife)) &
(says(husband::'a,statement-by(husband))) &
(a-truth(statement-by(husband)) & knight(husband) --> knight(wife)) &
(knight(husband) --> a-truth(statement-by(husband))) &
(a-truth(statement-by(husband)) | knight(wife)) &
(knight(wife) --> a-truth(statement-by(husband))) &
(~knight(husband)) --> False
by *meson*

lemma *PUZ025-1*:
(∀ X. a-truth(truthteller(X)) | a-truth(liar(X))) &
(∀ X. ~(a-truth(truthteller(X)) & a-truth(liar(X)))) &
(∀ Truthteller Statement. a-truth(truthteller(Truthteller)) & a-truth(says(Truthteller::'a,Statement)) --> a-truth(Statement)) &
(∀ Liar Statement. ~(a-truth(liar(Liar)) & a-truth(says(Liar::'a,Statement)) & a-truth(Statement))) &
(∀ Statement Truthteller. a-truth(Statement) & a-truth(says(Truthteller::'a,Statement)) --> a-truth(truthteller(Truthteller))) &
(∀ Statement Liar. a-truth(says(Liar::'a,Statement)) --> a-truth(Statement) | a-truth(liar(Liar))) &
(∀ Z X Y. people(X::'a,Y,Z) & a-truth(liar(X)) & a-truth(liar(Y)) --> a-truth(equal-type(X::'a,Y))) &
(∀ Z X Y. people(X::'a,Y,Z) & a-truth(truthteller(X)) & a-truth(truthteller(Y)) --> a-truth(equal-type(X::'a,Y))) &
(∀ X Y. a-truth(equal-type(X::'a,Y)) & a-truth(truthteller(X)) --> a-truth(truthteller(Y))) &
(∀ X Y. a-truth(equal-type(X::'a,Y)) & a-truth(liar(X)) --> a-truth(liar(Y))) &
(∀ X Y. a-truth(truthteller(X)) --> a-truth(equal-type(X::'a,Y)) | a-truth(liar(Y))) &
(∀ X Y. a-truth(liar(X)) --> a-truth(equal-type(X::'a,Y)) | a-truth(truthteller(Y))) &
(∀ Y X. a-truth(equal-type(X::'a,Y)) --> a-truth(equal-type(Y::'a,X))) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(truthteller(X)) & a-truth(Y) --> answer(yes)) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(truthteller(X)) --> a-truth(Y) | answer(no)) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(liar(X)) & a-truth(Y) --> answer(no)) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(liar(X)) --> a-truth(Y) | answer(yes)) &
(people(b::'a,c,a)) &

(*people*(*a*::'*a*,*b*,*a*)) &
(*people*(*a*::'*a*,*c*,*b*)) &
(*people*(*c*::'*a*,*b*,*a*)) &
(*a-truth*(*says*(*a*::'*a*,*equal-type*(*b*::'*a*,*c*)))) &
(*ask-1-if-2*(*c*::'*a*,*equal-type*(*a*::'*a*,*b*))) &
(∀ *Answer*. ~*answer*(*Answer*)) −−> *False*
oops

lemma *PUZ029-1*:
(∀ *X*. *dances-on-tightropes*(*X*) | *eats-pennybuns*(*X*) | *old*(*X*)) &
(∀ *X*. *pig*(*X*) & *liable-to-giddiness*(*X*) −−> *treated-with-respect*(*X*)) &
(∀ *X*. *wise*(*X*) & *balloonist*(*X*) −−> *has-umbrella*(*X*)) &
(∀ *X*. ~(*looks-ridiculous*(*X*) & *eats-pennybuns*(*X*) & *eats-lunch-in-public*(*X*))) &
(∀ *X*. *balloonist*(*X*) & *young*(*X*) −−> *liable-to-giddiness*(*X*)) &
(∀ *X*. *fat*(*X*) & *looks-ridiculous*(*X*) −−> *dances-on-tightropes*(*X*) | *eats-lunch-in-public*(*X*))
&
(∀ *X*. ~(*liable-to-giddiness*(*X*) & *wise*(*X*) & *dances-on-tightropes*(*X*))) &
(∀ *X*. *pig*(*X*) & *has-umbrella*(*X*) −−> *looks-ridiculous*(*X*)) &
(∀ *X*. *treated-with-respect*(*X*) −−> *dances-on-tightropes*(*X*) | *fat*(*X*)) &
(∀ *X*. *young*(*X*) | *old*(*X*)) &
(∀ *X*. ~(*young*(*X*) & *old*(*X*))) &
(*wise*(*piggy*)) &
(*young*(*piggy*)) &
(*pig*(*piggy*)) &
(*balloonist*(*piggy*)) −−> *False*
by *meson*

abbreviation *RNG001-0-ax equal additive-inverse add multiply product additive-identity sum* ≡
(∀ *X*. *sum*(*additive-identity*::'*a*,*X*,*X*)) &
(∀ *X*. *sum*(*X*::'*a*,*additive-identity*,*X*)) &
(∀ *X Y*. *product*(*X*::'*a*,*Y*,*multiply*(*X*::'*a*,*Y*))) &
(∀ *X Y*. *sum*(*X*::'*a*,*Y*,*add*(*X*::'*a*,*Y*))) &
(∀ *X*. *sum*(*additive-inverse*(*X*),*X*,*additive-identity*)) &
(∀ *X*. *sum*(*X*::'*a*,*additive-inverse*(*X*),*additive-identity*)) &
(∀ *Y U Z X V W*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*Y*::'*a*,*Z*,*V*) & *sum*(*U*::'*a*,*Z*,*W*) −−>
sum(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*Y*::'*a*,*Z*,*V*) & *sum*(*X*::'*a*,*V*,*W*) −−>
sum(*U*::'*a*,*Z*,*W*)) &
(∀ *Y X Z*. *sum*(*X*::'*a*,*Y*,*Z*) −−> *sum*(*Y*::'*a*,*X*,*Z*)) &
(∀ *Y U Z X V W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*U*::'*a*,*Z*,*W*)
−−> *product*(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*X*::'*a*,*V*,*W*)
−−> *product*(*U*::'*a*,*Z*,*W*)) &
(∀ *Y Z X V3 V1 V2 V4*. *product*(*X*::'*a*,*Y*,*V1*) & *product*(*X*::'*a*,*Z*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)
& *product*(*X*::'*a*,*V3*,*V4*) −−> *sum*(*V1*::'*a*,*V2*,*V4*)) &
(∀ *Y Z V1 V2 X V3 V4*. *product*(*X*::'*a*,*Y*,*V1*) & *product*(*X*::'*a*,*Z*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)

& sum(V1::'a,V2,V4) --> product(X::'a,V3,V4)) &
(∀ Y Z V3 X V1 V2 V4. product(Y::'a,X,V1) & product(Z::'a,X,V2) & sum(Y::'a,Z,V3)
& product(V3::'a,X,V4) --> sum(V1::'a,V2,V4)) &
(∀ Y Z V1 V2 V3 X V4. product(Y::'a,X,V1) & product(Z::'a,X,V2) & sum(Y::'a,Z,V3)
& sum(V1::'a,V2,V4) --> product(V3::'a,X,V4)) &
(∀ X Y U V. sum(X::'a,Y,U) & sum(X::'a,Y,V) --> equal(U::'a,V)) &
(∀ X Y U V. product(X::'a,Y,U) & product(X::'a,Y,V) --> equal(U::'a,V))

abbreviation *RNG001-0-eq product multiply sum add additive-inverse equal* ≡
(∀ X Y. equal(X::'a,Y) --> equal(additive-inverse(X),additive-inverse(Y))) &
(∀ X Y W. equal(X::'a,Y) --> equal(add(X::'a,W),add(Y::'a,W))) &
(∀ X W Y. equal(X::'a,Y) --> equal(add(W::'a,X),add(W::'a,Y))) &
(∀ X Y W Z. equal(X::'a,Y) & sum(X::'a,W,Z) --> sum(Y::'a,W,Z)) &
(∀ X W Y Z. equal(X::'a,Y) & sum(W::'a,X,Z) --> sum(W::'a,Y,Z)) &
(∀ X W Z Y. equal(X::'a,Y) & sum(W::'a,Z,X) --> sum(W::'a,Z,Y)) &
(∀ X Y W. equal(X::'a,Y) --> equal(multiply(X::'a,W),multiply(Y::'a,W)))
&
(∀ X W Y. equal(X::'a,Y) --> equal(multiply(W::'a,X),multiply(W::'a,Y)))
&
(∀ X Y W Z. equal(X::'a,Y) & product(X::'a,W,Z) --> product(Y::'a,W,Z))
&
(∀ X W Y Z. equal(X::'a,Y) & product(W::'a,X,Z) --> product(W::'a,Y,Z))
&
(∀ X W Z Y. equal(X::'a,Y) & product(W::'a,Z,X) --> product(W::'a,Z,Y))

lemma *RNG001-3*:
(∀ X. sum(additive-identity::'a,X,X)) &
(∀ X. sum(additive-inverse(X),X,additive-identity)) &
(∀ Y U Z X V W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(U::'a,Z,W) -->
sum(X::'a,V,W)) &
(∀ Y X V U Z W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(X::'a,V,W) -->
sum(U::'a,Z,W)) &
(∀ X Y. product(X::'a,Y,multiply(X::'a,Y))) &
(∀ Y Z X V3 V1 V2 V4. product(X::'a,Y,V1) & product(X::'a,Z,V2) & sum(Y::'a,Z,V3)
& product(X::'a,V3,V4) --> sum(V1::'a,V2,V4)) &
(∀ Y Z V1 V2 X V3 V4. product(X::'a,Y,V1) & product(X::'a,Z,V2) & sum(Y::'a,Z,V3)
& sum(V1::'a,V2,V4) --> product(X::'a,V3,V4)) &
(~product(a::'a,additive-identity,additive-identity)) --> False
oops

abbreviation *RNG-other-ax multiply add equal product additive-identity additive-inverse sum* ≡
(∀ X. sum(X::'a,additive-inverse(X),additive-identity)) &
(∀ Y U Z X V W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(U::'a,Z,W) -->
sum(X::'a,V,W)) &
(∀ Y X V U Z W. sum(X::'a,Y,U) & sum(Y::'a,Z,V) & sum(X::'a,V,W) -->
sum(U::'a,Z,W)) &
(∀ Y X Z. sum(X::'a,Y,Z) --> sum(Y::'a,X,Z)) &

$(\forall\ Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)\ \&\ product(Y::'a,Z,V)\ \&\ product(U::'a,Z,W)$
$--> product(X::'a,V,W))\ \&$
$(\forall\ Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)\ \&\ product(Y::'a,Z,V)\ \&\ product(X::'a,V,W)$
$--> product(U::'a,Z,W))\ \&$
$(\forall\ Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ product(X::'a,V3,V4)\ --> sum(V1::'a,V2,V4))\ \&$
$(\forall\ Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ sum(V1::'a,V2,V4)\ --> product(X::'a,V3,V4))\ \&$
$(\forall\ Y\ Z\ V3\ X\ V1\ V2\ V4.\ product(Y::'a,X,V1)\ \&\ product(Z::'a,X,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ product(V3::'a,X,V4)\ --> sum(V1::'a,V2,V4))\ \&$
$(\forall\ Y\ Z\ V1\ V2\ V3\ X\ V4.\ product(Y::'a,X,V1)\ \&\ product(Z::'a,X,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ sum(V1::'a,V2,V4)\ --> product(V3::'a,X,V4))\ \&$
$(\forall\ X\ Y\ U\ V.\ sum(X::'a,Y,U)\ \&\ sum(X::'a,Y,V)\ --> equal(U::'a,V))\ \&$
$(\forall\ X\ Y\ U\ V.\ product(X::'a,Y,U)\ \&\ product(X::'a,Y,V)\ --> equal(U::'a,V))$
$\&$
$(\forall\ X\ Y.\ equal(X::'a,Y)\ --> equal(additive\text{-}inverse(X),additive\text{-}inverse(Y)))\ \&$
$(\forall\ X\ Y\ W.\ equal(X::'a,Y)\ --> equal(add(X::'a,W),add(Y::'a,W)))\ \&$
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ sum(X::'a,W,Z)\ --> sum(Y::'a,W,Z))\ \&$
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ sum(W::'a,X,Z)\ --> sum(W::'a,Y,Z))\ \&$
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ sum(W::'a,Z,X)\ --> sum(W::'a,Z,Y))\ \&$
$(\forall\ X\ Y\ W.\ equal(X::'a,Y)\ --> equal(multiply(X::'a,W),multiply(Y::'a,W)))$
$\&$
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ product(X::'a,W,Z)\ --> product(Y::'a,W,Z))$
$\&$
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ product(W::'a,X,Z)\ --> product(W::'a,Y,Z))$
$\&$
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ product(W::'a,Z,X)\ --> product(W::'a,Z,Y))$

lemma *RNG001-5*:
EQU001-0-ax equal &
$(\forall\ X.\ sum(additive\text{-}identity::'a,X,X))\ \&$
$(\forall\ X.\ sum(X::'a,additive\text{-}identity,X))\ \&$
$(\forall\ X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))\ \&$
$(\forall\ X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))\ \&$
$(\forall\ X.\ sum(additive\text{-}inverse(X),X,additive\text{-}identity))\ \&$
RNG-other-ax multiply add equal product additive-identity additive-inverse sum
&
$(\sim product(a::'a,additive\text{-}identity,additive\text{-}identity))\ --> False$
oops

lemma *RNG011-5*:
EQU001-0-ax equal &
$(\forall\ A\ B\ C.\ equal(A::'a,B)\ --> equal(add(A::'a,C),add(B::'a,C)))\ \&$
$(\forall\ D\ F'\ E.\ equal(D::'a,E)\ --> equal(add(F'::'a,D),add(F'::'a,E)))\ \&$
$(\forall\ G\ H.\ equal(G::'a,H)\ --> equal(additive\text{-}inverse(G),additive\text{-}inverse(H)))\ \&$
$(\forall\ I'\ J\ K'.\ equal(I'::'a,J)\ --> equal(multiply(I'::'a,K'),multiply(J::'a,K')))\ \&$

(∀ L N M. equal(L::'a,M) --> equal(multiply(N::'a,L),multiply(N::'a,M))) &
(∀ A B C D. equal(A::'a,B) --> equal(associator(A::'a,C,D),associator(B::'a,C,D)))
&
(∀ E G F' H. equal(E::'a,F') --> equal(associator(G::'a,E,H),associator(G::'a,F',H)))
&
(∀ I' K' L J. equal(I'::'a,J) --> equal(associator(K'::'a,L,I'),associator(K'::'a,L,J)))
&
(∀ M N O'. equal(M::'a,N) --> equal(commutator(M::'a,O'),commutator(N::'a,O')))
&
(∀ P R Q. equal(P::'a,Q) --> equal(commutator(R::'a,P),commutator(R::'a,Q)))
&
(∀ 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)))) &
(∀ X. equal(add(X::'a,additive-identity),X)) &
(∀ X. equal(add(additive-identity::'a,X),X)) &
(∀ X. equal(add(X::'a,additive-inverse(X)),additive-identity)) &
(∀ X. equal(add(additive-inverse(X),X),additive-identity)) &
(equal(additive-inverse(additive-identity),additive-identity)) &
(∀ X Y. equal(add(X::'a,add(additive-inverse(X),Y)),Y)) &
(∀ X Y. equal(additive-inverse(add(X::'a,Y)),add(additive-inverse(X),additive-inverse(Y))))
&
(∀ X. equal(additive-inverse(additive-inverse(X)),X)) &
(∀ X. equal(multiply(X::'a,additive-identity),additive-identity)) &
(∀ X. equal(multiply(additive-identity::'a,X),additive-identity)) &
(∀ X Y. equal(multiply(additive-inverse(X),additive-inverse(Y)),multiply(X::'a,Y)))
&
(∀ X Y. equal(multiply(X::'a,additive-inverse(Y)),additive-inverse(multiply(X::'a,Y))))
&
(∀ X Y. equal(multiply(additive-inverse(X),Y),additive-inverse(multiply(X::'a,Y))))
&
(∀ Y X Z. equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))
&
(∀ X Y Z. equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))
&
(∀ X Y. equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))
&
(∀ X Y Z. equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive-inverse(multiply(X::'a,m
&
(∀ X Y. equal(commutator(X::'a,Y),add(multiply(Y::'a,X),additive-inverse(multiply(X::'a,Y)))))
&
(∀ X Y. equal(multiply(multiply(associator(X::'a,X,Y),X),associator(X::'a,X,Y)),additive-identity))
&
(~equal(multiply(multiply(associator(a::'a,a,b),a),associator(a::'a,a,b)),additive-identity))
--> False
by *meson*

lemma *RNG023-6*:
EQU001-0-ax equal &

(∀ Y X. equal(add(X::'a,Y),add(Y::'a,X))) &
(∀ X Y Z. equal(add(X::'a,add(Y::'a,Z)),add(add(X::'a,Y),Z))) &
(∀ X. equal(add(additive-identity::'a,X),X)) &
(∀ X. equal(add(X::'a,additive-identity),X)) &
(∀ X. equal(multiply(additive-identity::'a,X),additive-identity)) &
(∀ X. equal(multiply(X::'a,additive-identity),additive-identity)) &
(∀ X. equal(add(additive-inverse(X),X),additive-identity)) &
(∀ X. equal(add(X::'a,additive-inverse(X)),additive-identity)) &
(∀ Y X Z. equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))
&
(∀ X Y Z. equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))
&
(∀ X. equal(additive-inverse(additive-inverse(X)),X)) &
(∀ X Y. equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))
&
(∀ X Y. equal(multiply(multiply(X::'a,X),Y),multiply(X::'a,multiply(X::'a,Y))))
&
(∀ X Y Z. equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive-inverse(multiply(X::'a,m
&
(∀ X Y. equal(commutator(X::'a,Y),add(multiply(Y::'a,X),additive-inverse(multiply(X::'a,Y)))))
&
(∀ D E F'. equal(D::'a,E) --> equal(add(D::'a,F'),add(E::'a,F'))) &
(∀ G I' H. equal(G::'a,H) --> equal(add(I'::'a,G),add(I'::'a,H))) &
(∀ J K'. equal(J::'a,K') --> equal(additive-inverse(J),additive-inverse(K'))) &
(∀ L M N O'. equal(L::'a,M) --> equal(associator(L::'a,N,O'),associator(M::'a,N,O')))
&
(∀ P R Q S'. equal(P::'a,Q) --> equal(associator(R::'a,P,S'),associator(R::'a,Q,S')))
&
(∀ T' V W U. equal(T'::'a,U) --> equal(associator(V::'a,W,T'),associator(V::'a,W,U)))
&
(∀ X Y Z. equal(X::'a,Y) --> equal(commutator(X::'a,Z),commutator(Y::'a,Z)))
&
(∀ A1 C1 B1. equal(A1::'a,B1) --> equal(commutator(C1::'a,A1),commutator(C1::'a,B1)))
&
(∀ D1 E1 F1. equal(D1::'a,E1) --> equal(multiply(D1::'a,F1),multiply(E1::'a,F1)))
&
(∀ G1 I1 H1. equal(G1::'a,H1) --> equal(multiply(I1::'a,G1),multiply(I1::'a,H1)))
&
(~equal(associator(x::'a,x,y),additive-identity)) --> False
by *meson*

lemma *RNG028-2*:
EQU001-0-ax equal &
(∀ X. equal(add(additive-identity::'a,X),X)) &
(∀ X. equal(multiply(additive-identity::'a,X),additive-identity)) &
(∀ X. equal(multiply(X::'a,additive-identity),additive-identity)) &
(∀ X. equal(add(additive-inverse(X),X),additive-identity)) &
(∀ X Y. equal(additive-inverse(add(X::'a,Y)),add(additive-inverse(X),additive-inverse(Y))))

&
  (∀ X. equal(additive-inverse(additive-inverse(X)),X)) &
 (∀ Y X Z. equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))
&
 (∀ X Y Z. equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))
&
 (∀ X Y. equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))
&
 (∀ X Y. equal(multiply(multiply(X::'a,X),Y),multiply(X::'a,multiply(X::'a,Y))))
&
 (∀ X Y. equal(multiply(additive-inverse(X),Y),additive-inverse(multiply(X::'a,Y))))
&
 (∀ X Y. equal(multiply(X::'a,additive-inverse(Y)),additive-inverse(multiply(X::'a,Y))))
&
  (equal(additive-inverse(additive-identity),additive-identity)) &
  (∀ Y X. equal(add(X::'a,Y),add(Y::'a,X))) &
  (∀ X Y Z. equal(add(X::'a,add(Y::'a,Z)),add(add(X::'a,Y),Z))) &
  (∀ Z X Y. equal(add(X::'a,Z),add(Y::'a,Z)) --> equal(X::'a,Y)) &
  (∀ Z X Y. equal(add(Z::'a,X),add(Z::'a,Y)) --> equal(X::'a,Y)) &
  (∀ D E F'. equal(D::'a,E) --> equal(add(D::'a,F'),add(E::'a,F'))) &
  (∀ G I' H. equal(G::'a,H) --> equal(add(I'::'a,G),add(I'::'a,H))) &
  (∀ J K'. equal(J::'a,K') --> equal(additive-inverse(J),additive-inverse(K'))) &
 (∀ D1 E1 F1. equal(D1::'a,E1) --> equal(multiply(D1::'a,F1),multiply(E1::'a,F1)))
&
 (∀ G1 I1 H1. equal(G1::'a,H1) --> equal(multiply(I1::'a,G1),multiply(I1::'a,H1)))
&
 (∀ X Y Z. equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive-inverse(multiply(X::'a,m
&
 (∀ L M N O'. equal(L::'a,M) --> equal(associator(L::'a,N,O'),associator(M::'a,N,O')))
&
 (∀ P R Q S'. equal(P::'a,Q) --> equal(associator(R::'a,P,S'),associator(R::'a,Q,S')))
&
 (∀ T' V W U. equal(T'::'a,U) --> equal(associator(V::'a,W,T'),associator(V::'a,W,U)))
&
 (∀ X Y. ~equal(multiply(multiply(Y::'a,X),Y),multiply(Y::'a,multiply(X::'a,Y))))
&
 (∀ X Y Z. ~equal(associator(Y::'a,X,Z),additive-inverse(associator(X::'a,Y,Z))))
&
 (∀ X Y Z. ~equal(associator(Z::'a,Y,X),additive-inverse(associator(X::'a,Y,Z))))
&
 (~equal(multiply(multiply(cx::'a,multiply(cy::'a,cx)),cz),multiply(cx::'a,multiply(cy::'a,multiply(cx::'a,cz))))
--> False
  by *meson*

lemma *RNG038-2*:
  (∀ X. sum(X::'a,additive-identity,X)) &
  (∀ X Y. product(X::'a,Y,multiply(X::'a,Y))) &
  (∀ X Y. sum(X::'a,Y,add(X::'a,Y))) &

RNG-other-ax multiply add equal product additive-identity additive-inverse sum
&
(∀ *X*. *product*(*additive-identity*::'*a*,*X*,*additive-identity*)) &
(∀ *X*. *product*(*X*::'*a*,*additive-identity*,*additive-identity*)) &
(∀ *X Y*. *equal*(*X*::'*a*,*additive-identity*) −−> *product*(*X*::'*a*,*h*(*X*::'*a*,*Y*),*Y*)) &
(*product*(*a*::'*a*,*b*,*additive-identity*)) &
(~*equal*(*a*::'*a*,*additive-identity*)) &
(~*equal*(*b*::'*a*,*additive-identity*)) −−> *False*
by *meson*

lemma *RNG040-2*:
EQU001-0-ax equal &
RNG001-0-eq product multiply sum add additive-inverse equal &
(∀ *X*. *sum*(*additive-identity*::'*a*,*X*,*X*)) &
(∀ *X*. *sum*(*X*::'*a*,*additive-identity*,*X*)) &
(∀ *X Y*. *product*(*X*::'*a*,*Y*,*multiply*(*X*::'*a*,*Y*))) &
(∀ *X Y*. *sum*(*X*::'*a*,*Y*,*add*(*X*::'*a*,*Y*))) &
(∀ *X*. *sum*(*additive-inverse*(*X*),*X*,*additive-identity*)) &
(∀ *X*. *sum*(*X*::'*a*,*additive-inverse*(*X*),*additive-identity*)) &
(∀ *Y U Z X V W*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*Y*::'*a*,*Z*,*V*) & *sum*(*U*::'*a*,*Z*,*W*) −−>
sum(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*Y*::'*a*,*Z*,*V*) & *sum*(*X*::'*a*,*V*,*W*) −−>
sum(*U*::'*a*,*Z*,*W*)) &
(∀ *Y X Z*. *sum*(*X*::'*a*,*Y*,*Z*) −−> *sum*(*Y*::'*a*,*X*,*Z*)) &
(∀ *Y U Z X V W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*U*::'*a*,*Z*,*W*)
−−> *product*(*X*::'*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*Y*::'*a*,*Z*,*V*) & *product*(*X*::'*a*,*V*,*W*)
−−> *product*(*U*::'*a*,*Z*,*W*)) &
(∀ *Y Z X V3 V1 V2 V4*. *product*(*X*::'*a*,*Y*,*V1*) & *product*(*X*::'*a*,*Z*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)
& *product*(*X*::'*a*,*V3*,*V4*) −−> *sum*(*V1*::'*a*,*V2*,*V4*)) &
(∀ *Y Z V1 V2 X V3 V4*. *product*(*X*::'*a*,*Y*,*V1*) & *product*(*X*::'*a*,*Z*,*V2*) & *sum*(*Y*::'*a*,*Z*,*V3*)
& *sum*(*V1*::'*a*,*V2*,*V4*) −−> *product*(*X*::'*a*,*V3*,*V4*)) &
(∀ *X Y U V*. *sum*(*X*::'*a*,*Y*,*U*) & *sum*(*X*::'*a*,*Y*,*V*) −−> *equal*(*U*::'*a*,*V*)) &
(∀ *X Y U V*. *product*(*X*::'*a*,*Y*,*U*) & *product*(*X*::'*a*,*Y*,*V*) −−> *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 &
(∀ 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,b)),c)) &
(~equal(negate(add(c::'a,negate(add(negate(b),a)))),a)) --> False
by *meson*

lemma *ROB016-1*:
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))) &
(∀ J K' L. equal(J::'a,K') --> equal(multiply(J::'a,L),multiply(K'::'a,L))) &
(∀ M O' N. equal(M::'a,N) --> equal(multiply(O'::'a,M),multiply(O'::'a,N)))
&
(∀ P Q. equal(P::'a,Q) --> equal(successor(P),successor(Q))) &
(∀ R S'. equal(R::'a,S') & positive-integer(R) --> positive-integer(S')) &
(∀ X. equal(multiply(One::'a,X),X)) &
(∀ V X. positive-integer(X) --> equal(multiply(successor(V),X),add(X::'a,multiply(V::'a,X))))
&
(positive-integer(One)) &
(∀ X. positive-integer(X) --> positive-integer(successor(X))) &
(equal(negate(add(d::'a,e)),negate(e))) &
(positive-integer(k)) &
(∀ Vk X Y. equal(negate(add(negate(Y),negate(add(X::'a,negate(Y))))),X) &
positive-integer(Vk) --> equal(negate(add(Y::'a,multiply(Vk::'a,add(X::'a,negate(add(X::'a,negate(Y))))))
&
(~equal(negate(add(e::'a,multiply(k::'a,add(d::'a,negate(add(d::'a,negate(e))))))),negate(e)))
--> False
oops

lemma *ROB021-1*:
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))) &
(∀ 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))
&

($\forall$ *Subset Superset. member(member-of-1-not-of-2(Subset::'a,Superset),Superset) --> subset(Subset::'a,Superset))* &
($\forall$ *Subset Superset. equal-sets(Subset::'a,Superset) --> subset(Subset::'a,Superset))* &
($\forall$ *Subset Superset. equal-sets(Superset::'a,Subset) --> subset(Subset::'a,Superset))* &
($\forall$ *Set2 Set1. subset(Set1::'a,Set2)* & *subset(Set2::'a,Set1) --> equal-sets(Set2::'a,Set1))* &
($\forall$ *Set2 Intersection Element Set1. intersection(Set1::'a,Set2,Intersection)* & *member(Element::'a,Intersection) --> member(Element::'a,Set1))* &
($\forall$ *Set1 Intersection Element Set2. intersection(Set1::'a,Set2,Intersection)* & *member(Element::'a,Intersection) --> member(Element::'a,Set2))* &
($\forall$ *Set2 Set1 Element Intersection. intersection(Set1::'a,Set2,Intersection)* & *member(Element::'a,Set2)* & *member(Element::'a,Set1) --> member(Element::'a,Intersection))* &
($\forall$ *Set2 Intersection Set1. member(h(Set1::'a,Set2,Intersection),Intersection) | intersection(Set1::'a,Set2,Intersection) | member(h(Set1::'a,Set2,Intersection),Set1))* &
($\forall$ *Set1 Intersection Set2. member(h(Set1::'a,Set2,Intersection),Intersection) | intersection(Set1::'a,Set2,Intersection) | member(h(Set1::'a,Set2,Intersection),Set2))* &
($\forall$ *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*:
($\forall$ *Subset Element Superset. member(Element::'a,Subset)* & *ssubset(Subset::'a,Superset) --> member(Element::'a,Superset))* &
($\forall$ *Superset Subset. ssubset(Subset::'a,Superset) | member(member-of-1-not-of-2(Subset::'a,Superset),Subset))* &
($\forall$ *Subset Superset. member(member-of-1-not-of-2(Subset::'a,Superset),Superset) --> ssubset(Subset::'a,Superset))* &
($\forall$ *Subset Superset. equal-sets(Subset::'a,Superset) --> ssubset(Subset::'a,Superset))* &
($\forall$ *Subset Superset. equal-sets(Superset::'a,Subset) --> ssubset(Subset::'a,Superset))* &
($\forall$ *Set2 Set1. ssubset(Set1::'a,Set2)* & *ssubset(Set2::'a,Set1) --> equal-sets(Set2::'a,Set1))* &
($\forall$ *Set2 Difference Element Set1. difference(Set1::'a,Set2,Difference)* & *member(Element::'a,Difference) --> member(Element::'a,Set1))* &
($\forall$ *Element A-set Set1 Set2. ~(member(Element::'a,Set1)* & *member(Element::'a,Set2)* & *difference(A-set::'a,Set1,Set2)))* &

(∀ Set1 Difference Element Set2. member(Element::'a,Set1) & difference(Set1::'a,Set2,Difference) --> member(Element::'a,Difference) | member(Element::'a,Set2)) &
(∀ Set1 Set2 Difference. difference(Set1::'a,Set2,Difference) | member(k(Set1::'a,Set2,Difference),Set1) | member(k(Set1::'a,Set2,Difference),Difference)) &
(∀ Set1 Set2 Difference. member(k(Set1::'a,Set2,Difference),Set2) --> member(k(Set1::'a,Set2,Difference),Difference) | difference(Set1::'a,Set2,Difference)) &
(∀ Set1 Set2 Difference. member(k(Set1::'a,Set2,Difference),Difference) & member(k(Set1::'a,Set2,Difference),Set1) --> member(k(Set1::'a,Set2,Difference),Set2) | difference(Set1::'a,Set2,Difference)) &
(ssubset(d::'a,a)) &
(difference(b::'a,a,bDa)) &
(difference(b::'a,d,bDd)) &
(~ssubset(bDa::'a,bDd)) --> False
by *meson*

lemma *SET025-4*:
EQU001-0-ax equal &
(∀ Y X. member(X::'a,Y) --> little-set(X)) &
(∀ X Y. little-set(f1(X::'a,Y)) | equal(X::'a,Y)) &
(∀ X Y. member(f1(X::'a,Y),X) | member(f1(X::'a,Y),Y) | equal(X::'a,Y)) &
(∀ X Y. member(f1(X::'a,Y),X) & member(f1(X::'a,Y),Y) --> equal(X::'a,Y)) &
(∀ X U Y. member(U::'a,non-ordered-pair(X::'a,Y)) --> equal(U::'a,X) | equal(U::'a,Y)) &
(∀ Y U X. little-set(U) & equal(U::'a,X) --> member(U::'a,non-ordered-pair(X::'a,Y))) &
(∀ X U Y. little-set(U) & equal(U::'a,Y) --> member(U::'a,non-ordered-pair(X::'a,Y))) &
(∀ X Y. little-set(non-ordered-pair(X::'a,Y))) &
(∀ X. equal(singleton-set(X),non-ordered-pair(X::'a,X))) &
(∀ X Y. equal(ordered-pair(X::'a,Y),non-ordered-pair(singleton-set(X),non-ordered-pair(X::'a,Y)))) &
(∀ X. ordered-pair-predicate(X) --> little-set(f2(X))) &
(∀ X. ordered-pair-predicate(X) --> little-set(f3(X))) &
(∀ X. ordered-pair-predicate(X) --> equal(X::'a,ordered-pair(f2(X),f3(X)))) &
(∀ X Y Z. little-set(Y) & little-set(Z) & equal(X::'a,ordered-pair(Y::'a,Z)) --> ordered-pair-predicate(X)) &
(∀ Z X. member(Z::'a,first(X)) --> little-set(f4(Z::'a,X))) &
(∀ Z X. member(Z::'a,first(X)) --> little-set(f5(Z::'a,X))) &
(∀ Z X. member(Z::'a,first(X)) --> equal(X::'a,ordered-pair(f4(Z::'a,X),f5(Z::'a,X)))) &
(∀ Z X. member(Z::'a,first(X)) --> member(Z::'a,f4(Z::'a,X))) &
(∀ X V Z U. little-set(U) & little-set(V) & equal(X::'a,ordered-pair(U::'a,V)) & member(Z::'a,U) --> member(Z::'a,first(X))) &
(∀ Z X. member(Z::'a,second(X)) --> little-set(f6(Z::'a,X))) &
(∀ Z X. member(Z::'a,second(X)) --> little-set(f7(Z::'a,X))) &
(∀ Z X. member(Z::'a,second(X)) --> equal(X::'a,ordered-pair(f6(Z::'a,X),f7(Z::'a,X)))) &

$(\forall Z\ X.\ member(Z::'a,second(X)) \text{ --> } member(Z::'a,f7(Z::'a,X)))$ &
$(\forall X\ U\ Z\ V.\ little\text{-}set(U)$ & $little\text{-}set(V)$ & $equal(X::'a,ordered\text{-}pair(U::'a,V))$
& $member(Z::'a,V) \text{ --> } member(Z::'a,second(X)))$ &
$(\forall Z.\ member(Z::'a,estin) \text{ --> } ordered\text{-}pair\text{-}predicate(Z))$ &
$(\forall Z.\ member(Z::'a,estin) \text{ --> } member(first(Z),second(Z)))$ &
$(\forall Z.\ little\text{-}set(Z)$ & $ordered\text{-}pair\text{-}predicate(Z)$ & $member(first(Z),second(Z))$
$\text{--> } member(Z::'a,estin))$ &
$(\forall Y\ Z\ X.\ member(Z::'a,intersection(X::'a,Y)) \text{ --> } member(Z::'a,X))$ &
$(\forall X\ Z\ Y.\ member(Z::'a,intersection(X::'a,Y)) \text{ --> } member(Z::'a,Y))$ &
$(\forall X\ Z\ Y.\ member(Z::'a,X)$ & $member(Z::'a,Y) \text{ --> } member(Z::'a,intersection(X::'a,Y)))$
&
$(\forall Z\ X.\ \sim(member(Z::'a,complement(X))$ & $member(Z::'a,X)))$ &
$(\forall Z\ X.\ little\text{-}set(Z) \text{ --> } member(Z::'a,complement(X)) \mid member(Z::'a,X))$ &
$(\forall X\ Y.\ equal(union(X::'a,Y),complement(intersection(complement(X),complement(Y)))))$
&
$(\forall Z\ X.\ member(Z::'a,domain\text{-}of(X)) \text{ --> } ordered\text{-}pair\text{-}predicate(f8(Z::'a,X)))$
&
$(\forall Z\ X.\ member(Z::'a,domain\text{-}of(X)) \text{ --> } member(f8(Z::'a,X),X))$ &
$(\forall Z\ X.\ member(Z::'a,domain\text{-}of(X)) \text{ --> } equal(Z::'a,first(f8(Z::'a,X))))$ &
$(\forall X\ Z\ Xp.\ little\text{-}set(Z)$ & $ordered\text{-}pair\text{-}predicate(Xp)$ & $member(Xp::'a,X)$ &
$equal(Z::'a,first(Xp)) \text{ --> } member(Z::'a,domain\text{-}of(X)))$ &
$(\forall X\ Y\ Z.\ member(Z::'a,cross\text{-}product(X::'a,Y)) \text{ --> } ordered\text{-}pair\text{-}predicate(Z))$
&
$(\forall Y\ Z\ X.\ member(Z::'a,cross\text{-}product(X::'a,Y)) \text{ --> } member(first(Z),X))$ &
$(\forall X\ Z\ Y.\ member(Z::'a,cross\text{-}product(X::'a,Y)) \text{ --> } member(second(Z),Y))$
&
$(\forall X\ Z\ Y.\ little\text{-}set(Z)$ & $ordered\text{-}pair\text{-}predicate(Z)$ & $member(first(Z),X)$ &
$member(second(Z),Y) \text{ --> } member(Z::'a,cross\text{-}product(X::'a,Y)))$ &
$(\forall X\ Z.\ member(Z::'a,inv1\ X) \text{ --> } ordered\text{-}pair\text{-}predicate(Z))$ &
$(\forall Z\ X.\ member(Z::'a,inv1\ X) \text{ --> } member(ordered\text{-}pair(second(Z),first(Z)),X))$
&
$(\forall Z\ X.\ little\text{-}set(Z)$ & $ordered\text{-}pair\text{-}predicate(Z)$ & $member(ordered\text{-}pair(second(Z),first(Z)),X)$
$\text{--> } member(Z::'a,inv1\ X))$ &
$(\forall Z\ X.\ member(Z::'a,rot\text{-}right(X)) \text{ --> } little\text{-}set(f9(Z::'a,X)))$ &
$(\forall Z\ X.\ member(Z::'a,rot\text{-}right(X)) \text{ --> } little\text{-}set(f10(Z::'a,X)))$ &
$(\forall Z\ X.\ member(Z::'a,rot\text{-}right(X)) \text{ --> } little\text{-}set(f11(Z::'a,X)))$ &
$(\forall Z\ X.\ member(Z::'a,rot\text{-}right(X)) \text{ --> } equal(Z::'a,ordered\text{-}pair(f9(Z::'a,X),ordered\text{-}pair(f10(Z::'a,X),f11$
&
$(\forall Z\ X.\ member(Z::'a,rot\text{-}right(X)) \text{ --> } member(ordered\text{-}pair(f10(Z::'a,X),ordered\text{-}pair(f11(Z::'a,X),f9(Z$
&
$(\forall Z\ V\ W\ U\ X.\ little\text{-}set(Z)$ & $little\text{-}set(U)$ & $little\text{-}set(V)$ & $little\text{-}set(W)$ &
$equal(Z::'a,ordered\text{-}pair(U::'a,ordered\text{-}pair(V::'a,W)))$ & $member(ordered\text{-}pair(V::'a,ordered\text{-}pair(W::'a,U))$
$\text{--> } member(Z::'a,rot\text{-}right(X)))$ &
$(\forall Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X)) \text{ --> } little\text{-}set(f12(Z::'a,X)))$ &
$(\forall Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X)) \text{ --> } little\text{-}set(f13(Z::'a,X)))$ &
$(\forall Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X)) \text{ --> } little\text{-}set(f14(Z::'a,X)))$ &
$(\forall Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X)) \text{ --> } equal(Z::'a,ordered\text{-}pair(f12(Z::'a,X),ordered\text{-}pair(f13(Z::'a,X$
&
$(\forall Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X)) \text{ --> } member(ordered\text{-}pair(f12(Z::'a,X),ordered\text{-}pair(f14(Z::'a,X),$

&
(∀ Z U W V X. little-set(Z) & little-set(U) & little-set(V) & little-set(W) &
equal(Z::'a,ordered-pair(U::'a,ordered-pair(V::'a,W))) & member(ordered-pair(U::'a,ordered-pair(W::'a,V)),X)
--> member(Z::'a,flip-range-of(X))) &
(∀ X. equal(successor(X),union(X::'a,singleton-set(X)))) &
(∀ Z. ~member(Z::'a,empty-set)) &
(∀ Z. little-set(Z) --> member(Z::'a,universal-set)) &
(little-set(infinity)) &
(member(empty-set::'a,infinity)) &
(∀ X. member(X::'a,infinity) --> member(successor(X),infinity)) &
(∀ Z X. member(Z::'a,sigma(X)) --> member(f16(Z::'a,X),X)) &
(∀ Z X. member(Z::'a,sigma(X)) --> member(Z::'a,f16(Z::'a,X))) &
(∀ X Z Y. member(Y::'a,X) & member(Z::'a,Y) --> member(Z::'a,sigma(X)))
&
(∀ U. little-set(U) --> little-set(sigma(U))) &
(∀ X U Y. ssubset(X::'a,Y) & member(U::'a,X) --> member(U::'a,Y)) &
(∀ Y X. ssubset(X::'a,Y) | member(f17(X::'a,Y),X)) &
(∀ X Y. member(f17(X::'a,Y),Y) --> ssubset(X::'a,Y)) &
(∀ X Y. proper-subset(X::'a,Y) --> ssubset(X::'a,Y)) &
(∀ X Y. ~(proper-subset(X::'a,Y) & equal(X::'a,Y))) &
(∀ X Y. ssubset(X::'a,Y) --> proper-subset(X::'a,Y) | equal(X::'a,Y)) &
(∀ Z X. member(Z::'a,powerset(X)) --> ssubset(Z::'a,X)) &
(∀ Z X. little-set(Z) & ssubset(Z::'a,X) --> member(Z::'a,powerset(X))) &
(∀ U. little-set(U) --> little-set(powerset(U))) &
(∀ Z X. relation(Z) & member(X::'a,Z) --> ordered-pair-predicate(X)) &
(∀ Z. relation(Z) | member(f18(Z),Z)) &
(∀ Z. ordered-pair-predicate(f18(Z)) --> relation(Z)) &
(∀ U X V W. single-valued-set(X) & little-set(U) & little-set(V) & little-set(W)
& member(ordered-pair(U::'a,V),X) & member(ordered-pair(U::'a,W),X) -->
equal(V::'a,W)) &
(∀ X. single-valued-set(X) | little-set(f19(X))) &
(∀ X. single-valued-set(X) | little-set(f20(X))) &
(∀ X. single-valued-set(X) | little-set(f21(X))) &
(∀ X. single-valued-set(X) | member(ordered-pair(f19(X),f20(X)),X)) &
(∀ X. single-valued-set(X) | member(ordered-pair(f19(X),f21(X)),X)) &
(∀ X. equal(f20(X),f21(X)) --> single-valued-set(X)) &
(∀ Xf. function(Xf) --> relation(Xf)) &
(∀ Xf. function(Xf) --> single-valued-set(Xf)) &
(∀ Xf. relation(Xf) & single-valued-set(Xf) --> function(Xf)) &
(∀ Z X Xf. member(Z::'a,image'(X::'a,Xf)) --> ordered-pair-predicate(f22(Z::'a,X,Xf)))
&
(∀ Z X Xf. member(Z::'a,image'(X::'a,Xf)) --> member(f22(Z::'a,X,Xf),Xf))
&
(∀ Z Xf X. member(Z::'a,image'(X::'a,Xf)) --> member(first(f22(Z::'a,X,Xf)),X))
&
(∀ X Xf Z. member(Z::'a,image'(X::'a,Xf)) --> equal(second(f22(Z::'a,X,Xf)),Z))
&
(∀ Xf X Y Z. little-set(Z) & ordered-pair-predicate(Y) & member(Y::'a,Xf) &
member(first(Y),X) & equal(second(Y),Z) --> member(Z::'a,image'(X::'a,Xf)))

&
 $(\forall X\ Xf.\ \textit{little-set}(X)\ \&\ \textit{function}(Xf)\ {-}{-}{>}\ \textit{little-set}(\textit{image}'(X{::}'a,Xf)))\ \&$
 $(\forall X\ U\ Y.\ {\sim}(\textit{disjoint}(X{::}'a,Y)\ \&\ \textit{member}(U{::}'a,X)\ \&\ \textit{member}(U{::}'a,Y)))\ \&$
 $(\forall Y\ X.\ \textit{disjoint}(X{::}'a,Y)\ |\ \textit{member}(\textit{f23}(X{::}'a,Y),X))\ \&$
 $(\forall X\ Y.\ \textit{disjoint}(X{::}'a,Y)\ |\ \textit{member}(\textit{f23}(X{::}'a,Y),Y))\ \&$
 $(\forall X.\ \textit{equal}(X{::}'a,\textit{empty-set})\ |\ \textit{member}(\textit{f24}(X),X))\ \&$
 $(\forall X.\ \textit{equal}(X{::}'a,\textit{empty-set})\ |\ \textit{disjoint}(\textit{f24}(X),X))\ \&$
 $(\textit{function}(\textit{f25}))\ \&$
 $(\forall X.\ \textit{little-set}(X)\ {-}{-}{>}\ \textit{equal}(X{::}'a,\textit{empty-set})\ |\ \textit{member}(\textit{f26}(X),X))\ \&$
 $(\forall X.\ \textit{little-set}(X)\ {-}{-}{>}\ \textit{equal}(X{::}'a,\textit{empty-set})\ |\ \textit{member}(\textit{ordered-pair}(X{::}'a,\textit{f26}(X)),\textit{f25}))$
&
 $(\forall Z\ X.\ \textit{member}(Z{::}'a,\textit{range-of}(X))\ {-}{-}{>}\ \textit{ordered-pair-predicate}(\textit{f27}(Z{::}'a,X)))$
&
 $(\forall Z\ X.\ \textit{member}(Z{::}'a,\textit{range-of}(X))\ {-}{-}{>}\ \textit{member}(\textit{f27}(Z{::}'a,X),X))\ \&$
 $(\forall Z\ X.\ \textit{member}(Z{::}'a,\textit{range-of}(X))\ {-}{-}{>}\ \textit{equal}(Z{::}'a,\textit{second}(\textit{f27}(Z{::}'a,X))))\ \&$
 $(\forall X\ Z\ Xp.\ \textit{little-set}(Z)\ \&\ \textit{ordered-pair-predicate}(Xp)\ \&\ \textit{member}(Xp{::}'a,X)\ \&$
$\textit{equal}(Z{::}'a,\textit{second}(Xp))\ {-}{-}{>}\ \textit{member}(Z{::}'a,\textit{range-of}(X)))\ \&$
 $(\forall Z.\ \textit{member}(Z{::}'a,\textit{identity-relation})\ {-}{-}{>}\ \textit{ordered-pair-predicate}(Z))\ \&$
 $(\forall Z.\ \textit{member}(Z{::}'a,\textit{identity-relation})\ {-}{-}{>}\ \textit{equal}(\textit{first}(Z),\textit{second}(Z)))\ \&$
 $(\forall Z.\ \textit{little-set}(Z)\ \&\ \textit{ordered-pair-predicate}(Z)\ \&\ \textit{equal}(\textit{first}(Z),\textit{second}(Z))\ {-}{-}{>}$
$\textit{member}(Z{::}'a,\textit{identity-relation}))\ \&$
 $(\forall X\ Y.\ \textit{equal}(\textit{restrct}(X{::}'a,Y),\textit{intersection}(X{::}'a,\textit{cross-product}(Y{::}'a,\textit{universal-set}))))$
&
 $(\forall Xf.\ \textit{one-to-one-function}(Xf)\ {-}{-}{>}\ \textit{function}(Xf))\ \&$
 $(\forall Xf.\ \textit{one-to-one-function}(Xf)\ {-}{-}{>}\ \textit{function}(\textit{inv1}\ Xf))\ \&$
 $(\forall Xf.\ \textit{function}(Xf)\ \&\ \textit{function}(\textit{inv1}\ Xf)\ {-}{-}{>}\ \textit{one-to-one-function}(Xf))\ \&$
 $(\forall Z\ Xf\ Y.\ \textit{member}(Z{::}'a,\textit{apply}(Xf{::}'a,Y))\ {-}{-}{>}\ \textit{ordered-pair-predicate}(\textit{f28}(Z{::}'a,Xf,Y)))$
&
 $(\forall Z\ Y\ Xf.\ \textit{member}(Z{::}'a,\textit{apply}(Xf{::}'a,Y))\ {-}{-}{>}\ \textit{member}(\textit{f28}(Z{::}'a,Xf,Y),Xf))$
&
 $(\forall Z\ Xf\ Y.\ \textit{member}(Z{::}'a,\textit{apply}(Xf{::}'a,Y))\ {-}{-}{>}\ \textit{equal}(\textit{first}(\textit{f28}(Z{::}'a,Xf,Y)),Y))$
&
 $(\forall Z\ Xf\ Y.\ \textit{member}(Z{::}'a,\textit{apply}(Xf{::}'a,Y))\ {-}{-}{>}\ \textit{member}(Z{::}'a,\textit{second}(\textit{f28}(Z{::}'a,Xf,Y))))$
&
 $(\forall Xf\ Y\ Z\ W.\ \textit{ordered-pair-predicate}(W)\ \&\ \textit{member}(W{::}'a,Xf)\ \&\ \textit{equal}(\textit{first}(W),Y)$
$\&\ \textit{member}(Z{::}'a,\textit{second}(W))\ {-}{-}{>}\ \textit{member}(Z{::}'a,\textit{apply}(Xf{::}'a,Y)))\ \&$
 $(\forall Xf\ X\ Y.\ \textit{equal}(\textit{apply-to-two-arguments}(Xf{::}'a,X,Y),\textit{apply}(Xf{::}'a,\textit{ordered-pair}(X{::}'a,Y))))$
&
 $(\forall X\ Y\ Xf.\ \textit{maps}(Xf{::}'a,X,Y)\ {-}{-}{>}\ \textit{function}(Xf))\ \&$
 $(\forall Y\ Xf\ X.\ \textit{maps}(Xf{::}'a,X,Y)\ {-}{-}{>}\ \textit{equal}(\textit{domain-of}(Xf),X))\ \&$
 $(\forall X\ Xf\ Y.\ \textit{maps}(Xf{::}'a,X,Y)\ {-}{-}{>}\ \textit{ssubset}(\textit{range-of}(Xf),Y))\ \&$
 $(\forall X\ Xf\ Y.\ \textit{function}(Xf)\ \&\ \textit{equal}(\textit{domain-of}(Xf),X)\ \&\ \textit{ssubset}(\textit{range-of}(Xf),Y)$
${-}{-}{>}\ \textit{maps}(Xf{::}'a,X,Y))\ \&$
 $(\forall Xf\ Xs.\ \textit{closed}(Xs{::}'a,Xf)\ {-}{-}{>}\ \textit{little-set}(Xs))\ \&$
 $(\forall Xs\ Xf.\ \textit{closed}(Xs{::}'a,Xf)\ {-}{-}{>}\ \textit{little-set}(Xf))\ \&$
 $(\forall Xf\ Xs.\ \textit{closed}(Xs{::}'a,Xf)\ {-}{-}{>}\ \textit{maps}(Xf{::}'a,\textit{cross-product}(Xs{::}'a,Xs),Xs))\ \&$
 $(\forall Xf\ Xs.\ \textit{little-set}(Xs)\ \&\ \textit{little-set}(Xf)\ \&\ \textit{maps}(Xf{::}'a,\textit{cross-product}(Xs{::}'a,Xs),Xs)$
${-}{-}{>}\ \textit{closed}(Xs{::}'a,Xf))\ \&$
 $(\forall Z\ Xf\ Xg.\ \textit{member}(Z{::}'a,\textit{composition}(Xf{::}'a,Xg))\ {-}{-}{>}\ \textit{little-set}(\textit{f29}(Z{::}'a,Xf,Xg)))$

&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> little-set(f30(Z::'a,Xf,Xg)))
&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> little-set(f31(Z::'a,Xf,Xg)))
&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> equal(Z::'a,ordered-pair(f29(Z::'a,Xf,Xg),f30(Z::'a
&
(∀ Z Xg Xf. member(Z::'a,composition(Xf::'a,Xg)) --> member(ordered-pair(f29(Z::'a,Xf,Xg),f31(Z::'a,X
&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> member(ordered-pair(f31(Z::'a,Xf,Xg),f30(Z::'a,X
&
(∀ Z X Xf W Y Xg. little-set(Z) & little-set(X) & little-set(Y) & little-set(W) & equal(Z::'a,ordered-pair(X::'a,Y)) & member(ordered-pair(X::'a,W),Xf) & member(ordered-pair(W::'a,Y),Xg) --> member(Z::'a,composition(Xf::'a,Xg))) &
(∀ Xh Xs2 Xf2 Xs1 Xf1. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) --> closed(Xs1::'a,Xf1))
&
(∀ Xh Xs1 Xf1 Xs2 Xf2. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) --> closed(Xs2::'a,Xf2))
&
(∀ Xf1 Xf2 Xh Xs1 Xs2. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) --> maps(Xh::'a,Xs1,Xs2))
&
(∀ Xs2 Xs1 Xf1 Xf2 X Xh Y. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) & member(X::'a,Xs1) & member(Y::'a,Xs1) --> equal(apply(Xh::'a,apply-to-two-arguments(Xf1::'a,X,Y)),apply-t
&
(∀ Xh Xf1 Xs2 Xf2 Xs1. closed(Xs1::'a,Xf1) & closed(Xs2::'a,Xf2) & maps(Xh::'a,Xs1,Xs2) --> homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) | member(f32(Xh::'a,Xs1,Xf1,Xs2,Xf2),Xs1))
&
(∀ Xh Xf1 Xs2 Xf2 Xs1. closed(Xs1::'a,Xf1) & closed(Xs2::'a,Xf2) & maps(Xh::'a,Xs1,Xs2) --> homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) | member(f33(Xh::'a,Xs1,Xf1,Xs2,Xf2),Xs1))
&
(∀ Xh Xs1 Xf1 Xs2 Xf2. closed(Xs1::'a,Xf1) & closed(Xs2::'a,Xf2) & maps(Xh::'a,Xs1,Xs2) & equal(apply(Xh::'a,apply-to-two-arguments(Xf1::'a,f32(Xh::'a,Xs1,Xf1,Xs2,Xf2),f33(Xh::'a,Xs1,Xf1,Xs2,
--> homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2)) &
(∀ A B C. equal(A::'a,B) --> equal(f1(A::'a,C),f1(B::'a,C))) &
(∀ D F' E. equal(D::'a,E) --> equal(f1(F'::'a,D),f1(F'::'a,E))) &
(∀ A2 B2. equal(A2::'a,B2) --> equal(f2(A2),f2(B2))) &
(∀ G4 H4. equal(G4::'a,H4) --> equal(f3(G4),f3(H4))) &
(∀ O7 P7 Q7. equal(O7::'a,P7) --> equal(f4(O7::'a,Q7),f4(P7::'a,Q7))) &
(∀ R7 T7 S7. equal(R7::'a,S7) --> equal(f4(T7::'a,R7),f4(T7::'a,S7))) &
(∀ U7 V7 W7. equal(U7::'a,V7) --> equal(f5(U7::'a,W7),f5(V7::'a,W7))) &
(∀ X7 Z7 Y7. equal(X7::'a,Y7) --> equal(f5(Z7::'a,X7),f5(Z7::'a,Y7))) &
(∀ A8 B8 C8. equal(A8::'a,B8) --> equal(f6(A8::'a,C8),f6(B8::'a,C8))) &
(∀ D8 F8 E8. equal(D8::'a,E8) --> equal(f6(F8::'a,D8),f6(F8::'a,E8))) &
(∀ G8 H8 I8. equal(G8::'a,H8) --> equal(f7(G8::'a,I8),f7(H8::'a,I8))) &
(∀ J8 L8 K8. equal(J8::'a,K8) --> equal(f7(L8::'a,J8),f7(L8::'a,K8))) &
(∀ M8 N8 O8. equal(M8::'a,N8) --> equal(f8(M8::'a,O8),f8(N8::'a,O8))) &
(∀ P8 R8 Q8. equal(P8::'a,Q8) --> equal(f8(R8::'a,P8),f8(R8::'a,Q8))) &
(∀ S8 T8 U8. equal(S8::'a,T8) --> equal(f9(S8::'a,U8),f9(T8::'a,U8))) &
(∀ V8 X8 W8. equal(V8::'a,W8) --> equal(f9(X8::'a,V8),f9(X8::'a,W8))) &
(∀ G H I'. equal(G::'a,H) --> equal(f10(G::'a,I'),f10(H::'a,I'))) &

$(\forall J\ L\ K'.\ equal(J::'a,K')\ -->\ equal(f10(L::'a,J),f10(L::'a,K')))\ \&$
$(\forall M\ N\ O'.\ equal(M::'a,N)\ -->\ equal(f11(M::'a,O'),f11(N::'a,O')))\ \&$
$(\forall P\ R\ Q.\ equal(P::'a,Q)\ -->\ equal(f11(R::'a,P),f11(R::'a,Q)))\ \&$
$(\forall S'\ T'\ U.\ equal(S'::'a,T')\ -->\ equal(f12(S'::'a,U),f12(T'::'a,U)))\ \&$
$(\forall V\ X\ W.\ equal(V::'a,W)\ -->\ equal(f12(X::'a,V),f12(X::'a,W)))\ \&$
$(\forall Y\ Z\ A1.\ equal(Y::'a,Z)\ -->\ equal(f13(Y::'a,A1),f13(Z::'a,A1)))\ \&$
$(\forall B1\ D1\ C1.\ equal(B1::'a,C1)\ -->\ equal(f13(D1::'a,B1),f13(D1::'a,C1)))\ \&$
$(\forall E1\ F1\ G1.\ equal(E1::'a,F1)\ -->\ equal(f14(E1::'a,G1),f14(F1::'a,G1)))\ \&$
$(\forall H1\ J1\ I1.\ equal(H1::'a,I1)\ -->\ equal(f14(J1::'a,H1),f14(J1::'a,I1)))\ \&$
$(\forall K1\ L1\ M1.\ equal(K1::'a,L1)\ -->\ equal(f16(K1::'a,M1),f16(L1::'a,M1)))\ \&$
$(\forall N1\ P1\ O1.\ equal(N1::'a,O1)\ -->\ equal(f16(P1::'a,N1),f16(P1::'a,O1)))\ \&$
$(\forall Q1\ R1\ S1.\ equal(Q1::'a,R1)\ -->\ equal(f17(Q1::'a,S1),f17(R1::'a,S1)))\ \&$
$(\forall T1\ V1\ U1.\ equal(T1::'a,U1)\ -->\ equal(f17(V1::'a,T1),f17(V1::'a,U1)))\ \&$
$(\forall W1\ X1.\ equal(W1::'a,X1)\ -->\ equal(f18(W1),f18(X1)))\ \&$
$(\forall Y1\ Z1.\ equal(Y1::'a,Z1)\ -->\ equal(f19(Y1),f19(Z1)))\ \&$
$(\forall C2\ D2.\ equal(C2::'a,D2)\ -->\ equal(f20(C2),f20(D2)))\ \&$
$(\forall E2\ F2.\ equal(E2::'a,F2)\ -->\ equal(f21(E2),f21(F2)))\ \&$
$(\forall G2\ H2\ I2\ J2.\ equal(G2::'a,H2)\ -->\ equal(f22(G2::'a,I2,J2),f22(H2::'a,I2,J2)))$
$\&$
$(\forall K2\ M2\ L2\ N2.\ equal(K2::'a,L2)\ -->\ equal(f22(M2::'a,K2,N2),f22(M2::'a,L2,N2)))$
$\&$
$(\forall O2\ Q2\ R2\ P2.\ equal(O2::'a,P2)\ -->\ equal(f22(Q2::'a,R2,O2),f22(Q2::'a,R2,P2)))$
$\&$
$(\forall S2\ T2\ U2.\ equal(S2::'a,T2)\ -->\ equal(f23(S2::'a,U2),f23(T2::'a,U2)))\ \&$
$(\forall V2\ X2\ W2.\ equal(V2::'a,W2)\ -->\ equal(f23(X2::'a,V2),f23(X2::'a,W2)))$
$\&$
$(\forall Y2\ Z2.\ equal(Y2::'a,Z2)\ -->\ equal(f24(Y2),f24(Z2)))\ \&$
$(\forall A3\ B3.\ equal(A3::'a,B3)\ -->\ equal(f26(A3),f26(B3)))\ \&$
$(\forall C3\ D3\ E3.\ equal(C3::'a,D3)\ -->\ equal(f27(C3::'a,E3),f27(D3::'a,E3)))\ \&$
$(\forall F3\ H3\ G3.\ equal(F3::'a,G3)\ -->\ equal(f27(H3::'a,F3),f27(H3::'a,G3)))\ \&$
$(\forall I3\ J3\ K3\ L3.\ equal(I3::'a,J3)\ -->\ equal(f28(I3::'a,K3,L3),f28(J3::'a,K3,L3)))$
$\&$
$(\forall M3\ O3\ N3\ P3.\ equal(M3::'a,N3)\ -->\ equal(f28(O3::'a,M3,P3),f28(O3::'a,N3,P3)))$
$\&$
$(\forall Q3\ S3\ T3\ R3.\ equal(Q3::'a,R3)\ -->\ equal(f28(S3::'a,T3,Q3),f28(S3::'a,T3,R3)))$
$\&$
$(\forall U3\ V3\ W3\ X3.\ equal(U3::'a,V3)\ -->\ equal(f29(U3::'a,W3,X3),f29(V3::'a,W3,X3)))$
$\&$
$(\forall Y3\ A4\ Z3\ B4.\ equal(Y3::'a,Z3)\ -->\ equal(f29(A4::'a,Y3,B4),f29(A4::'a,Z3,B4)))$
$\&$
$(\forall C4\ E4\ F4\ D4.\ equal(C4::'a,D4)\ -->\ equal(f29(E4::'a,F4,C4),f29(E4::'a,F4,D4)))$
$\&$
$(\forall I4\ J4\ K4\ L4.\ equal(I4::'a,J4)\ -->\ equal(f30(I4::'a,K4,L4),f30(J4::'a,K4,L4)))$
$\&$
$(\forall M4\ O4\ N4\ P4.\ equal(M4::'a,N4)\ -->\ equal(f30(O4::'a,M4,P4),f30(O4::'a,N4,P4)))$
$\&$
$(\forall Q4\ S4\ T4\ R4.\ equal(Q4::'a,R4)\ -->\ equal(f30(S4::'a,T4,Q4),f30(S4::'a,T4,R4)))$
$\&$
$(\forall U4\ V4\ W4\ X4.\ equal(U4::'a,V4)\ -->\ equal(f31(U4::'a,W4,X4),f31(V4::'a,W4,X4)))$

&

$(\forall\, Y4\ A5\ Z4\ B5.\ equal(Y4{::}'a,Z4) \ {-}{-}{>}\ equal(f31(A5{::}'a,Y4,B5),f31(A5{::}'a,Z4,B5)))$

&

$(\forall\, C5\ E5\ F5\ D5.\ equal(C5{::}'a,D5) \ {-}{-}{>}\ equal(f31(E5{::}'a,F5,C5),f31(E5{::}'a,F5,D5)))$

&

$(\forall\, G5\ H5\ I5\ J5\ K5\ L5.\ equal(G5{::}'a,H5) \ {-}{-}{>}\ equal(f32(G5{::}'a,I5,J5,K5,L5),f32(H5{::}'a,I5,J5,K5,L5)))$

&

$(\forall\, M5\ O5\ N5\ P5\ Q5\ R5.\ equal(M5{::}'a,N5) \ {-}{-}{>}\ equal(f32(O5{::}'a,M5,P5,Q5,R5),f32(O5{::}'a,N5,P5,Q5,R5$

&

$(\forall\, S5\ U5\ V5\ T5\ W5\ X5.\ equal(S5{::}'a,T5) \ {-}{-}{>}\ equal(f32(U5{::}'a,V5,S5,W5,X5),f32(U5{::}'a,V5,T5,W5,X5)$

&

$(\forall\, Y5\ A6\ B6\ C6\ Z5\ D6.\ equal(Y5{::}'a,Z5) \ {-}{-}{>}\ equal(f32(A6{::}'a,B6,C6,Y5,D6),f32(A6{::}'a,B6,C6,Z5,D6))$

&

$(\forall\, E6\ G6\ H6\ I6\ J6\ F6.\ equal(E6{::}'a,F6) \ {-}{-}{>}\ equal(f32(G6{::}'a,H6,I6,J6,E6),f32(G6{::}'a,H6,I6,J6,F6)))$

&

$(\forall\, K6\ L6\ M6\ N6\ O6\ P6.\ equal(K6{::}'a,L6) \ {-}{-}{>}\ equal(f33(K6{::}'a,M6,N6,O6,P6),f33(L6{::}'a,M6,N6,O6,P6)$

&

$(\forall\, Q6\ S6\ R6\ T6\ U6\ V6.\ equal(Q6{::}'a,R6) \ {-}{-}{>}\ equal(f33(S6{::}'a,Q6,T6,U6,V6),f33(S6{::}'a,R6,T6,U6,V6))$

&

$(\forall\, W6\ Y6\ Z6\ X6\ A7\ B7.\ equal(W6{::}'a,X6) \ {-}{-}{>}\ equal(f33(Y6{::}'a,Z6,W6,A7,B7),f33(Y6{::}'a,Z6,X6,A7,B7)$

&

$(\forall\, C7\ E7\ F7\ G7\ D7\ H7.\ equal(C7{::}'a,D7) \ {-}{-}{>}\ equal(f33(E7{::}'a,F7,G7,C7,H7),f33(E7{::}'a,F7,G7,D7,H7))$

&

$(\forall\, I7\ K7\ L7\ M7\ N7\ J7.\ equal(I7{::}'a,J7) \ {-}{-}{>}\ equal(f33(K7{::}'a,L7,M7,N7,I7),f33(K7{::}'a,L7,M7,N7,J7)))$

&

$(\forall\, A\ B\ C.\ equal(A{::}'a,B) \ {-}{-}{>}\ equal(apply(A{::}'a,C),apply(B{::}'a,C)))$ &

$(\forall\, D\ F'\ E.\ equal(D{::}'a,E) \ {-}{-}{>}\ equal(apply(F'{::}'a,D),apply(F'{::}'a,E)))$ &

$(\forall\, G\ H\ I'\ J.\ equal(G{::}'a,H) \ {-}{-}{>}\ equal(apply\text{-}to\text{-}two\text{-}arguments(G{::}'a,I',J),apply\text{-}to\text{-}two\text{-}arguments(H{::}'a,I',$

&

$(\forall\, K'\ M\ L\ N.\ equal(K'{::}'a,L) \ {-}{-}{>}\ equal(apply\text{-}to\text{-}two\text{-}arguments(M{::}'a,K',N),apply\text{-}to\text{-}two\text{-}arguments(M{::}'a$

&

$(\forall\, O'\ Q\ R\ P.\ equal(O'{::}'a,P) \ {-}{-}{>}\ equal(apply\text{-}to\text{-}two\text{-}arguments(Q{::}'a,R,O'),apply\text{-}to\text{-}two\text{-}arguments(Q{::}'a,$

&

$(\forall\, S'\ T'.\ equal(S'{::}'a,T') \ {-}{-}{>}\ equal(complement(S'),complement(T')))$ &

$(\forall\, U\ V\ W.\ equal(U{::}'a,V) \ {-}{-}{>}\ equal(composition(U{::}'a,W),composition(V{::}'a,W)))$

&

$(\forall\, X\ Z\ Y.\ equal(X{::}'a,Y) \ {-}{-}{>}\ equal(composition(Z{::}'a,X),composition(Z{::}'a,Y)))$

&

$(\forall\, A1\ B1.\ equal(A1{::}'a,B1) \ {-}{-}{>}\ equal(inv1\ A1,inv1\ B1))$ &

$(\forall\, C1\ D1\ E1.\ equal(C1{::}'a,D1) \ {-}{-}{>}\ equal(cross\text{-}product(C1{::}'a,E1),cross\text{-}product(D1{::}'a,E1)))$

&

$(\forall\, F1\ H1\ G1.\ equal(F1{::}'a,G1) \ {-}{-}{>}\ equal(cross\text{-}product(H1{::}'a,F1),cross\text{-}product(H1{::}'a,G1)))$

&

$(\forall\, I1\ J1.\ equal(I1{::}'a,J1) \ {-}{-}{>}\ equal(domain\text{-}of(I1),domain\text{-}of(J1)))$ &

$(\forall\, I10\ J10.\ equal(I10{::}'a,J10) \ {-}{-}{>}\ equal(first(I10),first(J10)))$ &

$(\forall\, Q10\ R10.\ equal(Q10{::}'a,R10) \ {-}{-}{>}\ equal(flip\text{-}range\text{-}of(Q10),flip\text{-}range\text{-}of(R10)))$

&

$(\forall\, S10\ T10\ U10.\ equal(S10{::}'a,T10) \ {-}{-}{>}\ equal(image'(S10{::}'a,U10),image'(T10{::}'a,U10)))$

&

(∀ V10 X10 W10. equal(V10::'a,W10) --> equal(image'(X10::'a,V10),image'(X10::'a,W10)))
&
(∀ Y10 Z10 A11. equal(Y10::'a,Z10) --> equal(intersection(Y10::'a,A11),intersection(Z10::'a,A11)))
&
(∀ B11 D11 C11. equal(B11::'a,C11) --> equal(intersection(D11::'a,B11),intersection(D11::'a,C11)))
&
(∀ E11 F11 G11. equal(E11::'a,F11) --> equal(non-ordered-pair(E11::'a,G11),non-ordered-pair(F11::'a,G
&
(∀ H11 J11 I11. equal(H11::'a,I11) --> equal(non-ordered-pair(J11::'a,H11),non-ordered-pair(J11::'a,I11)
&
(∀ K11 L11 M11. equal(K11::'a,L11) --> equal(ordered-pair(K11::'a,M11),ordered-pair(L11::'a,M11)))
&
(∀ N11 P11 O11. equal(N11::'a,O11) --> equal(ordered-pair(P11::'a,N11),ordered-pair(P11::'a,O11)))
&
(∀ Q11 R11. equal(Q11::'a,R11) --> equal(powerset(Q11),powerset(R11))) &
(∀ S11 T11. equal(S11::'a,T11) --> equal(range-of(S11),range-of(T11))) &
(∀ U11 V11 W11. equal(U11::'a,V11) --> equal(restrct(U11::'a,W11),restrct(V11::'a,W11)))
&
(∀ X11 Z11 Y11. equal(X11::'a,Y11) --> equal(restrct(Z11::'a,X11),restrct(Z11::'a,Y11)))
&
(∀ A12 B12. equal(A12::'a,B12) --> equal(rot-right(A12),rot-right(B12))) &
(∀ C12 D12. equal(C12::'a,D12) --> equal(second(C12),second(D12))) &
(∀ K12 L12. equal(K12::'a,L12) --> equal(sigma(K12),sigma(L12))) &
(∀ M12 N12. equal(M12::'a,N12) --> equal(singleton-set(M12),singleton-set(N12)))
&
(∀ O12 P12. equal(O12::'a,P12) --> equal(successor(O12),successor(P12))) &
(∀ Q12 R12 S12. equal(Q12::'a,R12) --> equal(union(Q12::'a,S12),union(R12::'a,S12)))
&
(∀ T12 V12 U12. equal(T12::'a,U12) --> equal(union(V12::'a,T12),union(V12::'a,U12)))
&
(∀ W12 X12 Y12. equal(W12::'a,X12) & closed(W12::'a,Y12) --> closed(X12::'a,Y12))
&
(∀ Z12 B13 A13. equal(Z12::'a,A13) & closed(B13::'a,Z12) --> closed(B13::'a,A13))
&
(∀ C13 D13 E13. equal(C13::'a,D13) & disjoint(C13::'a,E13) --> disjoint(D13::'a,E13))
&
(∀ F13 H13 G13. equal(F13::'a,G13) & disjoint(H13::'a,F13) --> disjoint(H13::'a,G13))
&
(∀ I13 J13. equal(I13::'a,J13) & function(I13) --> function(J13)) &
(∀ K13 L13 M13 N13 O13 P13. equal(K13::'a,L13) & homomorphism(K13::'a,M13,N13,O13,P13)
--> homomorphism(L13::'a,M13,N13,O13,P13)) &
(∀ Q13 S13 R13 T13 U13 V13. equal(Q13::'a,R13) & homomorphism(S13::'a,Q13,T13,U13,V13)
--> homomorphism(S13::'a,R13,T13,U13,V13)) &
(∀ W13 Y13 Z13 X13 A14 B14. equal(W13::'a,X13) & homomorphism(Y13::'a,Z13,W13,A14,B14)
--> homomorphism(Y13::'a,Z13,X13,A14,B14)) &
(∀ C14 E14 F14 G14 D14 H14. equal(C14::'a,D14) & homomorphism(E14::'a,F14,G14,C14,H14)
--> homomorphism(E14::'a,F14,G14,D14,H14)) &
(∀ I14 K14 L14 M14 N14 J14. equal(I14::'a,J14) & homomorphism(K14::'a,L14,M14,N14,I14)
--> homomorphism(K14::'a,L14,M14,N14,J14)) &

(∀ O14 P14. equal(O14::'a,P14) & little-set(O14) --> little-set(P14)) &
(∀ Q14 R14 S14 T14. equal(Q14::'a,R14) & maps(Q14::'a,S14,T14) --> maps(R14::'a,S14,T14))
&
(∀ U14 W14 V14 X14. equal(U14::'a,V14) & maps(W14::'a,U14,X14) -->
maps(W14::'a,V14,X14)) &
(∀ Y14 A15 B15 Z14. equal(Y14::'a,Z14) & maps(A15::'a,B15,Y14) --> maps(A15::'a,B15,Z14))
&
(∀ C15 D15 E15. equal(C15::'a,D15) & member(C15::'a,E15) --> member(D15::'a,E15))
&
(∀ F15 H15 G15. equal(F15::'a,G15) & member(H15::'a,F15) --> member(H15::'a,G15))
&
(∀ I15 J15. equal(I15::'a,J15) & one-to-one-function(I15) --> one-to-one-function(J15))
&
(∀ K15 L15. equal(K15::'a,L15) & ordered-pair-predicate(K15) --> ordered-pair-predicate(L15))
&
(∀ M15 N15 O15. equal(M15::'a,N15) & proper-subset(M15::'a,O15) --> proper-subset(N15::'a,O15))
&
(∀ P15 R15 Q15. equal(P15::'a,Q15) & proper-subset(R15::'a,P15) --> proper-subset(R15::'a,Q15))
&
(∀ S15 T15. equal(S15::'a,T15) & relation(S15) --> relation(T15)) &
(∀ U15 V15. equal(U15::'a,V15) & single-valued-set(U15) --> single-valued-set(V15))
&
(∀ W15 X15 Y15. equal(W15::'a,X15) & ssubset(W15::'a,Y15) --> ssubset(X15::'a,Y15))
&
(∀ Z15 B16 A16. equal(Z15::'a,A16) & ssubset(B16::'a,Z15) --> ssubset(B16::'a,A16))
&
(~little-set(ordered-pair(a::'a,b))) --> False
oops

lemma *SET046-5*:
(∀ Y X. ~(element(X::'a,a) & element(X::'a,Y) & element(Y::'a,X))) &
(∀ X. element(X::'a,f(X)) | element(X::'a,a)) &
(∀ X. element(f(X),X) | element(X::'a,a)) --> False
by *meson*

lemma *SET047-5*:
(∀ X Z Y. set-equal(X::'a,Y) & element(Z::'a,X) --> element(Z::'a,Y)) &
(∀ Y Z X. set-equal(X::'a,Y) & element(Z::'a,Y) --> element(Z::'a,X)) &
(∀ X Y. element(f(X::'a,Y),X) | element(f(X::'a,Y),Y) | set-equal(X::'a,Y))
&
(∀ X Y. element(f(X::'a,Y),Y) & element(f(X::'a,Y),X) --> set-equal(X::'a,Y))
&
(set-equal(a::'a,b) | set-equal(b::'a,a)) &
(~(set-equal(b::'a,a) & set-equal(a::'a,b))) --> False
by *meson*

lemma *SYN034-1*:
$(\forall A.\ p(A::'a,a) \mid p(A::'a,f(A)))$ &
$(\forall A.\ p(A::'a,a) \mid p(f(A),A))$ &
$(\forall A\ B.\ \sim(p(A::'a,B)\ \&\ p(B::'a,A)\ \&\ p(B::'a,a)))$ $--\!>$ *False*
by *meson*

lemma *SYN071-1*:
EQU001-0-ax equal &
$(equal(a::'a,b) \mid equal(c::'a,d))$ &
$(equal(a::'a,c) \mid equal(b::'a,d))$ &
$(\sim equal(a::'a,d))$ &
$(\sim equal(b::'a,c))$ $--\!>$ *False*
by *meson*

lemma *SYN349-1*:
$(\forall X\ Y.\ f(w(X),g(X::'a,Y))\ --\!>\ f(X::'a,g(X::'a,Y)))$ &
$(\forall X\ Y.\ f(X::'a,g(X::'a,Y))\ --\!>\ f(w(X),g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(Y::'a,g(X::'a,Y))\ --\!>\ f(g(X::'a,Y),Y) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(g(X::'a,Y),Y)\ \&\ f(Y::'a,g(X::'a,Y))\ --\!>\ f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),Y) \mid f(Y::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),Y)\ --\!>\ f(Y::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X))\ --\!>\ f(g(X::'a,Y),Y) \mid f(Y::'a,g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(g(X::'a,Y),Y)\ \&\ f(g(X::'a,Y),w(X))\ --\!>\ f(X::'a,g(X::'a,Y)) \mid f(Y::'a,g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(Y::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X))\ --\!>\ f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),Y))$ &
$(\forall Y\ X.\ \sim(f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),Y)\ \&\ f(Y::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X))))$ $--\!>$ *False*
oops

lemma *SYN352-1*:
$(f(a::'a,b))$ &
$(\forall X\ Y.\ f(X::'a,Y)\ --\!>\ f(b::'a,z(X::'a,Y)) \mid f(Y::'a,z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(X::'a,Y) \mid f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(b::'a,z(X::'a,Y)) \mid f(X::'a,z(X::'a,Y)) \mid f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(b::'a,z(X::'a,Y))\ \&\ f(X::'a,z(X::'a,Y))\ --\!>\ f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ \sim(f(X::'a,Y)\ \&\ f(X::'a,z(X::'a,Y))\ \&\ f(Y::'a,z(X::'a,Y))))$ &
$(\forall X\ Y.\ f(X::'a,Y)\ --\!>\ f(X::'a,z(X::'a,Y)) \mid f(Y::'a,z(X::'a,Y)))$ $--\!>$ *False*
by *meson*

lemma *TOP001-2*:
$(\forall\ Vf\ U.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,f1(Vf::'a,U)))$
&
$(\forall\ U\ Vf.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \text{-}\text{-}{>}\ element\text{-}of\text{-}collection(f1(Vf::'a,U),Vf))$
&
$(\forall\ U\ \ Uu1\ \ Vf.\ element\text{-}of\text{-}set(U::'a,Uu1)\ \ \&\ \ element\text{-}of\text{-}collection(Uu1::'a,Vf)$
$\text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf)))\ \&$
$(\forall\ Vf\ X.\ basis(X::'a,Vf)\ \text{-}\text{-}{>}\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X))\ \&$
$(\forall\ Vf\ U\ X.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\text{-}\text{-}{>}\ element\text{-}of\text{-}set(X::'a,f10(Vf::'a,U,X)))\ \&$
$(\forall\ U\ X\ Vf.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$\text{-}\text{-}{>}\ element\text{-}of\text{-}collection(f10(Vf::'a,U,X),Vf))\ \&$
$(\forall\ X.\ subset\text{-}sets(X::'a,X))\ \&$
$(\forall\ X\ U\ Y.\ subset\text{-}sets(X::'a,Y)\ \&\ element\text{-}of\text{-}set(U::'a,X)\ \text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,Y))$
&
$(\forall\ X\ Y.\ equal\text{-}sets(X::'a,Y)\ \text{-}\text{-}{>}\ subset\text{-}sets(X::'a,Y))\ \&$
$(\forall\ Y\ X.\ subset\text{-}sets(X::'a,Y)\ |\ element\text{-}of\text{-}set(in\text{-}1st\text{-}set(X::'a,Y),X))\ \&$
$(\forall\ X\ Y.\ element\text{-}of\text{-}set(in\text{-}1st\text{-}set(X::'a,Y),Y)\ \text{-}\text{-}{>}\ subset\text{-}sets(X::'a,Y))\ \&$
$(basis(cx::'a,f))\ \&$
$(\sim subset\text{-}sets(union\text{-}of\text{-}members(top\text{-}of\text{-}basis(f)),cx))\ \text{-}\text{-}{>}\ False$
by *meson*

lemma *TOP002-2*:
$(\forall\ Vf\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ |\ element\text{-}of\text{-}set(f11(Vf::'a,U),U))$
&
$(\forall\ X.\ \sim element\text{-}of\text{-}set(X::'a,empty\text{-}set))\ \&$
$(\sim element\text{-}of\text{-}collection(empty\text{-}set::'a,top\text{-}of\text{-}basis(f)))\ \text{-}\text{-}{>}\ False$
by *meson*

lemma *TOP004-1*:
$(\forall\ Vf\ U.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,f1(Vf::'a,U)))$
&
$(\forall\ U\ Vf.\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf))\ \text{-}\text{-}{>}\ element\text{-}of\text{-}collection(f1(Vf::'a,U),Vf))$
&
$(\forall\ U\ \ Uu1\ \ Vf.\ element\text{-}of\text{-}set(U::'a,Uu1)\ \ \&\ \ element\text{-}of\text{-}collection(Uu1::'a,Vf)$
$\text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,union\text{-}of\text{-}members(Vf)))\ \&$
$(\forall\ Vf\ U\ Va.\ element\text{-}of\text{-}set(U::'a,intersection\text{-}of\text{-}members(Vf))\ \&\ element\text{-}of\text{-}collection(Va::'a,Vf)$
$\text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,Va))\ \&$
$(\forall\ U\ Vf.\ element\text{-}of\text{-}set(U::'a,intersection\text{-}of\text{-}members(Vf))\ |\ element\text{-}of\text{-}collection(f2(Vf::'a,U),Vf))$
&
$(\forall\ Vf\ U.\ element\text{-}of\text{-}set(U::'a,f2(Vf::'a,U))\ \text{-}\text{-}{>}\ element\text{-}of\text{-}set(U::'a,intersection\text{-}of\text{-}members(Vf)))$
&
$(\forall\ Vt\ X.\ topological\text{-}space(X::'a,Vt)\ \text{-}\text{-}{>}\ equal\text{-}sets(union\text{-}of\text{-}members(Vt),X))$
&
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \text{-}\text{-}{>}\ element\text{-}of\text{-}collection(empty\text{-}set::'a,Vt))$

&
($\forall$ X Vt. *topological-space*(X::'a, Vt) --> *element-of-collection*(X::'a, Vt)) &
($\forall$ X Y Z Vt. *topological-space*(X::'a, Vt) & *element-of-collection*(Y::'a, Vt) &
element-of-collection(Z::'a, Vt) --> *element-of-collection*(*intersection-of-sets*(Y::'a,Z), Vt))
&
($\forall$ X Vf Vt. *topological-space*(X::'a, Vt) & *subset-collections*(Vf::'a, Vt) -->
element-of-collection(*union-of-members*(Vf), Vt)) &
($\forall$ X Vt. *equal-sets*(*union-of-members*(Vt),X) & *element-of-collection*(*empty-set*::'a, Vt)
& *element-of-collection*(X::'a, Vt) --> *topological-space*(X::'a, Vt) | *element-of-collection*(*f3*(X::'a, Vt), Vt)
| *subset-collections*(*f5*(X::'a, Vt), Vt)) &
($\forall$ X Vt. *equal-sets*(*union-of-members*(Vt),X) & *element-of-collection*(*empty-set*::'a, Vt)
& *element-of-collection*(X::'a, Vt) & *element-of-collection*(*union-of-members*(*f5*(X::'a, Vt)), Vt)
--> *topological-space*(X::'a, Vt) | *element-of-collection*(*f3*(X::'a, Vt), Vt)) &
($\forall$ X Vt. *equal-sets*(*union-of-members*(Vt),X) & *element-of-collection*(*empty-set*::'a, Vt)
& *element-of-collection*(X::'a, Vt) --> *topological-space*(X::'a, Vt) | *element-of-collection*(*f4*(X::'a, Vt), Vt)
| *subset-collections*(*f5*(X::'a, Vt), Vt)) &
($\forall$ X Vt. *equal-sets*(*union-of-members*(Vt),X) & *element-of-collection*(*empty-set*::'a, Vt)
& *element-of-collection*(X::'a, Vt) & *element-of-collection*(*union-of-members*(*f5*(X::'a, Vt)), Vt)
--> *topological-space*(X::'a, Vt) | *element-of-collection*(*f4*(X::'a, Vt), Vt)) &
($\forall$ X Vt. *equal-sets*(*union-of-members*(Vt),X) & *element-of-collection*(*empty-set*::'a, Vt)
& *element-of-collection*(X::'a, Vt) & *element-of-collection*(*intersection-of-sets*(*f3*(X::'a, Vt),*f4*(X::'a, Vt)), Vt)
--> *topological-space*(X::'a, Vt) | *subset-collections*(*f5*(X::'a, Vt), Vt)) &
($\forall$ X Vt. *equal-sets*(*union-of-members*(Vt),X) & *element-of-collection*(*empty-set*::'a, Vt)
& *element-of-collection*(X::'a, Vt) & *element-of-collection*(*intersection-of-sets*(*f3*(X::'a, Vt),*f4*(X::'a, Vt)), Vt)
& *element-of-collection*(*union-of-members*(*f5*(X::'a, Vt)), Vt) --> *topological-space*(X::'a, Vt))
&
($\forall$ U X Vt. *open*(U::'a,X, Vt) --> *topological-space*(X::'a, Vt)) &
($\forall$ X U Vt. *open*(U::'a,X, Vt) --> *element-of-collection*(U::'a, Vt)) &
($\forall$ X U Vt. *topological-space*(X::'a, Vt) & *element-of-collection*(U::'a, Vt) -->
open(U::'a,X, Vt)) &
($\forall$ U X Vt. *closed*(U::'a,X, Vt) --> *topological-space*(X::'a, Vt)) &
($\forall$ U X Vt. *closed*(U::'a,X, Vt) --> *open*(*relative-complement-sets*(U::'a,X),X, Vt))
&
($\forall$ U X Vt. *topological-space*(X::'a, Vt) & *open*(*relative-complement-sets*(U::'a,X),X, Vt)
--> *closed*(U::'a,X, Vt)) &
($\forall$ Vs X Vt. *finer*(Vt::'a, Vs,X) --> *topological-space*(X::'a, Vt)) &
($\forall$ Vt X Vs. *finer*(Vt::'a, Vs,X) --> *topological-space*(X::'a, Vs)) &
($\forall$ X Vs Vt. *finer*(Vt::'a, Vs,X) --> *subset-collections*(Vs::'a, Vt)) &
($\forall$ X Vs Vt. *topological-space*(X::'a, Vt) & *topological-space*(X::'a, Vs) & *subset-collections*(Vs::'a, Vt)
--> *finer*(Vt::'a, Vs,X)) &
($\forall$ Vf X. *basis*(X::'a, Vf) --> *equal-sets*(*union-of-members*(Vf),X)) &
($\forall$ X Vf Y Vb1 Vb2. *basis*(X::'a, Vf) & *element-of-set*(Y::'a,X) & *element-of-collection*(Vb1::'a, Vf)
& *element-of-collection*(Vb2::'a, Vf) & *element-of-set*(Y::'a,*intersection-of-sets*(Vb1::'a, Vb2))
--> *element-of-set*(Y::'a,*f6*(X::'a, Vf, Y, Vb1, Vb2))) &
($\forall$ X Y Vb1 Vb2 Vf. *basis*(X::'a, Vf) & *element-of-set*(Y::'a,X) & *element-of-collection*(Vb1::'a, Vf)
& *element-of-collection*(Vb2::'a, Vf) & *element-of-set*(Y::'a,*intersection-of-sets*(Vb1::'a, Vb2))
--> *element-of-collection*(*f6*(X::'a, Vf, Y, Vb1, Vb2), Vf)) &
($\forall$ X Vf Y Vb1 Vb2. *basis*(X::'a, Vf) & *element-of-set*(Y::'a,X) & *element-of-collection*(Vb1::'a, Vf)
& *element-of-collection*(Vb2::'a, Vf) & *element-of-set*(Y::'a,*intersection-of-sets*(Vb1::'a, Vb2))

−−> subset-sets(f6(X::′a, Vf, Y, Vb1, Vb2),intersection-of-sets(Vb1::′a, Vb2))) &
(∀ Vf X. equal-sets(union-of-members(Vf),X) −−> basis(X::′a, Vf) | element-of-set(f7(X::′a, Vf),X))
&
(∀ X Vf. equal-sets(union-of-members(Vf),X) −−> basis(X::′a, Vf) | element-of-collection(f8(X::′a, Vf), Vf)
&
(∀ X Vf. equal-sets(union-of-members(Vf),X) −−> basis(X::′a, Vf) | element-of-collection(f9(X::′a, Vf), Vf)
&
(∀ X Vf. equal-sets(union-of-members(Vf),X) −−> basis(X::′a, Vf) | element-of-set(f7(X::′a, Vf),intersectio
&
(∀ Uu9 X Vf. equal-sets(union-of-members(Vf),X) & element-of-set(f7(X::′a, Vf), Uu9)
& element-of-collection(Uu9::′a, Vf) & subset-sets(Uu9::′a,intersection-of-sets(f8(X::′a, Vf),f9(X::′a, Vf)))
−−> basis(X::′a, 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))) &
(∀ U Y X Vt. element-of-collection(U::′a,subspace-topology(X::′a, Vt, Y)) −−>
topological-space(X::′a, Vt)) &
(∀ U Vt Y X. element-of-collection(U::′a,subspace-topology(X::′a, Vt, Y)) −−>
subset-sets(Y::′a,X)) &
(∀ X Y U Vt. element-of-collection(U::′a,subspace-topology(X::′a, Vt, Y)) −−>
element-of-collection(f12(X::′a, Vt, Y, U), Vt)) &
(∀ X Vt Y U. element-of-collection(U::′a,subspace-topology(X::′a, Vt, Y)) −−>
equal-sets(U::′a,intersection-of-sets(Y::′a,f12(X::′a, Vt, Y, U)))) &
(∀ X Vt U Y Uu12. topological-space(X::′a, Vt) & subset-sets(Y::′a,X) & element-of-collection(Uu12::′a, Vt)
& equal-sets(U::′a,intersection-of-sets(Y::′a, Uu12)) −−> element-of-collection(U::′a,subspace-topology(X::′a
&
(∀ U Y X Vt. element-of-set(U::′a,interior(Y::′a,X, Vt)) −−> topological-space(X::′a, Vt))
&
(∀ U Vt Y X. element-of-set(U::′a,interior(Y::′a,X, Vt)) −−> subset-sets(Y::′a,X))
&
(∀ Y X Vt U. element-of-set(U::′a,interior(Y::′a,X, Vt)) −−> element-of-set(U::′a,f13(Y::′a,X, Vt, U)))
&
(∀ X Vt U Y. element-of-set(U::′a,interior(Y::′a,X, Vt)) −−> subset-sets(f13(Y::′a,X, Vt, U), Y))
&
(∀ Y U X Vt. element-of-set(U::′a,interior(Y::′a,X, Vt)) −−> open(f13(Y::′a,X, Vt, U),X, Vt))
&
(∀ U Y Uu13 X Vt. topological-space(X::′a, Vt) & subset-sets(Y::′a,X) & element-of-set(U::′a, Uu13)
& subset-sets(Uu13::′a, Y) & open(Uu13::′a,X, Vt) −−> element-of-set(U::′a,interior(Y::′a,X, Vt)))
&
(∀ U Y X Vt. element-of-set(U::′a,closure(Y::′a,X, Vt)) −−> topological-space(X::′a, Vt))
&
(∀ U Vt Y X. element-of-set(U::′a,closure(Y::′a,X, Vt)) −−> subset-sets(Y::′a,X))

&
(∀ Y X Vt U V. *element-of-set*(U::'a,*closure*(Y::'a,X,Vt)) & *subset-sets*(Y::'a,V)
& *closed*(V::'a,X,Vt) --> *element-of-set*(U::'a,V)) &
(∀ Y X Vt U. *topological-space*(X::'a,Vt) & *subset-sets*(Y::'a,X) --> *element-of-set*(U::'a,*closure*(Y::'a,X,
| *subset-sets*(Y::'a,*f14*(Y::'a,X,Vt,U))) &
(∀ Y U X Vt. *topological-space*(X::'a,Vt) & *subset-sets*(Y::'a,X) --> *element-of-set*(U::'a,*closure*(Y::'a,X,
| *closed*(*f14*(Y::'a,X,Vt,U),X,Vt)) &
(∀ Y X Vt U. *topological-space*(X::'a,Vt) & *subset-sets*(Y::'a,X) & *element-of-set*(U::'a,*f14*(Y::'a,X,Vt,U))
--> *element-of-set*(U::'a,*closure*(Y::'a,X,Vt))) &
(∀ U Y X Vt. *neighborhood*(U::'a,Y,X,Vt) --> *topological-space*(X::'a,Vt)) &
(∀ Y U X Vt. *neighborhood*(U::'a,Y,X,Vt) --> *open*(U::'a,X,Vt)) &
(∀ X Vt Y U. *neighborhood*(U::'a,Y,X,Vt) --> *element-of-set*(Y::'a,U)) &
(∀ X Vt Y U. *topological-space*(X::'a,Vt) & *open*(U::'a,X,Vt) & *element-of-set*(Y::'a,U)
--> *neighborhood*(U::'a,Y,X,Vt)) &
(∀ Z Y X Vt. *limit-point*(Z::'a,Y,X,Vt) --> *topological-space*(X::'a,Vt)) &
(∀ Z Vt Y X. *limit-point*(Z::'a,Y,X,Vt) --> *subset-sets*(Y::'a,X)) &
(∀ Z X Vt U Y. *limit-point*(Z::'a,Y,X,Vt) & *neighborhood*(U::'a,Z,X,Vt) -->
element-of-set(*f15*(Z::'a,Y,X,Vt,U),*intersection-of-sets*(U::'a,Y))) &
(∀ Y X Vt U Z. ~(*limit-point*(Z::'a,Y,X,Vt) & *neighborhood*(U::'a,Z,X,Vt) &
eq-p(*f15*(Z::'a,Y,X,Vt,U),Z))) &
(∀ Y Z X Vt. *topological-space*(X::'a,Vt) & *subset-sets*(Y::'a,X) --> *limit-point*(Z::'a,Y,X,Vt)
| *neighborhood*(*f16*(Z::'a,Y,X,Vt),Z,X,Vt)) &
(∀ X Vt Y Uu16 Z. *topological-space*(X::'a,Vt) & *subset-sets*(Y::'a,X) & *element-of-set*(Uu16::'a,*intersection*
--> *limit-point*(Z::'a,Y,X,Vt) | *eq-p*(Uu16::'a,Z)) &
(∀ U Y X Vt. *element-of-set*(U::'a,*boundary*(Y::'a,X,Vt)) --> *topological-space*(X::'a,Vt))
&
(∀ U Y X Vt. *element-of-set*(U::'a,*boundary*(Y::'a,X,Vt)) --> *element-of-set*(U::'a,*closure*(Y::'a,X,Vt)))
&
(∀ U Y X Vt. *element-of-set*(U::'a,*boundary*(Y::'a,X,Vt)) --> *element-of-set*(U::'a,*closure*(*relative-comple*
&
(∀ U Y X Vt. *topological-space*(X::'a,Vt) & *element-of-set*(U::'a,*closure*(Y::'a,X,Vt))
& *element-of-set*(U::'a,*closure*(*relative-complement-sets*(Y::'a,X),X,Vt)) --> *element-of-set*(U::'a,*boundary*
&
(∀ X Vt. *hausdorff*(X::'a,Vt) --> *topological-space*(X::'a,Vt)) &
(∀ X-2 X-1 X Vt. *hausdorff*(X::'a,Vt) & *element-of-set*(X-1::'a,X) & *element-of-set*(X-2::'a,X)
--> *eq-p*(X-1::'a,X-2) | *neighborhood*(*f17*(X::'a,Vt,X-1,X-2),X-1,X,Vt)) &
(∀ X-1 X-2 X Vt. *hausdorff*(X::'a,Vt) & *element-of-set*(X-1::'a,X) & *element-of-set*(X-2::'a,X)
--> *eq-p*(X-1::'a,X-2) | *neighborhood*(*f18*(X::'a,Vt,X-1,X-2),X-2,X,Vt)) &
(∀ X Vt X-1 X-2. *hausdorff*(X::'a,Vt) & *element-of-set*(X-1::'a,X) & *element-of-set*(X-2::'a,X)
--> *eq-p*(X-1::'a,X-2) | *disjoint-s*(*f17*(X::'a,Vt,X-1,X-2),*f18*(X::'a,Vt,X-1,X-2)))
&
(∀ Vt X. *topological-space*(X::'a,Vt) --> *hausdorff*(X::'a,Vt) | *element-of-set*(*f19*(X::'a,Vt),X))
&
(∀ Vt X. *topological-space*(X::'a,Vt) --> *hausdorff*(X::'a,Vt) | *element-of-set*(*f20*(X::'a,Vt),X))
&
(∀ X Vt. *topological-space*(X::'a,Vt) & *eq-p*(*f19*(X::'a,Vt),*f20*(X::'a,Vt)) -->
hausdorff(X::'a,Vt)) &
(∀ 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) -->

hausdorff(X::'a, Vt)) &
(∀ Va1 Va2 X Vt. separation(Va1::'a, Va2, X, Vt) --> topological-space(X::'a, Vt))
&
(∀ Va2 X Vt Va1. ~(separation(Va1::'a, Va2, X, Vt) & equal-sets(Va1::'a, empty-set)))
&
(∀ Va1 X Vt Va2. ~(separation(Va1::'a, Va2, X, Vt) & equal-sets(Va2::'a, empty-set)))
&
(∀ Va2 X Va1 Vt. separation(Va1::'a, Va2, X, Vt) --> element-of-collection(Va1::'a, Vt))
&
(∀ Va1 X Va2 Vt. separation(Va1::'a, Va2, X, Vt) --> element-of-collection(Va2::'a, Vt))
&
(∀ Vt Va1 Va2 X. separation(Va1::'a, Va2, X, Vt) --> equal-sets(union-of-sets(Va1::'a, Va2), X))
&
(∀ X Vt Va1 Va2. separation(Va1::'a, Va2, X, Vt) --> disjoint-s(Va1::'a, Va2))
&
(∀ 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) --> separation(Va1::'a, Va2, X, Vt) | equal-sets(Va1::'a, empty-set)
| equal-sets(Va2::'a, empty-set)) &
(∀ X Vt. connected-space(X::'a, Vt) --> topological-space(X::'a, Vt)) &
(∀ Va1 Va2 X Vt. ~(connected-space(X::'a, Vt) & separation(Va1::'a, Va2, X, Vt)))
&
(∀ X Vt. topological-space(X::'a, Vt) --> connected-space(X::'a, Vt) | separation(f21(X::'a, Vt), f22(X::'a, Vt), X, Vt)) &
(∀ Va X Vt. connected-set(Va::'a, X, Vt) --> topological-space(X::'a, Vt)) &
(∀ Vt Va X. connected-set(Va::'a, X, Vt) --> subset-sets(Va::'a, X)) &
(∀ X Vt Va. connected-set(Va::'a, X, Vt) --> connected-space(Va::'a, subspace-topology(X::'a, Vt, Va)))
&
(∀ X Vt Va. topological-space(X::'a, Vt) & subset-sets(Va::'a, X) & connected-space(Va::'a, subspace-topology(X
--> connected-set(Va::'a, X, Vt)) &
(∀ Vf X Vt. open-covering(Vf::'a, X, Vt) --> topological-space(X::'a, Vt)) &
(∀ X Vf Vt. open-covering(Vf::'a, X, Vt) --> subset-collections(Vf::'a, Vt)) &
(∀ Vt Vf X. open-covering(Vf::'a, X, Vt) --> equal-sets(union-of-members(Vf), X))
&
(∀ Vt Vf X. topological-space(X::'a, Vt) & subset-collections(Vf::'a, Vt) & equal-sets(union-of-members(Vf), X
--> open-covering(Vf::'a, X, Vt)) &
(∀ X Vt. compact-space(X::'a, Vt) --> topological-space(X::'a, Vt)) &
(∀ X Vt Vf1. compact-space(X::'a, Vt) & open-covering(Vf1::'a, X, Vt) --> finite'(f23(X::'a, Vt, Vf1))) &
(∀ X Vt Vf1. compact-space(X::'a, Vt) & open-covering(Vf1::'a, X, Vt) --> subset-collections(f23(X::'a, Vt, V
&
(∀ Vf1 X Vt. compact-space(X::'a, Vt) & open-covering(Vf1::'a, X, Vt) --> open-covering(f23(X::'a, Vt, Vf1)
&
(∀ X Vt. topological-space(X::'a, Vt) --> compact-space(X::'a, Vt) | open-covering(f24(X::'a, Vt), X, Vt))
&
(∀ Uu24 X Vt. topological-space(X::'a, Vt) & finite'(Uu24) & subset-collections(Uu24::'a, f24(X::'a, Vt))
& open-covering(Uu24::'a, X, Vt) --> compact-space(X::'a, Vt)) &
(∀ Va X Vt. compact-set(Va::'a, X, Vt) --> topological-space(X::'a, Vt)) &
(∀ Vt Va X. compact-set(Va::'a, X, Vt) --> subset-sets(Va::'a, X)) &

(∀ X Vt Va. compact-set(Va::'a,X,Vt) --> compact-space(Va::'a,subspace-topology(X::'a,Vt,Va)))
&
(∀ X Vt Va. topological-space(X::'a,Vt) & subset-sets(Va::'a,X) & compact-space(Va::'a,subspace-topology(X:
--> compact-set(Va::'a,X,Vt)) &
(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 *TOP004-2*:
(∀ U Uu1 Vf. element-of-set(U::'a,Uu1) & element-of-collection(Uu1::'a,Vf) -->
element-of-set(U::'a,union-of-members(Vf))) &
(∀ Vf X. basis(X::'a,Vf) --> equal-sets(union-of-members(Vf),X)) &
(∀ X Vf Y Vb1 Vb2. basis(X::'a,Vf) & element-of-set(Y::'a,X) & element-of-collection(Vb1::'a,Vf)
& element-of-collection(Vb2::'a,Vf) & element-of-set(Y::'a,intersection-of-sets(Vb1::'a,Vb2))
--> element-of-set(Y::'a,f6(X::'a,Vf,Y,Vb1,Vb2))) &
(∀ X Y Vb1 Vb2 Vf. basis(X::'a,Vf) & element-of-set(Y::'a,X) & element-of-collection(Vb1::'a,Vf)
& element-of-collection(Vb2::'a,Vf) & element-of-set(Y::'a,intersection-of-sets(Vb1::'a,Vb2))
--> element-of-collection(f6(X::'a,Vf,Y,Vb1,Vb2),Vf)) &
(∀ X Vf Y Vb1 Vb2. basis(X::'a,Vf) & element-of-set(Y::'a,X) & element-of-collection(Vb1::'a,Vf)
& element-of-collection(Vb2::'a,Vf) & element-of-set(Y::'a,intersection-of-sets(Vb1::'a,Vb2))
--> subset-sets(f6(X::'a,Vf,Y,Vb1,Vb2),intersection-of-sets(Vb1::'a,Vb2))) &
(∀ 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))) &
(∀ 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
&
(∀ X U Y V. subset-sets(X::'a,Y) & subset-sets(U::'a,V) --> subset-sets(intersection-of-sets(X::'a,U),inte
&
(∀ 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*
&
(∀ *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

41 Examples for Ferrante and Rackoff's quantifier elimination procedure

theory *Dense-Linear-Order-Ex*
imports *Main*
begin

lemma
∃ (*y*::'*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) <*2*. *x* + *3*∗ *y* < *0* ∧ *x* − *y* >*0*
by *ferrack*

lemma ~ $(ALL\ x\ (y::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).$
$x < y \dashrightarrow 10*x < 11*y)$
by *ferrack*

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ x <$
$y \dashrightarrow (10*(x + 5*y + -1) < 60*y)$
by *ferrack*

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ x\ \tilde{}{=}$
$y \dashrightarrow x < y$
by *ferrack*

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ (x$
$\tilde{}{=}\ y\ \&\ 10*x\ \tilde{}{=}\ 9*y\ \&\ 10*x < y) \dashrightarrow x < y$
by *ferrack*

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ (x$
$\tilde{}{=}\ y\ \&\ 5*x <= y) \dashrightarrow 500*x <= 100*y$
by *ferrack*

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).\ (EX$
$(y::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).\ 4*x + 3*y <= 0$
$\&\ 4*x + 3*y >= -1)$
by *ferrack*

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}) < 0.$
$(EX\ (y::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}) > 0.\ 7*x + y$
$> 0\ \&\ x - y <= 9)$
by *ferrack*

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).\ (0 < x$
$\&\ x < 1) \dashrightarrow (ALL\ y > 1.\ x + y\ \tilde{}{=}\ 1)$
by *ferrack*

lemma $EX\ x.\ (ALL\ (y::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).$
$y < 2 \dashrightarrow\ 2*(y - x) \leq 0\ )$
by *ferrack*

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).\ x <$
$10\ |\ x > 20\ |\ (EX\ y.\ y>= 0\ \&\ y <= 10\ \&\ x+y = 20)$
by *ferrack*

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y\ z.\ x$
$+ y < z \dashrightarrow y >= z \dashrightarrow x < 0$
by *ferrack*

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y\ z.\ x$
$+ 7*y < 5*\ z\ \&\ 5*y >= 7*z\ \&\ x < 0$
by *ferrack*

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z. abs (x + y) <= z −−> (abs z = z)
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z. x + 7*y − 5* z < 0 & 5*y + 7*z + 3*x < 0
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z. (abs (5*x+3*y+z) <= 5*x+3*y+z & abs (5*x+3*y+z) >= − (5*x+3*y+z)) | (abs (5*x+3*y+z) >= 5*x+3*y+z & abs (5*x+3*y+z) <= − (5*x+3*y+z))
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. x < y −−> (EX z>0. x+z = y)
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. x < y −−> (EX z>0. x+z = y)
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. (EX z>0. abs (x − y) <= z)
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. (ALL z<0. (z < x −−> z <= y) & (z > y −−> z >= x))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. (ALL z>=0. abs (3*x+7*y) <= 2*z + 1)
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. (ALL z<0. (z < x −−> z <= y) & (z > y −−> z >= x))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero})>0. (ALL y. (EX z. 13* abs z ≠ abs (12*y − x) & 5*x − 3*(abs y) <= 7*z))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). abs (4*x + 17) < 4 & (ALL y . abs (x*34 − 34*y − 9) ≠ 0 ⟶ (EX z. 5*x − 3*abs y <= 7*z))
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX y > abs (23*x − 9). (ALL z > abs (3*y − 19* abs x). x+z > 2*y))

by *ferrack*

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX y< abs (3*x - 1). (ALL z >= (3*abs x - 1). abs (12*x - 13*y + 19*z) > abs (23*x)))
by *ferrack*

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). abs x < 100 & (ALL y > x. (EX z<2*y - x. 5*x - 3*y <= 7*z))
by *ferrack*

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z w. 7*x<3*y --> 5*y < 7*z --> z < 2*w --> 7*(2*w-x) > 2*y
by *ferrack*

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z w. 5*x + 3*z - 17*w + abs (y - 8*x + z) <= 89
by *ferrack*

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z w. 5*x + 3*z - 17*w + 7* (y - 8*x + z) <= max y (7*z - x + w)
by *ferrack*

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z w. min (5*x + 3*z) (17*w) + 5* abs (y - 8*x + z) <= max y (7*z - x + w)
by *ferrack*

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z. (EX w >= (x+y+z). w <= abs x + abs y + abs z)
by *ferrack*

lemma ~(ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (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::{ordered-field,recpower,number-ring, division-by-zero}) y. (EX z w. abs (x-y) = (z-w) & z*1234 < 233*x & w ~= y)
by *ferrack*

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX y z w. min (5*x + 3*z) (17*w) + 5* abs (y - 8*x + z) <= max y (7*z - x + w))
by *ferrack*

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z. (ALL w >= abs (x+y+z). w >= abs x + abs y + abs z)
by *ferrack*

lemma EX z. (ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero})

y. (EX w >= (x+y+z). w <= abs x + abs y + abs z))
by ferrack

lemma EX z. (ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) < abs z. (EX y w. x< y & x < z & x> w & 3*x < w + y))
by ferrack

lemma ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. (EX z. (ALL w. abs (x−y) = abs (z−w) −−> z < x & w ~= y))
by ferrack

lemma EX y. (ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX z w. min (5*x + 3*z) (17*w) + 5* abs (y − 8*x + z) <= max y (7*z − x + w)))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) z. (ALL w >= 13*x − 4*z. (EX y. w >= abs x + abs y + z))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (ALL y < x. (EX z > (x+y).
(ALL w. 5*w + 10*x − z >= y −−> w + 7*x + 3*z >= 2*y)))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (ALL y. (EX z > y.
(ALL w . w < 13 −−> w + 10*x − z >= y −−> 5*w + 7*x + 13*z >= 2*y)))
by ferrack

lemma EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z w. min (5*x + 3*z) (17*w) + 5* abs (y − 8*x + z) <= max y (7*z − 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 (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

42 Some examples for Presburger Arithmetic

theory *PresburgerEx*
imports *Presburger*
begin

lemma $\bigwedge m\ n\ ja\ ia.\ [\![\neg\ m \le j;\ \neg\ n \le i;\ e \ne 0;\ Suc\ j \le ja]\!] \Longrightarrow \exists\, m.\ \forall\, ja\ ia.\ m \le ja \longrightarrow (if\ j = ja \wedge i = ia\ then\ e\ else\ 0) = 0$ **by** *presburger*
lemma $(0::nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
by *presburger*
lemma $(0::nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
by *presburger*

theorem $(\forall\,(y::int).\ 3\ dvd\ y) ==> \forall\,(x::int).\ b < x --> a \le x$
by *presburger*

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 *presburger*

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 *presburger*

theorem $\forall\,(x::nat).\ \exists\,(y::nat).\ (0::nat) \le 5 --> y = 5 + x$
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 $\forall\,(x::int)\ y.\ x < y --> 2 * x + 1 < 2 * y$
by *presburger*

theorem $\forall\,(x::int)\ y.\ 2 * x + 1 \ne 2 * y$
by *presburger*

theorem $\exists\,(x::int)\ y.\ 0 < x\ \&\ 0 \le y\ \&\ 3 * x - 5 * y = 1$
by *presburger*

theorem $\sim\ (\exists\,(x::int)\ (y::int)\ (z::int).\ 4*x + (-6::int)*y = 1)$
by *presburger*

theorem ∀ (*x*::*int*). *b* < *x* −−> *a* ≤ *x*
  apply (*presburger elim*)
  oops

theorem ~ (∃ (*x*::*int*). *False*)
  by *presburger*

theorem ∀ (*x*::*int*). (*a*::*int*) < *3* ∗ *x* −−> *b* < *3* ∗ *x*
  apply (*presburger elim*)
  oops

theorem ∀ (*x*::*int*). (*2 dvd x*) −−> (∃ (*y*::*int*). *x* = *2*∗*y*)
  by *presburger*

theorem ∀ (*x*::*int*). (*2 dvd x*) −−> (∃ (*y*::*int*). *x* = *2*∗*y*)
  by *presburger*

theorem ∀ (*x*::*int*). (*2 dvd x*) = (∃ (*y*::*int*). *x* = *2*∗*y*)
  by *presburger*

theorem ∀ (*x*::*int*). ((*2 dvd x*) = (∀ (*y*::*int*). *x* ≠ *2*∗*y* + *1*))
  by *presburger*

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 *presburger*

theorem ~ (∀ (*i*::*int*). *4* ≤ *i* −−> (∃ *x y*. *0* ≤ *x* & *0* ≤ *y* & *3* ∗ *x* + *5* ∗ *y* = *i*))
  by *presburger*

theorem ∀ (*i*::*int*). *8* ≤ *i* −−> (∃ *x y*. *0* ≤ *x* & *0* ≤ *y* & *3* ∗ *x* + *5* ∗ *y* = *i*)
  by *presburger*

theorem ∃ (*j*::*int*). ∀ *i*. *j* ≤ *i* −−> (∃ *x y*. *0* ≤ *x* & *0* ≤ *y* & *3* ∗ *x* + *5* ∗ *y* = *i*)
  by *presburger*

theorem ~ (∀ *j* (*i*::*int*). *j* ≤ *i* −−> (∃ *x y*. *0* ≤ *x* & *0* ≤ *y* & *3* ∗ *x* + *5* ∗ *y* = *i*))
  by *presburger*

Slow: about 5 seconds on a 1.6GHz machine.

theorem (∃ *m*::*nat*. *n* = *2* ∗ *m*) −−> (*n* + *1*) *div 2* = *n 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 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 $[\![\, x3 = abs\ x2 - x1;\ x4 = abs\ x3 - x2;\ x5 = abs\ x4 - x3;$
$x6 = abs\ x5 - x4;\ x7 = abs\ x6 - x5;\ x8 = abs\ x7 - x6;$
$x9 = abs\ x8 - x7;\ x10 = abs\ x9 - x8;\ x11 = abs\ x10 - x9 \,]\!]$
$\Longrightarrow x1 = x10\ \&\ x2 = (x11::int)$
by *arith*

end

theory *Reflected-Presburger*
imports *GCD Efficient-Nat*
uses (*coopereif.ML*) (*coopertac.ML*)
begin

function
iupt :: *int* ⇒ *int* ⇒ *int list*
where
iupt i j = (*if j* < *i then* [] *else i* # *iupt* (*i*+*1*) *j*)
by *pat-completeness auto*
termination by (*relation measure* (λ (*i*, *j*). *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* ⇒ *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* ⇒ *num* ⇒ *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* ⇒ *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* ⇒ *int list* ⇒ *fm* ⇒ *bool*
primrec
Ifm bbs bs T = *True*
Ifm bbs bs F = *False*
Ifm bbs bs (*Lt a*) = (*Inum bs a* < *0*)
Ifm bbs bs (*Gt a*) = (*Inum bs a* > *0*)
Ifm bbs bs (*Le a*) = (*Inum bs a* ≤ *0*)
Ifm bbs bs (*Ge a*) = (*Inum bs a* ≥ *0*)
Ifm bbs bs (*Eq a*) = (*Inum bs a* = *0*)
Ifm bbs bs (*NEq a*) = (*Inum bs a* ≠ *0*)
Ifm bbs bs (*Dvd i b*) = (*i dvd Inum bs b*)
Ifm bbs bs (*NDvd i b*) = (¬(*i dvd Inum bs b*))
Ifm bbs bs (*NOT p*) = (¬ (*Ifm bbs bs p*))
Ifm bbs bs (*And p q*) = (*Ifm bbs bs p* ∧ *Ifm bbs bs q*)

$Ifm\ bbs\ bs\ (Or\ p\ q) = (Ifm\ bbs\ bs\ p \lor Ifm\ bbs\ bs\ q)$
$Ifm\ bbs\ bs\ (Imp\ p\ q) = ((Ifm\ bbs\ bs\ p) \longrightarrow (Ifm\ bbs\ bs\ q))$
$Ifm\ bbs\ bs\ (Iff\ p\ q) = (Ifm\ bbs\ bs\ p = Ifm\ bbs\ bs\ q)$
$Ifm\ bbs\ bs\ (E\ p) = (\exists\ x.\ Ifm\ bbs\ (x\#bs)\ p)$
$Ifm\ bbs\ bs\ (A\ p) = (\forall\ x.\ Ifm\ bbs\ (x\#bs)\ p)$
$Ifm\ bbs\ bs\ (Closed\ n) = bbs!n$
$Ifm\ bbs\ bs\ (NClosed\ n) = (\neg\ bbs!n)$

consts *prep* :: *fm* ⇒ *fm*
recdef *prep measure fmsize*
$prep\ (E\ T) = T$
$prep\ (E\ F) = F$
$prep\ (E\ (Or\ p\ q)) = Or\ (prep\ (E\ p))\ (prep\ (E\ q))$
$prep\ (E\ (Imp\ p\ q)) = Or\ (prep\ (E\ (NOT\ p)))\ (prep\ (E\ q))$
$prep\ (E\ (Iff\ p\ q)) = Or\ (prep\ (E\ (And\ p\ q)))\ (prep\ (E\ (And\ (NOT\ p)\ (NOT\ q))))$
$prep\ (E\ (NOT\ (And\ p\ q))) = Or\ (prep\ (E\ (NOT\ p)))\ (prep\ (E(NOT\ q)))$
$prep\ (E\ (NOT\ (Imp\ p\ q))) = prep\ (E\ (And\ p\ (NOT\ q)))$
$prep\ (E\ (NOT\ (Iff\ p\ q))) = Or\ (prep\ (E\ (And\ p\ (NOT\ q))))\ (prep\ (E(And\ (NOT\ p)\ q)))$
$prep\ (E\ p) = E\ (prep\ p)$
$prep\ (A\ (And\ p\ q)) = And\ (prep\ (A\ p))\ (prep\ (A\ q))$
$prep\ (A\ p) = prep\ (NOT\ (E\ (NOT\ p)))$
$prep\ (NOT\ (NOT\ p)) = prep\ p$
$prep\ (NOT\ (And\ p\ q)) = Or\ (prep\ (NOT\ p))\ (prep\ (NOT\ q))$
$prep\ (NOT\ (A\ p)) = prep\ (E\ (NOT\ p))$
$prep\ (NOT\ (Or\ p\ q)) = And\ (prep\ (NOT\ p))\ (prep\ (NOT\ q))$
$prep\ (NOT\ (Imp\ p\ q)) = And\ (prep\ p)\ (prep\ (NOT\ q))$
$prep\ (NOT\ (Iff\ p\ q)) = Or\ (prep\ (And\ p\ (NOT\ q)))\ (prep\ (And\ (NOT\ p)\ q))$
$prep\ (NOT\ p) = NOT\ (prep\ p)$
$prep\ (Or\ p\ q) = Or\ (prep\ p)\ (prep\ q)$
$prep\ (And\ p\ q) = And\ (prep\ p)\ (prep\ q)$
$prep\ (Imp\ p\ q) = prep\ (Or\ (NOT\ p)\ q)$
$prep\ (Iff\ p\ q) = Or\ (prep\ (And\ p\ q))\ (prep\ (And\ (NOT\ p)\ (NOT\ q)))$
$prep\ p = p$
(**hints** *simp add*: *fmsize-pos*)
lemma *prep*: $Ifm\ bbs\ bs\ (prep\ p) = Ifm\ bbs\ bs\ p$
by (*induct p arbitrary*: *bs rule*: *prep.induct*, *auto*)

consts *qfree*:: *fm* ⇒ *bool*
recdef *qfree measure size*
$qfree\ (E\ p) = False$
$qfree\ (A\ p) = False$
$qfree\ (NOT\ p) = qfree\ p$
$qfree\ (And\ p\ q) = (qfree\ p \land qfree\ q)$
$qfree\ (Or\ p\ q) = (qfree\ p \land qfree\ q)$
$qfree\ (Imp\ p\ q) = (qfree\ p \land qfree\ q)$

qfree (*Iff p q*) = (*qfree p* ∧ *qfree q*)
qfree p = *True*

consts
numbound0:: *num* ⇒ *bool*
bound0:: *fm* ⇒ *bool*
subst0:: *num* ⇒ *fm* ⇒ *fm*
primrec
numbound0 (*C c*) = *True*
numbound0 (*Bound n*) = (*n*>*0*)
numbound0 (*CN n i a*) = (*n* >*0* ∧ *numbound0 a*)
numbound0 (*Neg a*) = *numbound0 a*
numbound0 (*Add a b*) = (*numbound0 a* ∧ *numbound0 b*)
numbound0 (*Sub a b*) = (*numbound0 a* ∧ *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* ∧ *bound0 q*)
bound0 (*Or p q*) = (*bound0 p* ∧ *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* ⇒ *num*
decr :: *fm* ⇒ *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* $\Longrightarrow$ *qfree p*
by (*induct p*, *simp-all*)

constdefs *djf*:: ($'a \Rightarrow fm$) $\Rightarrow$ $'a \Rightarrow fm \Rightarrow fm$
djf f p q ≡ (*if q*=*T then T else if q*=*F then f p else*
(*let fp* = *f p in case fp of T* ⇒ *T* | *F* ⇒ *q* | - ⇒ *Or* (*f p*) *q*))
constdefs *evaldjf*:: ($'a \Rightarrow fm$) $\Rightarrow$ $'a$ *list* ⇒ *fm*
evaldjf f ps ≡ *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*) = (∃ *p* ∈ *set ps*. *Ifm bbs bs* (*f p*))
by(*induct ps*, *simp-all add*: *evaldjf-def djf-Or*)

lemma *evaldjf-bound0*:
assumes *nb*: ∀ *x*∈ *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*: ∀ *x*∈ *set xs*. *qfree* (*f x*)
shows *qfree* (*evaldjf f xs*)
using *nb* **by** (*induct xs*, *auto simp add*: *evaldjf-def djf-def Let-def*) (*case-tac f a*, *auto*)

consts *disjuncts* :: *fm* ⇒ *fm list*
recdef *disjuncts measure size*
disjuncts (*Or p q*) = (*disjuncts p*) @ (*disjuncts q*)
disjuncts F = []

disjuncts p = [*p*]

lemma *disjuncts*: (∃ *q*∈ *set* (*disjuncts p*). *Ifm bbs bs q*) = *Ifm bbs bs p*
by(*induct p rule*: *disjuncts.induct*, *auto*)

lemma *disjuncts-nb*: *bound0 p* ⟹ ∀ *q*∈ *set* (*disjuncts p*). *bound0 q*
proof−
  assume *nb*: *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*: *qfree p* ⟹ ∀ *q*∈ *set* (*disjuncts p*). *qfree q*
proof−
  assume *qf*: *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* ⇒ *fm*) ⇒ *fm* ⇒ *fm*
  DJ f p ≡ *evaldjf f* (*disjuncts p*)

lemma *DJ*: **assumes** *fdj*: ∀ *p q*. *f* (*Or p q*) = *Or* (*f p*) (*f q*)
  and *fF*: *f F* = *F*
  shows *Ifm bbs bs* (*DJ f p*) = *Ifm bbs bs* (*f p*)
proof−
  have *Ifm bbs bs* (*DJ f p*) = (∃ *q* ∈ *set* (*disjuncts p*). *Ifm bbs bs* (*f q*))
    by (*simp add*: *DJ-def evaldjf-ex*)
  also have ... = *Ifm bbs bs* (*f p*) **using** *fdj fF* **by** (*induct p rule*: *disjuncts.induct*, *auto*)
  finally show *?thesis* **.**
qed

lemma *DJ-qf*: **assumes**
  fqf: ∀ *p*. *qfree p* ⟶ *qfree* (*f p*)
  shows ∀ *p*. *qfree p* ⟶ *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** ∀ *q*∈ *set* (*disjuncts p*). *qfree q* **.**
  with *fqf* **have** *th'*:∀ *q*∈ *set* (*disjuncts p*). *qfree* (*f q*) **by** *blast*

  from *evaldjf-qf*[*OF th'*] *th* **show** *qfree* (*DJ f p*) **by** *simp*
qed

lemma *DJ-qe*: **assumes** *qe*: ∀ *bs p*. *qfree p* ⟶ *qfree* (*qe p*) ∧ (*Ifm bbs bs* (*qe p*) = *Ifm bbs bs* (*E p*))
  shows ∀ *bs p*. *qfree p* ⟶ *qfree* (*DJ qe p*) ∧ (*Ifm bbs bs* ((*DJ qe p*)) = *Ifm bbs*

bs (*E p*))
proof(*clarify*)
fix *p*::*fm* **and** *bs*
assume *qf*: *qfree p*
from *qe* **have** *qth*: $\forall$ *p*. *qfree p* $\longrightarrow$ *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*$\in$ *set* (*disjuncts p*). *Ifm bbs bs* (*qe q*))
by (*simp add*: *DJ-def evaldjf-ex*)
also have ... = ($\exists$ *q* $\in$ *set*(*disjuncts p*). *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* (λ (*xs*,*ys*). *length xs* + *length ys*)
lex-ns ([], *ms*) = *True*
lex-ns (*ns*, []) = *False*
lex-ns (*n*#*ns*, *m*#*ms*) = (*n*<*m* $\vee$ ((*n* = *m*) $\wedge$ *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* (λ (*t*,*s*). *num-size t* + *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 $\leq$ *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*: ⟦ *numbound0 t* ; *numbound0 s*⟧ ⟹ *numbound0* (*numsub t s*)
using *numsub-def numadd-nb numneg-nb* **by** *simp*

fun
  simpnum :: *num* ⇒ *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 ⟹ *numbound0* (*simpnum t*)
by (*induct t rule*: *simpnum.induct*, *auto simp add*: *numadd-nb numsub-nb nummul-nb numneg-nb*)

fun
not :: *fm* ⇒ *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* ⟹ *qfree* (*not p*)
by (*cases p*, *auto*)
lemma *not-bn*: *bound0 p* ⟹ *bound0* (*not p*)
by (*cases p*, *auto*)

constdefs *conj* :: *fm* ⇒ *fm* ⇒ *fm*
conj p q ≡ (*if* (*p* = *F* ∨ *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* ∨ *q*=*F*,*simp-all add*: *conj-def*) (*cases p*,*simp-all*)

lemma *conj-qf*: ⟦*qfree p* ; *qfree q*⟧ ⟹ *qfree* (*conj p q*)
using *conj-def* **by** *auto*
lemma *conj-nb*: ⟦*bound0 p* ; *bound0 q*⟧ ⟹ *bound0* (*conj p q*)
using *conj-def* **by** *auto*

constdefs *disj* :: *fm* ⇒ *fm* ⇒ *fm*
disj p q ≡ (*if* (*p* = *T* ∨ *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* (*disj p q*) = *Ifm bbs bs* (*Or p q*)
by (*cases p*=*T* ∨ *q*=*T*,*simp-all add*: *disj-def*) (*cases p*,*simp-all*)
lemma *disj-qf*: ⟦*qfree p* ; *qfree q*⟧ ⟹ *qfree* (*disj p q*)
using *disj-def* **by** *auto*
lemma *disj-nb*: ⟦*bound0 p* ; *bound0 q*⟧ ⟹ *bound0* (*disj p q*)

using *disj-def* **by** *auto*

constdefs *imp* :: *fm* ⇒ *fm* ⇒ *fm*
imp p q ≡ (*if* (*p* = *F* ∨ *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* (*imp p q*) = *Ifm bbs bs* (*Imp p q*)
by (*cases p*=*F* ∨ *q*=*T*,*simp-all add*: *imp-def*,*cases p*) (*simp-all add*: *not*)
lemma *imp-qf*: ⟦*qfree p* ; *qfree q*⟧ ⟹ *qfree* (*imp p q*)
using *imp-def* **by** (*cases p*=*F* ∨ *q*=*T*,*simp-all add*: *imp-def*,*cases p*) (*simp-all add*: *not-qf*)
lemma *imp-nb*: ⟦*bound0 p* ; *bound0 q*⟧ ⟹ *bound0* (*imp p q*)
using *imp-def* **by** (*cases p*=*F* ∨ *q*=*T*,*simp-all add*: *imp-def*,*cases p*) *simp-all*

constdefs *iff* :: *fm* ⇒ *fm* ⇒ *fm*
iff p q ≡ (*if* (*p* = *q*) *then T else if* (*p* = *not q* ∨ *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* (*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*: ⟦*qfree p* ; *qfree q*⟧ ⟹ *qfree* (*iff p q*)
by (*unfold iff-def*,*cases p*=*q*, *auto simp add*: *not-qf*)
lemma *iff-nb*: ⟦*bound0 p* ; *bound0 q*⟧ ⟹ *bound0* (*iff p q*)
using *iff-def* **by** (*unfold iff-def*,*cases p*=*q*, *auto simp add*: *not-bn*)

function (*sequential*)
simpfm :: *fm* ⇒ *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′* = *simpnum a in case a′ of C v* ⇒ *if* (*v* < *0*) *then T else F*
| - ⇒ *Lt a′*)
| *simpfm* (*Le a*) = (*let a′* = *simpnum a in case a′ of C v* ⇒ *if* (*v* ≤ *0*) *then T else F* | - ⇒ *Le a′*)
| *simpfm* (*Gt a*) = (*let a′* = *simpnum a in case a′ of C v* ⇒ *if* (*v* > *0*) *then T else F* | - ⇒ *Gt a′*)
| *simpfm* (*Ge a*) = (*let a′* = *simpnum a in case a′ of C v* ⇒ *if* (*v* ≥ *0*) *then T else F* | - ⇒ *Ge a′*)
| *simpfm* (*Eq a*) = (*let a′* = *simpnum a in case a′ of C v* ⇒ *if* (*v* = *0*) *then T else F* | - ⇒ *Eq a′*)
| *simpfm* (*NEq a*) = (*let a′* = *simpnum a in case a′ of C v* ⇒ *if* (*v* ≠ *0*) *then T else F* | - ⇒ *NEq a′*)
| *simpfm* (*Dvd i a*) = (*if i*=*0 then simpfm* (*Eq a*)
else if (*abs i* = *1*) *then T*

else let a' = simpnum a in case a' of C v ⇒ if (i dvd v) then T else F | - ⇒ Dvd i a')
| *simpfm (NDvd i a) = (if i=0 then simpfm (NEq a)*
else if (abs i = 1) then F
else let a' = simpnum a in case a' of C v ⇒ if (¬(i dvd v)) then T else F | - ⇒ NDvd i a')
| *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*)
case (*6 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 ¬ (∃ *v*. *?sa* = *C v*) **hence** *?case* **using** *sa*
by (*cases ?sa*, *simp-all add*: *Let-def*)**}**
ultimately show *?case* **by** *blast*
next
case (*7 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 ¬ (∃ *v*. *?sa* = *C v*) **hence** *?case* **using** *sa*
by (*cases ?sa*, *simp-all add*: *Let-def*)**}**
ultimately show *?case* **by** *blast*
next
case (*8 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 ¬ (∃ *v*. *?sa* = *C v*) **hence** *?case* **using** *sa*
by (*cases ?sa*, *simp-all add*: *Let-def*)**}**
ultimately show *?case* **by** *blast*
next
case (*9 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 ¬ (∃ *v*. *?sa* = *C v*) **hence** *?case* **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 ¬ (∃ *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** ¬ (∃ *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* ∨ *abs i* = *1* ∨ (*i*≠*0* ∧ (*abs i* ≠ *1*)) **by** *auto*
{**assume** *i=0* **hence** *?case* **using** *12.hyps* **by** (*simp add*: *dvd-def Let-def*)}
moreover
{**assume** *i1*: *abs i* = *1*
from *zdvd-1-left*[**where** *m* = *Inum bs a*] *uminus-dvd-conv*[**where** *d=1* **and** *t=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*≠*0* **and** *cond*: *abs i* ≠ *1*
{**fix** *v* **assume** *?sa* = *C v* **hence** *?case* **using** *sa*[*symmetric*] *inz cond*
by (*cases abs i* = *1*, *auto*) }
moreover {**assume** ¬ (∃ *v*. *?sa* = *C v*)
hence *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* ∨ *abs i* = *1* ∨ (*i*≠*0* ∧ (*abs i* ≠ *1*)) **by** *auto*
{**assume** *i=0* **hence** *?case* **using** *13.hyps* **by** (*simp add*: *dvd-def Let-def*)}
moreover
{**assume** *i1*: *abs i* = *1*
from *zdvd-1-left*[**where** *m* = *Inum bs a*] *uminus-dvd-conv*[**where** *d=1* **and** *t=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*≠*0* **and** *cond*: *abs i* ≠ *1*
{**fix** *v* **assume** *?sa* = *C v* **hence** *?case* **using** *sa*[*symmetric*] *inz cond*
by (*cases abs i* = *1*, *auto*) }
moreover {**assume** ¬ (∃ *v*. *?sa* = *C v*)
hence *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*: *bound0 p* ⟹ *bound0* (*simpfm p*)

proof(*induct p rule*: *simpfm.induct*)
  case (*6 a*) **hence** *nb*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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*: *numbound0 a* **by** *simp*
  hence *numbound0* (*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* $\Longrightarrow$ *qfree* (*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*: ∀ *bs p*. *qfree p* ⟶ *qfree* (*qe p*) ∧ (*Ifm bbs bs* (*qe p*) = *Ifm bbs bs* (*E p*))
shows ⋀ *bs*. *qfree* (*qelim p qe*) ∧ (*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
zsplit0 :: *num* ⇒ *int* × *num*
where
zsplit0 (*C c*) = (*0*,*C c*)
| *zsplit0* (*Bound n*) = (*if n*=*0 then* (*1*, *C 0*) *else* (*0*,*Bound n*))
| *zsplit0* (*CN n i a*) =
(*let* (*i'*,*a'*) = *zsplit0 a*
in if n=*0 then* (*i*+*i'*, *a'*) *else* (*i'*,*CN n i a'*))
| *zsplit0* (*Neg a*) = (*let* (*i'*,*a'*) = *zsplit0 a in* (−*i'*, *Neg a'*))
| *zsplit0* (*Add a b*) = (*let* (*ia*,*a'*) = *zsplit0 a* ;
(*ib*,*b'*) = *zsplit0 b*
in (*ia*+*ib*, *Add a' b'*))
| *zsplit0* (*Sub a b*) = (*let* (*ia*,*a'*) = *zsplit0 a* ;
(*ib*,*b'*) = *zsplit0 b*
in (*ia*−*ib*, *Sub a' b'*))
| *zsplit0* (*Mul i a*) = (*let* (*i'*,*a'*) = *zsplit0 a in* (*i*∗*i'*, *Mul i a'*))

lemma *zsplit0-I*:
shows ⋀ *n a*. *zsplit0 t* = (*n*,*a*) ⟹ (*Inum* ((*x*::*int*) #*bs*) (*CN 0 n a*) = *Inum* (*x* #*bs*) *t*) ∧ *numbound0 a*
(**is** ⋀ *n a*. *?S t* = (*n*,*a*) ⟹ (*?I x* (*CN 0 n a*) = *?I x t*) ∧ *?N a*)
proof(*induct t rule*: *zsplit0.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* (*zsplit0 a*)
let *?b* = *snd* (*zsplit0 a*)
have *abj*: *zsplit0 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* (*zsplit0 t*)
let *?at* = *snd* (*zsplit0 t*)
have *abj*: *zsplit0 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* (*zsplit0 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*=*Add ?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-distrib*)
next
case (*6 s t n a*)
let *?ns* = *fst* (*zsplit0 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) = zsplit0 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) = zsplit0 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*) = *zsplit0 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*) = *zsplit0 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*) = *zsplit0 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*) = *zsplit0 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*) = *zsplit0 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*) = *zsplit0 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*) ∧ *iszlfm* (*zlfm p*)
(**is** (*?I* (*?l p*) = *?I p*) ∧ *?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* = 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 (*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* = 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 (*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* = 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 (*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}
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*)
δ (*Or p q*) = *ilcm* (δ *p*) (δ *q*)
δ (*Dvd i* (*CN 0 c e*)) = *i*
δ (*NDvd i* (*CN 0 c e*)) = *i*
δ *p* = *1*

recdef *d*δ *measure size*
dδ (*And p q*) = (λ *d*. *d*δ *p d* ∧ *d*δ *q d*)
dδ (*Or p q*) = (λ *d*. *d*δ *p d* ∧ *d*δ *q d*)
dδ (*Dvd i* (*CN 0 c e*)) = (λ *d*. *i dvd d*)
dδ (*NDvd i* (*CN 0 c e*)) = (λ *d*. *i dvd d*)
dδ *p* = (λ *d*. *True*)

lemma *delta-mono*:
  assumes *lin*: *iszlfm p*
  and *d*: *d dvd d′*
  and *ad*: *dδ p d*
  shows *dδ 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 *δ* : **assumes** *lin*:*iszlfm p*
  shows *dδ p* (*δ p*) $\wedge$ *δ p* >*0*
using *lin*
proof (*induct p rule*: *iszlfm.induct*)
  case (*1 p q*)
  let *?d* = *δ* (*And p q*)
  from *prems ilcm-pos* **have** *dp*: *?d* >*0* **by** *simp*
  have *d1*: *δ p dvd δ* (*And p q*) **using** *prems* **by** *simp*
  hence *th*: *dδ p ?d* **using** *delta-mono prems*(*3*−*4*) **by**(*simp del*:*dvd-ilcm-self1*)
  have *δ q dvd δ* (*And p q*) **using** *prems* **by** *simp*
  hence *th′*: *dδ 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* = *δ* (*And p q*)
  from *prems ilcm-pos* **have** *dp*: *?d* >*0* **by** *simp*
  have *δ p dvd δ* (*And p q*) **using** *prems* **by** *simp*
  hence *th*: *dδ p ?d* **using** *delta-mono prems* **by**(*simp del*:*dvd-ilcm-self1*)
  have *δ q dvd δ* (*And p q*) **using** *prems* **by** *simp*
  hence *th′*: *dδ q ?d* **using** *delta-mono prems* **by**(*simp del*:*dvd-ilcm-self2*)
  from *th th′ dp* **show** *?case* **by** *simp*
qed *simp-all*

consts
  aβ :: *fm* ⇒ *int* ⇒ *fm*
  dβ :: *fm* ⇒ *int* ⇒ *bool*
  ζ :: *fm* ⇒ *int*
  β :: *fm* ⇒ *num list*
  α :: *fm* ⇒ *num list*

recdef *aβ measure size*
  aβ (*And p q*) = (*λ k*. *And* (*aβ p k*) (*aβ q k*))
  aβ (*Or p q*) = (*λ k*. *Or* (*aβ p k*) (*aβ q k*))
  aβ (*Eq* (*CN 0 c e*)) = (*λ k*. *Eq* (*CN 0 1* (*Mul* (*k div c*) *e*)))

```
  aβ (NEq (CN 0 c e)) = (λ k. NEq (CN 0 1 (Mul (k div c) e)))
  aβ (Lt  (CN 0 c e)) = (λ k. Lt (CN 0 1 (Mul (k div c) e)))
  aβ (Le  (CN 0 c e)) = (λ k. Le (CN 0 1 (Mul (k div c) e)))
  aβ (Gt  (CN 0 c e)) = (λ k. Gt (CN 0 1 (Mul (k div c) e)))
  aβ (Ge  (CN 0 c e)) = (λ k. Ge (CN 0 1 (Mul (k div c) e)))
  aβ (Dvd i (CN 0 c e)) =(λ k. Dvd ((k div c)∗i) (CN 0 1 (Mul (k div c) e)))
  aβ (NDvd i (CN 0 c e))=(λ k. NDvd ((k div c)∗i) (CN 0 1 (Mul (k div c) e)))
  aβ p = (λ k. p)

recdef dβ measure size
  dβ (And p q) = (λ k. (dβ p k) ∧ (dβ q k))
  dβ (Or p q) = (λ k. (dβ p k) ∧ (dβ q k))
  dβ (Eq  (CN 0 c e)) = (λ k. c dvd k)
  dβ (NEq (CN 0 c e)) = (λ k. c dvd k)
  dβ (Lt  (CN 0 c e)) = (λ k. c dvd k)
  dβ (Le  (CN 0 c e)) = (λ k. c dvd k)
  dβ (Gt  (CN 0 c e)) = (λ k. c dvd k)
  dβ (Ge  (CN 0 c e)) = (λ k. c dvd k)
  dβ (Dvd i (CN 0 c e)) =(λ k. c dvd k)
  dβ (NDvd i (CN 0 c e))=(λ k. c dvd k)
  dβ p = (λ k. True)

recdef ζ measure size
  ζ (And p q) = ilcm (ζ p) (ζ q)
  ζ (Or p q) = ilcm (ζ p) (ζ q)
  ζ (Eq  (CN 0 c e)) = c
  ζ (NEq (CN 0 c e)) = c
  ζ (Lt  (CN 0 c e)) = c
  ζ (Le  (CN 0 c e)) = c
  ζ (Gt  (CN 0 c e)) = c
  ζ (Ge  (CN 0 c e)) = c
  ζ (Dvd i (CN 0 c e)) = c
  ζ (NDvd i (CN 0 c e))= c
  ζ p = 1

recdef β measure size
  β (And p q) = (β p @ β q)
  β (Or p q) = (β p @ β q)
  β (Eq  (CN 0 c e)) = [Sub (C −1) e]
  β (NEq (CN 0 c e)) = [Neg e]
  β (Lt  (CN 0 c e)) = []
  β (Le  (CN 0 c e)) = []
  β (Gt  (CN 0 c e)) = [Neg e]
  β (Ge  (CN 0 c e)) = [Sub (C −1) e]
  β p = []

recdef α measure size
  α (And p q) = (α p @ α q)
  α (Or p q) = (α p @ α q)
```

α (*Eq* (*CN 0 c e*)) = [*Add* (*C* −*1*) *e*]
α (*NEq* (*CN 0 c e*)) = [*e*]
α (*Lt* (*CN 0 c e*)) = [*e*]
α (*Le* (*CN 0 c e*)) = [*Add* (*C* −*1*) *e*]
α (*Gt* (*CN 0 c e*)) = []
α (*Ge* (*CN 0 c e*)) = []
α *p* = []
consts *mirror* :: *fm* ⇒ *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* ⟹ (*x*::*int*) *dvd 1* = (*x* = *1*)
by *auto*

lemma *minusinf-inf*:
assumes *linp*: *iszlfm p*
and *u*: *dβ p 1*
shows ∃ (*z*::*int*). ∀ *x* < *z*. *Ifm bbs* (*x*#*bs*) (*minusinf p*) = *Ifm bbs* (*x*#*bs*) *p*
(**is** *?P p* **is** ∃ (*z*::*int*). ∀ *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 za* **in** *exI*,*simp*)
next
case (*2 p q*) **thus** *?case*
by (*auto simp add*: *dvd1-eq1*) (*rule-tac x*=*min z za* **in** *exI*,*simp*)
next
case (*3 c e*) **hence** *c1*: *c*=*1* **and** *nb*: *numbound0 e* **using** *dvd1-eq1* **by** *simp*+
hence ∀ *x*<(− *Inum* (*a*#*bs*) *e*). *c*∗*x* + *Inum* (*x*#*bs*) *e* ≠ *0*
proof(*clarsimp*)
fix *x* **assume** *x* < (− *Inum* (*a*#*bs*) *e*) **and***x* + *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*+
hence ∀ *x*<(− *Inum* (*a*#*bs*) *e*). *c*∗*x* + *Inum* (*x*#*bs*) *e* ≠ *0*
proof(*clarsimp*)

fix *x* **assume** $x < (- \mathit{Inum}\ (a\#bs)\ e)$ **and**$x + \mathit{Inum}\ (x\#bs)\ e = 0$
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*+
hence $\forall\ x<(- \mathit{Inum}\ (a\#bs)\ e).\ c*x + \mathit{Inum}\ (x\#bs)\ e < 0$
proof(*clarsimp*)
fix *x* **assume** $x < (- \mathit{Inum}\ (a\#bs)\ e)$
with *numbound0-I*[*OF nb*, **where** *bs*=*bs* **and** *b*=*a* **and** *b'*=*x*]
show $x + \mathit{Inum}\ (x\#bs)\ e < 0$ **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*+
hence $\forall\ x<(- \mathit{Inum}\ (a\#bs)\ e).\ c*x + \mathit{Inum}\ (x\#bs)\ e \leq 0$
proof(*clarsimp*)
fix *x* **assume** $x < (- \mathit{Inum}\ (a\#bs)\ e)$
with *numbound0-I*[*OF nb*, **where** *bs*=*bs* **and** *b*=*a* **and** *b'*=*x*]
show $x + \mathit{Inum}\ (x\#bs)\ e \leq 0$ **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*+
hence $\forall\ x<(- \mathit{Inum}\ (a\#bs)\ e).\ \neg\ (c*x + \mathit{Inum}\ (x\#bs)\ e > 0)$
proof(*clarsimp*)
fix *x* **assume** $x < (- \mathit{Inum}\ (a\#bs)\ e)$ **and**$x + \mathit{Inum}\ (x\#bs)\ e > 0$
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*+
hence $\forall\ x<(- \mathit{Inum}\ (a\#bs)\ e).\ \neg\ (c*x + \mathit{Inum}\ (x\#bs)\ e \geq 0)$
proof(*clarsimp*)
fix *x* **assume** $x < (- \mathit{Inum}\ (a\#bs)\ e)$ **and**$x + \mathit{Inum}\ (x\#bs)\ e \geq 0$
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*: *iszlfm p*
shows $\mathit{Ifm}\ bbs\ ((x - k*d)\#bs)\ (\mathit{minusinf}\ p) = \mathit{Ifm}\ bbs\ (x\ \#bs)\ (\mathit{minusinf}\ p)$
using *linp d*
proof(*induct p rule*: *iszlfm.induct*)
case (*9 i c e*) **hence** *nbe*: *numbound0 e* **and** *id*: *i dvd d* **by** *simp*+

hence ∃ *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 ∃ (*l*::*int*). *?rt* = *i* ∗ *l* **by** (*simp add*: *dvd-def*)
hence ∃ (*l*::*int*). *c*∗*x*+ *?I x e* = *i*∗*l*+*c*∗(*k* ∗ *i*∗*di*)
by (*simp add*: *ring-simps di-def*)
hence ∃ (*l*::*int*). *c*∗*x*+ *?I x e* = *i*∗(*l* + *c*∗*k*∗*di*)
by (*simp add*: *ring-simps*)
hence ∃ (*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 ∃ (*l*::*int*). *c*∗*x*+*?e* = *i*∗*l* **by** (*simp add*: *dvd-def*)
hence ∃ (*l*::*int*). *c*∗*x* − *c*∗(*k*∗*d*) +*?e* = *i*∗*l* − *c*∗(*k*∗*d*) **by** *simp*
hence ∃ (*l*::*int*). *c*∗*x* − *c*∗(*k*∗*d*) +*?e* = *i*∗*l* − *c*∗(*k*∗*i*∗*di*) **by** (*simp add*: *di-def*)
hence ∃ (*l*::*int*). *c*∗*x* − *c*∗(*k*∗*d*) +*?e* = *i*∗((*l* − *c*∗*k*∗*di*)) **by** (*simp add*: *ring-simps*)
hence ∃ (*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
next
case (*10 i c e*) **hence** *nbe*: *numbound0 e* **and** *id*: *i dvd d* **by** *simp*+
hence ∃ *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 ∃ (*l*::*int*). *?rt* = *i* ∗ *l* **by** (*simp add*: *dvd-def*)
hence ∃ (*l*::*int*). *c*∗*x*+ *?I x e* = *i*∗*l*+*c*∗(*k* ∗ *i*∗*di*)
by (*simp add*: *ring-simps di-def*)
hence ∃ (*l*::*int*). *c*∗*x*+ *?I x e* = *i*∗(*l* + *c*∗*k*∗*di*)
by (*simp add*: *ring-simps*)
hence ∃ (*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 ∃ (*l*::*int*). *c*∗*x*+*?e* = *i*∗*l* **by** (*simp add*: *dvd-def*)

hence ∃ (*l*::*int*). *c*∗*x* − *c*∗(*k*∗*d*) +*?e* = *i*∗*l* − *c*∗(*k*∗*d*) **by** *simp*
hence ∃ (*l*::*int*). *c*∗*x* − *c*∗(*k*∗*d*) +*?e* = *i*∗*l* − *c*∗(*k*∗*i*∗*di*) **by** (*simp add*: *di-def*)
hence ∃ (*l*::*int*). *c*∗*x* − *c*∗(*k*∗*d*) +*?e* = *i*∗((*l* − *c*∗*k*∗*di*)) **by** (*simp add*: *ring-simps*)
hence ∃ (*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 *minusinf-ex*:
assumes *lin*: *iszlfm p* **and** *u*: *dβ p 1*
and *exmi*: ∃ (*x*::*int*). *Ifm bbs* (*x*#*bs*) (*minusinf p*) (**is** ∃ *x*. *?P1 x*)
shows ∃ (*x*::*int*). *Ifm bbs* (*x*#*bs*) *p* (**is** ∃ *x*. *?P x*)
proof−
let *?d* = *δ p*
from *δ* [*OF lin*] **have** *dpos*: *?d* >*0* **by** *simp*
from *δ* [*OF lin*] **have** *alld*: *dδ p ?d* **by** *simp*
from *minusinf-repeats*[*OF alld lin*] **have** *th1*:∀ *x k*. *?P1 x* = *?P1* (*x* − (*k* ∗ *?d*)) **by** *simp*
from *minusinf-inf*[*OF lin u*] **have** *th2*:∃ *z*. ∀ *x*. *x*<*z* ⟶ (*?P x* = *?P1 x*) **by** *blast*
from *minusinfinity* [*OF dpos th1 th2*] *exmi* **show** *?thesis* **by** *blast*
qed

lemma *minusinf-bex*:
assumes *lin*: *iszlfm p*
shows (∃ (*x*::*int*). *Ifm bbs* (*x*#*bs*) (*minusinf p*)) =
(∃ (*x*::*int*)∈ {*1*..*δ p*}. *Ifm bbs* (*x*#*bs*) (*minusinf p*))
(**is** (∃ *x*. *?P x*) = -)
proof−
let *?d* = *δ p*
from *δ* [*OF lin*] **have** *dpos*: *?d* >*0* **by** *simp*
from *δ* [*OF lin*] **have** *alld*: *dδ p ?d* **by** *simp*
from *minusinf-repeats*[*OF alld lin*] **have** *th1*:∀ *x k*. *?P x* = *?P* (*x* − (*k* ∗ *?d*)) **by** *simp*
from *periodic-finite-ex*[*OF dpos th1*] **show** *?thesis* **by** *blast*
qed

lemma *mirrorαβ*:
assumes *lp*: *iszlfm p*
shows (*Inum* (*i*#*bs*)) ‘ *set* (*α p*) = (*Inum* (*i*#*bs*)) ‘ *set* (*β* (*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*)
      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* .
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* ∧ *dβ p 1*
  =⇒ *iszlfm* (*mirror p*) ∧ *dβ* (*mirror p*) *1*
by (*induct p rule*: *mirror.induct*, *auto*)

lemma *mirror-δ*: *iszlfm p* =⇒ *δ* (*mirror p*) = *δ p*
by (*induct p rule*: *mirror.induct*,*auto*)

lemma *β-numbound0*: **assumes** *lp*: *iszlfm p*
  shows ∀ *b*∈ *set* (*β p*). *numbound0 b*
  using *lp* **by** (*induct p rule*: *β.induct*,*auto*)

lemma *dβ-mono*:
  assumes *linp*: *iszlfm p*
  and *dr*: *dβ 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 α-*l*: **assumes** *lp*: *iszlfm p*
shows $\forall$ $b\in$ *set* (α p). *numbound0 b*
using *lp*
by(*induct p rule*: α.*induct*, *auto*)

lemma ζ:
assumes *linp*: *iszlfm p*
shows ζ p > *0* $\wedge$ $d\beta$ p (ζ p)
using *linp*
proof(*induct p rule*: *iszlfm.induct*)
case (*1 p q*)
from *prems* **have** *dl1*: ζ p *dvd ilcm* (ζ p) (ζ q) **by** *simp*
from *prems* **have** *dl2*: ζ q *dvd ilcm* (ζ p) (ζ q) **by** *simp*
from *prems* $d\beta$-*mono*[**where** p = p **and** l=ζ p **and** l'=*ilcm* (ζ p) (ζ q)]
$d\beta$-*mono*[**where** p = q **and** l=ζ q **and** l'=*ilcm* (ζ p) (ζ q)]
dl1 dl2 **show** *?case* **by** (*auto simp add*: *ilcm-pos*)
next
case (*2 p q*)
from *prems* **have** *dl1*: ζ p *dvd ilcm* (ζ p) (ζ q) **by** *simp*
from *prems* **have** *dl2*: ζ q *dvd ilcm* (ζ p) (ζ q) **by** *simp*
from *prems* $d\beta$-*mono*[**where** p = p **and** l=ζ p **and** l'=*ilcm* (ζ p) (ζ q)]
$d\beta$-*mono*[**where** p = q **and** l=ζ q **and** l'=*ilcm* (ζ p) (ζ q)]
dl1 dl2 **show** *?case* **by** (*auto simp add*: *ilcm-pos*)
qed (*auto simp add*: *ilcm-pos*)

lemma $a\beta$: **assumes** *linp*: *iszlfm p* **and** d: $d\beta$ p l **and** *lp*: l > *0*
shows *iszlfm* ($a\beta$ p l) $\wedge$ $d\beta$ ($a\beta$ p l) *1* $\wedge$ (*Ifm bbs* (l*x #*bs*) ($a\beta$ p l) = *Ifm bbs* (x#*bs*) p)
using *linp d*
proof (*induct p rule*: *iszlfm.induct*)
case (*5 c e*) **hence** *cp*: c>*0* **and** *be*: *numbound0 e* **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 *div* $c\leq$ l *div* c
by (*simp add*: *zdiv-mono1* [*OF clel cp*])
then have *ldcp*:*0* < l *div* c
by (*simp add*: *zdiv-self* [*OF cnz*])
have c * (l *div* c) = c* (l *div* c) + l *mod* c **using** d' *zdvd-iff-zmod-eq-0*[**where** m=c **and** n=l] **by** *simp*
hence *cl*:c * (l *div* c) =l **using** *zmod-zdiv-equality*[**where** a=l **and** b=c, *symmetric*]
by *simp*
hence (l*x + (l *div* c) * *Inum* (x # *bs*) e < *0*) =
((c * (l *div* c)) * x + (l *div* c) * *Inum* (x # *bs*) e < *0*)

by *simp*
also have ... = ((*l div c*) ∗ (*c*∗*x* + *Inum* (*x* # *bs*) *e*) < (*l div c*) ∗ *0*) **by** (*simp add*: *ring-simps*)
also have ... = (*c*∗*x* + *Inum* (*x* # *bs*) *e* < *0*)
using *mult-less-0-iff* [**where** *a*=(*l 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 (*6 c e*) **hence** *cp*: *c*>*0* **and** *be*: *numbound0 e* **and** *d'*: *c dvd l* **by** *simp*+
from *lp cp* **have** *clel*: *c*≤*l* **by** (*simp add*: *zdvd-imp-le* [*OF d' lp*])
from *cp* **have** *cnz*: *c* ≠ *0* **by** *simp*
have *c div c*≤ *l div c*
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:*0* < *l div c*
by (*simp add*: *zdiv-self*[*OF cnz*])
have *c* ∗ (*l div c*) = *c*∗ (*l div c*) + *l mod c* **using** *d' zdvd-iff-zmod-eq-0*[**where** *m*=*c* **and** *n*=*l*] **by** *simp*
hence *cl*:*c* ∗ (*l div c*) =*l* **using** *zmod-zdiv-equality*[**where** *a*=*l* **and** *b*=*c*, *symmetric*]
by *simp*
hence (*l*∗*x* + (*l div c*) ∗ *Inum* (*x*# *bs*) *e* ≤ *0*) =
((*c* ∗ (*l div c*)) ∗ *x* + (*l div c*) ∗ *Inum* (*x* # *bs*) *e* ≤ *0*)
by *simp*
also have ... = ((*l div c*) ∗ (*c* ∗ *x* + *Inum* (*x* # *bs*) *e*) ≤ ((*l div c*)) ∗ *0*) **by** (*simp add*: *ring-simps*)
also have ... = (*c*∗*x* + *Inum* (*x* # *bs*) *e* ≤ *0*)
using *mult-le-0-iff* [**where** *a*=(*l 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 dvd l* **by** *simp*+
from *lp cp* **have** *clel*: *c*≤*l* **by** (*simp add*: *zdvd-imp-le* [*OF d' lp*])
from *cp* **have** *cnz*: *c* ≠ *0* **by** *simp*
have *c div c*≤ *l div c*
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:*0* < *l div c*
by (*simp add*: *zdiv-self*[*OF cnz*])
have *c* ∗ (*l div c*) = *c*∗ (*l div c*) + *l mod c* **using** *d' zdvd-iff-zmod-eq-0*[**where** *m*=*c* **and** *n*=*l*] **by** *simp*
hence *cl*:*c* ∗ (*l div c*) =*l* **using** *zmod-zdiv-equality*[**where** *a*=*l* **and** *b*=*c*, *symmetric*]
by *simp*
hence (*l*∗*x* + (*l div c*)∗ *Inum* (*x* # *bs*) *e* > *0*) =
((*c* ∗ (*l div c*)) ∗ *x* + (*l div c*) ∗ *Inum* (*x* # *bs*) *e* > *0*)
by *simp*
also have ... = ((*l div c*) ∗ (*c* ∗ *x* + *Inum* (*x* # *bs*) *e*) > ((*l div c*)) ∗ *0*) **by** (*simp add*: *ring-simps*)

also have ... = ($c * x$ + *Inum* (x # *bs*) $e > 0$)
using *zero-less-mult-iff* [**where** a=(l *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 *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 *div* $c \leq l$ *div* c
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:$0 < l$ *div* c
by (*simp add*: *zdiv-self*[*OF cnz*])
have c * (l *div* c) = c* (l *div* c) + l *mod* c **using** d' *zdvd-iff-zmod-eq-0*[**where** m=c **and** n=l] **by** *simp*
hence *cl*:c * (l *div* c) =l **using** *zmod-zdiv-equality*[**where** a=l **and** b=c, *symmetric*]
by *simp*
hence (l*x + (l *div* c)* *Inum* (x # *bs*) $e \geq 0$) =
((c*(l *div* c))*x + (l *div* c)* *Inum* (x # *bs*) $e \geq 0$)
by *simp*
also have ... = ((l *div* c)*(c*x + *Inum* (x # *bs*) e) $\geq$ ((l *div* c)) * 0)
by (*simp add*: *ring-simps*)
also have ... = (c*x + *Inum* (x # *bs*) $e \geq 0$) **using** *ldcp*
zero-le-mult-iff [**where** a=l *div* c **and** b=c*x + *Inum* (x # *bs*) 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 *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 *div* $c \leq l$ *div* c
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:$0 < l$ *div* c
by (*simp add*: *zdiv-self*[*OF cnz*])
have c * (l *div* c) = c* (l *div* c) + l *mod* c **using** d' *zdvd-iff-zmod-eq-0*[**where** m=c **and** n=l] **by** *simp*
hence *cl*:c * (l *div* c) =l **using** *zmod-zdiv-equality*[**where** a=l **and** b=c, *symmetric*]
by *simp*
hence ($l * x$ + (l *div* c) * *Inum* (x # *bs*) $e = 0$) =
((c * (l *div* c)) * x + (l *div* c) * *Inum* (x # *bs*) $e = 0$)
by *simp*
also have ... = ((l *div* c) * ($c * x$ + *Inum* (x # *bs*) e) = ((l *div* c)) * 0) **by** (*simp add*: *ring-simps*)
also have ... = ($c * x$ + *Inum* (x # *bs*) $e = 0$)
using *mult-eq-0-iff* [**where** a=(l *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 (*4 c e*) **hence** *cp*: $c>0$ **and** *be*: *numbound0 e* **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 div* $c\leq$ *l div c*
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:$0 <$ *l div c*
by (*simp add*: *zdiv-self*[*OF cnz*])
have $c *$ (*l div c*) $= c*$ (*l div c*) $+$ *l mod c* **using** d' *zdvd-iff-zmod-eq-0*[**where** $m=c$ **and** $n=l$] **by** *simp*
hence *cl*:$c *$ (*l div c*) $=l$ **using** *zmod-zdiv-equality*[**where** $a=l$ **and** $b=c$, *symmetric*]
by *simp*
hence ($l * x +$ (*l div c*) $*$ *Inum* (x # *bs*) $e \neq 0$) $=$
(($c *$ (*l div c*)) $* x +$ (*l div c*) $*$ *Inum* (x # *bs*) $e \neq 0$)
by *simp*
also have ... $=$ ((*l div c*) $*$ ($c * x +$ *Inum* (x # *bs*) e) $\neq$ ((*l div c*)) $* 0$) **by** (*simp add*: *ring-simps*)
also have ... $=$ ($c * x +$ *Inum* (x # *bs*) $e \neq 0$)
using *zero-le-mult-iff* [**where** $a=$(*l 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 (*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 div* $c\leq$ *l div c*
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:$0 <$ *l div c*
by (*simp add*: *zdiv-self*[*OF cnz*])
have $c *$ (*l div c*) $= c*$ (*l div c*) $+$ *l mod c* **using** d' *zdvd-iff-zmod-eq-0*[**where** $m=c$ **and** $n=l$] **by** *simp*
hence *cl*:$c *$ (*l div c*) $=l$ **using** *zmod-zdiv-equality*[**where** $a=l$ **and** $b=c$, *symmetric*]
by *simp*
hence ($\exists$ (*k*::*int*). $l * x +$ (*l div c*) $*$ *Inum* (x # *bs*) $e =$ ((*l div c*) $* j$) $* k$) $=$ ($\exists$ (*k*::*int*). ($c *$ (*l div c*)) $* x +$ (*l div c*) $*$ *Inum* (x # *bs*) $e =$ ((*l div c*) $* j$) $* k$) **by** *simp*
also have ... $=$ ($\exists$ (*k*::*int*). (*l div c*) $*$ ($c * x +$ *Inum* (x # *bs*) $e - j * k$) $=$ (*l div c*)$*0$) **by** (*simp add*: *ring-simps*)
also have ... $=$ ($\exists$ (*k*::*int*). $c * x +$ *Inum* (x # *bs*) $e - j * k = 0$)
using *zero-le-mult-iff* [**where** $a=$(*l div c*) **and** $b=c * x +$ *Inum* (x # *bs*) $e - j * k$] *ldcp* **by** *simp*
also have ... $=$ ($\exists$ (*k*::*int*). $c * x +$ *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 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≤l* **by** (*simp add*: *zdvd-imp-le* [*OF d' lp*])
from *cp* **have** *cnz*: *c ≠ 0* **by** *simp*
have *c div c≤ l div c*
by (*simp add*: *zdiv-mono1*[*OF clel cp*])
then have *ldcp*:*0* < *l div c*
by (*simp add*: *zdiv-self*[*OF cnz*])
have *c* ∗ (*l div c*) = *c*∗ (*l div c*) + *l mod c* **using** *d' zdvd-iff-zmod-eq-0*[**where** *m=c* **and** *n=l*] **by** *simp*
hence *cl*:*c* ∗ (*l div c*) =*l* **using** *zmod-zdiv-equality*[**where** *a=l* **and** *b=c*, *symmetric*]
by *simp*
hence (∃ (*k*::*int*). *l* ∗ *x* + (*l div c*) ∗ *Inum* (*x* # *bs*) *e* = ((*l div c*) ∗ *j*) ∗ *k*) = (∃ (*k*::*int*). (*c* ∗ (*l div c*)) ∗ *x* + (*l div c*) ∗ *Inum* (*x* # *bs*) *e* = ((*l div c*) ∗ *j*) ∗ *k*) **by** *simp*
also have . . . = (∃ (*k*::*int*). (*l div c*) ∗ (*c* ∗ *x* + *Inum* (*x* # *bs*) *e* − *j* ∗ *k*) = (*l div c*)∗*0*) **by** (*simp add*: *ring-simps*)
also have . . . = (∃ (*k*::*int*). *c* ∗ *x* + *Inum* (*x* # *bs*) *e* − *j* ∗ *k* = *0*)
using *zero-le-mult-iff* [**where** *a*=(*l div c*) **and** *b=c* ∗ *x* + *Inum* (*x* # *bs*) *e* − *j* ∗ *k*] *ldcp* **by** *simp*
also have . . . = (∃ (*k*::*int*). *c* ∗ *x* + *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 ldcp jp ldcp*] **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 *aβ-ex*: **assumes** *linp*: *iszlfm p* **and** *d*: *dβ p l* **and** *lp*: *l>0*
shows (∃ *x*. *l dvd x* ∧ *Ifm bbs* (*x* #*bs*) (*aβ p l*)) = (∃ (*x*::*int*). *Ifm bbs* (*x*#*bs*) *p*)
(**is** (∃ *x*. *l dvd x* ∧ *?P x*) = (∃ *x*. *?P' x*))
proof−
have (∃ *x*. *l dvd x* ∧ *?P x*) = (∃ (*x*::*int*). *?P* (*l*∗*x*))
using *unity-coeff-ex*[**where** *l=l* **and** *P=?P*, *simplified*] **by** *simp*
also have . . . = (∃ (*x*::*int*). *?P' x*) **using** *aβ*[*OF linp d lp*] **by** *simp*
finally show *?thesis* **.**
qed

lemma *β*:
assumes *lp*: *iszlfm p*
and *u*: *dβ p 1*
and *d*: *dδ p d*
and *dp*: *d* > *0*
and *nob*: ¬(∃(*j*::*int*) ∈ {*1* .. *d*}. ∃ *b*∈ (*Inum* (*a*#*bs*)) ' *set*(*β p*). *x* = *b* + *j*)
and *p*: *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*:*numbound0 e* **using** *dvd1-eq1*[**where** *x=c*] **by** *simp+*
with *dp p c1 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*:*numbound0 e* **using** *dvd1-eq1*[**where** *x=c*] **by** *simp+*
with *dp p c1 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*: *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 bex-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 bex-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 $\exists$ $(j{::}int) \in \{1 \, .. \, d\}$. $j = x + ?e + 1$ **by** *simp*
hence $\exists$ $(j{::}int) \in \{1 \, .. \, d\}$. $x = - ?e - 1 + j$ **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*: $d\beta$ *p 1*
and *d*: $d\delta$ *p d*
and *dp*: $d > 0$
shows $\forall$ *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*: ¬(∃(*j*::*int*) ∈ {*1* .. *d*}. ∃ *b*∈ (*Inum* (*a*#*bs*)) ' *set*(β *p*). *x* = *b* + *j*)
by *auto*
from β[*OF lp u d dp nb2 px*] **show** *?P* (*x* −*d*) **.**
qed
lemma *cpmi-eq*: *0* < *D* ⟹ (*EX z*::*int*. *ALL x*. *x* < *z* −−> (*P x* = *P1 x*))
==> *ALL x*.~(*EX* (*j*::*int*) : {*1*..*D*}. *EX* (*b*::*int*) : *B*. *P*(*b*+*j*)) −−> *P* (*x*) −−> *P* (*x* − *D*)
==> (*ALL* (*x*::*int*). *ALL* (*k*::*int*). ((*P1 x*)= (*P1* (*x*−*k*∗*D*))))
==> (*EX* (*x*::*int*). *P*(*x*)) = ((*EX* (*j*::*int*) : {*1*..*D*} . (*P1*(*j*))) | (*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*<=*k* ⟹ !*x*. *P x* ⟶ *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* <= (|*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 (∃ (*x::int*). *Ifm bbs* (*x* #*bs*) *p*) = (∃ *j*∈ {*1*.. *d*}. *Ifm bbs* (*j* #*bs*) (*minusinf p*) ∨ (∃ *b* ∈ *set* (β *p*). *Ifm bbs* ((*Inum* (*i*#*bs*) *b* + *j*) #*bs*) *p*))
(**is** (∃ (*x::int*). *?P* (*x*)) = (∃ *j*∈ *?D*. *?M j* ∨ (∃ *b*∈ *?B*. *?P* (*?I b* + *j*))))
proof−
from *minusinf-inf*[*OF lp u*]
have *th*: ∃(*z::int*). ∀*x*<*z*. *?P* (*x*) = *?M x* **by** *blast*
let *?B′* = {*?I b* | *b*. *b*∈ *?B*}
have *BB′*: (∃*j*∈*?D*. ∃*b*∈ *?B*. *?P* (*?I b* +*j*)) = (∃ *j* ∈ *?D*. ∃ *b* ∈ *?B′*. *?P* (*b* + *j*)) **by** *auto*
hence *th2*: ∀ *x*. ¬ (∃ *j* ∈ *?D*. ∃ *b* ∈ *?B′*. *?P* ((*b* + *j*))) ⟶ *?P* (*x*) ⟶ *?P* ((*x* − *d*))
using β′[*OF lp u d dp*, **where** *a*=*i* **and** *bbs* = *bbs*] **by** *blast*
from *minusinf-repeats*[*OF d lp*]
have *th3*: ∀ *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 (∃ *x*. *Ifm bbs* (*x*#*bs*) (*mirror p*)) = (∃ *x*. *Ifm bbs* (*x*#*bs*) *p*)
(**is** (∃ *x*. *?I x ?mp*) = (∃ *x*. *?I x p*))
proof(*auto*)
fix *x* **assume** *?I x ?mp* **hence** *?I* (− *x*) *p* **using** *mirror*[*OF lp*] **by** *blast*
thus ∃ *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 ∃ *x*. *?I x ?mp* **by** *blast*
qed

lemma *cp-thm′*:
assumes *lp*: *iszlfm p*
and *up*: *d*β *p 1* **and** *dd*: *d*δ *p d* **and** *dp*: *d* > *0*
shows (∃ *x*. *Ifm bbs* (*x*#*bs*) *p*) = ((∃ *j*∈ {*1* .. *d*}. *Ifm bbs* (*j*#*bs*) (*minusinf p*)) ∨ (∃ *j*∈ {*1*.. *d*}. ∃ *b*∈ (*Inum* (*i*#*bs*)) ‘ *set* (β *p*). *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*))) (*a*β *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 ⋀ *q B d*. *unit p* = (*q*,*B*,*d*) ⟹ ((∃ *x*. *Ifm bbs* (*x*#*bs*) *p*) = (∃ *x*. *Ifm bbs* (*x*#*bs*) *q*)) ∧ (*Inum* (*i*#*bs*)) ‘ *set B* = (*Inum* (*i*#*bs*)) ‘ *set* (β *q*) ∧ *d*β *q 1* ∧ *d*δ

q d ∧ *d* >*0* ∧ *iszlfm q* ∧ (∀ *b*∈ *set B*. *numbound0 b*)
proof−
fix *q B d*
assume *qBd*: *unit p* = (*q*,*B*,*d*)
let *?thes* = ((∃ *x*. *Ifm bbs* (*x*#*bs*) *p*) = (∃ *x*. *Ifm bbs* (*x*#*bs*) *q*)) ∧
Inum (*i*#*bs*) ' *set B* = *Inum* (*i*#*bs*) ' *set* (β *q*) ∧
*d*β *q 1* ∧ *d*δ *q d* ∧ *0* < *d* ∧ *iszlfm q* ∧ (∀ *b*∈ *set B*. *numbound0 b*)
let *?I* = λ *x p*. *Ifm bbs* (*x*#*bs*) *p*
let *?p′* = *zlfm p*
let *?l* = ζ *?p′*
let *?q* = *And* (*Dvd ?l* (*CN 0 1* (*C 0*))) (*a*β *?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*: *d*β *?p′ ?l* **by** *auto*
from *a*β*-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 a*β[*OF lp′ dl lp*] **have** *lq*:*iszlfm ?q* **and** *uq*: *d*β *?q 1* **by** *auto*
from δ[*OF lq*] **have** *dp*:*?d* >*0* **and** *dd*: *d*δ *?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′ mirror*αβ[*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*
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* = (∃ *j*∈ {*1*.. *?d*}. *?I j ?mq* ∨ (∃ *b*∈ *?N* ‘ *set ?B*. *Ifm bbs* ((*b*+ *j*)#*bs*) *?q*)) **by** *auto*
also have ... = (∃ *j*∈ {*1*.. *?d*}. *?I j ?mq* ∨ (∃ *b*∈ *set ?B*. *Ifm bbs* ((*?N b*+ *j*)#*bs*) *?q*)) **by** *simp*
also have ... = ((∃ *j*∈ {*1*.. *?d*}. *?I j ?mq*) ∨ (∃ *j*∈ {*1*.. *?d*}. ∃ *b*∈ *set ?B*. *Ifm bbs* ((*?N* (*Add b* (*C j*)))#*bs*) *?q*)) **by** (*simp only*: *Inum.simps*) *blast*
also have ... = ((∃ *j*∈ {*1*.. *?d*}. *?I j ?smq*) ∨ (∃ *j*∈ {*1*.. *?d*}. ∃ *b*∈ *set ?B*. *Ifm bbs* ((*?N* (*Add b* (*C j*)))#*bs*) *?q*)) **by** (*simp add*: *simpfm*)
also have ... = ((∃ *j*∈ *set ?js*. (λ *j*. *?I i* (*simpfm* (*subst0* (*C j*) *?smq*))) *j*) ∨ (∃ *j*∈ *set ?js*. ∃ *b*∈ *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* (λ *j*. *simpfm* (*subst0* (*C j*) *?smq*)) *?js*) ∨ (∃ *j*∈ *set ?js*. ∃ *b*∈ *set ?B*. *?I i* (*subst0* (*Add b* (*C j*)) *?q*)))
by (*simp only*: *evaldjf-ex subst0-I*[*OF qfq*])
also have ...= (*?I i ?md* ∨ (∃ (*b*,*j*) ∈ *set ?Bjs*. (λ (*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* ∨ (*?I i* (*evaldjf* (λ (*b*,*j*). *simpfm* (*subst0* (*Add b* (*C j*)) *?q*)) *?Bjs*)))
by (*simp only*: *evaldjf-ex*[**where** *bs=i#bs* **and** *f*=λ (*b*,*j*). *simpfm* (*subst0* (*Add b* (*C j*)) *?q*) **and** *ps=?Bjs*]) (*auto simp add*: *split-def*)
finally have *mdqd*: *?lhs* = (*?I i ?md* ∨ *?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* ≠ *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* ⇒ *fm*
pa ≡ (λ *p*. *qelim* (*prep p*) *cooper*)

theorem *mirqe*: (*Ifm bbs bs* (*pa p*) = *Ifm bbs bs p*) ∧ *qfree* (*pa p*)
using *qelim-ci cooper prep* **by** (*auto simp add*: *pa-def*)

definition
cooper-test :: *unit* ⇒ *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 ∃ (*j*::*int*). ∀ *x*≥*j*. (∃ *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 (∀(*y*::*int*). *3 dvd y*) ==> ∀(*x*::*int*). *b* < *x* −−> *a* ≤ *x*
by *cooper*

theorem !! (*y*::*int*) (*z*::*int*) (*n*::*int*). *3 dvd z* ==> *2 dvd* (*y*::*int*) ==>
(∃(*x*::*int*). *2*∗*x* = *y*) & (∃(*k*::*int*). *3*∗*k* = *z*)
by *cooper*

theorem !! (*y*::*int*) (*z*::*int*) *n*. *Suc*(*n*::*nat*) < *6* ==> *3 dvd z* ==>
2 dvd (*y*::*int*) ==> (∃(*x*::*int*). *2*∗*x* = *y*) & (∃(*k*::*int*). *3*∗*k* = *z*)
by *cooper*

theorem ∀(*x*::*nat*). ∃(*y*::*nat*). (*0*::*nat*) ≤ *5* −−> *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 $ALL(x::int)\ y.\ 2 * x + 1\ \tilde{}{=}\ 2 * y$ **by** *cooper*
lemma $EX(x::int)\ y.\ 0 < x\ \ \&\ 0 <= y\ \ \&\ 3 * x - 5 * y = 1$ **by** *cooper*
lemma $\tilde{}\ (EX(x::int)\ (y::int)\ (z::int).\ 4{*}x + (-6::int){*}y = 1)$ **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 $ALL(x::int).\ ((2\ dvd\ x) = (ALL(y::int).\ x\ \tilde{}{=}\ 2{*}y + 1))$ **by** *cooper*
lemma $\tilde{}\ (ALL(x::int).\ ((2\ dvd\ x) = (ALL(y::int).\ x\ \tilde{}{=}\ 2{*}y{+}1) \mid (EX(q::int)$ $(u::int)\ i.\ 3{*}i + 2{*}q - u < 17)\ {-}{-}{>}\ 0 < x \mid ((\tilde{}\ 3\ dvd\ x)\ \&(x + 8 = 0))))$ **by** *cooper*
lemma $\tilde{}\ (ALL(i::int).\ 4 <= i\ {-}{-}{>}\ (EX\ x\ y.\ 0 <= x\ \&\ 0 <= y\ \&\ 3 * x + 5 * y = i))$
  by *cooper*
lemma $EX\ j.\ ALL\ (x::int) >= j.\ EX\ i\ j.\ 5{*}i + 3{*}j = x$ **by** *cooper*

theorem $(\forall (y::int).\ 3\ dvd\ y) ==> \forall (x::int).\ b < x\ {-}{-}{>}\ a \le x$
  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 $\forall (x::nat).\ \exists (y::nat).\ (0::nat) \le 5\ {-}{-}{>}\ y = 5 + x$
  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 $\forall (x::int)\ y.\ x < y\ {-}{-}{>}\ 2 * x + 1 < 2 * y$
  by *cooper*

theorem $\forall (x::int)\ y.\ 2 * x + 1 \neq 2 * y$
  by *cooper*

theorem $\exists (x::int)\ y.\ 0 < x\ \ \&\ 0 \le y\ \ \&\ 3 * x - 5 * y = 1$
  by *cooper*

theorem $\tilde{}\ (\exists (x::int)\ (y::int)\ (z::int).\ 4{*}x + (-6::int){*}y = 1)$
  by *cooper*

theorem $\tilde{}\ (\exists (x::int).\ False)$

by *cooper*

theorem $\forall(x::int). (2\ dvd\ x) \dashrightarrow (\exists(y::int). x = 2*y)$
by *cooper*

theorem $\forall(x::int). (2\ dvd\ x) \dashrightarrow (\exists(y::int). x = 2*y)$
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 ~ $(\forall(x::int).$
$((2\ dvd\ x) = (\forall(y::int). x \neq 2*y+1) \mid$
$(\exists(q::int)\ (u::int)\ i.\ 3*i + 2*q - u < 17)$
$\dashrightarrow 0 < x \mid ((\sim 3\ dvd\ x)\ \&(x + 8 = 0))))$
by *cooper*

theorem ~ $(\forall(i::int). 4 \leq i \dashrightarrow (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
by *cooper*

theorem $\forall(i::int). 8 \leq i \dashrightarrow (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
by *cooper*

theorem $\exists(j::int). \forall i.\ j \leq i \dashrightarrow (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
by *cooper*

theorem ~ $(\forall j\ (i::int).\ j \leq i \dashrightarrow (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
by *cooper*

theorem $(\exists m::nat.\ n = 2 * m) \dashrightarrow (n + 1)\ div\ 2 = n\ div\ 2$
by *cooper*

end

43 Generic reflection and reification

theory *Reflection*
imports *Main*
uses *reflection-data.ML* (*reflection.ML*)
begin

setup ⟪ *Reify-Data.setup*⟫

lemma *ext2*: $(\forall x.\ f\ x = g\ x) \Longrightarrow f = g$
  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-eqs = eqs@ceqs
     in Method.SIMPLE-METHOD′ (Reflection.reflection-tac ctxt corr-thms raw-eqs
to)
     end) end
⟫ *reflection method*
end

44 Implementation of finite sets by lists

theory *Executable-Set*
imports *Main*
begin

44.1 Definitional rewrites

lemma [*code target*: *Set*]:
  $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$

by *blast*

lemma [*code*]:
$a \in A \longleftrightarrow (\exists x{\in}A.\ x = a)$
unfolding *bex-triv-one-point1* **..**

definition
filter-set :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
filter-set $P\ xs = \{x{\in}xs.\ P\ x\}$

44.2 Operations on lists

44.2.1 Basic definitions

definition
flip :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
flip $f\ a\ b = f\ b\ a$

definition
member :: $'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
member $xs\ x \longleftrightarrow x \in set\ xs$

definition
insertl :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
insertl $x\ xs = (if\ member\ xs\ x\ then\ xs\ else\ x\#xs)$

lemma [*code target*: *List*]: *member* $[]\ y \longleftrightarrow False$
and [*code target*: *List*]: *member* $(x\#xs)\ y \longleftrightarrow y = x \vee member\ xs\ y$
unfolding *member-def* **by** (*induct xs*) *simp-all*

fun
drop-first :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
drop-first $f\ [] = []$
| *drop-first* $f\ (x\#xs) = (if\ f\ x\ then\ xs\ else\ x\ \#\ drop\text{-}first\ f\ xs)$
declare *drop-first.simps* [*code del*]
declare *drop-first.simps* [*code target*: *List*]

declare *remove1.simps* [*code del*]
lemma [*code target*: *List*]:
remove1 $x\ xs = (if\ member\ xs\ x\ then\ drop\text{-}first\ (\lambda y.\ y = x)\ xs\ else\ xs)$
proof (*cases member xs x*)
case *False* **thus** *?thesis* **unfolding** *member-def* **by** (*induct xs*) *auto*
next
case *True*
have *remove1* $x\ xs = drop\text{-}first\ (\lambda y.\ y = x)\ xs$ **by** (*induct xs*) *simp-all*
with *True* **show** *?thesis* **by** *simp*
qed

lemma *member-nil* [*simp*]:
member $[] = (\lambda x.\ False)$

proof
  fix *x*
  show *member* [] *x* = *False* **unfolding** *member-def* **by** *simp*
qed

lemma *member-insertl* [*simp*]:
  $x \in set\ (insertl\ x\ xs)$
  unfolding *insertl-def member-def mem-iff* **by** *simp*

lemma *insertl-member* [*simp*]:
  fixes *xs x*
  assumes *member*: *member xs x*
  shows *insertl x xs* = *xs*
  using *member* **unfolding** *insertl-def* **by** *simp*

lemma *insertl-not-member* [*simp*]:
  fixes *xs x*
  assumes *member*: ¬ (*member xs x*)
  shows *insertl x xs* = *x* # *xs*
  using *member* **unfolding** *insertl-def* **by** *simp*

lemma *foldr-remove1-empty* [*simp*]:
  foldr remove1 xs [] = []
  by (*induct xs*) *simp-all*

44.2.2 Derived definitions

function *unionl* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
  unionl [] *ys* = *ys*
| *unionl xs ys* = *foldr insertl xs ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *unionl-def* = *unionl.simps*(*2*)

function *intersect* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
  intersect [] *ys* = []
| *intersect xs* [] = []
| *intersect xs ys* = *filter* (*member xs*) *ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *intersect-def* = *intersect.simps*(*3*)

function *subtract* :: $'a\ list \Rightarrow 'a\ list \Rightarrow 'a\ list$
where
  subtract [] *ys* = *ys*

| *subtract xs* [] = []
| *subtract xs ys* = *foldr remove1 xs ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *subtract-def* = *subtract.simps*(*3*)

function *map-distinct* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list*
where
  map-distinct f [] = []
| *map-distinct f xs* = *foldr* (*insertl o f*) *xs* []
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *map-distinct-def* = *map-distinct.simps*(*2*)

function *unions* :: *'a list list* ⇒ *'a list*
where
  unions [] = []
| *unions xs* = *foldr unionl xs* []
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *unions-def* = *unions.simps*(*2*)

consts *intersects* :: *'a list list* ⇒ *'a list*
primrec
  intersects (*x#xs*) = *foldr intersect xs x*

definition
  map-union :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
  map-union xs f = *unions* (*map f xs*)

definition
  map-inter :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
  map-inter xs f = *intersects* (*map f xs*)

44.3 Isomorphism proofs

lemma *iso-member*:
  member xs x ⟷ *x* ∈ *set xs*
  unfolding *member-def mem-iff* **..**

lemma *iso-insert*:
  set (*insertl x xs*) = *insert x* (*set xs*)
  unfolding *insertl-def iso-member* **by** (*simp add*: *Set.insert-absorb*)

lemma *iso-remove1*:
  assumes *distnct*: *distinct xs*

shows *set* (*remove1 x xs*) = *set xs* − {*x*}
using *distnct set-remove1-eq* **by** *auto*

lemma *iso-union*:
set (*unionl xs ys*) = *set xs* ∪ *set ys*
unfolding *unionl-def*
by (*induct xs arbitrary*: *ys*) (*simp-all add*: *iso-insert*)

lemma *iso-intersect*:
set (*intersect xs ys*) = *set xs* ∩ *set ys*
unfolding *intersect-def Int-def* **by** (*simp add*: *Int-def iso-member*) *auto*

definition
subtract' :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
subtract' = *flip subtract*

lemma *iso-subtract*:
fixes *ys*
assumes *distnct*: *distinct ys*
shows *set* (*subtract' ys xs*) = *set ys* − *set xs*
and *distinct* (*subtract' ys xs*)
unfolding *subtract'-def flip-def subtract-def*
using *distnct* **by** (*induct xs arbitrary*: *ys*) *auto*

lemma *iso-map-distinct*:
set (*map-distinct f xs*) = *image f* (*set xs*)
unfolding *map-distinct-def* **by** (*induct xs*) (*simp-all add*: *iso-insert*)

lemma *iso-unions*:
set (*unions xss*) = $\bigcup$ *set* (*map set xss*)
unfolding *unions-def*
proof (*induct xss*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons xs xss*) **thus** *?case* **by** (*induct xs*) (*simp-all add*: *iso-insert*)
qed

lemma *iso-intersects*:
set (*intersects* (*xs#xss*)) = $\bigcap$ *set* (*map set* (*xs#xss*))
by (*induct xss*) (*simp-all add*: *Int-def iso-member*, *auto*)

lemma *iso-UNION*:
set (*map-union xs f*) = *UNION* (*set xs*) (*set o f*)
unfolding *map-union-def iso-unions* **by** *simp*

lemma *iso-INTER*:
set (*map-inter* (*x#xs*) *f*) = *INTER* (*set* (*x#xs*)) (*set o f*)
unfolding *map-inter-def iso-intersects* **by** (*induct xs*) (*simp-all add*: *iso-member*, *auto*)

definition
Blall :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blall = *flip list-all*
definition
Blex :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blex = *flip list-ex*

lemma *iso-Ball*:
Blall xs f = *Ball* (*set xs*) *f*
unfolding *Blall-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-Bex*:
Blex xs f = *Bex* (*set xs*) *f*
unfolding *Blex-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-filter*:
set (*filter P xs*) = *filter-set P* (*set xs*)
unfolding *filter-set-def* **by** (*induct xs*) *auto*

44.4 code generator setup

ML ⟪
nonfix inter;
nonfix union;
nonfix subset;
⟫

44.4.1 type serializations

types-code
set (- *list*)
attach (*term-of*) ⟪
fun term-of-set f T [] = *Const* ({}, *Type* (*set*, [*T*]))
| *term-of-set f T* (*x* :: *xs*) = *Const* (*insert*,
T −−> *Type* (*set*, [*T*]) −−> *Type* (*set*, [*T*])) $ *f x* $ *term-of-set f T xs*;
⟫
attach (*test*) ⟪
fun gen-set' aG i j = *frequency*
[(*i*, *fn* () => *aG j* :: *gen-set' aG* (*i*−*1*) *j*), (*1*, *fn* () => [])] ()
and gen-set aG i = *gen-set' aG i i*;
⟫

44.4.2 const serializations

consts-code
{} ({∗[]∗})
insert ({∗*insertl*∗})
op ∪ ({∗*unionl*∗})
op ∩ ({∗*intersect*∗})

op − :: *'a set* ⇒ *'a set* ⇒ *'a set* ({∗ *flip subtract* ∗})
image ({∗*map-distinct*∗})
Union ({∗*unions*∗})
Inter ({∗*intersects*∗})
UNION ({∗*map-union*∗})
INTER ({∗*map-inter*∗})
Ball ({∗*Blall*∗})
Bex ({∗*Blex*∗})
filter-set ({∗*filter*∗})

end

theory *NBE* **imports** *Main Executable-Set* **begin**

axiomatization where *unproven*: *PROP A*

declare *Let-def* [*simp*]

consts-code *undefined* ((*raise Match*))

types *lam-var-name* = *nat*
ml-var-name = *nat*
const-name = *nat*

datatype *tm* = *Ct const-name* | *Vt lam-var-name* | *Lam tm* | *At tm tm*
| *term-of ml*
and *ml* =
C const-name ml list | *V lam-var-name ml list*
| *Fun ml ml list nat*
| *apply ml ml*

| *V-ML ml-var-name* | *A-ML ml ml list* | *Lam-ML ml*
| *CC const-name*

lemma [*simp*]: $x \in set\ vs \Longrightarrow size\ x < Suc\ (ml\text{-}list\text{-}size1\ vs)$
by (*induct vs*) *auto*
lemma [*simp*]: $x \in set\ vs \Longrightarrow size\ x < Suc\ (ml\text{-}list\text{-}size2\ vs)$
by (*induct vs*) *auto*
lemma [*simp*]:$x \in set\ vs \Longrightarrow size\ x < Suc\ (size\ v + ml\text{-}list\text{-}size3\ vs)$
by (*induct vs*) *auto*
lemma [*simp*]: $x \in set\ vs \Longrightarrow size\ x < Suc\ (size\ v + ml\text{-}list\text{-}size4\ vs)$
by (*induct vs*) *auto*

locale *Vars* =
fixes *r s t*:: *tm*
and *rs ss ts* :: *tm list*

and *u v w* :: *ml*
and *us vs ws* :: *ml list*
and *nm* :: *const-name*
and *x* :: *lam-var-name*
and *X* :: *ml-var-name*

inductive-set *Pure-tms* :: *tm set*
where
Ct s : *Pure-tms*
| *Vt x* : *Pure-tms*
| *t* : *Pure-tms* ==> *Lam t* : *Pure-tms*
| *s* : *Pure-tms* ==> *t* : *Pure-tms* ==> *At s t* : *Pure-tms*

consts
R :: (*const-name* ∗ *tm list* ∗ *tm*)*set*
compR :: (*const-name* ∗ *ml list* ∗ *ml*)*set*

fun
lift-tm :: *nat* ⇒ *tm* ⇒ *tm* (*lift*) **and**
lift-ml :: *nat* ⇒ *ml* ⇒ *ml* (*lift*)
where
lift i (*Ct nm*) = *Ct nm* |
lift i (*Vt x*) = *Vt*(*if x* < *i then x else x+1*) |
lift i (*Lam t*) = *Lam* (*lift* (*i+1*) *t*) |
lift i (*At s t*) = *At* (*lift i s*) (*lift i t*) |
lift i (*term-of v*) = *term-of* (*lift i v*) |

lift i (*C nm vs*) = *C nm* (*map* (*lift i*) *vs*) |
lift i (*V x vs*) = *V* (*if x* < *i then x else x+1*) (*map* (*lift i*) *vs*) |
lift i (*Fun v vs n*) = *Fun* (*lift i v*) (*map* (*lift i*) *vs*) *n* |
lift i (*apply u v*) = *apply* (*lift i u*) (*lift i v*) |
lift i (*V-ML X*) = *V-ML X* |
lift i (*A-ML v vs*) = *A-ML* (*lift i v*) (*map* (*lift i*) *vs*) |
lift i (*Lam-ML v*) = *Lam-ML* (*lift i v*) |
lift i (*CC nm*) = *CC nm*

fun
lift-tm-ML :: *nat* ⇒ *tm* ⇒ *tm* ($lift_{ML}$) **and**
lift-ml-ML :: *nat* ⇒ *ml* ⇒ *ml* ($lift_{ML}$)
where
$lift_{ML}$ *i* (*Ct nm*) = *Ct nm* |
$lift_{ML}$ *i* (*Vt x*) = *Vt x* |
$lift_{ML}$ *i* (*Lam t*) = *Lam* ($lift_{ML}$ *i t*) |
$lift_{ML}$ *i* (*At s t*) = *At* ($lift_{ML}$ *i s*) ($lift_{ML}$ *i t*) |
$lift_{ML}$ *i* (*term-of v*) = *term-of* ($lift_{ML}$ *i v*) |

$lift_{ML}$ *i* (*C nm vs*) = *C nm* (*map* ($lift_{ML}$ *i*) *vs*) |
$lift_{ML}$ *i* (*V x vs*) = *V x* (*map* ($lift_{ML}$ *i*) *vs*) |

$lift_{ML}$ *i* (*Fun v vs n*) = *Fun* ($lift_{ML}$ *i v*) (*map* ($lift_{ML}$ *i*) *vs*) *n* |
$lift_{ML}$ *i* (*apply u v*) = *apply* ($lift_{ML}$ *i u*) ($lift_{ML}$ *i v*) |
$lift_{ML}$ *i* (*V-ML X*) = *V-ML* (*if X* < *i then X else X+1*) |
$lift_{ML}$ *i* (*A-ML v vs*) = *A-ML* ($lift_{ML}$ *i v*) (*map* ($lift_{ML}$ *i*) *vs*) |
$lift_{ML}$ *i* (*Lam-ML v*) = *Lam-ML* ($lift_{ML}$ (*i+1*) *v*) |
$lift_{ML}$ *i* (*CC nm*) = *CC nm*

constdefs
cons :: *tm* ⇒ (*nat* ⇒ *tm*) ⇒ (*nat* ⇒ *tm*) (**infix** ## *65*)
t##f ≡ λ*i*. *case i of 0* ⇒ *t* | *Suc j* ⇒ *lift 0* (*f j*)
cons-ML :: *ml* ⇒ (*nat* ⇒ *ml*) ⇒ (*nat* ⇒ *ml*) (**infix** ## *65*)
v##f ≡ λ*i*. *case i of 0* ⇒ *v*::*ml* | *Suc j* ⇒ $lift_{ML}$ *0* (*f j*)

consts *subst* :: (*nat* ⇒ *tm*) ⇒ *tm* ⇒ *tm*
primrec
subst f (*Ct nm*) = *Ct nm*
subst f (*Vt x*) = *f x*
subst f (*Lam t*) = *Lam* (*subst* (*Vt 0* ## *f*) *t*)
subst f (*At s t*) = *At* (*subst f s*) (*subst f t*)

lemma *size-lift*[*simp*]: **shows**
size(*lift i t*) = *size*(*t*::*tm*) **and** *size*(*lift i* (*v*::*ml*)) = *size v*
and *ml-list-size1* (*map* (*lift i*) *vs*) = *ml-list-size1 vs*
and *ml-list-size2* (*map* (*lift i*) *vs*) = *ml-list-size2 vs*
and *ml-list-size3* (*map* (*lift i*) *vs*) = *ml-list-size3 vs*
and *ml-list-size4* (*map* (*lift i*) *vs*) = *ml-list-size4 vs*
by (*induct arbitrary*: *i* **and** *i* **and** *i* **and** *i* **and** *i* **and** *i rule*: *tm-ml.inducts*)
simp-all

lemma *size-lift-ML*[*simp*]: **shows**
size($lift_{ML}$ *i t*) = *size*(*t*::*tm*) **and** *size*($lift_{ML}$ *i* (*v*::*ml*)) = *size v*
and *ml-list-size1* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size1 vs*
and *ml-list-size2* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size2 vs*
and *ml-list-size3* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size3 vs*
and *ml-list-size4* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size4 vs*
by (*induct arbitrary*: *i* **and** *i* **and** *i* **and** *i* **and** *i* **and** *i rule*: *tm-ml.inducts*)
simp-all

fun
subst-ml-ML :: (*nat* ⇒ *ml*) ⇒ *ml* ⇒ *ml* ($subst_{ML}$) **and**
subst-tm-ML :: (*nat* ⇒ *ml*) ⇒ *tm* ⇒ *tm* ($subst_{ML}$)
where
$subst_{ML}$ *f* (*Ct nm*) = *Ct nm* |
$subst_{ML}$ *f* (*Vt x*) = *Vt x* |
$subst_{ML}$ *f* (*Lam t*) = *Lam* ($subst_{ML}$ (*lift 0 o f*) *t*) |
$subst_{ML}$ *f* (*At s t*) = *At* ($subst_{ML}$ *f s*) ($subst_{ML}$ *f t*) |
$subst_{ML}$ *f* (*term-of v*) = *term-of* ($subst_{ML}$ *f v*) |

$subst_{ML}$ *f* (*C nm vs*) = *C nm* (*map* ($subst_{ML}$ *f*) *vs*) |
$subst_{ML}$ *f* (*V x vs*) = *V x* (*map* ($subst_{ML}$ *f*) *vs*) |
$subst_{ML}$ *f* (*Fun v vs n*) = *Fun* ($subst_{ML}$ *f v*) (*map* ($subst_{ML}$ *f*) *vs*) *n* |
$subst_{ML}$ *f* (*apply u v*) = *apply* ($subst_{ML}$ *f u*) ($subst_{ML}$ *f v*) |
$subst_{ML}$ *f* (*V-ML X*) = *f X* |
$subst_{ML}$ *f* (*A-ML v vs*) = *A-ML* ($subst_{ML}$ *f v*) (*map* ($subst_{ML}$ *f*) *vs*) |
$subst_{ML}$ *f* (*Lam-ML v*) = *Lam-ML* ($subst_{ML}$ (*V-ML 0 ## f*) *v*) |
$subst_{ML}$ *f* (*CC nm*) = *CC nm*

lemmas [*code*] = *lift-tm-ML.simps lift-ml-ML.simps*
lemmas [*code*] = *lift-tm.simps lift-ml.simps*
lemmas [*code*] = *subst-tm-ML.simps subst-ml-ML.simps*

abbreviation
subst-decr :: *nat* ⇒ *tm* ⇒ *nat* ⇒ *tm* **where**
subst-decr k t == %*n. if n<k then Vt n else if n=k then t else Vt*(*n* − *1*)
abbreviation
subst-decr-ML :: *nat* ⇒ *ml* ⇒ *nat* ⇒ *ml* **where**
subst-decr-ML k v == %*n. if n<k then V-ML n else if n=k then v else V-ML*(*n* − *1*)
abbreviation
subst1 :: *tm* ⇒ *tm* ⇒ *nat* ⇒ *tm* ((-/[-'/-]) [*300*, *0*, *0*] *300*) **where**
s[*t*/*k*] == *subst* (*subst-decr k t*) *s*
abbreviation
subst1-ML :: *ml* ⇒ *ml* ⇒ *nat* ⇒ *ml* ((-/[-'/-]) [*300*, *0*, *0*] *300*) **where**
u[*v*/*k*] == $subst_{ML}$ (*subst-decr-ML k v*) *u*

lemma *size-subst-ML*[*simp*]: **shows**
(!*x*. *size*(*f x*) = *0*) ⟶ *size*($subst_{ML}$ *f t*) = *size*(*t*::*tm*) **and**
(!*x*. *size*(*f x*) = *0*) ⟶ *size*($subst_{ML}$ *f* (*v*::*ml*)) = *size v*
and (!*x*. *size*(*f x*) = *0*) ⟶ *ml-list-size1* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size1 vs*
and (!*x*. *size*(*f x*) = *0*) ⟶ *ml-list-size2* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size2 vs*
and (!*x*. *size*(*f x*) = *0*) ⟶ *ml-list-size3* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size3 vs*
and (!*x*. *size*(*f x*) = *0*) ⟶ *ml-list-size4* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size4 vs*
apply (*induct arbitrary*: *f* **and** *f* **and** *f* **and** *f* **and** *f* **and** *f rule*: *tm-ml.inducts*)
apply (*simp-all add*:*cons-ML-def split*:*nat.split*)
done

lemma *lift-lift*: **includes** *Vars* **shows**
i < *k+1* ⟹ *lift* (*Suc k*) (*lift i t*) = *lift i* (*lift k t*)
and *i* < *k+1* ⟹ *lift* (*Suc k*) (*lift i v*) = *lift i* (*lift k v*)
apply(*induct t* **and** *v arbitrary*: *i* **and** *i rule*:*lift-tm-lift-ml.induct*)
apply(*simp-all add*:*map-compose*[*symmetric*])
done

corollary *lift-o-lift*: **shows**
i < *k+1* ⟹ *lift-tm* (*Suc k*) *o* (*lift-tm i*) = *lift-tm i o lift-tm k* **and**

$i < k{+}1 \Longrightarrow$ *lift-ml* (*Suc k*) *o* (*lift-ml i*) = *lift-ml i o lift-ml k*
by(*rule ext*, *simp add*:*lift-lift*)+

lemma *lift-lift-ML*: **includes** *Vars* **shows**
$i < k{+}1 \Longrightarrow lift_{ML}\ (Suc\ k)\ (lift_{ML}\ i\ t) = lift_{ML}\ i\ (lift_{ML}\ k\ t)$
and $i < k{+}1 \Longrightarrow lift_{ML}\ (Suc\ k)\ (lift_{ML}\ i\ v) = lift_{ML}\ i\ (lift_{ML}\ k\ v)$
apply(*induct t* **and** *v arbitrary*: *i* **and** *i rule*:*lift-tm-ML-lift-ml-ML.induct*)
apply(*simp-all add*:*map-compose*[*symmetric*])
done

lemma *lift-lift-ML-comm*: **includes** *Vars* **shows**
$lift\ j\ (lift_{ML}\ i\ t) = lift_{ML}\ i\ (lift\ j\ t)$ **and**
$lift\ j\ (lift_{ML}\ i\ v) = lift_{ML}\ i\ (lift\ j\ v)$
apply(*induct t* **and** *v arbitrary*: *i j* **and** *i j rule*:*lift-tm-ML-lift-ml-ML.induct*)
apply(*simp-all add*:*map-compose*[*symmetric*])
done

lemma [*simp*]:
V-ML 0 ## subst-decr-ML k v = *subst-decr-ML* (*Suc k*) ($lift_{ML}$ *0 v*)
by(*rule ext*)(*simp add*:*cons-ML-def split*:*nat.split*)

lemma [*simp*]: *lift 0 o subst-decr-ML k v* = *subst-decr-ML k* (*lift 0 v*)
by(*rule ext*)(*simp add*:*cons-ML-def split*:*nat.split*)

lemma *subst-lift-id*[*simp*]: **includes** *Vars* **shows**
$subst_{ML}\ (subst\text{-}decr\text{-}ML\ k\ v)\ (lift_{ML}\ k\ t) = t$ **and** $(lift_{ML}\ k\ u)[v/k] = u$
apply(*induct k t* **and** *k u arbitrary*: *v* **and** *v rule*: *lift-tm-ML-lift-ml-ML.induct*)
apply (*simp-all add*:*map-idI map-compose*[*symmetric*])
apply (*simp cong*:*if-cong*)
done

inductive-set
tRed :: (*tm* ∗ *tm*)*set*
and *tred* :: [*tm*, *tm*] => *bool* (**infixl** → *50*)
where
$s \to t == (s,\ t) \in tRed$
| $At\ (Lam\ t)\ s \to t[s/0]$
| $(nm,ts,t) : R ==> foldl\ At\ (Ct\ nm)\ (map\ (subst\ rs)\ ts) \to subst\ rs\ t$
| $t \to t' ==> Lam\ t \to Lam\ t'$
| $s \to s' ==> At\ s\ t \to At\ s'\ t$
| $t \to t' ==> At\ s\ t \to At\ s\ t'$

abbreviation
treds :: [*tm*, *tm*] => *bool* (**infixl** →∗ *50*) **where**
$s \to\!* \ t == (s,\ t) \in tRed\hat{\ }*$

inductive-set
tRed-list :: (*tm list* ∗ *tm list*) *set*

and *treds-list* :: [*tm list*, *tm list*] ⇒ *bool* (**infixl** →∗ *50*)
where
ss →∗ *ts* == (*ss*, *ts*) ∈ *tRed-list*
| [] →∗ []
| *ts* →∗ *ts′* ==> *t* →∗ *t′* ==> *t#ts* →∗ *t′#ts′*

declare *tRed-list.intros*[*simp*]

lemma *tRed-list-refl*[*simp*]: **includes** *Vars* **shows** *ts* →∗ *ts*
by(*induct ts*) *auto*

fun *ML-closed* :: *nat* ⇒ *ml* ⇒ *bool*
and *ML-closed-t* :: *nat* ⇒ *tm* ⇒ *bool* **where**
ML-closed i (*C nm vs*) = (*ALL v*:*set vs*. *ML-closed i v*) |
ML-closed i (*V nm vs*) = (*ALL v*:*set vs*. *ML-closed i v*) |
ML-closed i (*Fun f vs n*) = (*ML-closed i f* & (*ALL v*:*set vs*. *ML-closed i v*)) |
ML-closed i (*A-ML v vs*) = (*ML-closed i v* & (*ALL v*:*set vs*. *ML-closed i v*)) |
ML-closed i (*apply v w*) = (*ML-closed i v* & *ML-closed i w*) |
ML-closed i (*CC nm*) = *True* |
ML-closed i (*V-ML X*) = (*X*<*i*) |
ML-closed i (*Lam-ML v*) = *ML-closed* (*i*+*1*) *v* |
ML-closed-t i (*term-of v*) = *ML-closed i v* |
ML-closed-t i (*At r s*) = (*ML-closed-t i r* & *ML-closed-t i s*) |
ML-closed-t i (*Lam t*) = (*ML-closed-t i t*) |
ML-closed-t i v = *True*
thm *ML-closed.simps ML-closed-t.simps*

inductive-set
Red :: (*ml* ∗ *ml*)*set*
and *Redt* :: (*tm* ∗ *tm*)*set*
and *Redl* :: (*ml list* ∗ *ml list*)*set*
and *red* :: [*ml*, *ml*] => *bool* (**infixl** ⇒ *50*)
and *redl* :: [*ml list*, *ml list*] => *bool* (**infixl** ⇒ *50*)
and *redt* :: [*tm*, *tm*] => *bool* (**infixl** ⇒ *50*)
and *reds* :: [*ml*, *ml*] => *bool* (**infixl** ⇒∗ *50*)
and *redts* :: [*tm*, *tm*] => *bool* (**infixl** ⇒∗ *50*)
where
s ⇒ *t* == (*s*, *t*) ∈ *Red*
| *s* ⇒ *t* == (*s*, *t*) ∈ *Redl*
| *s* ⇒ *t* == (*s*, *t*) ∈ *Redt*
| *s* ⇒∗ *t* == (*s*, *t*) ∈ *Red*ˆ∗
| *s* ⇒∗ *t* == (*s*, *t*) ∈ *Redt*ˆ∗

| *A-ML* (*Lam-ML u*) [*v*] ⇒ *u*[*v*/*0*]

| (*nm*,*vs*,*v*) : *compR* ==> *ALL i*. *ML-closed 0* (*f i*) ⟹ *A-ML* (*CC nm*) (*map* ($subst_{ML}$ *f*) *vs*) ⇒ $subst_{ML}$ *f v*

| *apply-Fun1*: *apply* (*Fun f vs* (*Suc 0*)) $v \Rightarrow$ *A-ML f* (*vs* @ [v])
| *apply-Fun2*: $n > 0$ ==>
apply (*Fun f vs* (*Suc n*)) $v \Rightarrow$ *Fun f* (*vs* @ [v]) n
| *apply-C*: *apply* (*C nm vs*) $v \Rightarrow$ *C nm* (*vs* @ [v])
| *apply-V*: *apply* (*V x vs*) $v \Rightarrow$ *V x* (*vs* @ [v])

| *term-of-C*: *term-of* (*C nm vs*) $\Rightarrow$ *foldl At* (*Ct nm*) (*map term-of vs*)
| *term-of-V*: *term-of* (*V x vs*) $\Rightarrow$ *foldl At* (*Vt x*) (*map term-of vs*)
| *term-of-Fun*: *term-of*(*Fun vf vs n*) $\Rightarrow$
Lam (*term-of* ((*apply* (*lift 0* (*Fun vf vs n*)) (*V-ML 0*))[*V 0* []/*0*]))

| *ctxt-Lam*: $t \Rightarrow t'$ ==> *Lam* $t \Rightarrow$ *Lam* t'
| *ctxt-At1*: $s \Rightarrow s'$ ==> *At* s $t \Rightarrow$ *At* s' t
| *ctxt-At2*: $t \Rightarrow t'$ ==> *At* s $t \Rightarrow$ *At* s t'
| *ctxt-term-of*: $v \Rightarrow v'$ ==> *term-of* $v \Rightarrow$ *term-of* v'
| *ctxt-C*: $vs \Rightarrow vs'$ ==> *C nm vs* $\Rightarrow$ *C nm* vs'
| *ctxt-V*: $vs \Rightarrow vs'$ ==> *V x vs* $\Rightarrow$ *V x* vs'
| *ctxt-Fun1*: $f \Rightarrow f'$ ==> *Fun f vs n* $\Rightarrow$ *Fun* f' *vs n*
| *ctxt-Fun3*: $vs \Rightarrow vs'$ ==> *Fun f vs n* $\Rightarrow$ *Fun f* vs' n
| *ctxt-apply1*: $s \Rightarrow s'$ ==> *apply* s $t \Rightarrow$ *apply* s' t
| *ctxt-apply2*: $t \Rightarrow t'$ ==> *apply* s $t \Rightarrow$ *apply* s t'
| *ctxt-A-ML1*: $f \Rightarrow f'$ ==> *A-ML f vs* $\Rightarrow$ *A-ML* f' *vs*
| *ctxt-A-ML2*: $vs \Rightarrow vs'$ ==> *A-ML f vs* $\Rightarrow$ *A-ML f* vs'
| *ctxt-list1*: $v \Rightarrow v'$ ==> $v\#vs \Rightarrow v'\#vs$
| *ctxt-list2*: $vs \Rightarrow vs'$ ==> $v\#vs \Rightarrow v\#vs'$

consts
ar :: *const-name* $\Rightarrow$ *nat*

axioms
ar-pos: *ar nm* > 0

types *env* = *ml list*

consts *eval* :: *tm* $\Rightarrow$ *env* $\Rightarrow$ *ml*
primrec
eval (*Vt x*) *e* = *e*!*x*
eval (*Ct nm*) *e* = *Fun* (*CC nm*) [] (*ar nm*)
eval (*At s t*) *e* = *apply* (*eval s e*) (*eval t e*)
eval (*Lam t*) *e* = *Fun* (*Lam-ML* (*eval t* ((*V-ML 0*) # *map* ($lift_{ML}$ *0*) *e*))) [] *1*

fun *size'* :: *ml* $\Rightarrow$ *nat* **where**
size' (*C nm vs*) = ($\sum$ *v*←*vs*. *size' v*)+*1* |
size' (*V nm vs*) = ($\sum$ *v*←*vs*. *size' v*)+*1* |
size' (*Fun f vs n*) = (*size' f* + ($\sum$ *v*←*vs*. *size' v*))+*1* |
size' (*A-ML v vs*) = (*size' v* + ($\sum$ *v*←*vs*. *size' v*))+*1* |
size' (*apply v w*) = (*size' v* + *size' w*)+*1* |
size' (*CC nm*) = *1* |

size′ (*V-ML X*) = *1* |
size′ (*Lam-ML v*) = *size′ v* + *1*

lemma *listsum-size′*[*simp*]:
v ∈ *set vs* ⟹ *size′ v* < *Suc*(*listsum* (*map size′ vs*))
by (*rule unproven*)

corollary *cor-listsum-size′*[*simp*]:
v ∈ *set vs* ⟹ *size′ v* < *Suc*(*m* + *listsum* (*map size′ vs*))
using *listsum-size′*[*of v vs*] **by** *arith*

lemma
size-subst-ML[*simp*]: **includes** *Vars* **assumes** *A*: !*i*. *size*(*f i*) = *0*
shows *size*($subst_{ML}$ *f t*) = *size*(*t*)
and *size*($subst_{ML}$ *f v*) = *size*(*v*)
and *ml-list-size1* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size1 vs*
and *ml-list-size2* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size2 vs*
and *ml-list-size3* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size3 vs*
and *ml-list-size4* (*map* ($subst_{ML}$ *f*) *vs*) = *ml-list-size4 vs*
by (*induct rule*: *tm-ml.inducts*) (*simp-all add*: *A cons-ML-def split*:*nat.split*)

lemma [*simp*]:
∀ *i j*. *size′*(*f i*) = *size′*(*V-ML j*) ⟹ *size′* ($subst_{ML}$ *f v*) = *size′ v*
by (*rule unproven*)

lemma [*simp*]: *size′* (*lift i v*) = *size′ v*
by (*rule unproven*)

function *kernel* :: *ml* ⇒ *tm* (-! *300*) **where**
(*C nm vs*)! = *foldl At* (*Ct nm*) (*map kernel vs*) |
(*Lam-ML v*)! = *Lam* (((*lift 0 v*)[*V 0* []/*0*])!) |
(*Fun f vs n*)! = *foldl At* (*f*!) (*map kernel vs*) |
(*A-ML v vs*)! = *foldl At* (*v*!) (*map kernel vs*) |
(*apply v w*)! = *At* (*v*!) (*w*!) |
(*CC nm*)! = *Ct nm* |
(*V x vs*)! = *foldl At* (*Vt x*) (*map kernel vs*) |
(*V-ML X*)! = *undefined*
by *pat-completeness auto*
termination by(*relation measure size′*) *auto*

consts *kernelt* :: *tm* ⇒ *tm* (-! *300*)
primrec
(*Ct nm*)! = *Ct nm*
(*term-of v*)! = *v*!
(*Vt x*)! = *Vt x*
(*At s t*)! = *At* (*s*!) (*t*!)
(*Lam t*)! = *Lam* (*t*!)

abbreviation
kernels :: *ml list* ⇒ *tm list* (-! *300*) **where**
vs ! == *map kernel vs*

axioms
compiler-correct:
(*nm*, *vs*, *v*) : *compR* ==> *ALL i*. *ML-closed 0* (*f i*) ⟹ (*nm*, (*map* ($subst_{ML}$ *f*) *vs*)!, ($subst_{ML}$ *f v*)!) : *R*

consts
free-vars :: *tm* ⇒ *lam-var-name set*
primrec
free-vars (*Ct nm*) = {}
free-vars (*Vt x*) = {*x*}
free-vars (*Lam t*) = {*i*. *EX j* : *free-vars t*. *j* = *i*+*1*}
free-vars (*At s t*) = *free-vars s* ∪ *free-vars t*

lemma [*simp*]: *t* : *Pure-tms* ⟹ $lift_{ML}$ *k t* = *t*
by (*erule Pure-tms.induct*) *simp-all*

lemma *kernel-pure*: **includes** *Vars* **assumes** *t* : *Pure-tms* **shows** *t*! = *t*
using *assms* **by** (*induct*) *simp-all*

lemma *lift-eval*:
t : *Pure-tms* ⟹ *ALL e k*. (*ALL i* : *free-vars t*. *i* < *size e*) --> *lift k* (*eval t e*) = *eval t* (*map* (*lift k*) *e*)
apply(*induct set*:*Pure-tms*)
apply *simp-all*
apply *clarsimp*
apply(*erule-tac x* = *V-ML 0* # *map* ($lift_{ML}$ *0*) *e* **in** *allE*)
apply *simp*
apply(*erule impE*)
apply *clarsimp*
apply(*case-tac i*)**apply** *simp*
apply *simp*
apply (*simp add*: *map-compose*[*symmetric*])
apply (*simp add*: *o-def lift-lift-ML-comm*)
done

lemma *lift-ML-eval*[*rule-format*]:
t : *Pure-tms* ⟹ *ALL e k*. (*ALL i* : *free-vars t*. *i* < *size e*) --> $lift_{ML}$ *k* (*eval t e*) = *eval t* (*map* ($lift_{ML}$ *k*) *e*)
apply(*induct set*:*Pure-tms*)
apply *simp-all*
apply *clarsimp*
apply(*erule-tac x* = *V-ML 0* # *map* ($lift_{ML}$ *0*) *e* **in** *allE*)

apply *simp*
apply(*erule impE*)
 apply *clarsimp*
 apply(*case-tac i*)**apply** *simp*
 apply *simp*
apply (*simp add*: *map-compose*[*symmetric*])
apply (*simp add*:*o-def lift-lift-ML*)
done

lemma [*simp*]: **includes** *Vars* **shows** $(v \ \#\# \ f) \ 0 = v$
by(*simp add*:*cons-ML-def*)

lemma [*simp*]: **includes** *Vars* **shows** $(v \ \#\# \ f) \ (Suc \ n) = lift_{ML} \ 0 \ (f \ n)$
by(*simp add*:*cons-ML-def*)

lemma *lift-o-shift*: $lift \ k \ o \ (V\text{-}ML \ 0 \ \#\# \ f) = (V\text{-}ML \ 0 \ \#\# \ (lift \ k \circ f))$
apply(*rule ext*)
apply (*simp add*:*cons-ML-def lift-lift-ML-comm split*:*nat.split*)
done

lemma *lift-subst-ML*: **shows**
 $lift\text{-}tm \ k \ (subst_{ML} \ f \ t) = subst_{ML} \ (lift\text{-}ml \ k \ o \ f) \ (lift\text{-}tm \ k \ t)$ **and**
 $lift\text{-}ml \ k \ (subst_{ML} \ f \ v) = subst_{ML} \ (lift\text{-}ml \ k \ o \ f) \ (lift\text{-}ml \ k \ v)$
apply (*induct t* **and** *v arbitrary*: *f k* **and** *f k rule*: *lift-tm-lift-ml.induct*)
apply (*simp-all add*:*map-compose*[*symmetric*] *o-assoc lift-o-lift lift-o-shift*)
done

corollary *lift-subst-ML1*: $\forall v \ k. \ lift\text{-}ml \ 0 \ (u[v/k]) = (lift\text{-}ml \ 0 \ u)[lift \ 0 \ v/k]$
apply(*rule measure-induct*[**where** $f = size$ **and** $a = u$])
apply(*case-tac x*)
apply(*simp-all add*:*lift-lift map-compose*[*symmetric*] *lift-subst-ML*)
apply(*subst lift-lift-ML-comm*)**apply** *simp*
done

lemma *lift-ML-lift-ML*: **includes** *Vars* **shows**
 $i < k+1 \Longrightarrow lift_{ML} \ (Suc \ k) \ (lift_{ML} \ i \ t) = lift_{ML} \ i \ (lift_{ML} \ k \ t)$
and $i < k+1 \Longrightarrow lift_{ML} \ (Suc \ k) \ (lift_{ML} \ i \ v) = lift_{ML} \ i \ (lift_{ML} \ k \ v)$
apply (*induct k t* **and** *k v arbitrary*: *i k* **and** *i k*
 rule: *lift-tm-ML-lift-ml-ML.induct*)
apply(*simp-all add*:*map-compose*[*symmetric*])
done

corollary *lift-ML-o-lift-ML*: **shows**
 $i < k+1 \Longrightarrow lift\text{-}tm\text{-}ML \ (Suc \ k) \ o \ (lift\text{-}tm\text{-}ML \ i) = lift\text{-}tm\text{-}ML \ i \ o \ lift\text{-}tm\text{-}ML \ k$
and
 $i < k+1 \Longrightarrow lift\text{-}ml\text{-}ML \ (Suc \ k) \ o \ (lift\text{-}ml\text{-}ML \ i) = lift\text{-}ml\text{-}ML \ i \ o \ lift\text{-}ml\text{-}ML \ k$
by(*rule ext*, *simp add*:*lift-ML-lift-ML*)+

abbreviation *insrt* **where**

insrt k f == (%*i*. *if i*<*k then lift-ml-ML k* (*f i*) *else if i*=*k then V-ML k else lift-ml-ML k* (*f*(*i* − *1*)))

lemma *subst-insrt-lift*: **includes** *Vars* **shows**
$subst_{ML}$ (*insrt k f*) ($lift_{ML}$ *k t*) = $lift_{ML}$ *k* ($subst_{ML}$ *f t*) **and**
$subst_{ML}$ (*insrt k f*) ($lift_{ML}$ *k v*) = $lift_{ML}$ *k* ($subst_{ML}$ *f v*)
apply (*induct k t* **and** *k v arbitrary*: *f k* **and** *f k rule*: *lift-tm-ML-lift-ml-ML.induct*)
apply (*simp-all add*:*map-compose*[*symmetric*] *o-assoc lift-o-lift lift-o-shift*)
apply(*subgoal-tac lift 0* ∘ *insrt k f* = *insrt k* (*lift 0* ∘ *f*))
apply *simp*
apply(*rule ext*)
apply (*simp add*:*lift-lift-ML-comm*)
apply(*subgoal-tac V-ML 0* ## *insrt k f* = *insrt* (*Suc k*) (*V-ML 0* ## *f*))
apply *simp*
apply(*rule ext*)
apply (*simp add*:*lift-ML-lift-ML cons-ML-def split*:*nat.split*)
done

corollary *subst-cons-lift*: **includes** *Vars* **shows**
$subst_{ML}$ (*V-ML 0* ## *f*) *o* (*lift-ml-ML 0*) = *lift-ml-ML 0 o* (*subst-ml-ML f*)
apply(*rule ext*)
apply(*simp add*: *cons-ML-def subst-insrt-lift*[*symmetric*])
apply(*subgoal-tac nat-case* (*V-ML 0*) (λ*j*. $lift_{ML}$ *0* (*f j*)) = (λ*i*. *if i* = *0 then V-ML 0 else* $lift_{ML}$ *0* (*f* (*i* − *1*))))
apply *simp*
apply(*rule ext*, *simp split*:*nat.split*)
done

lemma *subst-eval*[*rule-format*]: *t* : *Pure-tms* ⟹
ALL f e. (*ALL i* : *free-vars t*. *i* < *size e*) ⟶ $subst_{ML}$ *f* (*eval t e*) = *eval t* (*map* ($subst_{ML}$ *f*) *e*)
apply(*induct set*:*Pure-tms*)
apply *simp-all*
apply *clarsimp*
apply(*erule-tac x*=*V-ML 0* ## *f* **in** *allE*)
apply(*erule-tac x*= (*V-ML 0* # *map* ($lift_{ML}$ *0*) *e*) **in** *allE*)
apply(*erule impE*)
apply *clarsimp*
apply(*case-tac i*)**apply** *simp*
apply *simp*
apply (*simp add*:*subst-cons-lift map-compose*[*symmetric*])
done

theorem *kernel-eval*[*rule-format*]: **includes** *Vars* **shows**
t : *Pure-tms* ==>
ALL e. (*ALL i* : *free-vars t*. *i* < *size e*) ⟶ (*ALL i* < *size e*. *e*!*i* = *V i* []) −−> (*eval t e*)! = *t*!
apply(*induct set*:*Pure-tms*)

apply *simp-all*
apply *clarsimp*
apply(*subst lift-eval*) **apply** *simp*
 apply *clarsimp*
 apply(*case-tac i*)**apply** *simp*
 apply *simp*
apply(*subst subst-eval*) **apply** *simp*
 apply *clarsimp*
 apply(*case-tac i*)**apply** *simp*
 apply *simp*
apply(*erule-tac x=map* (*subst*$_{ML}$ (λ*n. if n* = *0 then V 0* [] *else V-ML* (*n* − *1*)))
 (*map* (*lift 0*) (*V-ML 0* # *map* (*lift*$_{ML}$ *0*) *e*)) **in** *allE*)
apply(*erule impE*)
apply(*clarsimp*)
 apply(*case-tac i*)**apply** *simp*
 apply *simp*
apply(*erule impE*)
apply(*clarsimp*)
 apply(*case-tac i*)**apply** *simp*
 apply *simp*
apply *simp*
done

lemma *map-eq-iff-nth*:
 (*map f xs* = *map g xs*) = (!*i*<*size xs*. *f*(*xs*!*i*) = *g*(*xs*!*i*))
by (*rule unproven*)

lemma [*simp*]: **includes** *Vars* **shows** *ML-closed k v* $\Longrightarrow$ *lift*$_{ML}$ *k v* = *v*
by (*rule unproven*)
lemma [*simp*]: **includes** *Vars* **shows** *ML-closed 0 v* $\Longrightarrow$ *subst*$_{ML}$ *f v* = *v*
by (*rule unproven*)
lemma [*simp*]: **includes** *Vars* **shows** *ML-closed k v* $\Longrightarrow$ *ML-closed k* (*lift m v*)
by (*rule unproven*)

lemma *red-Lam*[*simp*]: **includes** *Vars* **shows** *t* →∗ *t'* ==> *Lam t* →∗ *Lam t'*
apply(*induct rule*:*rtrancl-induct*)
apply(*simp-all*)
apply(*blast intro*: *rtrancl-into-rtrancl tRed.intros*)
done

lemma *red-At1*[*simp*]: **includes** *Vars* **shows** *t* →∗ *t'* ==> *At t s* →∗ *At t' s*
apply(*induct rule*:*rtrancl-induct*)
apply(*simp-all*)
apply(*blast intro*: *rtrancl-into-rtrancl tRed.intros*)
done

lemma *red-At2*[*simp*]: **includes** *Vars* **shows** *t* →∗ *t'* ==> *At s t* →∗ *At s t'*

apply(*induct rule*:*rtrancl-induct*)
apply(*simp-all*)
apply(*blast intro*:*rtrancl-into-rtrancl tRed.intros*)
done

lemma *tRed-list-foldl-At*:
$ts \to\!* \ ts' \Longrightarrow s \to\!* \ s' \Longrightarrow foldl\ At\ s\ ts \to\!* \ foldl\ At\ s'\ ts'$
apply(*induct arbitrary*:*s s′ rule*:*tRed-list.induct*)
apply *simp*
apply *simp*
apply(*blast dest*: *red-At1 red-At2 intro*:*rtrancl-trans*)
done

lemma [*trans*]: $s = t \Longrightarrow t \to t' \Longrightarrow s \to t'$
by *simp*

lemma *subst-foldl*[*simp*]:
$subst\ f\ (foldl\ At\ s\ ts) = foldl\ At\ (subst\ f\ s)\ (map\ (subst\ f)\ ts)$
by (*induct ts arbitrary*: *s*) *auto*

lemma *foldl-At-size*: $size\ ts = size\ ts' \Longrightarrow$
$foldl\ At\ s\ ts = foldl\ At\ s'\ ts' \longleftrightarrow s = s'\ \&\ ts = ts'$
by (*induct arbitrary*: *s s′ rule*:*list-induct2*) *simp-all*

consts $depth\text{-}At :: tm \Rightarrow nat$
primrec
$depth\text{-}At(Ct\ cn) = 0$
$depth\text{-}At(Vt\ x) = 0$
$depth\text{-}At(Lam\ t) = 0$
$depth\text{-}At(At\ s\ t) = depth\text{-}At\ s + 1$
$depth\text{-}At(term\text{-}of\ v) = 0$

lemma *depth-At-foldl*:
$depth\text{-}At(foldl\ At\ s\ ts) = depth\text{-}At\ s + size\ ts$
by (*induct ts arbitrary*: *s*) *simp-all*

lemma *foldl-At-eq-length*:
$foldl\ At\ s\ ts = foldl\ At\ s\ ts' \Longrightarrow length\ ts = length\ ts'$
apply(*subgoal-tac* $depth\text{-}At(foldl\ At\ s\ ts) = depth\text{-}At(foldl\ At\ s\ ts')$)
apply(*erule thin-rl*)
apply (*simp add*:*depth-At-foldl*)
apply *simp*
done

lemma *foldl-At-eq*[*simp*]: $foldl\ At\ s\ ts = foldl\ At\ s\ ts' \longleftrightarrow ts = ts'$
apply(*rule*)
prefer *2* **apply** *simp*

apply(*blast dest*:*foldl-At-size foldl-At-eq-length*)
done

lemma [*simp*]: *foldl At s ts* ! = *foldl At* (*s*!) (*map kernelt ts*)
by (*induct ts arbitrary*: *s*) *simp-all*

lemma [*simp*]: (*kernelt* ∘ *term-of*) = *kernel*
by(*rule ext*) *simp*

lemma *shift-subst-decr*:
Vt 0 ## subst-decr k t = *subst-decr* (*Suc k*) (*lift 0 t*)
apply(*rule ext*)
apply (*simp add*:*cons-def split*:*nat.split*)
done

lemma [*simp*]: *lift k* (*foldl At s ts*) = *foldl At* (*lift k s*) (*map* (*lift k*) *ts*)
by(*induct ts arbitrary*:*s*) *simp-all*

44.5 Horrible detour

definition *liftn n* == *lift-ml 0* ˆ *n*

lemma [*simp*]: *liftn n* (*C i vs*) = *C i* (*map* (*liftn n*) *vs*)
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all add*: *map-compose*[*symmetric*])
done

lemma [*simp*]: *liftn n* (*CC nm*) = *CC nm*
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all add*: *map-compose*[*symmetric*])
done

lemma [*simp*]: *liftn n* (*apply v w*) = *apply* (*liftn n v*) (*liftn n w*)
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all add*: *map-compose*[*symmetric*])
done

lemma [*simp*]: *liftn n* (*A-ML v vs*) = *A-ML* (*liftn n v*) (*map* (*liftn n*) *vs*)
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all add*: *map-compose*[*symmetric*])
done

lemma [*simp*]:
liftn n (*Fun v vs i*) = *Fun* (*liftn n v*) (*map* (*liftn n*) *vs*) *i*
apply(*unfold liftn-def*)

apply(*induct n*)
apply (*simp-all add*: *map-compose*[*symmetric*] *id-def*)
done

lemma [*simp*]: *liftn n* (*Lam-ML v*) = *Lam-ML* (*liftn n v*)
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all add*: *map-compose*[*symmetric*] *id-def*)
done

lemma *liftn-liftn-add*: *liftn m* (*liftn n v*) = *liftn* (*m*+*n*) *v*
by(*simp add*:*liftn-def funpow-add*)

lemma [*simp*]: *liftn n* (*V-ML k*) = *V-ML k*
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all*)
done

lemma *liftn-lift-ML-comm*: *liftn n* ($lift_{ML}$ *0 v*) = $lift_{ML}$ *0* (*liftn n v*)
apply(*unfold liftn-def*)
apply(*induct n*)
apply (*simp-all add*:*lift-lift-ML-comm*)
done

lemma *liftn-cons*: *liftn n* ((*V-ML 0 ## f*) *x*) = (*V-ML 0 ##* (*liftn n o f*)) *x*
apply(*simp add*:*cons-ML-def liftn-lift-ML-comm split*:*nat.split*)
done

End of horrible detour

lemma *kernel-subst1*:
ML-closed 1 u $\Longrightarrow$ *ML-closed 0 v* $\Longrightarrow$ *kernel*(*u*[*v*/*0*]) = (*kernel*((*lift 0 u*)[*V 0* []/*0*]))[*kernel v*/*0*]
by (*rule unproven*)

lemma includes *Vars* **shows** *foldl-Pure*[*simp*]:
t : *Pure-tms* $\Longrightarrow$ $\forall$ *t*$\in$*set ts*. *t* : *Pure-tms* $\Longrightarrow$
(!!*s t*. *s* : *Pure-tms* $\Longrightarrow$ *t* : *Pure-tms* $\Longrightarrow$ *f s t* : *Pure-tms*) $\Longrightarrow$
foldl f t ts $\in$ *Pure-tms*
by(*induct ts arbitrary*: *t*) *simp-all*

declare *Pure-tms.intros*[*simp*]

lemma includes *Vars* **shows** *ML-closed 0 v* $\Longrightarrow$ *kernel v* : *Pure-tms*
apply(*induct rule*:*kernel.induct*)
apply *simp-all*
apply(*rule Pure-tms.intros*)

by (*rule unproven*)

lemma *subst-Vt*: **includes** *Vars* **shows** *subst Vt* = *id*
by (*rule unproven*)

theorem *Red-sound*: **includes** *Vars*
shows $v \Rightarrow v' \Longrightarrow$ *ML-closed 0 v* $\Longrightarrow$ $v! \rightarrow* v'!$ & *ML-closed 0 v'*
and $t \Rightarrow t' \Longrightarrow$ *ML-closed-t 0 t* $\Longrightarrow$ *kernelt t* $\rightarrow*$ *kernelt t'* & *ML-closed-t 0 t'*
and (*vs* :: *ml list*) $\Rightarrow$ *vs'* $\Longrightarrow$!*v* : *set vs* . *ML-closed 0 v* $\Longrightarrow$ *map kernel vs* $\rightarrow*$ *map kernel vs'* & (! *v'*:*set vs'*. *ML-closed 0 v'*)
proof(*induct rule*:*Red-Redt-Redl.inducts*)
fix *u v*
let *?v* = *A-ML* (*Lam-ML u*) [*v*]
assume *cl*: *ML-closed 0* (*A-ML* (*Lam-ML u*) [*v*])
let *?u'* = (*lift-ml 0 u*)[*V 0* []/*0*]
have *?v*! = *At* (*Lam* ((*?u'*)!)) (*v* !) **by** *simp*
also have ... → (*?u'* !)[*v*!/*0*] (**is** - → *?R*) **by**(*rule tRed.intros*)
also have *?R* = *u*[*v*/*0*]! **using** *cl*
apply(*cut-tac u* = *u* **and** *v* = *v* **in** *kernel-subst1*)
apply(*simp-all*)
done
finally have *kernel*(*A-ML* (*Lam-ML u*) [*v*]) →∗ *kernel*(*u*[*v*/*0*]) (**is** *?A*) **by**(*rule r-into-rtrancl*)
moreover have *ML-closed 0* (*u*[*v*/*0*]) (**is** *?C*) **using** *cl* **apply** *simp* **by** (*rule unproven*)
ultimately show *?A* & *?C* ..
next
case *term-of-C* **thus** *?case* **apply** (*auto simp*:*map-compose*[*symmetric*])**by** (*rule unproven*)
next
fix *f* :: *nat* ⇒ *ml* **and** *nm vs v*
assume *f*: ∀ *i*. *ML-closed 0* (*f i*) **and** *compR*: (*nm*, *vs*, *v*) ∈ *compR*
note *tRed.intros*(*2*)[*OF compiler-correct*[*OF compR f*], *of Vt*,*simplified map-compose*[*symmetric*]]
hence *red*: *foldl At* (*Ct nm*) (*map* (*kernel o* $subst_{ML}$ *f*) *vs*) →
($subst_{ML}$ *f v*)! (**is** - → *?R*) **apply**(*simp add*:*map-compose*) **by** (*rule unproven*)
have *A-ML* (*CC nm*) (*map* ($subst_{ML}$ *f*) *vs*)! =
foldl At (*Ct nm*) (*map* (*kernel o* $subst_{ML}$ *f*) *vs*) **by** (*simp add*:*map-compose*)
also note *red*

finally have *A-ML* (*CC nm*) (*map* ($subst_{ML}$ *f*) *vs*)! →∗ $subst_{ML}$ *f v*! (**is** *?A*)
by(*rule r-into-rtrancl*)
moreover have *ML-closed 0* ($subst_{ML}$ *f v*) (**is** *?C*) **by** (*rule unproven*)
ultimately show *?A* & *?C* ..
next
case *term-of-V* **thus** *?case* **apply** (*auto simp*:*map-compose*[*symmetric*]) **by** (*rule unproven*)
next
case (*term-of-Fun vf vs n*)

hence *term-of* (*Fun vf vs n*)! →∗
Lam (*term-of* (*apply* (*lift 0* (*Fun vf vs n*)) (*V-ML 0*)[*V 0* []/*0*]))! **by** − (*rule unproven*)
moreover
have *ML-closed-t 0*
(*Lam* (*term-of* (*apply* (*lift 0* (*Fun vf vs n*)) (*V-ML 0*)[*V 0* []/*0*]))) **by** (*rule unproven*)
ultimately show *?case* **..**
next
case *apply-Fun1* **thus** *?case* **by** *simp*
next
case *apply-Fun2* **thus** *?case* **by** *simp*
next
case *apply-C* **thus** *?case* **by** *simp*
next
case *apply-V* **thus** *?case* **by** *simp*
next
case *ctxt-Lam* **thus** *?case* **by**(*auto*)
next
case *ctxt-At1* **thus** *?case* **by**(*auto*)
next
case *ctxt-At2* **thus** *?case* **by** (*auto*)
next
case *ctxt-term-of* **thus** *?case* **by** (*auto*)
next
case *ctxt-C* **thus** *?case* **by** (*fastsimp simp*:*tRed-list-foldl-At*)
next
case *ctxt-V* **thus** *?case* **by** (*fastsimp simp*:*tRed-list-foldl-At*)
next
case *ctxt-Fun1* **thus** *?case* **by** (*fastsimp simp*:*tRed-list-foldl-At*)
next
case *ctxt-Fun3* **thus** *?case* **by** (*fastsimp simp*:*tRed-list-foldl-At*)
next
case *ctxt-apply1* **thus** *?case* **by** *auto*
next
case *ctxt-apply2* **thus** *?case* **by** *auto*
next
case *ctxt-A-ML1* **thus** *?case* **by** (*fastsimp simp*:*tRed-list-foldl-At*)
next
case *ctxt-A-ML2* **thus** *?case* **by** (*fastsimp simp*:*tRed-list-foldl-At*)
next
case *ctxt-list1* **thus** *?case* **by** *simp*
next
case *ctxt-list2* **thus** *?case* **by** *simp*
qed

inductive-cases *tRedE*: *Ct n* → *u*
thm *tRedE*

lemma [*simp*]: *Ct n* = *foldl At t ts* ⟷ *t* = *Ct n* & *ts* = []
by (*induct ts arbitrary*:*t*) *auto*

corollary *kernel-inv*: **includes** *Vars* **shows**
(*t* :: *tm*) ⇒∗ *t*′ ==> *ML-closed-t 0 t* ==> *t*! →∗ *t*′!
by (*rule unproven*)

theorem includes *Vars*
assumes *t*: *t* : *Pure-tms* **and** *t*′: *t*′ : *Pure-tms* **and**
closed: *free-vars t* = {} **and** *reds*: *term-of* (*eval t* []) ⇒∗ *t*′
shows *t* →∗ *t*′
proof −
  have *ML-cl*: *ML-closed-t 0* (*term-of* (*eval t* [])) **by** (*rule unproven*)
  have (*eval t* [])! = *t*!
    using *kernel-eval*[*OF t*, **where** *e*=[]] *closed* **by** *simp*
  hence (*term-of* (*eval t* []))! = *t*! **by** *simp*
  moreover have *term-of* (*eval t* [])! →∗ *t*′!
    using *kernel-inv*[*OF reds ML-cl*] **by** *auto*
  ultimately have *t*! →∗ *t*′! **by** *simp*
  thus *?thesis* **using** *kernel-pure t t*′ **by** *auto*
qed

end

45 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.aeconv (!pairs) t of
                      SOME v => v
                    | NONE   => insert t)
    (*abstraction of a numeric literal*)
    fun lit (t as Const(@{const-name HOL.zero}, -)) = t
      | lit (t as Const(@{const-name HOL.one}, -)) = t
      | lit (t as Const(@{const-name Numeral.number-of}, -) $ w) = t
      | lit t = 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

46 Examples for the 'refute' command

theory *Refute-Examples* **imports** *Main*
begin

refute-params [*satsolver=dpll*]

lemma $P \wedge Q$
  apply (*rule conjI*)
  refute *1* — refutes P
  refute *2* — refutes Q
  refute — equivalent to 'refute 1'
    — here 'refute 3' would cause an exception, since we only have 2 subgoals
  refute [*maxsize=5*] — we can override parameters ...
  refute [*satsolver=dpll*] *2* — ... and specify a subgoal at the same time
oops

46.1 Examples and Test Cases

46.1.1 Propositional logic

lemma *True*
  refute
  apply *auto*
done

lemma *False*
  refute
oops

lemma P
  refute
oops

lemma $\sim P$
  refute
oops

lemma $P \ \& \ Q$
  refute
oops

lemma $P \mid Q$
  refute
oops

lemma $P \longrightarrow Q$
  refute
oops

lemma $(P::bool) = Q$
  refute
oops

lemma $(P \mid Q) \longrightarrow (P \ \& \ Q)$
  refute
oops

46.1.2 Predicate logic

lemma $P\ x\ y\ z$
  refute
oops

lemma $P\ x\ y \longrightarrow P\ y\ x$
  refute
oops

lemma $P\ (f\ (f\ x)) \longrightarrow P\ x \longrightarrow P\ (f\ x)$
  refute
oops

46.1.3 Equality

lemma $P = True$
  refute
oops

lemma $P = False$
  refute
oops

lemma $x = y$
  refute

oops

lemma $f\ x = g\ x$
refute
oops

lemma $(f::'a \Rightarrow 'b) = g$
refute
oops

lemma $(f::('d \Rightarrow 'd) \Rightarrow ('c \Rightarrow 'd)) = g$
refute
oops

lemma *distinct* $[a,b]$
refute
apply *simp*
refute
oops

46.1.4 First-Order Logic

lemma $\exists\, x.\ P\ x$
refute
oops

lemma $\forall\, x.\ P\ x$
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 $(\exists\, x.\ P\ x) \longrightarrow (\forall\, x.\ P\ x)$
refute
oops

lemma $(\forall x.\ \exists y.\ P\ x\ y) \longrightarrow (\exists y.\ \forall x.\ P\ x\ y)$
  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 $[\![\ \forall x.\ P\ x\ x;\ \forall x\ y.\ P\ x\ y \longrightarrow P\ y\ x\ ]\!] \Longrightarrow P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z$
  refute
oops

The "Drinker's theorem" ...

lemma $\exists x.\ f\ x = g\ x \longrightarrow f = g$
  refute [*maxsize=4*]
  apply (*auto simp add*: *ext*)
done

... and an incorrect version of it

lemma $(\exists x.\ f\ x = g\ x) \longrightarrow f = g$
  refute
oops

"Every function has a fixed point."

lemma $\exists x.\ f\ x = x$
  refute
oops

"Function composition is commutative."

lemma $f\ (g\ x) = g\ (f\ x)$

refute
oops

"Two functions that are equivalent wrt. the same predicate 'P' are equal."

lemma $((P::('a \Rightarrow 'b) \Rightarrow bool)\ f = P\ g) \longrightarrow (f\ x = g\ x)$
refute
oops

46.1.5 Higher-Order Logic

lemma $\exists P.\ P$
refute
apply *auto*
done

lemma $\forall P.\ P$
refute
oops

lemma $EX!\ P.\ P$
refute
apply *auto*
done

lemma $EX!\ P.\ P\ x$
refute
oops

lemma $P\ Q \mid Q\ x$
refute
oops

lemma $x \neq All$
refute
oops

lemma $x \neq Ex$
refute
oops

lemma $x \neq Ex1$
refute
oops

"The transitive closure 'T' of an arbitrary relation 'P' is non-empty."

constdefs
$trans :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
$trans\ P == (ALL\ x\ y\ z.\ P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z)$
$subset :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$

$subset\ P\ Q == (ALL\ x\ y.\ P\ x\ y \longrightarrow Q\ x\ y)$
$trans\text{-}closure :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
$trans\text{-}closure\ P\ Q == (subset\ Q\ P)\ \&\ (trans\ P)\ \&\ (ALL\ R.\ subset\ Q\ R \longrightarrow trans\ R \longrightarrow subset\ P\ R)$

lemma $trans\text{-}closure\ T\ P \longrightarrow (\exists\, x\ y.\ T\ x\ y)$
refute
oops

"The union of transitive closures is equal to the transitive closure of unions."

lemma $(\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow T\ x\ y) \longrightarrow trans\ T \longrightarrow (\forall\, Q.\ (\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow Q\ x\ y) \longrightarrow trans\ Q \longrightarrow subset\ T\ Q)$
$\longrightarrow trans\text{-}closure\ TP\ P$
$\longrightarrow trans\text{-}closure\ TR\ R$
$\longrightarrow (T\ x\ y = (TP\ x\ y \mid TR\ x\ y))$
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 $(rule\text{-}tac\ x=\lambda x.\ x$ **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

46.1.6 Meta-logic

lemma $!!x.\ P\ x$
  refute
oops

lemma $f\ x == g\ x$
  refute
oops

lemma $P \Longrightarrow Q$
  refute
oops

lemma $[\![\ P;\ Q;\ R\ ]\!] \Longrightarrow S$
  refute
oops

lemma $(x == all) \Longrightarrow False$
  refute
oops

lemma $(x == (op ==)) \Longrightarrow False$
  refute
oops

lemma $(x == (op \Longrightarrow)) \Longrightarrow False$
  refute
oops

46.1.7 Schematic variables

lemma $?P$
  refute
  apply *auto*
done

lemma $x = ?y$
  refute
  apply *auto*
done

46.1.8 Abstractions

lemma $(\lambda x.\ x) = (\lambda x.\ y)$

refute
oops

lemma $(\lambda f.\ f\ x) = (\lambda f.\ \mathit{True})$
refute
oops

lemma $(\lambda x.\ x) = (\lambda y.\ y)$
refute
apply *simp*
done

46.1.9 Sets

lemma $P\ (A::'a\ set)$
refute
oops

lemma $P\ (A::'a\ set\ set)$
refute
oops

lemma $\{x.\ P\ x\} = \{y.\ P\ y\}$
refute
apply *simp*
done

lemma $x : \{x.\ P\ x\}$
refute
oops

lemma $P\ op{:}$
refute
oops

lemma $P\ (op{:}\ x)$
refute
oops

lemma $P\ \mathit{Collect}$
refute
oops

lemma $A\ \mathit{Un}\ B = A\ \mathit{Int}\ B$
refute
oops

lemma $(A\ \mathit{Int}\ B)\ \mathit{Un}\ C = (A\ \mathit{Un}\ C)\ \mathit{Int}\ B$
refute

oops

lemma *Ball A P* $\longrightarrow$ *Bex A P*
  refute
oops

46.1.10 arbitrary

lemma *arbitrary*
  refute
oops

lemma *P arbitrary*
  refute
oops

lemma *arbitrary x*
  refute
oops

lemma *arbitrary arbitrary*
  refute
oops

46.1.11 The

lemma *The P*
  refute
oops

lemma *P The*
  refute
oops

lemma *P* (*The P*)
  refute
oops

lemma (*THE x. x=y*) = *z*
  refute
oops

lemma *Ex P* $\longrightarrow$ *P* (*The P*)
  refute
oops

46.1.12 Eps

lemma *Eps P*
  refute

oops

lemma *P Eps*
  refute
oops

lemma *P* (*Eps P*)
  refute
oops

lemma (*SOME x. x=y*) = *z*
  refute
oops

lemma *Ex P* $\longrightarrow$ *P* (*Eps P*)
  refute [*maxsize=3*]
  apply (*auto simp add*: *someI*)
done

46.1.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of *'a*:

typedef *'a myTdef* = *insert* (*arbitrary*::*'a*) (*arbitrary*::*'a set*)
  by *auto*

lemma (*x*::*'a myTdef*) = *y*
  refute
oops

typedecl *myTdecl*

typedef *'a T-bij* = {(*f*::*'a*⇒*'a*). ∀ *y*. ∃!*x*. *f x* = *y*}
  by *auto*

lemma *P* (*f*::(*myTdecl myTdef*) *T-bij*)
  refute
oops

46.1.14 Inductive datatypes

With *quick-and-dirty* set, the datatype package does not generate certain axioms for recursion operators. Without these axioms, refute may find spurious countermodels.

unit

lemma *P* (*x*::*unit*)
  refute
oops

lemma $\forall x::unit.\ P\ x$
  refute
oops

lemma $P\ ()$
  refute
oops

lemma *unit-rec* $u\ x = u$
  refute
  apply *simp*
done

lemma P (*unit-rec* $u\ x$)
  refute
oops

lemma P (*case* x *of* $() \Rightarrow u$)
  refute
oops

option

lemma $P\ (x::'a\ option)$
  refute
oops

lemma $\forall x::'a\ option.\ P\ x$
  refute
oops

lemma P *None*
  refute
oops

lemma P (*Some* x)
  refute
oops

lemma *option-rec* $n\ s$ *None* $= n$
  refute
  apply *simp*
done

lemma *option-rec* $n\ s$ (*Some* x) $= s\ x$
  refute [*maxsize=4*]
  apply *simp*
done

lemma P (*option-rec n s x*)
  refute
oops

lemma P (*case x of None* $\Rightarrow$ *n* | *Some u* $\Rightarrow$ *s u*)
  refute
oops

*

lemma P (x::$'a*'b$)
  refute
oops

lemma $\forall x$::$'a*'b$. $P\ x$
  refute
oops

lemma P (x, y)
  refute
oops

lemma P (*fst x*)
  refute
oops

lemma P (*snd x*)
  refute
oops

lemma P *Pair*
  refute
oops

lemma *prod-rec p* (a, b) = *p a b*
  refute [*maxsize=2*]
  apply *simp*
oops

lemma P (*prod-rec p x*)
  refute
oops

lemma P (*case x of Pair a b* $\Rightarrow$ *p a b*)
  refute
oops

+

lemma P (x::$'a+'b$)
  refute

oops

lemma $\forall x::'a+'b.\ P\ x$
  refute
oops

lemma $P\ (Inl\ x)$
  refute
oops

lemma $P\ (Inr\ x)$
  refute
oops

lemma $P\ Inl$
  refute
oops

lemma $sum\text{-}rec\ l\ r\ (Inl\ x) = l\ x$
  refute $[maxsize=3]$
  apply $simp$
done

lemma $sum\text{-}rec\ l\ r\ (Inr\ x) = r\ x$
  refute $[maxsize=3]$
  apply $simp$
done

lemma $P\ (sum\text{-}rec\ l\ r\ x)$
  refute
oops

lemma $P\ (case\ x\ of\ Inl\ a \Rightarrow l\ a\ |\ Inr\ b \Rightarrow r\ b)$
  refute
oops

Non-recursive datatypes

datatype $T1 = A\ |\ B$

lemma $P\ (x::T1)$
  refute
oops

lemma $\forall x::T1.\ P\ x$
  refute
oops

lemma $P\ A$
  refute

oops

lemma *P B*
  refute
oops

lemma *T1-rec a b A* = *a*
  refute
  apply *simp*
done

lemma *T1-rec a b B* = *b*
  refute
  apply *simp*
done

lemma *P* (*T1-rec a b x*)
  refute
oops

lemma *P* (*case x of A* ⇒ *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 | D $v \Rightarrow d$ v)
refute
oops

datatype ('*a*,'*b*) *T3* = *E* '*a* ⇒ '*b*

lemma P (x::('*a*,'*b*) *T3*)
refute
oops

lemma ∀ x::('*a*,'*b*) *T3*. P x
refute
oops

lemma P E
refute
oops

lemma *T3-rec* e (E x) = e x
refute [*maxsize=2*]
apply *simp*
done

lemma P (*T3-rec* e x)
refute
oops

lemma P (*case* x *of* E $f \Rightarrow e$ f)
refute
oops

Recursive datatypes

nat

lemma P (x::*nat*)
refute
oops

lemma ∀ x::*nat*. P x
refute
oops

lemma P (*Suc* *0*)
refute
oops

lemma P *Suc*

refute — *Suc* is a partial function (regardless of the size of the model), hence *P Suc* is undefined, hence no model will be found
oops

lemma *nat-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 ⇒ zero | Suc n ⇒ suc n)*
refute
oops

'a list

lemma *P (xs::'a list)*
refute
oops

lemma ∀ *xs::'a list. P xs*
refute
oops

lemma *P [x, y]*
refute
oops

lemma *list-rec nil cons [] = nil*
refute [*maxsize=3*]
apply *simp*
done

lemma *list-rec nil cons (x#xs) = cons x xs (list-rec nil cons xs)*
refute [*maxsize=2*]
apply *simp*
done

lemma *P (list-rec nil cons xs)*
refute
oops

lemma P (*case* x *of Nil* $\Rightarrow$ *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 $\forall\, x$::*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 $\forall x::'a\ BinTree.\ P\ x$
  refute
oops

lemma $P\ (Node\ (Leaf\ x)\ (Leaf\ y))$
  refute
oops

lemma $BinTree\text{-}rec\ l\ n\ (Leaf\ x) = l\ x$
  refute $[maxsize=1]$
  apply *simp*
done

lemma $BinTree\text{-}rec\ l\ n\ (Node\ x\ y) = n\ x\ y\ (BinTree\text{-}rec\ l\ n\ x)\ (BinTree\text{-}rec\ l\ n\ y)$
  refute $[maxsize=1]$
  apply *simp*
done

lemma $P\ (BinTree\text{-}rec\ l\ n\ x)$
  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* ⇒ *number a* | *ITE b a1 a2* ⇒ *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* ⇒ *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 $X\text{-}Y\text{-}rec\text{-}1\ a\ b\ c\ d\ e\ f\ A = a$
refute [*maxsize=3*]
apply *simp*
done

lemma $X\text{-}Y\text{-}rec\text{-}1\ a\ b\ c\ d\ e\ f\ (B\ x) = b\ x\ (X\text{-}Y\text{-}rec\text{-}1\ a\ b\ c\ d\ e\ f\ x)$
refute [*maxsize=1*]
apply *simp*
done

lemma $X\text{-}Y\text{-}rec\text{-}1\ a\ b\ c\ d\ e\ f\ (C\ y) = c\ y\ (X\text{-}Y\text{-}rec\text{-}2\ a\ b\ c\ d\ e\ f\ y)$
refute [*maxsize=1*]
apply *simp*
done

lemma *X-Y-rec-2 a b c d e f* (*D x*) = *d x* (*X-Y-rec-1 a b c d e f x*)
  refute [*maxsize=1*]
  apply *simp*
done

lemma *X-Y-rec-2 a b c d e f* (*E y*) = *e y* (*X-Y-rec-2 a b c d e f y*)
  refute [*maxsize=1*]
  apply *simp*
done

lemma *X-Y-rec-2 a b c d e f F* = *f*
  refute [*maxsize=3*]
  apply *simp*
done

lemma *P* (*X-Y-rec-1 a b c d e f x*)
  refute
oops

lemma *P* (*X-Y-rec-2 a b c d e f y*)
  refute
oops

Other datatype examples

Indirect recursion is implemented via mutual recursion.

datatype *XOpt* = *CX XOpt option* | *DX bool* ⇒ *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* (λ*b*. *XOpt-rec-3 cx dx n1 s1 n2 s2* (*x b*))
  refute [*maxsize=1*]

apply *simp*
done

lemma $XOpt\text{-}rec\text{-}2\ cx\ dx\ n1\ s1\ n2\ s2\ None = n1$
refute [$maxsize=2$]
apply *simp*
done

lemma $XOpt\text{-}rec\text{-}2\ cx\ dx\ n1\ s1\ n2\ s2\ (Some\ x) = s1\ x\ (XOpt\text{-}rec\text{-}1\ cx\ dx\ n1\ s1\ n2\ s2\ x)$
refute [$maxsize=1$]
apply *simp*
done

lemma $XOpt\text{-}rec\text{-}3\ cx\ dx\ n1\ s1\ n2\ s2\ None = n2$
refute [$maxsize=2$]
apply *simp*
done

lemma $XOpt\text{-}rec\text{-}3\ cx\ dx\ n1\ s1\ n2\ s2\ (Some\ x) = s2\ x\ (XOpt\text{-}rec\text{-}1\ cx\ dx\ n1\ s1\ n2\ s2\ x)$
refute [$maxsize=1$]
apply *simp*
done

lemma $P\ (XOpt\text{-}rec\text{-}1\ cx\ dx\ n1\ s1\ n2\ s2\ x)$
refute
oops

lemma $P\ (XOpt\text{-}rec\text{-}2\ cx\ dx\ n1\ s1\ n2\ s2\ x)$
refute
oops

lemma $P\ (XOpt\text{-}rec\text{-}3\ cx\ dx\ n1\ s1\ n2\ s2\ x)$
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* (λ*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 ∀ *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* (λ*n*. *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* (λ*n*. *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* ⇒ *bool* | *D 'b list*
datatype *'c U* = *E* (*'c*, *'c U*) *T*

lemma *P* (*x*::*'c U*)
refute
oops

lemma ∀ *x*::*'c U*. *P x*
refute
oops

lemma *P* (*E* (*C* ($\lambda a.$ *True*)))
refute
oops

lemma *U-rec-1 e c d nil cons* (*E x*) = *e x* (*U-rec-2 e c d nil cons x*)
refute [*maxsize=1*]
apply *simp*
done

lemma *U-rec-2 e c d nil cons* (*C x*) = *c x*
  refute [*maxsize=1*]
  apply *simp*
done

lemma *U-rec-2 e c d nil cons* (*D x*) = *d x* (*U-rec-3 e c d nil cons x*)
  refute [*maxsize=1*]
  apply *simp*
done

lemma *U-rec-3 e c d nil cons* [] = *nil*
  refute [*maxsize=2*]
  apply *simp*
done

lemma *U-rec-3 e c d nil cons* (*x#xs*) = *cons x xs* (*U-rec-1 e c d nil cons x*) (*U-rec-3 e c d nil cons xs*)
  refute [*maxsize=1*]
  apply *simp*
done

lemma *P* (*U-rec-1 e c d nil cons x*)
  refute
oops

lemma *P* (*U-rec-2 e c d nil cons x*)
  refute
oops

lemma *P* (*U-rec-3 e c d nil cons x*)
  refute
oops

46.1.15 Records

record ('*a*, '*b*) *point* =
  xpos :: '*a*
  ypos :: '*b*

lemma (*x*::('*a*, '*b*) *point*) = *y*
  refute
oops

record ('*a*, '*b*, '*c*) *extpoint* = ('*a*, '*b*) *point* +
  ext :: '*c*

lemma (*x*::('*a*, '*b*, '*c*) *extpoint*) = *y*
  refute
oops

46.1.16 Inductively defined sets

inductive-set *arbitrarySet* :: *'a set*
where
  arbitrary : *arbitrarySet*

lemma *x* : *arbitrarySet*
  refute
oops

inductive-set *evenCard* :: *'a set set*
where
  {} : *evenCard*
| ⟦ *S* : *evenCard*; *x* ∉ *S*; *y* ∉ *S*; *x* ≠ *y* ⟧ ⟹ *S* ∪ {*x*, *y*} : *evenCard*

lemma *S* : *evenCard*
  refute
oops

inductive-set
  even :: *nat set*
  and *odd* :: *nat set*
where
  0 : *even*
| *n* : *even* ⟹ *Suc n* : *odd*
| *n* : *odd* ⟹ *Suc n* : *even*

lemma *n* : *odd*

oops

consts *f* :: *'a* ⇒ *'a*

inductive-set
  a-even :: *'a set*
  and *a-odd* :: *'a set*
where
  arbitrary : *a-even*
| *x* : *a-even* ⟹ *f x* : *a-odd*
| *x* : *a-odd* ⟹ *f x* : *a-even*

lemma *x* : *a-odd*
  refute — finds a model of size 2, as expected
oops

46.1.17 Examples involving special functions

lemma *card x* = *0*
  refute
oops

lemma *finite x*
  refute — no finite countermodel exists
oops

lemma $(x::nat) + y = 0$
  refute
oops

lemma $(x::nat) = x + x$
  refute
oops

lemma $(x::nat) - y + y = x$
  refute
oops

lemma $(x::nat) = x * x$
  refute
oops

lemma $(x::nat) < x + y$
  refute
oops

lemma *xs* @ [] = *ys* @ []
  refute
oops

lemma *xs* @ *ys* = *ys* @ *xs*
  refute
oops

lemma *f* (*lfp f*) = *lfp f*
  refute
oops

lemma *f* (*gfp f*) = *gfp f*
  refute
oops

lemma *lfp f* = *gfp f*
  refute
oops

46.1.18 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-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-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-const :: $'a \Rightarrow 'a$

axclass *classD* < *type*
  classD-ax: *classD-const* (*classD-const* x) = *classD-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 $OFCLASS('a::type,\ type\text{-}class)$
  refute — no countermodel exists
  apply *intro-classes*
done

lemma $OFCLASS('a::classC,\ type\text{-}class)$
  refute — no countermodel exists
  apply *intro-classes*
done

lemma $OFCLASS('a,\ classB\text{-}class)$
  refute — no countermodel exists
  apply *intro-classes*
  apply *simp*
done

lemma $OFCLASS('a::type,\ classC\text{-}class)$
  refute
oops

Overloading:

consts *inverse* :: $'a \Rightarrow 'a$

defs (**overloaded**)
  inverse-bool: *inverse* $(b::bool)$ $==$ ~ 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

47 Examples for the 'quickcheck' command

theory *Quickcheck-Examples* **imports** *Main* **begin**

The 'quickcheck' command allows to find counterexamples by evaluating formulae under an assignment of free variables to random values. In contrast to 'refute', it can deal with inductive datatypes, but cannot handle quantifiers.

47.1 Lists

theorem *map g (map f xs) = map (g o f) xs*
  quickcheck
  oops

theorem *map g (map f xs) = map (f o g) xs*
  quickcheck
  oops

theorem *rev (xs @ ys) = rev ys @ rev xs*
  quickcheck
  oops

theorem *rev (xs @ ys) = rev xs @ rev ys*
  quickcheck
  oops

theorem *rev (rev xs) = xs*
  quickcheck
  oops

theorem *rev xs = xs*
  quickcheck
  oops

consts
  occurs :: *'a ⇒ 'a list ⇒ nat*
primrec
  occurs a [] = 0
  occurs a (x#xs) = (if (x=a) then Suc(occurs a xs) else occurs a xs)

consts
  del1 :: *'a ⇒ 'a list ⇒ 'a list*
primrec
  del1 a [] = []
  del1 a (x#xs) = (if (x=a) then xs else (x#del1 a xs))

lemma *Suc (occurs a (del1 a xs)) = occurs a xs*
  — Wrong. Precondition needed.
  quickcheck
  oops

lemma *xs ~= [] ⟶ 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 $occurs\ b\ xs = 0 \lor a=b \longrightarrow occurs\ a\ xs = occurs\ b\ (replace\ a\ b\ xs)$
quickcheck
apply (*induct-tac xs*)
apply *auto*
done

47.2 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 ⇒ 'a list
primrec
  inOrder (Tip a)= [a]
  inOrder (Node f x y) = (inOrder x)@[f]@(inOrder y)

consts
  root :: 'a ntree ⇒ '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.