Miscellaneous HOL Examples

April 19, 2009

Contents

1 An experimental alternative numeral representation.

theory *Numeral*
imports *Int Inductive*
begin

1.1 The *num* type

datatype *num* = *One* | *Dig0 num* | *Dig1 num*

Increment function for type *Numeral.num*

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

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

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

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

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

lemma *nat-of-num-neq-0*: *nat-of-num x* ≠ *0*
  by (*induct x*) *simp-all*

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

lemma *num-of-nat-double*:
  0 < *n* ⟹ *num-of-nat* (*n* + *n*) = *Dig0* (*num-of-nat n*)
  by (*induct n*) *simp-all*

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

lemma *nat-of-num-inverse*: *num-of-nat* (*nat-of-num x*) = *x*

by (*induct x*) (*simp-all add*: *num-of-nat-double nat-of-num-pos*)

lemma *num-of-nat-inverse*: $0 < n \Longrightarrow$ *nat-of-num* (*num-of-nat n*) = *n*
by (*induct n*) (*simp-all add*: *nat-of-num-inc*)

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

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

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

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

1.2 Numeral operations

ML ⟪
structure DigSimps =
NamedThmsFun(*val name* = *numeral*; *val description* = *Simplification rules for numerals*)
⟫

setup *DigSimps.setup*

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

definition *plus-num* :: *num* ⇒ *num* ⇒ *num* **where**
[*code del*]: *m* + *n* = *num-of-nat* (*nat-of-num m* + *nat-of-num n*)

definition *times-num* :: *num* ⇒ *num* ⇒ *num* **where**
[*code del*]: *m* ∗ *n* = *num-of-nat* (*nat-of-num m* ∗ *nat-of-num n*)

definition *less-eq-num* :: *num* ⇒ *num* ⇒ *bool* **where**
[*code del*]: *m* ≤ *n* ⟷ *nat-of-num m* ≤ *nat-of-num n*

definition *less-num* :: *num* ⇒ *num* ⇒ *bool* **where**
[*code del*]: *m* < *n* ⟷ *nat-of-num m* < *nat-of-num n*

instance ..

end

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

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

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

lemma *Dig-times* [*numeral*, *simp*, *code*]:
One ∗ *One* = *One*
One ∗ *Dig0 n* = *Dig0 n*
One ∗ *Dig1 n* = *Dig1 n*
Dig0 n ∗ *One* = *Dig0 n*
Dig0 n ∗ *Dig0 m* = *Dig0* (*n* ∗ *Dig0 m*)
Dig0 n ∗ *Dig1 m* = *Dig0* (*n* ∗ *Dig1 m*)
Dig1 n ∗ *One* = *Dig1 n*
Dig1 n ∗ *Dig0 m* = *Dig0* (*n* ∗ *Dig0 m* + *m*)
Dig1 n ∗ *Dig1 m* = *Dig1* (*n* ∗ *Dig1 m* + *m*)
by (*simp-all add*: *num-eq-iff nat-of-num-add nat-of-num-mult*
left-distrib right-distrib)

lemma *Dig-eq*:
One = *One* ⟷ *True*
One = *Dig0 n* ⟷ *False*
One = *Dig1 n* ⟷ *False*
Dig0 m = *One* ⟷ *False*
Dig1 m = *One* ⟷ *False*
Dig0 m = *Dig0 n* ⟷ *m* = *n*
Dig0 m = *Dig1 n* ⟷ *False*
Dig1 m = *Dig0 n* ⟷ *False*
Dig1 m = *Dig1 n* ⟷ *m* = *n*
by *simp-all*

lemma *less-eq-num-code* [*numeral*, *simp*, *code*]:
One ≤ *n* ⟷ *True*
Dig0 m ≤ *One* ⟷ *False*
Dig1 m ≤ *One* ⟷ *False*
Dig0 m ≤ *Dig0 n* ⟷ *m* ≤ *n*
Dig0 m ≤ *Dig1 n* ⟷ *m* ≤ *n*
Dig1 m ≤ *Dig1 n* ⟷ *m* ≤ *n*
Dig1 m ≤ *Dig0 n* ⟷ *m* < *n*
using *nat-of-num-pos* [*of n*] *nat-of-num-pos* [*of m*]
by (*auto simp add*: *less-eq-num-def less-num-def*)

lemma *less-num-code* [*numeral*, *simp*, *code*]:
m < *One* ⟷ *False*
One < *One* ⟷ *False*
One < *Dig0 n* ⟷ *True*
One < *Dig1 n* ⟷ *True*
Dig0 m < *Dig0 n* ⟷ *m* < *n*
Dig0 m < *Dig1 n* ⟷ *m* ≤ *n*
Dig1 m < *Dig1 n* ⟷ *m* < *n*
Dig1 m < *Dig0 n* ⟷ *m* < *n*
using *nat-of-num-pos* [*of n*] *nat-of-num-pos* [*of m*]
by (*auto simp add*: *less-eq-num-def less-num-def*)

Rules using *One* and *inc* as constructors

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

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

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

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

A double-and-decrement function

primrec *DigM* :: *num* ⇒ *num* **where**
DigM One = *One*
| *DigM* (*Dig0 n*) = *Dig1* (*DigM n*)
| *DigM* (*Dig1 n*) = *Dig1* (*Dig0 n*)

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

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

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

Squaring and exponentiation

primrec *square* :: *num* ⇒ *num* **where**
square One = *One*
| *square* (*Dig0 n*) = *Dig0* (*Dig0* (*square n*))
| *square* (*Dig1 n*) = *Dig1* (*Dig0* (*square n* + *n*))

primrec *pow* :: *num* ⇒ *num* ⇒ *num*
where
pow x One = *x*
| *pow x* (*Dig0 y*) = *square* (*pow x y*)
| *pow x* (*Dig1 y*) = *x* ∗ *square* (*pow x y*)

1.3 Binary numerals

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

class *semiring-numeral* = *semiring* + *monoid-mult*
begin

primrec *of-num* :: *num* ⇒ *'a* **where**
of-num-one [*numeral*]: *of-num One* = *1*
| *of-num* (*Dig0 n*) = *of-num n* + *of-num n*
| *of-num* (*Dig1 n*) = *of-num n* + *of-num n* + *1*

lemma *of-num-inc*: *of-num* (*inc x*) = *of-num x* + *1*
by (*induct x*) (*simp-all add*: *add-ac*)

declare *of-num.simps* [*simp del*]

end

ML stuff and syntax.

ML ⟪
fun mk-num 1 = @{*term One*}

```
  | mk-num k =
      let
        val (l, b) = Integer.div-mod k 2;
        val bit = (if b = 0 then @{term Dig0} else @{term Dig1});
      in bit $ (mk-num l) end;

fun dest-num @{term One} = 1
  | dest-num (@{term Dig0} $ n) = 2 * dest-num n
  | dest-num (@{term Dig1} $ n) = 2 * dest-num n + 1;

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

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

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

syntax
  -Numerals :: xnum ⇒ 'a    (-)

parse-translation ⟨⟨
let
  fun num-of-int n = if n > 0 then case IntInf.quotRem (n, 2)
     of (0, 1) => Const (@{const-name One}, dummyT)
      | (n, 0) => Const (@{const-name Dig0}, dummyT) $ num-of-int n
      | (n, 1) => Const (@{const-name Dig1}, dummyT) $ num-of-int n
    else raise Match;
  fun numeral-tr [Free (num, -)] =
        let
          val {leading-zeros, value, ...} = Syntax.read-xnum num;
          val - = leading-zeros = 0 andalso value > 0
            orelse error (Bad numeral: ^ num);
        in Const (@{const-name of-num}, @{typ num} --> dummyT) $ num-of-int
value end
    | numeral-tr ts = raise TERM (numeral-tr, ts);
in [(-Numerals, numeral-tr)] end
⟩⟩

typed-print-translation ⟨⟨
let
  fun dig b n = b + 2 * n;
  fun int-of-num' (Const (@{const-syntax Dig0}, -) $ n) =
        dig 0 (int-of-num' n)
    | int-of-num' (Const (@{const-syntax Dig1}, -) $ n) =
        dig 1 (int-of-num' n)
    | int-of-num' (Const (@{const-syntax One}, -)) = 1;
```

```
  fun num-tr' show-sorts T [n] =
    let
      val k = int-of-num' n;
      val t' = Syntax.const -Numerals $ Syntax.free (# ^ string-of-int k);
    in case T
     of Type (fun, [-, T']) =>
        if not (! show-types) andalso can Term.dest-Type T' then t'
         else Syntax.const Syntax.constrainC $ t' $ Syntax.term-of-typ show-sorts
T'
      | T' => if T' = dummyT then t' else raise Match
    end;
in [(@{const-syntax of-num}, num-tr')] end
⟩⟩
```

1.4 Class-specific numeral rules

of-num is a morphism.

1.4.1 Class *semiring-numeral*

context *semiring-numeral*
begin

abbreviation *Num1* ≡ *of-num One*

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

lemma *of-num-plus-one* [*numeral*]:
  of-num n + 1 = of-num (*n + One*)
  by (*rule sym*, *induct n*) (*simp-all add*: *of-num.simps add-ac*)

lemma *of-num-one-plus* [*numeral*]:
  1 + of-num n = of-num (*n + One*)
  unfolding *of-num-plus-one* [*symmetric*] *add-commute* **..**

lemma *of-num-plus* [*numeral*]:
  of-num m + of-num n = of-num (*m + n*)
  by (*induct n rule*: *num-induct*)
    (*simp-all add*: *add-One add-inc of-num-one of-num-inc add-ac*)

lemma *of-num-times-one* [*numeral*]:
  *of-num n * 1 = of-num n*
  by *simp*

lemma *of-num-one-times* [*numeral*]:
  *1 * of-num n = of-num n*

by *simp*

lemma *of-num-times* [*numeral*]:
of-num m ∗ *of-num n* = *of-num* (*m* ∗ *n*)
by (*induct n rule*: *num-induct*)
(*simp-all add*: *of-num-plus* [*symmetric*] *mult-One mult-inc*
semiring-class.right-distrib right-distrib of-num-one of-num-inc)

end

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

context *semiring-1*
begin

subclass *semiring-numeral* **..**

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

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

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

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

end

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

1.4.3 Equality: class *semiring-char-0*

context *semiring-char-0*

begin

lemma *of-num-eq-iff* [*numeral*]:
  of-num m = of-num n ⟷ m = n
  unfolding *of-nat-of-num* [*symmetric*] *nat-of-num-of-num* [*symmetric*]
    of-nat-eq-iff num-eq-iff **..**

lemma *of-num-eq-one-iff* [*numeral*]:
  of-num n = 1 ⟷ n = One
proof −
  have *of-num n = of-num One ⟷ n = One* **unfolding** *of-num-eq-iff* **..**
  then show *?thesis* **by** (*simp add*: *of-num-one*)
qed

lemma *one-eq-of-num-iff* [*numeral*]:
  1 = of-num n ⟷ n = One
  unfolding *of-num-eq-one-iff* [*symmetric*] **by** *auto*

end

1.4.4 Comparisons: class *ordered-semidom*

Could be perhaps more general than here.

context *ordered-semidom*
begin

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

lemma *of-num-less-eq-iff* [*numeral*]: *of-num m ≤ of-num n ⟷ m ≤ n*
proof −
  have *of-nat* (*of-num m*) ≤ *of-nat* (*of-num n*) ⟷ *m ≤ n*
    unfolding *less-eq-num-def nat-of-num-of-num of-nat-le-iff* **..**
  then show *?thesis* **by** (*simp add*: *of-nat-of-num*)
qed

lemma *of-num-less-eq-one-iff* [*numeral*]: *of-num n ≤ 1 ⟷ n = One*
proof −
  have *of-num n ≤ of-num One ⟷ n = One*
    by (*cases n*) (*simp-all add*: *of-num-less-eq-iff*)
  then show *?thesis* **by** (*simp add*: *of-num-one*)
qed

lemma *one-less-eq-of-num-iff* [*numeral*]: *1 ≤ of-num n*
proof −
  have *of-num One ≤ of-num n*
    by (*cases n*) (*simp-all add*: *of-num-less-eq-iff*)
  then show *?thesis* **by** (*simp add*: *of-num-one*)
qed

lemma *of-num-less-iff* [*numeral*]: *of-num m* < *of-num n* ⟷ *m* < *n*
proof −
have *of-nat* (*of-num m*) < *of-nat* (*of-num n*) ⟷ *m* < *n*
unfolding *less-num-def nat-of-num-of-num of-nat-less-iff* **..**
then show *?thesis* **by** (*simp add*: *of-nat-of-num*)
qed

lemma *of-num-less-one-iff* [*numeral*]: ¬ *of-num n* < *1*
proof −
have ¬ *of-num n* < *of-num One*
by (*cases n*) (*simp-all add*: *of-num-less-iff*)
then show *?thesis* **by** (*simp add*: *of-num-one*)
qed

lemma *one-less-of-num-iff* [*numeral*]: *1* < *of-num n* ⟷ *n* ≠ *One*
proof −
have *of-num One* < *of-num n* ⟷ *n* ≠ *One*
by (*cases n*) (*simp-all add*: *of-num-less-iff*)
then show *?thesis* **by** (*simp add*: *of-num-one*)
qed

lemma *of-num-nonneg* [*numeral*]: *0* ≤ *of-num n*
by (*induct n*) (*simp-all add*: *of-num.simps add-nonneg-nonneg*)

lemma *of-num-less-zero-iff* [*numeral*]: ¬ *of-num n* < *0*
by (*simp add*: *not-less of-num-nonneg*)

lemma *of-num-le-zero-iff* [*numeral*]: ¬ *of-num n* ≤ *0*
by (*simp add*: *not-le of-num-pos*)

end

context *ordered-idom*
begin

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

lemma *minus-of-num-less-one-iff*: − *of-num n* < *1*
using *minus-of-num-less-of-num-iff* [*of n One*] **by** (*simp add*: *of-num-one*)

lemma *minus-one-less-of-num-iff*: − *1* < *of-num n*
using *minus-of-num-less-of-num-iff* [*of One n*] **by** (*simp add*: *of-num-one*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

1.4.5 Structures with subtraction: class *semiring-1-minus*

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

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

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

end

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

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

lemma *Dig-of-num-zero*:
shows *of-num n* − *of-num n* = *0*
by (*rule minus-inverts-plus1*) *simp*

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

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

simproc-setup *numeral-minus* (*of-num m* − *of-num n*) = ⟪
let

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

lemma Dig-of-num-minus-zero [numeral]:
  of-num n - 0 = of-num n
  by (simp add: minus-inverts-plus1)

lemma Dig-one-minus-zero [numeral]:
  1 - 0 = 1
  by (simp add: minus-inverts-plus1)

lemma Dig-one-minus-one [numeral]:
  1 - 1 = 0
  by (simp add: minus-inverts-plus1)

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

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

end
```

1.4.6 Structures with negation: class *ring-1*

context *ring-1*
begin

subclass *semiring-1-minus*
proof qed (*simp-all add*: *algebra-simps*)

lemma *Dig-zero-minus-of-num* [*numeral*]:
0 − of-num n = − of-num n
by *simp*

lemma *Dig-zero-minus-one* [*numeral*]:
0 − 1 = − 1
by *simp*

lemma *Dig-uminus-uminus* [*numeral*]:
− (− of-num n) = of-num n
by *simp*

lemma *Dig-plus-uminus* [*numeral*]:
of-num m + − of-num n = of-num m − of-num n
− of-num m + of-num n = of-num n − of-num m
− of-num m + − of-num n = − (of-num m + of-num n)
of-num m − − of-num n = of-num m + of-num n
− of-num m − of-num n = − (of-num m + of-num n)
− of-num m − − of-num n = of-num n − of-num m
by (*simp-all add*: *diff-minus add-commute*)

lemma *Dig-times-uminus* [*numeral*]:
− of-num n ∗ of-num m = − (of-num n ∗ of-num m)
of-num n ∗ − of-num m = − (of-num n ∗ of-num m)
− of-num n ∗ − of-num m = of-num n ∗ of-num m
by (*simp-all add*: *minus-mult-left* [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *of-int-of-num* [*numeral*]: *of-int* (*of-num n*) = *of-num n*
by (*induct n*)
(*simp-all only*: *of-num.simps semiring-numeral-class.of-num.simps of-int-add*, *simp-all*)

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

end

1.4.7 Structures with exponentiation

lemma *of-num-square*: *of-num* (*square x*) = *of-num x ∗ of-num x*
by (*induct x*)
(*simp-all add*: *of-num.simps of-num-plus* [*symmetric*] *algebra-simps*)

lemma *of-num-pow*:
(*of-num* (*pow x y*)::'*a*::{*semiring-numeral*,*recpower*}) = *of-num x* ^ *of-num y*
by (*induct y*)
(*simp-all add*: *of-num.simps of-num-square of-num-times* [*symmetric*]
power-Suc power-add)

lemma *power-of-num* [*numeral*]:
of-num x ^ *of-num y* = (*of-num* (*pow x y*)::'*a*::{*semiring-numeral*,*recpower*})
by (*rule of-num-pow* [*symmetric*])

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

lemma *power-minus-one-double*:
(− *1*) ^ (*n* + *n*) = (*1*::'*a*::{*ring-1*,*recpower*})
by (*induct n*) (*simp-all add*: *power-Suc*)

lemma *power-minus-Dig0* [*numeral*]:
fixes *x* :: '*a*::{*ring-1*,*recpower*}
shows (− *x*) ^ *of-num* (*Dig0 n*) = *x* ^ *of-num* (*Dig0 n*)
by (*subst power-minus*)
(*simp add*: *of-num.simps power-minus-one-double*)

lemma *power-minus-Dig1* [*numeral*]:
fixes *x* :: '*a*::{*ring-1*,*recpower*}
shows (− *x*) ^ *of-num* (*Dig1 n*) = − (*x* ^ *of-num* (*Dig1 n*))
by (*subst power-minus*)
(*simp add*: *of-num.simps power-Suc power-minus-one-double*)

declare *power-one* [*numeral*]

1.4.8 Greetings to *nat*.

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

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

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

declare *diff-0-eq-0* [*numeral*]

1.5 Code generator setup for *int*

definition *Pls* :: *num* ⇒ *int* **where**
  [*simp*, *code post*]: *Pls n* = *of-num n*

definition *Mns* :: *num* ⇒ *int* **where**
  [*simp*, *code post*]: *Mns n* = − *of-num n*

code-datatype *0*::*int Pls Mns*

lemmas [*code inline*] = *Pls-def* [*symmetric*] *Mns-def* [*symmetric*]

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

definition *dup* :: *int* ⇒ *int* **where**
  [*code del*]: *dup k* = *2* ∗ *k*

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

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

lemma [*code*, *code del*]:
  (*1* :: *int*) = *1*
  (*op* + :: *int* ⇒ *int* ⇒ *int*) = *op* +
  (*uminus* :: *int* ⇒ *int*) = *uminus*
  (*op* − :: *int* ⇒ *int* ⇒ *int*) = *op* −
  (*op* ∗ :: *int* ⇒ *int* ⇒ *int*) = *op* ∗
  (*eq-class.eq* :: *int* ⇒ *int* ⇒ *bool*) = *eq-class.eq*
  (*op* ≤ :: *int* ⇒ *int* ⇒ *bool*) = *op* ≤
  (*op* < :: *int* ⇒ *int* ⇒ *bool*) = *op* <
  by *rule*+

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

lemma *plus-int-code* [*code*]:
k + 0 = (k::int)
0 + l = (l::int)
Pls m + Pls n = Pls (m + n)
Pls m − Pls n = sub m n
Mns m + Mns n = Mns (m + n)
Mns m − Mns n = sub n m
by (*simp-all add*: *of-num-plus* [*symmetric*])

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

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

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

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

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

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

lemma [*code inline del*]: (*0*::*int*) ≡ *Numeral0* **by** *simp*
lemma [*code inline del*]: (*1*::*int*) ≡ *Numeral1* **by** *simp*
declare *zero-is-num-zero* [*code inline del*]
declare *one-is-num-one* [*code inline del*]

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

1.6 Numeral equations as default simplification rules

TODO. Be more precise here with respect to subsumed facts. Or use named theorems anyway.

declare (**in** *semiring-numeral*) *numeral* [*simp*]
declare (**in** *semiring-1*) *numeral* [*simp*]
declare (**in** *semiring-char-0*) *numeral* [*simp*]
declare (**in** *ring-1*) *numeral* [*simp*]
thm *numeral*

Toy examples

definition *bar* ⟷ *#4* ∗ *#2* + *#7* = (*#8* :: *nat*) ∧ *#4* ∗ *#2* + *#7* ≥ (*#8* :: *int*) − *#3*
code-thms *bar*

export-code *bar* **in** *Haskell* **file** –
export-code *bar* **in** *OCaml* **module-name** *Foo* **file** –
ML ‹‹ @{*code bar*} ››

end

2 Foundations of HOL

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

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

2.1 Pure Logic

classes *type*
defaultsort *type*

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

2.1.1 Basic logical connectives

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

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

2.1.2 Extensional equality

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

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

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

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

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

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

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

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

2.1.3 Derived connectives

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

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

definition
  not :: $o \Rightarrow o$ $(\neg\ \text{-}\ [40]\ 40)$ **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
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 ‹A›
also note ‹B›
finally show C **.**
qed

```
  qed
qed

theorem conjE [elim]: A ∧ B ⟹ (A ⟹ B ⟹ C) ⟹ C
proof (unfold conj-def)
  assume c: ∀ C. (A ⟶ B ⟶ C) ⟶ C
  assume A ⟹ B ⟹ C
  moreover {
    from c have (A ⟶ B ⟶ A) ⟶ A ..
    also have A ⟶ B ⟶ A
    proof
      assume A
      then show B ⟶ A ..
    qed
    finally have A .
  } moreover {
    from c have (A ⟶ B ⟶ B) ⟶ B ..
    also have A ⟶ B ⟶ B
    proof
      show B ⟶ B ..
    qed
    finally have B .
  } ultimately show C .
qed

theorem disjI1 [intro]: A ⟹ A ∨ B
proof (unfold disj-def)
  assume A
  show ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  proof
    fix C show (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
    proof
      assume A ⟶ C
      also note ‹A›
      finally have C .
      then show (B ⟶ C) ⟶ C ..
    qed
  qed
qed

theorem disjI2 [intro]: B ⟹ A ∨ B
proof (unfold disj-def)
  assume B
  show ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  proof
    fix C show (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
    proof
      show (B ⟶ C) ⟶ C
      proof
```

```
      assume B ⟶ C
      also note ‹B›
      finally show C .
    qed
  qed
 qed

theorem disjE [elim]: A ∨ B ⟹ (A ⟹ C) ⟹ (B ⟹ C) ⟹ C
proof (unfold disj-def)
  assume c: ∀ C. (A ⟶ C) ⟶ (B ⟶ C) ⟶ C
  assume r1: A ⟹ C and r2: B ⟹ C
  from c have (A ⟶ C) ⟶ (B ⟶ C) ⟶ C ..
  also have A ⟶ C
  proof
    assume A then show C by (rule r1)
  qed
  also have B ⟶ C
  proof
    assume B then show C by (rule r2)
  qed
  finally show C .
qed

theorem exI [intro]: P a ⟹ ∃ x. P x
proof (unfold Ex-def)
  assume P a
  show ∀ C. (∀ x. P x ⟶ C) ⟶ C
  proof
    fix C show (∀ x. P x ⟶ C) ⟶ C
    proof
      assume ∀ x. P x ⟶ C
      then have P a ⟶ C ..
      also note ‹P a›
      finally show C .
    qed
  qed
qed

theorem exE [elim]: ∃ x. P x ⟹ (⋀x. P x ⟹ C) ⟹ C
proof (unfold Ex-def)
  assume c: ∀ C. (∀ x. P x ⟶ C) ⟶ C
  assume r: ⋀x. P x ⟹ C
  from c have (∀ x. P x ⟶ C) ⟶ C ..
  also have ∀ x. P x ⟶ C
  proof
    fix x show P x ⟶ C
    proof
      assume P x
```

```
    then show C by (rule r)
   qed
  qed
  finally show C .
qed
```

2.2 Classical logic

```
locale classical =
  assumes classical: (¬ A ⟹ A) ⟹ A

theorem (in classical)
  Peirce's-Law: ((A ⟶ B) ⟶ A) ⟶ A
proof
  assume a: (A ⟶ B) ⟶ A
  show A
  proof (rule classical)
    assume ¬ A
    have A ⟶ B
    proof
      assume A
      with ‹¬ A› show B by (rule contradiction)
    qed
    with a show A ..
  qed
qed

theorem (in classical)
  double-negation: ¬ ¬ A ⟹ A
proof −
  assume ¬ ¬ A
  show A
  proof (rule classical)
    assume ¬ A
    with ‹¬ ¬ A› show ?thesis by (rule contradiction)
  qed
qed

theorem (in classical)
  tertium-non-datur: A ∨ ¬ A
proof (rule double-negation)
  show ¬ ¬ (A ∨ ¬ A)
  proof
    assume ¬ (A ∨ ¬ A)
    have ¬ A
    proof
      assume A then have A ∨ ¬ A ..
      with ‹¬ (A ∨ ¬ A)› show ⊥ 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
```

3 Abstract Natural Numbers primitive recursion

```
theory Abstract-NAT
imports Main
begin
```

Axiomatic Natural Numbers (Peano) – a monomorphic theory.

```
locale NAT =
  fixes zero :: 'n
    and succ :: 'n ⇒ '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* ≠ *0* **by** *simp*
fix *P*
assume *zero*: *P 0*
and *succ*: ⋀*n*. *P n* ⟹ *P* (*Suc n*)
show *P n*
proof (*induct n*)
case *0* **show** *?case* **by** (*rule zero*)
next
case *Suc* **then show** *?case* **by** (*rule succ*)
qed
qed

end

4 Proof by guessing

theory *Guess*
imports *Main*
begin

lemma *True*
proof

have *1*: ∃ *x*. *x* = *x* **by** *simp*

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

have *2*: ∃ *x y*. *x* = *x* & *y* = *y* **by** *simp*
from *2* **guess** **apply** − **apply** (*erule exE conjE*)+ **done**
from *2* **guess** *x* **apply** − **apply** (*erule exE conjE*)+ **done**
from *2* **guess** *x y* **apply** − **apply** (*erule exE conjE*)+ **done**
from *2* **guess** *x* :: *'a* **and** *y* :: *'b* **apply** − **apply** (*erule exE conjE*)+ **done**
from *2* **guess** *x y* :: *nat* **apply** − **apply** (*erule exE conjE*)+ **done**

qed

end

5 Simple and efficient binary numerals

theory *Binary*
imports *Main*
begin

5.1 Binary representation of natural numbers

definition
bit :: *nat* ⇒ *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* (@{*const-name False*}, -)) = *0*
| *dest-bit* (*Const* (@{*const-name True*}, -)) = *1*
| *dest-bit t* = *raise TERM* (*dest-bit*, [*t*]);

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

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

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

5.2 Direct operations – plain normalization

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

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

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

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

5.3 Indirect operations – ML will produce witnesses

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

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

lemma *binary-diff*:
 fixes *n* :: *nat*

```
  shows m ≡ n + k ⟹ m − n ≡ k
    and n ≡ m + k ⟹ m − n ≡ 0
  by simp-all

lemma binary-divmod:
  fixes n :: nat
  assumes m ≡ n * k + l and 0 < n and l < n
  shows m div n ≡ k
    and m mod n ≡ l
proof −
  from ‹m ≡ n * k + l› have m = l + k * n by simp
  with ‹0 < n› and ‹l < n› show m div n ≡ k and m mod n ≡ 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 = ⟪
  Scan.succeed (K (SIMPLE-METHOD'
    (full-simp-tac
      (HOL-basic-ss
        addsimps @{thms binary-simps}
        addsimprocs
         [@{simproc binary-nat-less-eq},
          @{simproc binary-nat-less},
          @{simproc binary-nat-diff},
          @{simproc binary-nat-div},
          @{simproc binary-nat-mod}])))))
⟫ binary simplification
```

5.4 Concrete syntax

syntax
-Binary :: *num-const* ⇒ *'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*
⟫

5.5 Examples

lemma $*6* = *6*
by (*simp add*: *bit-simps*)

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

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

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

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

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

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

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

lemma $*11111111112222222222333333333344444444444* - $*9988776655443322111*
=
$*11111111112222222222323344556689001122333*

by *binary-simp*

lemma (*11111111112222222222333333333344444444444*::*nat*) − *998877665544332211*
=
11111111112222222222323344556689001122333
by *simp*

lemma (*11111111112222222222333333333344444444444*::*int*) − *998877665544332211*
=
11111111112222222222323344556689001122333
by *simp*

lemma $*11111111112222222222333333333344444444444* ∗ $*998877665544332211*
=
$*1109864072938022197293802219729380221972383090160869185684*
by *binary-simp*

lemma $*11111111112222222222333333333344444444444* ∗ $*998877665544332211*
−
$*55555555556666666666777777777788888888888* =
$*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 $*11111111112222222222333333333344444444444* < $*998877665544332211*
= *False*
by *binary-simp*

lemma $*998877665544332211* < $*11111111112222222222333333333344444444444*
= *True*
by *binary-simp*

lemma $*11111111112222222222333333333344444444444* <= $*998877665544332211*
= *False*
by *binary-simp*

lemma $*998877665544332211* <= $*11111111112222222222333333333344444444444*
= *True*

by *binary-simp*

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

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

lemma $*111111111122222222223333333334444444444 div $998877665544332211*
=
$*11123595506730337078 75*
by *binary-simp*

lemma $*111111111122222222223333333334444444444 mod $998877665544332211*
=
$*422451743175828 19*
by *binary-simp*

lemma (*111111111122222222223333333334444444444*::*int*) *div 998877665544332211*
=
11123595506730337078 75
by *simp* — legacy numerals: 30 times slower

lemma (*111111111122222222223333333334444444444*::*int*) *mod 998877665544332211*
=
422451743175828 19
by *simp* — legacy numerals: 30 times slower

end

6 Examples of recdef definitions

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

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

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

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

lemma *fib 7 = 13*
by *simp*

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

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

Not handled automatically: too complicated.

consts *variant* :: *nat* * *nat list => nat*
recdef (**permissive**) *variant measure* (λ(*n*,*ns*). *size* (*filter* (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* ((λ*y*. *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 λ*x*. *mapf x* instead of *mapf*.

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

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

Removing the termination condition from the generated thms:

lemma *mapf* (*Suc n*) = *concat* (*map mapf* (*replicate n n*))

apply (*simp add*: *mapf.simps mapf-tc*)
done

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

end

7 Examples of function definitions

theory *Fundefs*
imports *Main*
begin

7.1 Very basic

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

partial simp and induction rules:

thm *fib.psimps*
thm *fib.pinduct*

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

thm *fib.cases*

total simp and induction rules:

thm *fib.simps*
thm *fib.induct*

7.2 Currying

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

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

7.3 Nested recursion

function *nz*
where
  nz 0 = *0*

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

lemma nz-is-zero: — A lemma we need to prove termination
  assumes trm: nz-dom x
  shows nz x = 0
using trm
by induct auto

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

thm nz.simps
thm nz.induct
```

Here comes McCarthy's 91-function

```
function f91 :: nat => nat
where
  f91 n = (if 100 < n then n - 10 else f91 (f91 (n + 11)))
by pat-completeness auto


lemma f91-estimate:
  assumes trm: f91-dom n
  shows n < f91 n + 11
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
```

Now trivial (even though it does not belong here):

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

7.4 More general patterns

7.4.1 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
```

7.4.2 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
```

7.5 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
```

7.6 Definitions in local contexts

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

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

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

thm *foldL.simps*

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

thm *foldR-foldL*

end

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

7.7 Regression tests

The following examples mainly serve as tests for the function package

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

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

function *h* :: *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 ⇒ nat
where
  k x = (let a = x; b = x in k x)
  by pat-completeness auto



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



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



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

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

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

declare lem[simp]

fun tinc :: nat tree ⇒ nat tree
where
  tinc (Leaf n) = Leaf (Suc n)
```

```
| tinc (Branch l) = Branch (map tinc l)



record point =
  Xcoord :: int
  Ycoord :: int

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

8 Examples of automatically derived induction rules

theory *Induction-Scheme*
imports *Main*
begin

8.1 Some simple induction principles on nat

lemma *nat-standard-induct*:
⟦*P 0*; ⋀*n*. *P n* ⟹ *P* (*Suc n*)⟧ ⟹ *P x*
by *induct-scheme* (*pat-completeness*, *lexicographic-order*)

lemma *nat-induct2*:
⟦ *P 0*; *P* (*Suc 0*); ⋀*k*. *P k* ==> *P* (*Suc k*) ==> *P* (*Suc* (*Suc k*)) ⟧
⟹ *P n*
by *induct-scheme* (*pat-completeness*, *lexicographic-order*)

lemma *minus-one-induct*:
⟦⋀*n*::*nat*. (*n* ≠ *0* ⟹ *P* (*n* − *1*)) ⟹ *P n*⟧ ⟹ *P x*
by *induct-scheme* (*pat-completeness*, *lexicographic-order*)

theorem *diff-induct*:
(!!*x*. *P x 0*) ==> (!!*y*. *P 0* (*Suc y*)) ==>
(!!*x y*. *P x y* ==> *P* (*Suc x*) (*Suc y*)) ==> *P m n*
by *induct-scheme* (*pat-completeness*, *lexicographic-order*)

lemma *list-induct2′*:

⟦ P [] []; ⋀x xs. P (x#xs) []; ⋀y ys. P [] (y#ys); ⋀x xs y ys. P xs ys ⟹ P (x#xs) (y#ys) ⟧ ⟹ P xs ys
by *induct-scheme* (*pat-completeness*, *lexicographic-order*)

theorem *even-odd-induct*:
assumes R *0*
assumes Q *0*
assumes ⋀n. Q n ⟹ R (*Suc* n)
assumes ⋀n. R n ⟹ Q (*Suc* n)
shows R n Q n
using *assms*
by *induct-scheme* (*pat-completeness+*, *lexicographic-order*)

end

9 Some of the results in Inductive Invariants for Nested Recursion

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

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

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

definition
indinv :: ('*a* ∗ '*a*) *set* => ('*a* => '*b* => *bool*) => (('*a* => '*b*) => ('*a* => '*b*)) => *bool* **where**
indinv r S F = (∀f x. (∀y. (y,x) : r −−> S y (f y)) −−> S x (F f x))

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

definition
indinv-on :: ('*a* ∗ '*a*) *set* => '*a set* => ('*a* => '*b* => *bool*) => (('*a* => '*b*) => ('*a* => '*b*)) => *bool* **where**
indinv-on r D S F = (∀f. ∀x∈D. (∀y∈D. (y,x) ∈ r −−> S y (f y)) −−> S x (F f x))

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

theorem *indinv-wfrec*:
assumes *wf*: *wf* r **and**

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

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

theorem *ind-fixpoint-on-lemma*:
assumes *WF*: *wf r* **and**
INV: ∀ *f*. ∀ *x*∈*D*. (∀ *y*∈*D*. (*y*,*x*) ∈ *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*∈*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*: ∀ *y*∈*D*. (*y*, *x*) ∈ *r* −−> *F* (*wfrec r F*) *y* = *f y* & *wfrec r F y* = *f y* & *S y* (*f y*)
assume *D'*: *x*∈*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** ∀ *y*∈*D*. (*y*, *x*) ∈ *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: $\forall f\ x.\ (\forall y.\ (y,x) \in r \longrightarrow S\ y\ (wfrec\ r\ F\ y)\ \&\ f\ y = wfrec\ r\ F\ y) \longrightarrow S\ x\ (wfrec\ r\ F\ x)\ \&\ F\ f\ x = wfrec\ r\ F\ x$
shows $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x\ \&\ S\ x\ (wfrec\ r\ F\ x)$
apply (*rule ind-fixpoint-on-lemma* [*OF WF - UNIV-I*, *simplified*])
by (*rule INV*)

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

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

end

10 Example use if an inductive invariant to solve termination conditions

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

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

consts $g :: nat \Rightarrow nat$

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

Proving the termination condition using the strengthened inductive invariant.

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

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

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

end

11 Test of Locale Interpretation

theory *LocaleTest2*
imports *Main GCD*
begin

12 Interpretation of Defined Concepts

Naming convention for global objects: prefixes D and d

12.1 Lattices

Much of the lattice proofs are from HOL/Lattice.

12.1.1 Definitions

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

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 ($x \sqsubset y$) = ($x \sqsubseteq y$ & $x \sim= y$)

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

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

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* ⊓ *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 ⊓ *y* ⊑ *x*
  by (*rule is-inf-lower*) (*rule is-inf-meet*)

lemma *meet-right* [*intro?*]:
  x ⊓ *y* ⊑ *y*
  by (*rule is-inf-lower*) (*rule is-inf-meet*)

lemma *meet-le* [*intro?*]:
  [| *z* ⊑ *x*; *z* ⊑ *y* |] ==> *z* ⊑ *x* ⊓ *y*
  by (*rule is-inf-greatest*) (*rule is-inf-meet*)

lemma *is-supI* [*intro?*]: *x* ⊑ *s* ⟹ *y* ⊑ *s* ⟹
    (⋀*z*. *x* ⊑ *z* ⟹ *y* ⊑ *z* ⟹ *s* ⊑ *z*) ⟹ *is-sup x y s*
  by (*unfold is-sup-def*) *blast*

lemma *is-sup-least* [*elim?*]:
    is-sup x y s ⟹ *x* ⊑ *z* ⟹ *y* ⊑ *z* ⟹ *s* ⊑ *z*
  by (*unfold is-sup-def*) *blast*

lemma *is-sup-upper* [*elim?*]:
    is-sup x y s ⟹ (*x* ⊑ *s* ⟹ *y* ⊑ *s* ⟹ *C*) ⟹ *C*
  by (*unfold is-sup-def*) *blast*

theorem *is-sup-uniq*: *is-sup x y s* ⟹ *is-sup x y s'* ⟹ *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* ⊑ *s'*
    proof (*rule is-sup-least*)
      from *sup'* **show** *x* ⊑ *s'* **..**
      from *sup'* **show** *y* ⊑ *s'* **..**
    qed
    from *sup'* **show** *s'* ⊑ *s*
    proof (*rule is-sup-least*)
      from *sup* **show** *x* ⊑ *s* **..**
      from *sup* **show** *y* ⊑ *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*. *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*. *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 ⊓ (y ⊓ z) ⊑ y ⊓ z ..
      also have ... ⊑ y ..
      finally show ?thesis .
    qed
  qed
  show x ⊓ (y ⊓ z) ⊑ z
  proof −
    have x ⊓ (y ⊓ z) ⊑ y ⊓ z ..
    also have ... ⊑ 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*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
proof (*rule joinI*)
  show $x \sqcup y \sqsubseteq x \sqcup (y \sqcup z)$
  proof
    show $x \sqsubseteq x \sqcup (y \sqcup z)$ **..**
    show $y \sqsubseteq x \sqcup (y \sqcup z)$
    proof −
      have $y \sqsubseteq y \sqcup z$ **..**
      also have ... $\sqsubseteq x \sqcup (y \sqcup z)$ **..**
      finally show *?thesis* **.**
    qed
  qed
  show $z \sqsubseteq x \sqcup (y \sqcup z)$
  proof −
    have $z \sqsubseteq y \sqcup z$ **..**
    also have ... $\sqsubseteq x \sqcup (y \sqcup z)$ **..**
    finally show *?thesis* **.**
  qed
  fix w **assume** $x \sqcup y \sqsubseteq w$ **and** $z \sqsubseteq w$
  show $x \sqcup (y \sqcup z) \sqsubseteq w$
  proof
    show $x \sqsubseteq w$
    proof −
      have $x \sqsubseteq x \sqcup y$ **..**
      also have ... $\sqsubseteq w$ **by** *fact*
      finally show *?thesis* **.**
    qed
    show $y \sqcup z \sqsubseteq w$
    proof
      show $y \sqsubseteq w$
      proof −
        have $y \sqsubseteq x \sqcup y$ **..**
        also have ... $\sqsubseteq w$ **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

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

sublocale *dlo* < *ddlat*
proof
  fix *x y z*
  show $x \sqcap (y \sqcup z) = x \sqcap y \sqcup x \sqcap z$ (**is** *?l* = *?r*)

Jacobson I, p. 462

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

12.1.2 Total order <= 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*)
    proof qed *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*

proof qed *arith*

Interpreted theorems from the locales, involving defined terms.

thm *int.less-def*

from dpo

thm *int.meet-left*

from dlat

thm *int.meet-distr*

from ddlat

thm *int.less-total*

from dlo

12.1.3 Total order <= on *nat*

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

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

proof −
  show *dpo* (*op* <= :: [*nat*, *nat*] => *bool*)
    proof qed *auto*
  then interpret *nat*: *dpo op* <= :: [*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
  proof qed arith
```

Interpreted theorems from the locales, involving defined terms.

thm *nat.less-def*

from dpo

thm *nat.meet-left*

from dlat

thm *nat.meet-distr*

from ddlat

thm *nat.less-total*

from ldo

12.1.4 Lattice *dvd* on *nat*

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

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

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

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

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

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

```
    and dlat.join (op dvd) (x::nat) y = lcm x y
proof −
  show dlat (op dvd :: [nat, nat] => bool)
    apply unfold-locales
    apply (unfold nat-dvd.is-inf-def nat-dvd.is-sup-def)
    apply (rule-tac x = gcd x y in exI)
    apply auto [1]
    apply (rule-tac x = lcm x y in exI)
    apply (auto intro: lcm-dvd1 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
```

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

12.2.1 Locale declarations and lemmas

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

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

and *r-one* [*simp*]: $x ** one = x$

begin

definition
inv **where** $inv\ x = (THE\ y.\ x ** y = one\ \&\ y ** x = one)$
definition
unit **where** $unit\ x = (EX\ y.\ x ** y = one\ \&\ y ** x = one)$

lemma *inv-unique*:
assumes *eq*: $y ** x = one\ x ** y' = one$
shows $y = y'$
proof −
from *eq* **have** $y = y ** (x ** y')$ **by** (*simp add*: *r-one*)
also have $... = (y ** x) ** y'$ **by** (*simp add*: *assoc*)
also from *eq* **have** $... = y'$ **by** (*simp add*: *l-one*)
finally show *?thesis* **.**
qed

lemma *unit-one* [*intro*, *simp*]:
unit one
by (*unfold unit-def*) *auto*

lemma *unit-l-inv-ex*:
$unit\ x ==> \exists y.\ y ** x = one$
by (*unfold unit-def*) *auto*

lemma *unit-r-inv-ex*:
$unit\ x ==> \exists y.\ x ** y = one$
by (*unfold unit-def*) *auto*

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

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

lemma *unit-inv-unit* [*intro*, *simp*]:
$unit\ x ==> unit\ (inv\ x)$

proof −
assume x: *unit* x
show *unit* (*inv* x)
by (*auto simp add*: *unit-def*
intro: *unit-l-inv unit-r-inv* x)
qed

lemma *unit-l-cancel* [*simp*]:
unit x ==> $(x ** y = x ** z) = (y = z)$
proof
assume *eq*: $x ** y = x ** z$
and G: *unit* x
then have (*inv* $x ** x$) $** y = (inv\ x ** x) ** z$
by (*simp add*: *assoc*)
with G **show** $y = z$ **by** (*simp add*: *unit-l-inv*)
next
assume *eq*: $y = z$
and G: *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*]:
  ∃ *y*. *y* ** *x* = *one*
  using *unit-l-inv-ex* **by** *simp*

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

begin

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

end
```

12.2.2 Interpretation of Functions

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

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

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

thm Dmonoid.unit-def Dfun.unit-def

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

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

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

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

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

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

end

13 Monoids and Groups as predicates over record schemes

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

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

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

definition
  monoid :: (| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) => *bool* **where**
  monoid M = (∀ *x y z*.
    times M (*times M x y*) *z* = *times M x* (*times M y z*) ∧
    times M (*one M*) *x* = *x* ∧ *times M x* (*one M*) = *x*)

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

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

end

14 Binary arithmetic examples

theory *BinEx*
imports *Complex-Main*
begin

14.1 Regression Testing for Cancellation Simprocs

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

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

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

lemma (*i* + *j* + *12* + (*k*::*int*)) − *5* = *y*
apply *simp* **oops**

lemma *y* − *b* < (*b*::*int*)
apply *simp* **oops**

lemma *y* − (*3***b* + *c*) < (*b*::*int*) − *2***c*
apply *simp* **oops**

lemma (*2***x* − (*u***v*) + *y*) − *v***3***u* = (*w*::*int*)
apply *simp* **oops**

lemma (*2***x***u***v* + (*u***v*)**4* + *y*) − *v***u***4* = (*w*::*int*)
apply *simp* **oops**

lemma (*2***x***u***v* + (*u***v*)**4* + *y*) − *v***u* = (*w*::*int*)
apply *simp* **oops**

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

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

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

lemma $2*y + 3*z + 6*w + 2*y + 3*z + 2*u = 2*y' + 3*z' + 6*w' + 2*y' + 3*z' + u + (vv::int)$
apply *simp* **oops**

lemma $a + -(b+c) + b = (d::int)$
apply *simp* **oops**

lemma $a + -(b+c) - b = (d::int)$
apply *simp* **oops**

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

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

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

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

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

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

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

14.2 Arithmetic Method Tests

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

14.3 The Integers

Addition

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

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

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

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

Negation

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

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

Multiplication

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

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

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

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

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

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

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

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

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

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

No integer overflow!

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

Quotient and Remainder

lemma (*10*::*int*) *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*

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

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

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

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

lemma *100094 mod 144 = (14::int)*
by *simp*

Powers

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

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

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

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

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

14.4 The Natural Numbers

Successor

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

Addition

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

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

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

Subtraction

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

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

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

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

Multiplication

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

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

Quotient and Remainder

lemma (*10*::*nat*) *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* ^ *12* = (*4096*::*nat*)
by *simp*

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

lemma *12* ^ *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*

14.5 Real Arithmetic

14.5.1 Addition

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

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

14.5.2 Negation

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

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

14.5.3 Multiplication

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

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

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

14.5.4 Inequalities

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

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

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

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

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

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

14.5.5 Powers

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

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

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

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

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

by *simp*

14.5.6 Tests

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by *arith*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma $(a + b) - (c + d) = (a - c) + (b - (d::real))$
by *arith*

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

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

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

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

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

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

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

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

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

lemma $(a - b) + (b - c) = a - (c::real)$
by *arith*

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

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

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

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

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

lemma $(0::real) < x \Longrightarrow \neg(x = 0)$
by *arith*

lemma $(x + y) * (x - y) = (x * x) - (y * y)$

oops

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

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

lemma $!!a::real.\ a \leq b ==> c \leq d ==> x + y < z ==> a + c \leq b + d$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a < b ==> c < d ==> a - d \leq b + (-c)$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a \leq b ==> b + b \leq c ==> a + a \leq c$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a + b \leq i + j ==> a \leq b ==> i \leq j ==> a + a \leq j + j$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a + b < i + j ==> a < b ==> i < j ==> a + a < j + j$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a + b + c \leq i + j + k \wedge a \leq b \wedge b \leq c \wedge i \leq j \wedge j \leq k --> a + a + a \leq k + k + k$
by *arith*

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l ==> a \leq b ==> b \leq c ==> c \leq d ==> i \leq j ==> j \leq k ==> k \leq l ==> a \leq l$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l ==> a \leq b ==> b \leq c ==> c \leq d ==> i \leq j ==> j \leq k ==> k \leq l ==> a + a + a + a \leq l + l + l + l$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l ==> a \leq b ==> b \leq c ==> c \leq d ==> i \leq j ==> j \leq k ==> k \leq l ==> a + a + a + a + a \leq l + l + l + l + i$
by (*tactic fast-arith-tac* @{*context*} *1*)

lemma $!!a::real.\ a + b + c + d \leq i + j + k + l ==> a \leq b ==> b \leq c ==> c \leq d ==> i \leq j ==> j \leq k ==> k \leq l ==> a + a + a + a + a + a \leq l + l + l + l + i + l$
by (*tactic fast-arith-tac* @{*context*} *1*)

14.6 Complex Arithmetic

lemma $(1359 + 93746*ii) - (2468 + 46375*ii) = -1109 + 47371*ii$

by *simp*

lemma $-(65745 + -47371*ii) = -65745 + 47371*ii$
by *simp*

Multiplication requires distributive laws. Perhaps versions instantiated to literal constants should be added to the simpset.

lemma $(1 + ii) * (1 - ii) = 2$
by (*simp add*: *ring-distribs*)

lemma $(1 + 2*ii) * (1 + 3*ii) = -5 + 5*ii$
by (*simp add*: *ring-distribs*)

lemma $(-84 + 255*ii) + (51 * 255*ii) = -84 + 13260 * ii$
by (*simp add*: *ring-distribs*)

No inequalities or linear arithmetic: the complex numbers are unordered!

No powers (not supported yet)

end

15 Examples for hexadecimal and binary numerals

theory *Hex-Bin-Examples* **imports** *Main*
begin

Hex and bin numerals can be used like normal decimal numerals in input

lemma *0xFF = 255* **by** (*rule refl*)
lemma *0xF = 0b1111* **by** (*rule refl*)

Just like decimal numeral they are polymorphic, for arithmetic they need to be constrained

lemma *0x0A + 0x10 = (0x1A :: nat)* **by** *simp*

The number of leading zeros is irrelevant

lemma *0b00010000 = 0x10* **by** (*rule refl*)

Unary minus works as for decimal numerals

lemma $- 0x0A = -10$ **by** (*rule refl*)

Hex and bin numerals are printed as decimal: $2::'a$

term *0b10*
term *0x0A*

The numerals 0 and 1 are syntactically different from the constants 0 and 1. For the usual numeric types, their values are the same, though.

lemma *0x01 = 1* **oops**
lemma *0x00 = 0* **oops**

lemma *0x01 = (1::nat)* **by** *simp*
lemma *0b0000 = (0::int)* **by** *simp*

end

16 Antiquotations

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

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

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

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

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

parse-translation ⟪ [*Syntax.quote-antiquote-tr -Expr var Expr*] ⟫
print-translation ⟪ [*Syntax.quote-antiquote-tr' -Expr var Expr*] ⟫

term *EXPR (a + b + c)*
term *EXPR (a + b + c + VAR x + VAR y + 1)*
term *EXPR (VAR (f w) + VAR x)*

term *Expr (λenv. env x)*
term *Expr (λenv. f env)*
term *Expr (λenv. f env + env x)*
term *Expr (λenv. f env y z)*
term *Expr (λenv. f env + g y env)*
term *Expr (λenv. f env + g env y + h a env z)*

end

17 Multiple nested quotations and anti-quotations

theory *Multiquote* **imports** *Main* **begin**

Multiple nested quotations and anti-quotations – basically a generalized

version of de-Bruijn representation.

syntax
-quote :: *'b* => (*'a* => *'b*) (≪-≫ [*0*] *1000*)
-antiquote :: (*'a* => *'b*) => *'b* (´- [*1000*] *1000*)

parse-translation ⟪
let
fun antiquote-tr i (*Const* (*-antiquote*, -) $ (*t as Const* (*-antiquote*, -) $ -)) =
skip-antiquote-tr i t
| *antiquote-tr i* (*Const* (*-antiquote*, -) $ *t*) =
antiquote-tr i t $ *Bound i*
| *antiquote-tr i* (*t* $ *u*) = *antiquote-tr i t* $ *antiquote-tr i u*
| *antiquote-tr i* (*Abs* (*x*, *T*, *t*)) = *Abs* (*x*, *T*, *antiquote-tr* (*i* + *1*) *t*)
| *antiquote-tr* - *a* = *a*
and skip-antiquote-tr i ((*c as Const* (*-antiquote*, -)) $ *t*) =
c $ *skip-antiquote-tr i t*
| *skip-antiquote-tr i t* = *antiquote-tr i t*;

fun quote-tr [*t*] = *Abs* (*s*, *dummyT*, *antiquote-tr 0* (*Term.incr-boundvars 1 t*))
| *quote-tr ts* = *raise TERM* (*quote-tr*, *ts*);
in [(*-quote*, *quote-tr*)] *end*
⟫

basic examples

term ≪*a* + *b* + *c*≫
term ≪*a* + *b* + *c* + ´*x* + ´*y* + *1*≫
term ≪´(*f w*) + ´*x*≫
term ≪*f* ´*x* ´*y z*≫

advanced examples

term ≪≪´´*x* + ´*y*≫≫
term ≪≪´´*x* + ´*y*≫ *o* ´*f*≫
term ≪´(*f o* ´*g*)≫
term ≪≪´´(*f o* ´*g*)≫≫

end

18 Partial equivalence relations

theory *PER* **imports** *Main* **begin**

Higher-order quotients are defined over partial equivalence relations (PERs) instead of total ones. We provide axiomatic type classes *equiv* < *partial-equiv* and a type constructor *'a quot* with basic operations. This development is based on:

Oscar Slotosch: *Higher Order Quotients and their Implementation in Isabelle HOL.* Elsa L. Gunter and Amy Felty, editors, Theorem Proving in

Higher Order Logics: TPHOLs '97, Springer LNCS 1275, 1997.

18.1 Partial equivalence

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

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

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

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*]: $x \sim x \Longrightarrow x \in domain$
  unfolding *domain-def* **by** *blast*

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

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

18.2 Equivalence on function spaces

The $\sim$ relation is lifted to function spaces. It is important to note that this is *not* the direct product, but a structural one corresponding to the congruence property.

defs (**overloaded**)
  eqv-fun-def: $f \sim g == \forall x \in domain.\ \forall y \in domain.\ x \sim y \longrightarrow f\ x \sim g\ y$

lemma *partial-equiv-funI* [*intro?*]:
  $(!!x\ y.\ x \in domain \Longrightarrow y \in domain \Longrightarrow x \sim y \Longrightarrow f\ x \sim g\ y) \Longrightarrow f \sim g$
  unfolding *eqv-fun-def* **by** *blast*

lemma *partial-equiv-funD* [*dest?*]:
  $f \sim g \Longrightarrow x \in \mathit{domain} \Longrightarrow y \in \mathit{domain} \Longrightarrow x \sim y \Longrightarrow f\ x \sim g\ y$
 unfolding *eqv-fun-def* **by** *blast*

The class of partial equivalence relations is closed under function spaces (in *both* argument positions).

instance *fun* :: (*partial-equiv*, *partial-equiv*) *partial-equiv*
proof
 fix $f\ g\ h :: 'a{::}\mathit{partial\text{-}equiv} \Rightarrow 'b{::}\mathit{partial\text{-}equiv}$
 assume *fg*: $f \sim g$
 show $g \sim f$
 proof
  fix $x\ y :: 'a$
  assume x: $x \in \mathit{domain}$ **and** y: $y \in \mathit{domain}$
  assume $x \sim y$ **then have** $y \sim x$ **..**
  with *fg* $y\ x$ **have** $f\ y \sim g\ x$ **..**
  then show $g\ x \sim f\ y$ **..**
 qed
 assume *gh*: $g \sim h$
 show $f \sim h$
 proof
  fix $x\ y :: 'a$
  assume x: $x \in \mathit{domain}$ **and** y: $y \in \mathit{domain}$ **and** $x \sim y$
  with *fg* **have** $f\ x \sim g\ y$ **..**
  also from y **have** $y \sim y$ **..**
  with *gh* $y\ y$ **have** $g\ y \sim h\ y$ **..**
  finally show $f\ x \sim h\ y$ **.**
 qed
qed

18.3 Total equivalence

The class of total equivalence relations on top of PERs. It coincides with the standard notion of equivalence, i.e. $\sim :: 'a \Rightarrow 'a \Rightarrow \mathit{bool}$ is required to be reflexive, transitive and symmetric.

axclass *equiv* < *partial-equiv*
 eqv-refl [*intro*]: $x \sim x$

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

theorem *equiv-domain* [*intro*]: $(x{::}'a{::}\mathit{equiv}) \in \mathit{domain}$
proof
 show $x \sim x$ **..**
qed

theorem *equiv-cong* [*dest?*]: $f \sim g \Longrightarrow x \sim y \Longrightarrow f\ x \sim g\ (y{::}'a{::}\mathit{equiv})$
proof −
 assume $f \sim g$

moreover have $x \in domain$ **..**
moreover have $y \in domain$ **..**
moreover assume $x \sim y$
ultimately show *?thesis* **..**
qed

18.4 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*]: $R \in quot \Longrightarrow (!!a.\ R = \{x.\ a \sim x\} \Longrightarrow C) \Longrightarrow C$
unfolding *quot-def* **by** *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition
eqv-class :: $('a::partial\text{-}equiv) \Rightarrow 'a\ quot$ $(\lfloor\text{-}\rfloor)$ **where**
$\lfloor a \rfloor = Abs\text{-}quot\ \{x.\ a \sim x\}$

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

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

18.5 Equality on quotients

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

theorem *eqv-class-eqI* [*intro*]: $a \sim b \Longrightarrow \lfloor a \rfloor = \lfloor b \rfloor$
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 ∼ a ..
      also assume a ∼ x
      finally show b ∼ x .
    next
      note ab
      also assume b ∼ x
      finally show a ∼ x .
    qed
  qed
  then show ?thesis by (simp only: eqv-class-def)
qed

theorem eqv-class-eqD' [dest?]: ⌊a⌋ = ⌊b⌋ ==> a ∈ domain ==> a ∼ b
proof (unfold eqv-class-def)
  assume Abs-quot {x. a ∼ x} = Abs-quot {x. b ∼ x}
  then have {x. a ∼ x} = {x. b ∼ x} by (simp only: Abs-quot-inject quotI)
  moreover assume a ∈ domain then have a ∼ a ..
  ultimately have a ∈ {x. b ∼ x} by blast
  then have b ∼ a by blast
  then show a ∼ b ..
qed

theorem eqv-class-eqD [dest?]: ⌊a⌋ = ⌊b⌋ ==> a ∼ (b::'a::equiv)
proof (rule eqv-class-eqD')
  show a ∈ domain ..
qed

lemma eqv-class-eq' [simp]: a ∈ domain ==> (⌊a⌋ = ⌊b⌋) = (a ∼ b)
  using eqv-class-eqI eqv-class-eqD' by (blast del: eqv-refl)

lemma eqv-class-eq [simp]: (⌊a⌋ = ⌊b⌋) = (a ∼ (b::'a::equiv))
  using eqv-class-eqI eqv-class-eqD by blast
```

18.6 Picking representing elements

```
definition
  pick :: 'a::partial-equiv quot => 'a where
  pick A = (SOME a. A = ⌊a⌋)

theorem pick-eqv' [intro?, simp]: a ∈ domain ==> pick ⌊a⌋ ∼ a
proof (unfold pick-def)
  assume a: a ∈ domain
  show (SOME x. ⌊a⌋ = ⌊x⌋) ∼ a
  proof (rule someI2)
    show ⌊a⌋ = ⌊a⌋ ..
    fix x assume ⌊a⌋ = ⌊x⌋
    from this and a have a ∼ x ..
    then show x ∼ a ..
```

qed
qed

theorem *pick-eqv* [*intro*, *simp*]: *pick* ⌊*a*⌋ ∼ (*a*::′*a*::*equiv*)
proof (*rule pick-eqv*′)
show *a* ∈ *domain* **..**
qed

theorem *pick-inverse*: ⌊*pick A*⌋ = (*A*::′*a*::*equiv quot*)
proof (*cases A*)
fix *a* **assume** *a*: *A* = ⌊*a*⌋
then have *pick A* ∼ *a* **by** *simp*
then have ⌊*pick A*⌋ = ⌊*a*⌋ **by** *simp*
with *a* **show** *?thesis* **by** *simp*
qed

end

19 Summing natural numbers

theory *NatSum* **imports** *Main Parity* **begin**

Summing natural numbers, squares, cubes, etc.

Thanks to Sloane's On-Line Encyclopedia of Integer Sequences, http://www.research.att.com/~njas/sequences/.

lemmas [*simp*] =
ring-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*: (∑ *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 ∗ (∑ *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*:
(∑ *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-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*:
 $30 * of\text{-}nat\ (\sum i{=}0..{<}m.\ i * i * i * i) =$

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

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

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

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

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

end

20 Three Divides Theorem

theory *ThreeDivides*
imports *Main LaTeXsugar*
begin

20.1 Abstract

The following document presents a proof of the Three Divides N theorem formalised in the Isabelle/Isar theorem proving system.

Theorem: 3 divides n if and only if 3 divides the sum of all digits in n.

Informal Proof: Take $n = \sum n_j * 10^j$ where n_j is the j'th least significant digit of the decimal denotation of the number n and the sum ranges over all digits. Then

$$(n - \sum n_j) = \sum n_j * (10^j - 1)$$

We know $\forall j\ 3|(10^j - 1)$ and hence $3|LHS$, therefore

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

□

20.2 Formal proof

20.2.1 Miscellaneous summation lemmas

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

lemma *div-sum*:
fixes *a*::*nat* **and** *n*::*nat*
shows $\forall x.\ a\ dvd\ A\ x \Longrightarrow a\ dvd\ (\sum x{<}n.\ A\ x * D\ x)$
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* ((∑ *x*<*n*. *A x* ∗ *D x*) + (*A n* ∗ *D n*)) **by** (*simp add*: *dvd-add*)
thus *?case* **by** *simp*
qed

20.2.2 Generalised Three Divides

This section solves a generalised form of the three divides problem. Here we show that for any sequence of numbers the theorem holds. In the next section we specialise this theorem to apply directly to the decimal expansion of the natural numbers.

Here we show that the first statement in the informal proof is true for all natural numbers. Note we are using $D\ i$ to denote the i'th element in a sequence of numbers.

lemma *digit-diff-split*:
fixes *n*::*nat* **and** *nd*::*nat* **and** *x*::*nat*
shows $n = (\sum x{\in}\{..{<}nd\}.\ (D\ x){*}((10{::}nat)\hat{}x)) \Longrightarrow$
$(n - (\sum x{<}nd.\ (D\ x))) = (\sum x{<}nd.\ (D\ x){*}(10\hat{}x - 1))$
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*^*x* − *1*)
proof (*induct x*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)
let *?thr* = (*3*::*nat*)
have *?thr dvd 9* **by** *simp*
moreover
have *?thr dvd* (*10*∗(*10*^*n* − *1*)) **by** (*rule dvd-mult*) (*rule Suc*)
hence *?thr dvd* (*10*^(*n*+*1*) − *10*) **by** (*simp add*: *nat-distrib*)
ultimately
have*?thr dvd* ((*10*^(*n*+*1*) − *10*) + *9*)
by (*simp only*: *add-ac*) (*rule dvd-add*)
thus *?case* **by** *simp*
qed

Expanding on the previous lemma and lemma *div-sum*.

lemma *three-divs-1*:
fixes $D :: nat \Rightarrow nat$
shows *3 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*⇒*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

20.2.3 Three Divides Natural

This section shows that for all natural numbers we can generate a sequence of digits less than ten that represent the decimal expansion of the number. We then use the lemma *three-div-general* to prove our final theorem.

Definitions of length and digit sum.

This section introduces some functions to calculate the required properties of natural numbers. We then proceed to prove some properties of these functions.

The function *nlen* returns the number of digits in a natural number n.

```
consts nlen :: nat ⇒ 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 = (∑x < nlen n. n div 10^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 = (∑x<nlen m. (m div (10::nat)^x mod 10) * 10^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 = (∑x<nd. m div 10 div 10^x mod 10 * 10^x)
    by blast
  obtain c where mexp: m = 10*(m div 10) + c ∧ c < 10
    and cdef: c = m mod 10 by simp
  show m = (∑x<nlen m. m div 10^x mod 10 * 10^x)
  proof −
    from ‹Suc nd = nlen m›
    have nd = nlen (m div 10) by (rule nlen-suc)
    with IH have
      m div 10 = (∑x<nd. m div 10 div 10^x mod 10 * 10^x) by simp
    with mexp have
      m = 10*(∑x<nd. m div 10 div 10^x mod 10 * 10^x) + c by simp
    also have
```

```
    ... = (∑x<nd. m div 10 div 10^x mod 10 * 10^(x+1)) + c
    by (subst setsum-right-distrib) (simp add: mult-ac)
  also have
    ... = (∑x<nd. m div 10^(Suc x) mod 10 * 10^(Suc x)) + c
    by (simp add: div-mult2-eq[symmetric])
  also have
    ... = (∑x∈{Suc 0..<Suc nd}. m div 10^x  mod 10 * 10^x) + c
    by (simp only: setsum-shift-bounds-Suc-ivl)
      (simp add: atLeast0LessThan)
  also have
    ... = (∑x<Suc nd. m div 10^x mod 10 * 10^x)
    by (simp add: atLeast0LessThan[symmetric] setsum-head-upt-Suc cdef)
  also note ‹Suc nd = nlen m›
  finally
  show m = (∑x<nlen m. m div 10^x mod 10 * 10^x) .
 qed
qed
```

Final theorem.

We now combine the general theorem *three-div-general* and existence result of *exp-exists* to prove our final theorem.

```
theorem three-divides-nat:
 shows (3 dvd n) = (3 dvd sumdig n)
proof (unfold sumdig-def)
 have n = (∑x<nlen n. (n div (10::nat)^x mod 10) * 10^x)
   by (rule exp-exists)
 moreover
 have 3 dvd (∑x<nlen n. (n div (10::nat)^x mod 10) * 10^x) =
       (3 dvd (∑x<nlen n. n div 10^x mod 10))
   by (rule three-div-general)
 ultimately
 show 3 dvd n = (3 dvd (∑x<nlen n. n div 10^x mod 10)) by simp
qed

end
```

21 Higher-Order Logic: Intuitionistic predicate calculus problems

```
theory Intuitionistic imports Main begin



lemma (~~(P&Q)) = ((~~P) & (~~Q))
 by iprover
```

lemma ~~ ((~P --> Q) --> (~P --> ~Q) --> P)
by *iprover*

lemma (~~(P-->Q)) = (~~P --> ~~Q)
by *iprover*

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

lemma ~~((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 --> ~~P
by *iprover*

lemma ~~(~~P --> P)
by *iprover*

lemma ~~P & ~~(P --> Q) --> ~~Q
by *iprover*

lemma ((P=Q) --> P&Q&R) &
((Q=R) --> P&Q&R) &
((R=P) --> P&Q&R) --> P&Q&R
by *iprover*

lemma ((P=Q) --> P&Q&R&S&T) &

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

lemma $(ALL\ x.\ EX\ y.\ ALL\ z.\ p(x)\ \&\ q(y)\ \&\ r(z)) =$
$(ALL\ z.\ EX\ y.\ ALL\ x.\ p(x)\ \&\ q(y)\ \&\ r(z))$
by (*iprover del*: *allE elim 2*: *allE'*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma $(EX\ x.\ P(x))\ \&$
$(ALL\ x.\ L(x)\ \text{-->}\ \sim\ (M(x)\ \&\ R(x)))\ \&$
$(ALL\ x.\ P(x)\ \text{-->}\ (M(x)\ \&\ L(x)))\ \&$
$((ALL\ x.\ P(x)\text{-->}Q(x))\ |\ (EX\ x.\ P(x)\&R(x)))$

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

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

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

lemma $\sim(EX\ x.\ P(x)\ \&\ (Q(x)\ |\ R(x)))\ \&$
$(EX\ x.\ L(x)\ \&\ P(x))\ \&$
$(ALL\ x.\ \sim\ R(x)\ -->\ M(x))$
$-->$ $(EX\ x.\ L(x)\ \&\ M(x))$
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 $(ALL\ x.\ \sim\sim(P(a)\ \&\ (P(x)-->P(b))-->P(c)))\ =$
$(ALL\ x.\ \sim\sim((\sim P(a)\ |\ P(x)\ |\ P(c))\ \&\ (\sim P(a)\ |\ \sim P(b)\ |\ P(c))))$
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

22 CTL formulae

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

We formalize basic concepts of Computational Tree Logic (CTL) [3, 2] within the simply-typed set theory of HOL.

By using the common technique of "shallow embedding", a CTL formula is identified with the corresponding set of states where it holds. Consequently, CTL operations such as negation, conjunction, disjunction simply become complement, intersection, union of sets. We only require a separate 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 ⇒ 'a ctl ⇒ 'a ctl* (**infixr** → *75*) **where**
p → q = − p ∪ q

lemma [*intro*!]: *p ∩ p → q ⊆ q* **unfolding** *imp-def* **by** *auto*
lemma [*intro*!]: *p ⊆ (q → 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 × '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 [3].

definition
EX (EX - [*80*] *90*) **where** EX *p* = {*s*. ∃ *s'*. (*s*, *s'*) ∈ $\mathcal{M}$ ∧ *s'* ∈ *p*}
definition

EF (EF - [*80*] *90*) **where** EF p = *lfp* ($\lambda s.\ p \cup$ EX s)
definition
EG (EG - [*80*] *90*) **where** EG p = *gfp* ($\lambda s.\ p \cap$ EX s)

AX, AF and AG are now defined dually in terms of EX, EF and EG.

definition
AX (AX - [*80*] *90*) **where** AX $p = -$ EX $-\ p$
definition
AF (AF - [*80*] *90*) **where** AF $p = -$ EG $-\ p$
definition
AG (AG - [*80*] *90*) **where** AG $p = -$ EF $-\ p$

lemmas [*simp*] = *EX-def EG-def AX-def EF-def AF-def AG-def*

22.1 Basic fixed point properties

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

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

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

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

in order to give dual fixed point representations of AF p and AG p:

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

lemma *EF-fp*: EF $p = p \cup$ EX EF p
proof −
  have *mono* ($\lambda s.\ p \cup$ 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 $p = p \cup$ AX AF p
proof −
  have *mono* ($\lambda s.\ p \cup$ 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 $p = p \cap$ EX EG p
proof −
  have *mono* ($\lambda s.\ p \cap$ 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 $\lambda s.\ p \cap$ AX s.

lemma *AG-fp*: AG $p = p \cap$ AX AG p
proof −
  have *mono* ($\lambda s.\ p \cap$ 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** $p \cap$ 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*: $q \subseteq p \cap \text{AX } q \Longrightarrow q \subseteq \text{AG } p$
  by (*simp only*: *AG-gfp*) (*rule gfp-upperbound*)

22.2 The tree induction principle

With the most basic facts available, we are now able to establish a few more interesting results, leading to the *tree induction* principle for AG (see below). We will use some elementary monotonicity and distributivity rules.

lemma *AX-int*: $\text{AX } (p \cap q) = \text{AX } p \cap \text{AX } q$ **by** *auto*
lemma *AX-mono*: $p \subseteq q \Longrightarrow \text{AX } p \subseteq \text{AX } q$ **by** *auto*
lemma *AG-mono*: $p \subseteq q \Longrightarrow \text{AG } p \subseteq \text{AG } q$
  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*: $\text{AG } p \subseteq \text{AX } p$
proof −
  have $\text{AG } p \subseteq \text{AX AG } p$ **by** (*rule AG-fp-2*)
  also have $\text{AG } p \subseteq p$ **by** (*rule AG-fp-1*) **moreover note** *AX-mono*
  finally show *?thesis* **.**
qed

Furthermore we show idempotency of the AG operator. The proof is a good example of how accumulated facts may get used to feed a single rule step.

lemma *AG-AG*: $\text{AG AG } p = \text{AG } p$
proof
  show $\text{AG AG } p \subseteq \text{AG } p$ **by** (*rule AG-fp-1*)
next
  show $\text{AG } p \subseteq \text{AG AG } p$
  proof (*rule AG-I*)
    have $\text{AG } p \subseteq \text{AG } p$ **..**
    moreover have $\text{AG } p \subseteq \text{AX AG } p$ **by** (*rule AG-fp-2*)
    ultimately show $\text{AG } p \subseteq \text{AG } p \cap \text{AX AG } p$ **..**
  qed
qed

We now give an alternative characterization of the AG operator, which describes the AG operator in an "operational" way by tree induction: In a state holds AG p iff in that state holds p, and in all reachable states s follows from the fact that p holds in s, that p also holds in all successor states of s. We use the co-induction principle *AG-I* to establish this in a purely algebraic manner.

theorem *AG-induct*: $p \cap \text{AG } (p \rightarrow \text{AX } p) = \text{AG } p$
proof
  show $p \cap \text{AG } (p \rightarrow \text{AX } p) \subseteq \text{AG } p$ (**is** *?lhs* $\subseteq$ -)

```
  proof (rule AG-I)
    show ?lhs ⊆ p ∩ AX ?lhs
    proof
      show ?lhs ⊆ p ..
      show ?lhs ⊆ AX ?lhs
      proof −
        {
          have AG (p → AX p) ⊆ p → AX p by (rule AG-fp-1)
          moreover have p ∩ p → AX p ⊆ AX p ..
          ultimately have ?lhs ⊆ AX p by auto
        }
        moreover
        {
          have p ∩ AG (p → AX p) ⊆ AG (p → AX p) ..
          also have ... ⊆ AX ... by (rule AG-fp-2)
          finally have ?lhs ⊆ AX AG (p → AX p) .
        }
        ultimately have ?lhs ⊆ AX p ∩ AX AG (p → AX p)
          by (rule Int-greatest)
        also have ... = AX ?lhs by (simp only: AX-int)
        finally show ?thesis .
      qed
    qed
  qed
next
  show AG p ⊆ p ∩ AG (p → AX p)
  proof
    show AG p ⊆ p by (rule AG-fp-1)
    show AG p ⊆ AG (p → AX p)
    proof −
      have AG p = AG AG p by (simp only: AG-AG)
      also have AG p ⊆ AX p by (rule AG-AX) moreover note AG-mono
      also have AX p ⊆ (p → AX p) .. moreover note AG-mono
      finally show ?thesis .
    qed
  qed
qed
```

22.3 An application of tree induction

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

```
theorem AG-AX-commute: AG AX p = AX AG p
proof −
  have AG AX p = AX p ∩ AX AG AX p by (rule AG-fp)
  also have ... = AX (p ∩ AG AX p) by (simp only: AX-int)
  also have p ∩ AG AX p = AG p  (is ?lhs = -)
  proof
    have AX p ⊆ p → AX p ..
```

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

end
```

23 Arithmetic

theory *Arith-Examples* **imports** *Main* **begin**

The *arith* method is used frequently throughout the Isabelle distribution. This file merely contains some additional tests and special corner cases. Some rather technical remarks:

`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. `linear_arith_tac` performs meta-to-object-logic conversion, full splitting of operators, and NNF normalization of the goal. The *arith* method combines them both, and tries other methods (e.g. *presburger*) as well. This is the one that you should use in your proofs!

An *arith*-based simproc is available as well (see `Lin_Arith.lin_arith_simproc`), which—for performance reasons—however does even less splitting than `fast_arith_tac` at the moment (namely inequalities only). (On the other hand, it does take apart conjunctions, which `fast_arith_tac` currently does not do.)

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

```
lemma (i::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* ⟪ *linear-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* ⟫)

23.2 Meta-Logic

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

lemma $((x::nat) == z \Longrightarrow x \sim= y) \Longrightarrow x \sim= y \mid z \sim= y$
by (*tactic* ⟪ *linear-arith-tac* @{*context*} *1* ⟫)

23.3 Various Other Examples

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

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

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

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

lemma $(P::bool) = Q \Longrightarrow Q = P$
by (*tactic* ⟪ *linear-arith-tac* @{*context*} *1* ⟫)

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

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

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

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

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

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

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

lemma $[\!|\ (x::nat) < y;\ P\ (x - y)\ |\!] \Longrightarrow P\ 0$
by (*tactic* ⟪ *linear-arith-tac* @{*context*} *1* ⟫)

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

lemma $[\!|\ (a::nat) < b;\ c < d\ |\!] \Longrightarrow (a - b) = (c - d)$
by (*tactic* ⟪ *fast-arith-tac* @{*context*} *1* ⟫)

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

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

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

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

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

Splitting of inequalities of different type.

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

Again, but different order.

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

end

24 Binary trees

theory *BT* **imports** *Main* **begin**

datatype *'a bt* =
  Lf
| *Br 'a 'a bt 'a bt*

consts
  n-nodes :: *'a bt => nat*
  n-leaves :: *'a bt => nat*
  depth :: *'a bt => nat*
  reflect :: *'a bt => 'a bt*
  bt-map :: *('a => 'b) => ('a bt => 'b bt)*
  preorder :: *'a bt => 'a list*
  inorder :: *'a bt => 'a list*
  postorder :: *'a bt => 'a list*
  append :: *'a bt => 'a bt => 'a bt*

primrec
  n-nodes Lf = 0
  n-nodes (Br a t1 t2) = Suc (n-nodes t1 + n-nodes t2)

primrec
  n-leaves Lf = Suc 0
  n-leaves (Br a t1 t2) = n-leaves t1 + n-leaves t2

primrec
  depth Lf = 0
  depth (Br a t1 t2) = Suc (max (depth t1) (depth t2))

primrec
  reflect Lf = Lf
  reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)

primrec
  bt-map f Lf = Lf
  bt-map f (Br a t1 t2) = Br (f a) (bt-map f t1) (bt-map f t2)

primrec
  preorder Lf = []
  preorder (Br a t1 t2) = [a] @ (preorder t1) @ (preorder t2)

primrec
  inorder Lf = []
  inorder (Br a t1 t2) = (inorder t1) @ [a] @ (inorder t2)

primrec
postorder Lf = []
postorder (*Br a t1 t2*) = (*postorder t1*) @ (*postorder t2*) @ [*a*]

primrec
append Lf t = *t*
append (*Br a t1 t2*) *t* = *Br a* (*append t1 t*) (*append t2 t*)

BT simplification

lemma *n-leaves-reflect*: *n-leaves* (*reflect t*) = *n-leaves t*
apply (*induct t*)
apply *auto*
done

lemma *n-nodes-reflect*: *n-nodes* (*reflect t*) = *n-nodes t*
apply (*induct t*)
apply *auto*
done

lemma *depth-reflect*: *depth* (*reflect t*) = *depth t*
apply (*induct t*)
apply *auto*
done

The famous relationship between the numbers of leaves and nodes.

lemma *n-leaves-nodes*: *n-leaves t* = *Suc* (*n-nodes t*)
apply (*induct t*)
apply *auto*
done

lemma *reflect-reflect-ident*: *reflect* (*reflect t*) = *t*
apply (*induct t*)
apply *auto*
done

lemma *bt-map-reflect*: *bt-map f* (*reflect t*) = *reflect* (*bt-map f t*)
apply (*induct t*)
apply *simp-all*
done

lemma *preorder-bt-map*: *preorder* (*bt-map f t*) = *map f* (*preorder t*)
apply (*induct t*)
apply *simp-all*
done

lemma *inorder-bt-map*: *inorder* (*bt-map f t*) = *map f* (*inorder t*)
apply (*induct t*)
apply *simp-all*

done

lemma *postorder-bt-map*: *postorder* (*bt-map f t*) = *map f* (*postorder t*)
apply (*induct t*)
apply *simp-all*
done

lemma *depth-bt-map* [*simp*]: *depth* (*bt-map f t*) = *depth t*
apply (*induct t*)
apply *simp-all*
done

lemma *n-leaves-bt-map* [*simp*]: *n-leaves* (*bt-map f t*) = *n-leaves t*
apply (*induct t*)
apply (*simp-all add*: *left-distrib*)
done

lemma *preorder-reflect*: *preorder* (*reflect t*) = *rev* (*postorder t*)
apply (*induct t*)
apply *simp-all*
done

lemma *inorder-reflect*: *inorder* (*reflect t*) = *rev* (*inorder t*)
apply (*induct t*)
apply *simp-all*
done

lemma *postorder-reflect*: *postorder* (*reflect t*) = *rev* (*preorder t*)
apply (*induct t*)
apply *simp-all*
done

Analogues of the standard properties of the append function for lists.

lemma *append-assoc* [*simp*]:
append (*append t1 t2*) *t3* = *append t1* (*append t2 t3*)
apply (*induct t1*)
apply *simp-all*
done

lemma *append-Lf2* [*simp*]: *append t Lf* = *t*
apply (*induct t*)
apply *simp-all*
done

lemma *depth-append* [*simp*]: *depth* (*append t1 t2*) = *depth t1* + *depth t2*
apply (*induct t1*)
apply (*simp-all add*: *max-add-distrib-left*)
done

lemma *n-leaves-append* [*simp*]:
  n-leaves (*append t1 t2*) = *n-leaves t1* ∗ *n-leaves t2*
 apply (*induct t1*)
 apply (*simp-all add*: *left-distrib*)
 done

lemma *bt-map-append*:
  bt-map f (*append t1 t2*) = *append* (*bt-map f t1*) (*bt-map f t2*)
 apply (*induct t1*)
 apply *simp-all*
 done

end

25 Sorting: Basic Theory

theory *Sorting*
imports *Main Multiset*
begin

consts
 sorted1:: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool*
 sorted :: ('*a* ⇒ '*a* ⇒ *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* ⇒ '*a* ⇒ *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

26 Merge Sort

theory *MergeSort*
imports *Sorting*
begin

context *linorder*
begin

fun *merge* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  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* ≤) (*merge xs ys*) = (*sorted* (*op* ≤) *xs* & *sorted* (*op* ≤) *ys*)
apply(*induct xs ys rule*: *merge.induct*)
apply(*simp-all add*: *ball-Un not-le less-le*)
apply(*blast intro*: *order-trans*)
done

fun *msort* :: *'a list* ⇒ *'a list*

where
  msort [] = []
| *msort* [*x*] = [*x*]
| *msort xs* = *merge* (*msort* (*take* (*size xs div 2*) *xs*))
                       (*msort* (*drop* (*size xs div 2*) *xs*))

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

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

end

end

27 A lemma for Lagrange's theorem

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

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

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

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

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

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

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

A challenge by John Harrison. Takes about 12s on a 1.6GHz machine.

lemma fixes *p1* :: *'a*::*comm-ring* **shows**
(sq p1 + sq q1 + sq r1 + sq s1 + sq t1 + sq u1 + sq v1 + sq w1) ∗
(sq p2 + sq q2 + sq r2 + sq s2 + sq t2 + sq u2 + sq v2 + sq w2)
= sq (p1∗p2 − q1∗q2 − r1∗r2 − s1∗s2 − t1∗t2 − u1∗u2 − v1∗v2 − w1∗w2)
+
sq (p1∗q2 + q1∗p2 + r1∗s2 − s1∗r2 + t1∗u2 − u1∗t2 − v1∗w2 + w1∗v2)
+
sq (p1∗r2 − q1∗s2 + r1∗p2 + s1∗q2 + t1∗v2 + u1∗w2 − v1∗t2 − w1∗u2)
+
sq (p1∗s2 + q1∗r2 − r1∗q2 + s1∗p2 + t1∗w2 − u1∗v2 + v1∗u2 − w1∗t2)
+
sq (p1∗t2 − q1∗u2 − r1∗v2 − s1∗w2 + t1∗p2 + u1∗q2 + v1∗r2 + w1∗s2)
+
sq (p1∗u2 + q1∗t2 − r1∗w2 + s1∗v2 − t1∗q2 + u1∗p2 − v1∗s2 + w1∗r2)
+
sq (p1∗v2 + q1∗w2 + r1∗t2 − s1∗u2 − t1∗r2 + u1∗s2 + v1∗p2 − w1∗q2)
+
sq (p1∗w2 − q1∗v2 + r1∗u2 + s1∗t2 − t1∗s2 − u1∗r2 + v1∗q2 + w1∗p2)
by (*simp only*: *sq-def ring-simps*)

end

28 Groebner Basis Examples

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

28.1 Basic examples

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

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

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

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

lemma *((x::int) + y) ^3 − 1 = (x − z) ^2 − 10 ⟹ x = z + 3 ⟹ x = − y*
by *algebra*

lemma $(4::nat) + 4 = 3 + 5$
by *algebra*

lemma $(4::int) + 0 = 4$
apply *algebra?*
by *simp*

lemma
assumes $a * x\hat{}2 + b * x + c = (0::int)$ **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 $(algebra\ add: sq\text{-}def)$

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 $point = int \times int$

definition *collinear* ::$point \Rightarrow point \Rightarrow point \Rightarrow bool$ **where**
$collinear \equiv \lambda(Ax,Ay)\ (Bx,By)\ (Cx,Cy).$
$((Ax - Bx) * (By - Cy) = (Ay - By) * (Bx - Cx))$

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 $EX\ (d::int).\ a*y - a*x = n*d \Longrightarrow EX\ u\ v.\ a*u + n*v = 1 \Longrightarrow EX\ e.\ y - x = n*e$
by *algebra*

end

29 Milner-Tofte: Co-induction in Relational Semantics

theory *MT*
imports *Main*
begin

typedecl *Const*

typedecl *ExVar*
typedecl *Ex*

typedecl *TyConst*

typedecl *Ty*

typedecl *Clos*
typedecl *Val*

typedecl *ValEnv*
typedecl *TyEnv*

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

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

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

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

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

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

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

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

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

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

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

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

axioms

e-const-inj: *e-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** · *60*)

types
  'a subst = (*'a* × *'a trm*) *list*

Applying a substitution to a variable:

fun *assoc* :: *'a* ⇒ *'b* ⇒ (*'a* × *'b*) *list* ⇒ *'b*
where
  assoc x d [] = *d*
| *assoc x d* ((*p*,*q*)#*t*) = (*if x* = *p then q else assoc x d t*)

Applying a substitution to a term:

fun *apply-subst* :: *'a trm* ⇒ *'a subst* ⇒ *'a trm* (**infixl** ◁ *60*)
where
  (*Var v*) ◁ *s* = *assoc v* (*Var v*) *s*
| (*Const c*) ◁ *s* = (*Const c*)
| (*M* · *N*) ◁ *s* = (*M* ◁ *s*) · (*N* ◁ *s*)

Composition of substitutions:

fun
  compose :: *'a subst* ⇒ *'a subst* ⇒ *'a subst* (**infixl** · *80*)
where
  [] · *bl* = *bl*
| ((*a*,*b*) # *al*) · *bl* = (*a*, *b* ◁ *bl*) # (*al* · *bl*)

Equivalence of substitutions:

definition *eqv* (**infix** =$_s$ *50*)
where
  s1 =$_s$ *s2* ≡ ∀ *t*. *t* ◁ *s1* = *t* ◁ *s2*

30.2 Basic lemmas

lemma *apply-empty*[*simp*]: *t* ◁ [] = *t*
by (*induct t*) *auto*

lemma *compose-empty*[*simp*]: *σ* · [] = *σ*
by (*induct σ*) *auto*

lemma *apply-compose*[*simp*]: *t* ◁ (*s1* · *s2*) = *t* ◁ *s1* ◁ *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*: $[\![\sigma =_s \sigma';\ \vartheta =_s \vartheta']\!] \Longrightarrow (\sigma \cdot \vartheta) =_s (\sigma' \cdot \vartheta')$
by (*auto simp*:*eqv-def*)

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

30.3 Specification: Most general unifiers

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

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

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

lemma *MGU-sym*[*sym*]:
$\mathit{MGU}\ \sigma\ s\ t \Longrightarrow \mathit{MGU}\ \sigma\ t\ s$

by (*auto simp*:*MGU-def Unifier-def*)

30.4 The unification algorithm

Occurs check: Proper subterm relation

fun *occ* :: *'a trm* ⇒ *'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 ◁ ϑ = s ◁ [(v,t)] ⋅ ϑ using th
      by (induct s) auto
  qed
qed

declare MGU-Var[symmetric, intro]

lemma MGU-Const[simp]: MGU [] (Const c) (Const d) = (c = d)
  unfolding MGU-def Unifier-def
  by auto
```

If unification terminates, then it computes most general unifiers:

```
lemma unify-partial-correctness:
  assumes unify-dom (M, N)
  assumes unify M N = Some σ
  shows MGU σ M N
using assms
proof (induct M N arbitrary: σ)
  case (7 M N M' N' σ) — The interesting case

  then obtain ϑ1 ϑ2
    where unify M M' = Some ϑ1
    and unify (N ◁ ϑ1) (N' ◁ ϑ1) = Some ϑ2
    and σ: σ = ϑ1 ⋅ ϑ2
    and MGU-inner: MGU ϑ1 M M'
    and MGU-outer: MGU ϑ2 (N ◁ ϑ1) (N' ◁ ϑ1)
    by (auto split:option.split-asm)

  show ?case
  proof
    from MGU-inner and MGU-outer
    have M ◁ ϑ1 = M' ◁ ϑ1
      and N ◁ ϑ1 ◁ ϑ2 = N' ◁ ϑ1 ◁ ϑ2
      unfolding MGU-def Unifier-def
      by auto
    thus M ⋅ N ◁ σ = M' ⋅ N' ◁ σ unfolding σ
      by simp
  next
    fix σ' assume M ⋅ N ◁ σ' = M' ⋅ N' ◁ σ'
    hence M ◁ σ' = M' ◁ σ'
      and Ns: N ◁ σ' = N' ◁ σ' by auto

    with MGU-inner obtain δ
      where eqv: σ' =s ϑ1 ⋅ δ
      unfolding MGU-def Unifier-def
      by auto

    from Ns have N ◁ ϑ1 ◁ δ = N' ◁ ϑ1 ◁ δ
      by (simp add:eqv-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-intro*, *auto simp*:*eqv-dest*[*OF eqv*]
eqv-dest[*OF eqv2*])
thus $\exists \gamma.\ \sigma' =_s \sigma \cdot \gamma$ **..**
qed
qed (*auto split*:*split-if-asm*) — Solve the remaining cases automatically

30.6 Properties used in termination proof

The variables of a term:

fun *vars-of*:: *'a trm* ⇒ *'a set*
where
vars-of (*Var v*) = { *v* }
| *vars-of* (*Const c*) = {}
| *vars-of* (*M* · *N*) = *vars-of M* ∪ *vars-of N*

lemma *vars-of-finite*[*intro*]: *finite* (*vars-of t*)
by (*induct t*) *simp-all*

Elimination of variables by a substitution:

definition
elim σ *v* ≡ ∀ *t*. *v* ∉ *vars-of* (*t* ◁ σ)

lemma *elim-intro*[*intro*]: (⋀*t*. *v* ∉ *vars-of* (*t* ◁ σ)) ⟹ *elim* σ *v*
by (*auto simp*:*elim-def*)

lemma *elim-dest*[*dest*]: *elim* σ *v* ⟹ *v* ∉ *vars-of* (*t* ◁ σ)
by (*auto simp*:*elim-def*)

lemma *elim-eqv*: σ =$_s$ ϑ ⟹ *elim* σ *x* = *elim* ϑ *x*
by (*auto simp*:*elim-def eqv-def*)

Replacing a variable by itself yields an identity subtitution:

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

lemma *var-same*: (*t* = *Var v*) = ([(*v*, *t*)] =$_s$ [])
proof
assume *t-v*: *t* = *Var v*

```
  thus [(v, t)] =s []
    by auto
next
  assume id: [(v, t)] =s []
  show t = Var v
  proof −
    have t = Var v ◁ [(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)^4 − b^4 = (a − b) * (a + b)*(a^2 + b^2)

by *comm-ring*

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

end

32 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.

32.1 Ackermann's Function

fun *ack* :: *nat* => *nat* => *nat* **where**
ack 0 n = *Suc n* |
ack (*Suc m*) *0* = *ack m 1* |
ack (*Suc m*) (*Suc n*) = *ack m* (*ack* (*Suc m*) *n*)

PROPERTY A 4

lemma *less-ack2* [*iff*]: *j* < *ack i j*
by (*induct i j rule*: *ack.induct*) *simp-all*

PROPERTY A 5-, the single-step lemma

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

PROPERTY A 5, monotonicity for <

lemma *ack-less-mono2*: *j* < *k* ==> *ack i j* < *ack i k*
using *lift-Suc-mono-less*[**where** *f* = *ack i*]
by (*metis ack-less-ack-Suc2*)

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*) ≤ *ack* (*Suc i*) *j*
proof (*induct j*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc j*) **show** *?case*
by (*auto intro*!: *ack-le-mono2*)
(*metis Suc Suc-leI Suc-lessI less-ack2 linorder-not-less*)
qed

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: *ack i j* < *ack* (*Suc i*) *j*
by (*blast intro*: *ack-less-mono2 less-le-trans*)

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

lemma *less-ack1* [*iff*]: *i* < *ack i j*
apply (*induct i*)
apply *simp-all*
apply (*blast intro*: *Suc-leI le-less-trans*)
done

PROPERTY A 8

lemma *ack-1* [*simp*]: *ack* (*Suc 0*) *j* = *j* + *2*
by (*induct j*) *simp-all*

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

lemma *ack-2* [*simp*]: *ack* (*Suc* (*Suc 0*)) *j* = *2* ∗ *j* + *3*
by (*induct j*) *simp-all*

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

lemma *ack-less-mono1-aux*: *ack i k* < *ack* (*Suc* (*i* + *i'*)) *k*
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 ≤

lemma *ack-le-mono1*: *i* ≤ *j* ==> *ack i k* ≤ *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

32.2 Primitive Recursive Functions

primrec *hd0* :: *nat list* => *nat* **where**
hd0 [] = *0* |
hd0 (*m* # *ms*) = *m*

Inductive definition of the set of primitive recursive functions of type *nat list* ⇒ *nat*.

definition *SC* :: *nat list* => *nat* **where**
SC l = *Suc* (*hd0 l*)

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

definition *PROJ* :: *nat* => *nat list* => *nat* **where**
PROJ i l = *hd0* (*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*
by (*simp add*: *SC-def*)

lemma *CONSTANT* [*simp*]: *CONSTANT k l* = *k*
by (*simp add*: *CONSTANT-def*)

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

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

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

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

MAIN RESULT

lemma *SC-case*: *SC l* < *ack 1* (*listsum l*)
apply (*unfold SC-def*)
apply (*induct l*)
apply (*simp-all add*: *le-add1 le-imp-less-Suc*)

done

lemma *CONSTANT-case*: *CONSTANT k l* < *ack k* (*listsum l*)
by *simp*

lemma *PROJ-case*: *PROJ i l* < *ack 0* (*listsum l*)
apply (*simp add*: *PROJ-def*)
apply (*induct l arbitrary*:*i*)
apply (*auto simp add*: *drop-Cons split*: *nat.split*)
apply (*blast intro*: *less-le-trans le-add2*)
done

COMP case

lemma *COMP-map-aux*: ∀*f* ∈ *set fs*. *PRIMREC f* ∧ (∃*kf*. ∀*l*. *f l* < *ack kf* (*listsum l*))
==> ∃*k*. ∀*l*. *listsum* (*map* (λ*f*. *f l*) *fs*) < *ack k* (*listsum l*)
apply (*induct fs*)
apply (*rule-tac x* = *0* **in** *exI*)
apply *simp*
apply *simp*
apply (*blast intro*: *add-less-mono ack-add-bound less-trans*)
done

lemma *COMP-case*:
∀*l*. *g l* < *ack kg* (*listsum l*) ==>
∀*f* ∈ *set fs*. *PRIMREC f* ∧ (∃*kf*. ∀*l*. *f l* < *ack kf* (*listsum l*))
==> ∃*k*. ∀*l*. *COMP g fs l* < *ack k* (*listsum 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* + *listsum l* < *ack kf* (*listsum l*) ==>
∀*l*. *g l* + *listsum l* < *ack kg* (*listsum l*) ==>
PREC f g l + *listsum l* < *ack* (*Suc* (*kf* + *kg*)) (*listsum l*)
apply (*unfold PREC-def*)
apply (*case-tac l*)
apply *simp-all*
apply (*blast intro*: *less-trans*)
apply (*erule ssubst*) — get rid of the needless assumption
apply (*induct-tac a*)

apply *simp-all*

base case

apply (*blast intro*: *le-add1* [*THEN le-imp-less-Suc*, *THEN ack-less-mono1*] *less-trans*)

induction step

apply (*rule Suc-leI* [*THEN le-less-trans*])
apply (*rule le-refl* [*THEN add-le-mono*, *THEN le-less-trans*])
prefer *2*
apply (*erule spec*)
apply (*simp add*: *le-add2*)

final part of the simplification

apply *simp*
apply (*rule le-add2* [*THEN ack-le-mono1*, *THEN le-less-trans*])
apply (*erule ack-less-mono2*)
done

lemma *PREC-case*:
$\forall l.\ f\ l < ack\ kf\ (listsum\ l) \Longrightarrow$
$\forall l.\ g\ l < ack\ kg\ (listsum\ l) \Longrightarrow$
$\exists k.\ \forall l.\ PREC\ f\ g\ l < ack\ k\ (listsum\ l)$
by (*metis le-less-trans* [*OF le-add1 PREC-case-aux*] *ack-add-bound2*)

lemma *ack-bounds-PRIMREC*: $PRIMREC\ f \Longrightarrow \exists k.\ \forall l.\ f\ l < ack\ k\ (listsum\ l)$
apply (*erule PRIMREC.induct*)
apply (*blast intro*: *SC-case CONSTANT-case PROJ-case COMP-case PREC-case*)+
done

theorem *ack-not-PRIMREC*:
$\neg\ PRIMREC\ (\lambda l.\ case\ l\ of\ [] \Rightarrow 0 \mid x \# l' \Rightarrow ack\ x\ x)$
apply (*rule notI*)
apply (*erule ack-bounds-PRIMREC* [*THEN exE*])
apply (*rule less-irrefl* [*THEN notE*])
apply (*drule-tac* $x = [x]$ **in** *spec*)
apply *simp*
done

end

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

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

definition
*sublattice :: ('a potype * 'a set)set* **where**
sublattice =
(SIGMA cl: CompleteLattice.
{S. S ⊆ 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 *S =*
fixes *cl :: 'a potype*
and *A :: 'a set*
and *r :: ('a * 'a) set*
defines *A-def: A == pset cl*
and *r-def: r == order cl*

locale *PO* = *S* +
  assumes *cl-po*: *cl* : *PartialOrder*

locale *CL* = *S* +
  assumes *cl-co*: *cl* : *CompleteLattice*

sublocale *CL* < *PO*
apply (*simp-all add*: *A-def r-def*)
apply *unfold-locales*
using *cl-co* **unfolding** *CompleteLattice-def* **by** *auto*

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

sublocale *CLF* < *CL*
apply (*simp-all add*: *A-def r-def*)
apply *unfold-locales*
using *f-cl* **unfolding** *CLF-set-def* **by** *auto*

locale *Tarski* = *CLF* +
  fixes *Y* :: *'a set*
    and *intY1* :: *'a set*
    and *v* :: *'a*
  assumes
    Y-ss: *Y* ⊆ *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*|)

33.1 Partial Order

lemma (**in** *PO*) *dual*:
  PO (*dual cl*)
apply *unfold-locales*
using *cl-po*
unfolding *PartialOrder-def dual-def*
by *auto*

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

lemma (**in** *PO*) *PO-imp-sym* [*simp*]: *antisym r*

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

lemma (**in** *PO*) *PO-imp-trans* [*simp*]: *trans r*
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def r-def*)
done

lemma (**in** *PO*) *reflE*: $x \in A \Longrightarrow (x, x) \in r$
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def refl-on-def A-def r-def*)
done

lemma (**in** *PO*) *antisymE*: $[\!| (a, b) \in r; (b, a) \in r |\!] \Longrightarrow a = b$
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def antisym-def r-def*)
done

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

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

lemma (**in** *PO*) *po-subset-po*:
  $S \subseteq A \Longrightarrow (|$ *pset* $= S$, *order* $=$ *induced S r* $|) \in$ *PartialOrder*
apply (*simp* (*no-asm*) *add*: *PartialOrder-def*)
apply *auto*
— refl
apply (*simp add*: *refl-on-def induced-def*)
apply (*blast intro*: *reflE*)
— antisym
apply (*simp add*: *antisym-def induced-def*)
apply (*blast intro*: *antisymE*)
— trans
apply (*simp add*: *trans-def induced-def*)
apply (*blast intro*: *transE*)
done

lemma (**in** *PO*) *indE*: $[\!| (x, y) \in$ *induced S r*; $S \subseteq A$ $|\!] \Longrightarrow (x, y) \in r$
by (*simp add*: *add*: *induced-def*)

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

lemma (**in** *CL*) *CL-imp-ex-isLub*: $S \subseteq A \Longrightarrow \exists L.\ isLub\ S\ cl\ L$
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* $\in$ *PartialOrder*
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def dual-def refl-on-converse*
trans-converse antisym-converse)
done

lemma *Rdual*:
$\forall S.\ (S \subseteq A \dashrightarrow (\exists L.\ isLub\ S\ (|\ pset = A,\ order = r|)\ L))$
$\Longrightarrow \forall S.\ (S \subseteq A \dashrightarrow (\exists G.\ isGlb\ S\ (|\ pset = A,\ order = r|)\ G))$
apply *safe*
apply (*rule-tac x* = *lub* $\{y.\ y \in A\ \&\ (\forall k \in S.\ (y,\ k) \in r)\}$
(|*pset* = *A*, *order* = *r*|) **in** *exI*)
apply (*drule-tac x* = $\{y.\ y \in A\ \&\ (\forall k \in S.\ (y,k) \in r)\ \}$ **in** *spec*)
apply (*drule mp*, *fast*)
apply (*simp add*: *isLub-lub isGlb-def*)
apply (*simp add*: *isLub-def*, *blast*)
done

lemma *lub-dual-glb*: *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* $\subseteq$ *PartialOrder*
by (*simp add*: *PartialOrder-def CompleteLattice-def*, *fast*)

lemmas *CL-imp-PO* = *CL-subset-PO* [*THEN subsetD*]

lemma (**in** *CL*) *CO-refl-on*: *refl-on A r*
by (*rule PO-imp-refl-on*)

lemma (**in** *CL*) *CO-antisym*: *antisym r*
by (*rule PO-imp-sym*)

lemma (**in** *CL*) *CO-trans*: *trans r*
by (*rule PO-imp-trans*)

lemma *CompleteLatticeI*:
[| *po* ∈ *PartialOrder*; (∀ *S*. *S* ⊆ *pset po* −−> (∃ *L*. *isLub S po L*));
(∀ *S*. *S* ⊆ *pset po* −−> (∃ *G*. *isGlb S po G*))|]
==> *po* ∈ *CompleteLattice*
apply (*unfold CompleteLattice-def*, *blast*)
done

lemma (**in** *CL*) *CL-dualCL*: *dual cl* ∈ *CompleteLattice*
apply (*insert cl-co*)
apply (*simp add*: *CompleteLattice-def dual-def*)
apply (*fold dual-def*)
apply (*simp add*: *isLub-dual-isGlb* [*symmetric*] *isGlb-dual-isLub* [*symmetric*]
dualPO)
done

lemma (**in** *PO*) *dualA-iff*: *pset* (*dual cl*) = *pset cl*
by (*simp add*: *dual-def*)

lemma (**in** *PO*) *dualr-iff*: ((*x*, *y*) ∈ (*order*(*dual cl*))) = ((*y*, *x*) ∈ *order cl*)
by (*simp add*: *dual-def*)

lemma (**in** *PO*) *monotone-dual*:
monotone f (*pset cl*) (*order cl*)
==> *monotone f* (*pset* (*dual cl*)) (*order*(*dual cl*))
by (*simp add*: *monotone-def dualA-iff dualr-iff*)

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

lemma (**in** *PO*) *trans*:
(*x*, *y*) ∈ *r* ⟹ (*y*, *z*) ∈ *r* ⟹ (*x*, *z*) ∈ *r*
using *cl-po* **apply** (*auto simp add*: *PartialOrder-def r-def*)
unfolding *trans-def* **by** *blast*

lemma (**in** *PO*) *interval-not-empty*:

interval r a b $\neq \{\}$ ==> $(a, b) \in r$
apply (*simp add*: *interval-def*)
using *trans* **by** *blast*

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

33.2 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*)

lemma (**in** *CL*) *dual*:
CL (*dual cl*)
apply *unfold-locales*
using *cl-co* **unfolding** *CompleteLattice-def*
apply (*simp add*: *dualPO isGlb-dual-isLub* [*symmetric*] *isLub-dual-isGlb* [*symmetric*] *dualA-iff*)
done

33.3 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*:
    [| $S \subseteq A$; $L \in A$; $\forall x \in S.\ (x,L) \in r$;
       $\forall z \in A.\ (\forall y \in S.\ (y,z) \in r) \dashrightarrow (L,z) \in r$ |] ==> 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*:
[| $L \in A$; $\forall y \in S.\ (y, L) \in r$;
$(\forall z \in A.\ (\forall y \in S.\ (y, z){:}r) \longrightarrow (L, z) \in r)$|] ==> *isLub S cl L*
by (*simp add*: *isLub-def A-def r-def*)

33.4 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 dual*)
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 dual*)
apply (*simp add*: *dualA-iff A-def*, *assumption*)
done

Reduce the sublattice property by using substructural properties; abandoned see *Tarski-4.ML*.

lemma (**in** *CLF*) [*simp*]:
f: *pset cl* −> *pset cl* & *monotone f* (*pset cl*) (*order cl*)
apply (*insert f-cl*)
apply (*simp add*: *CLF-set-def*)
done

declare (**in** *CLF*) *f-cl* [*simp*]

lemma (**in** *CLF*) *f-in-funcset*: $f \in A \rightarrow A$
by (*simp add*: *A-def*)

lemma (**in** *CLF*) *monotone-f*: *monotone f A r*
by (*simp add*: *A-def r-def*)

lemma (**in** *CLF*) *CLF-dual*: (*dual cl*, f) $\in$ *CLF-set*
apply (*simp add*: *CLF-set-def CL-dualCL monotone-dual*)
apply (*simp add*: *dualA-iff*)
done

lemma (**in** *CLF*) *dual*:

CLF (*dual cl*) *f*
apply (*rule CLF.intro*)
apply (*rule CLF-dual*)
done

33.5 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*)

33.6 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*:
[| *H* = {*x*. (*x*, *f x*) $\in$ *r* & *x* $\in$ *A*} |] ==> (*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*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} \Longrightarrow lub\ H\ cl \in fix\ f\ A$
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*:
  $[|\ H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\};\ \ x \in P\ |] \Longrightarrow x \in H$
by (*simp add*: *P-def fix-imp-eq* [*of - f A*] *reflE CO-refl-on*
    fix-subset [*of f A*, *THEN subsetD*])

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

33.7 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*: $H = \{x.\ (f\ x,\ x) \in r\ \&\ x \in A\} \Longrightarrow glb\ H\ cl \in P$

— 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 dual*)
apply (*simp add*: *dualr-iff dualA-iff*)
done

lemma (**in** *CLF*) *T-thm-1-glb*: *glb P cl* = *glb* {*x*. (*f x*, *x*) ∈ *r* & *x* ∈ *A*} *cl*
apply (*simp add*: *glb-dual-lub P-def A-def r-def*)
apply (*rule dualA-iff* [*THEN subst*])
apply (*simp add*: *CLF.T-thm-1-lub* [*of - f*, *OF dual*]
dualPO CL-dualCL CLF-dual dualr-iff)
done

33.8 interval

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

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

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

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

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

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

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

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

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

lemma (**in** *CLF*) *G-in-interval*:
[| $a \in A$; $b \in A$; *interval r a b* $\neq \{\}$; $S \subseteq$ *interval r a b*; *isGlb S cl G*;
$S \neq \{\}$ |] ==> $G \in$ *interval r a b*
apply (*simp add*: *interval-dual*)
apply (*simp add*: *CLF.L-in-interval* [*of - f*, *OF dual*]
dualA-iff A-def isGlb-dual-isLub)
done

lemma (**in** *CLF*) *intervalPO*:
[| $a \in A$; $b \in A$; *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 (*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*]

33.9 Top and Bottom

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

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

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

lemma (**in** *CLF*) *Top-in-lattice*: *Top cl* ∈ *A*
apply (*simp add*: *Top-dual-Bot A-def*)
apply (*rule dualA-iff* [*THEN subst*])
apply (*rule CLF.Bot-in-lattice* [*OF dual*])
done

lemma (**in** *CLF*) *Top-prop*: *x* ∈ *A* ==> (*x*, *Top cl*) ∈ *r*
apply (*simp add*: *Top-def greatest-def*)
apply (*rule-tac a*=*lub A cl* **in** *someI2*)
apply (*rule someI2*)
apply (*simp-all add*: *lub-in-lattice lub-upper*
r-def [*symmetric*] *A-def* [*symmetric*])
done

lemma (**in** *CLF*) *Bot-prop*: *x* ∈ *A* ==> (*Bot cl*, *x*) ∈ *r*
apply (*simp add*: *Bot-dual-Top r-def*)
apply (*rule dualr-iff* [*THEN subst*])
apply (*rule CLF.Top-prop* [*OF dual*])
apply (*simp add*: *dualA-iff A-def*)
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-on-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 (*rule CLF.Top-in-lattice* [*OF dual*])
apply (*rule CLF.Top-intv-not-empty* [*OF dual*])
apply (*simp add*: *dualA-iff A-def*)
done

33.10 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* ⊆ *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*)) ∈ *r to* (*x*, *lub Y cl*) ∈ *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* ⊆ *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* ∈ *intY1* ⟹ *f x* ∈ *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*) ∈ *r*
apply (*rule-tac f=f* **in** *monotoneE*)
apply (*rule monotone-f*)
apply (*rule lubY-in-A*)
apply (*simp add*: *intY1-def interval-def intY1-elem*)
apply (*simp add*: *intY1-def interval-def*)
— (*f x*, *Top cl*) ∈ *r*
apply (*rule Top-prop*)
apply (*rule f-in-funcset* [*THEN funcset-mem*])
apply (*simp add*: *intY1-def interval-def intY1-elem*)
done

lemma (**in** *Tarski*) *intY1-func*: (%*x*: *intY1*. *f x*) ∈ *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*)
unfolding *v-def*
apply (*rule CLF.glbH-is-fixp* [*OF CLF.intro*, *unfolded CLF-set-def*, *of* (⌊*pset* = *intY1*, *order* = *induced intY1 r*⌋), *simplified*])
apply *auto*
apply (*rule intY1-is-cl*)
apply (*rule intY1-func*)
apply (*rule intY1-mono*)
done

lemma (**in** *Tarski*) *z-in-interval*:
[| $z \in P$; $\forall y{\in}Y.\ (y,\ z) \in$ *induced P r* |] ==> $z \in$ *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*: [| $z \in P$; $\forall y{\in}Y.\ (y,\ z) \in$ *induced P r* |]
==> ((%x: *intY1*. *f x*) *z*, *z*) $\in$ *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*:
$\exists L.$ *isLub Y* (| *pset* = *P*, *order* = *induced P r* |) *L*
apply (*rule-tac x* = *v* **in** *exI*)
apply (*simp add*: *isLub-def*)
— $v \in P$
apply (*simp add*: *v-in-P*)
apply (*rule conjI*)
— v is lub
— *1*. $\forall y{:}Y.\ (y,\ v) \in$ *induced P r*
apply (*rule ballI*)
apply (*simp add*: *induced-def subsetD v-in-P*)
apply (*rule conjI*)
apply (*erule Y-ss* [*THEN subsetD*])
apply (*rule-tac b* = *lub Y cl* **in** *transE*)
apply (*rule lub-upper*)
apply (*rule Y-subset-A*, *assumption*)
apply (*rule-tac b* = *Top cl* **in** *interval-imp-mem*)
apply (*simp add*: *v-def*)
apply (*fold intY1-def*)
apply (*rule CL.glb-in-lattice* [*OF CL.intro* [*OF intY1-is-cl*], *simplified*])
apply *auto*
apply (*rule indI*)
prefer *3* **apply** *assumption*
prefer *2* **apply** (*simp add*: *v-in-P*)
apply (*unfold v-def*)
apply (*rule indE*)
apply (*rule-tac* [*2*] *intY1-subset*)
apply (*rule CL.glb-lower* [*OF CL.intro* [*OF intY1-is-cl*], *simplified*])
apply (*simp add*: *CL-imp-PO intY1-is-cl*)
apply *force*

apply (*simp add*: *induced-def intY1-f-closed z-in-interval*)
apply (*simp add*: *P-def fix-imp-eq* [*of - f A*] *reflE*
fix-subset [*of f A*, *THEN subsetD*])
done

lemma *CompleteLatticeI-simp*:
[| (| *pset* = *A*, *order* = *r* |) ∈ *PartialOrder*;
∀ *S*. *S* ⊆ *A* −−> (∃ *L*. *isLub S* (| *pset* = *A*, *order* = *r* |) *L*) |]
==> (| *pset* = *A*, *order* = *r* |) ∈ *CompleteLattice*
by (*simp add*: *CompleteLatticeI Rdual*)

theorem (**in** *CLF*) *Tarski-full*:
(| *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 (*rule Tarski.tarski-full-lemma* [*OF Tarski.intro* [*OF - Tarski-axioms.intro*]])
proof − **show** *CLF cl f* **.. qed**

end

34 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 ≠ [] ==> *bv-to-nat bv* = *bitval* (*hd bv*) ∗ *2* ^ (*length bv* − *1*) + *bv-to-nat* (*tl bv*)
by (*cases bv*) (*simp-all add*: *bv-to-nat-helper*)

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

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

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

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

lemma *full-adder-correct*:
bv-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*)

34.1 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

35 Classical Predicate Calculus Problems

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

35.1 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 $(P \dashrightarrow Q \mid R) \dashrightarrow (P \dashrightarrow Q) \mid (P \dashrightarrow R)$
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.

35.1.1 Pelletier's examples

1

lemma $(P \dashrightarrow Q) \;=\; (\sim Q \dashrightarrow \sim P)$
by *blast*

2

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

3

lemma $\sim (P \dashrightarrow Q) \dashrightarrow (Q \dashrightarrow P)$
by *blast*

4

lemma $(\sim P \dashrightarrow Q) \;=\; (\sim Q \dashrightarrow P)$
by *blast*

5

lemma $((P|Q) \dashrightarrow (P|R)) \dashrightarrow (P|(Q \dashrightarrow R))$
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*

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

From Wishnu Prasetya

lemma $(\forall s.\ q(s) --> r(s)) \ \& \ \sim r(s) \ \& \ (\forall s.\ \sim r(s) \ \& \ \sim q(s) --> p(t) \mid q(t))$
$--> p(t) \mid r(t)$
by *blast*

35.1.3 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*

35.1.4 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 $(\forall x\ y.\ \exists z.\ \forall w.\ (P(x)\&Q(y)-->R(z)\&S(w)))$
$--> (\exists x\ y.\ P(x) \ \& \ Q(y)) --> (\exists z.\ R(z))$

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

Problem 24

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

Problem 25

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

Problem 26

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

Problem 27

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

Problem 28. AMENDED

lemma $(\forall x.\ P(x)\ {-->}\ (\forall x.\ Q(x)))\ \&$
$(((\forall x.\ Q(x)|R(x))\ {-->}\ (\exists x.\ Q(x)\&S(x)))\ \&$
$((\exists x.\ S(x))\ {-->}\ (\forall x.\ L(x)\ {-->}\ M(x)))$
$\ {-->}\ (\forall x.\ P(x)\ \&\ L(x)\ {-->}\ M(x))$

by *blast*

Problem 29. Essentially the same as Principia Mathematica *11.71

lemma $(\exists x.\ F(x)) \ \&\ (\exists y.\ G(y))$
$\quad --> (\ ((\forall x.\ F(x) --> H(x)) \ \&\ (\forall y.\ G(y) --> J(y))) \ =$
$\qquad (\forall x\ y.\ F(x) \ \&\ G(y) --> H(x) \ \&\ J(y)))$
by *blast*

Problem 30

lemma $(\forall x.\ P(x) \mid Q(x) --> \sim R(x)) \ \&$
$\qquad (\forall x.\ (Q(x) --> \sim S(x)) --> P(x) \ \&\ R(x))$
$\quad --> (\forall x.\ S(x))$
by *blast*

Problem 31

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

Problem 32

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

Problem 33

lemma $(\forall x.\ P(a) \ \&\ (P(x) --> P(b)) --> P(c)) \ =$
$\quad (\forall x.\ (\sim P(a) \mid P(x) \mid P(c)) \ \&\ (\sim P(a) \mid \sim P(b) \mid P(c)))$
by *blast*

Problem 34 AMENDED (TWICE!!)

Andrews's challenge

lemma $((\exists x.\ \forall y.\ p(x) = p(y)) \ =$
$\qquad ((\exists x.\ q(x)) = (\forall y.\ p(y)))) \ =$
$\qquad ((\exists x.\ \forall y.\ q(x) = q(y)) \ =$
$\qquad ((\exists x.\ p(x)) = (\forall y.\ q(y))))$
by *blast*

Problem 35

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

Problem 36

lemma $(\forall x.\ \exists y.\ J\ x\ y) \ \&$

```
      (∀x. ∃y. G x y) &
      (∀x y. J x y | G x y −−>
      (∀z. J y z | G y z −−> H x z))
   −−> (∀x. ∃y. H x y)
by blast
```

Problem 37

```
lemma (∀z. ∃w. ∀x. ∃y.
         (P x z −−>P y w) & P y z & (P y w −−> (∃u. Q u w))) &
      (∀x z. ~(P x z) −−> (∃y. Q y z)) &
      ((∃x y. Q x y) −−> (∀x. R x x))
   −−> (∀x. ∃y. R x y)
by blast
```

Problem 38

```
lemma (∀x. p(a) & (p(x) −−> (∃y. p(y) & r x y)) −−>
         (∃z. ∃w. p(z) & r x w & r w z)) =
    (∀x. (~p(a) | p(x) | (∃z. ∃w. p(z) & r x w & r w z)) &
         (~p(a) | ~(∃y. p(y) & r x y) |
          (∃z. ∃w. p(z) & r x w & r w z)))
by blast
```

Problem 39

```
lemma ~ (∃x. ∀y. F y x = (~ F y y))
by blast
```

Problem 40. AMENDED

```
lemma (∃y. ∀x. F x y = F x x)
      −−>  ~ (∀x. ∃y. ∀z. F z y = (~ F z x))
by blast
```

Problem 41

```
lemma (∀z. ∃y. ∀x. f x y = (f x z & ~ f x x))
            −−> ~ (∃z. ∀x. f x z)
by blast
```

Problem 42

```
lemma ~ (∃y. ∀x. p x y = (~ (∃z. p x z & p z x)))
by blast
```

Problem 43!!

```
lemma (∀x::'a. ∀y::'a. q x y = (∀z. p z x = (p z y::bool)))
  −−> (∀x. (∀y. q x y = (q y x::bool)))
by blast
```

Problem 44

```
lemma (∀x. f(x) −−>
```

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

Problem 45

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

35.1.5 Problems (mainly) involving equality or functions

Problem 48

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

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

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

Problem 51

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

Problem 52. Almost the same as 51.

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

Problem 55

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

lemma $\mathit{lives}(\mathit{agatha})\ \&\ \mathit{lives}(\mathit{butler})\ \&\ \mathit{lives}(\mathit{charles})\ \&$
$(\mathit{killed}\ \mathit{agatha}\ \mathit{agatha}\ |\ \mathit{killed}\ \mathit{butler}\ \mathit{agatha}\ |\ \mathit{killed}\ \mathit{charles}\ \mathit{agatha})\ \&$

$(\forall x\ y.\ killed\ x\ y \longrightarrow hates\ x\ y\ \&\ \sim richer\ x\ y)\ \&$
$(\forall x.\ hates\ agatha\ x \longrightarrow \sim hates\ charles\ x)\ \&$
$(hates\ agatha\ agatha\ \&\ hates\ agatha\ charles)\ \&$
$(\forall x.\ lives(x)\ \&\ \sim richer\ x\ agatha \longrightarrow hates\ butler\ x)\ \&$
$(\forall x.\ hates\ agatha\ x \longrightarrow hates\ butler\ x)\ \&$
$(\forall x.\ \sim hates\ x\ agatha \mid \sim hates\ x\ butler \mid \sim hates\ x\ charles) \longrightarrow$
$killed\ ?who\ agatha$
by *fast*

Problem 56

lemma $(\forall x.\ (\exists y.\ P(y)\ \&\ x{=}f(y)) \longrightarrow P(x)) = (\forall x.\ P(x) \longrightarrow P(f(x)))$
by *blast*

Problem 57

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

Problem 58 NOT PROVED AUTOMATICALLY

lemma $(\forall x\ y.\ f(x){=}g(y)) \longrightarrow (\forall x\ y.\ f(f(x)){=}f(g(y)))$
by (*fast intro*: *arg-cong* [*of concl*: f])

Problem 59

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

Problem 60

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

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

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

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

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

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

lemma $(\forall x.\ F(x)\ \&\ \sim G(x) \longrightarrow (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y) \longrightarrow K(y)))\ \&$

$(\forall x.\ K(x) \longrightarrow \sim G(x)) \quad \longrightarrow \quad (\exists x.\ K(x) \longrightarrow \sim G(x))$
by *fast*

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

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

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

35.2 Model Elimination Prover

Trying out meson with arguments

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

The "small example" from Bezem, Hendriks and de Nivelle, Automatic Proof Construction in Type Theory Using Resolution, JAR 29: 3-4 (2002), pages 253-275

lemma $(\forall x\ y\ z.\ R(x,y)\ \&\ R(y,z) \longrightarrow R(x,z))\ \&$
$(\forall x.\ \exists y.\ R(x,y)) \longrightarrow$
$\sim (\forall x.\ P\ x = (\forall y.\ R(x,y) \longrightarrow \sim\ P\ y))$
by (*tactic*⟪*Meson.safe-best-meson-tac* @{*claset*} *1*⟫)
— In contrast, *meson* is SLOW: 7.6s on griffon

35.2.1 Pelletier's examples

1

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

2

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

3

lemma $\sim(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P)$
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 $P \mid \sim P$
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*

35.2.2 Classical Logic: examples with quantifiers

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

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

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

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

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

Needs double instantiation of EXISTS

lemma $\exists x.\ P\ x \ --> \ P\ a \ \& \ P\ b$
by *blast*

lemma $\exists z.\ P\ z \ --> \ (\forall x.\ P\ x)$
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*

35.2.3 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 $(\forall x\ y.\ \exists z.\ \forall w.\ (P\ x\ \&\ Q\ y\ \longrightarrow\ R\ z\ \&\ S\ w))$
$\longrightarrow\ (\exists x\ y.\ P\ x\ \&\ Q\ y)\ \longrightarrow\ (\exists z.\ R\ z)$
by *blast*

Problem 21

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

Problem 22

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

Problem 23

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

Problem 24

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

Problem 25

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

Problem 26; has 24 Horn clauses

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

Problem 27; has 13 Horn clauses

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

Problem 28. AMENDED; has 14 Horn clauses

lemma $(\forall x.\ P\ x\ \longrightarrow\ (\forall x.\ Q\ x))\ \&$

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

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

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

Problem 30

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

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

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

Problem 32

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

Problem 33; has 55 Horn clauses

lemma $(\forall x.\ P\ a\ \&\ (P\ x \longrightarrow P\ b) \longrightarrow P\ c)\ =$
$(\forall x.\ (\sim P\ a \mid P\ x \mid P\ c)\ \&\ (\sim P\ a \mid \sim P\ b \mid P\ c))$
by *blast*

Problem 34: Andrews's challenge has 924 Horn clauses

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

Problem 35

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

Problem 36; has 15 Horn clauses

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&\ (\forall x.\ \exists y.\ G\ x\ y)\ \&$

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

Problem 37; has 10 Horn clauses

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

Problem 38

Quite hard: 422 Horn clauses!!

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

Problem 39

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

Problem 40. AMENDED

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

Problem 41

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

Problem 42

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

Problem 43 NOW PROVED AUTOMATICALLY!!

lemma $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}bool)))$
$\longrightarrow (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}bool)))$
by *blast*

Problem 44: 13 Horn clauses; 7-step proof

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

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

Problem 45; has 27 Horn clauses; 54-step proof

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

Problem 46; has 26 Horn clauses; 21-step proof

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

Problem 47. Schubert's Steamroller. 26 clauses; 63 Horn clauses. 87094 inferences so far. Searching to depth 36

lemma $(\forall x.\ \mathit{wolf}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{wolf}\ x)\ \&$
$(\forall x.\ \mathit{fox}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{fox}\ x)\ \&$
$(\forall x.\ \mathit{bird}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{bird}\ x)\ \&$
$(\forall x.\ \mathit{caterpillar}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{caterpillar}\ x)\ \&$
$(\forall x.\ \mathit{snail}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{snail}\ x)\ \&$
$(\forall x.\ \mathit{grain}\ x \longrightarrow \mathit{plant}\ x)\ \&\ (\exists x.\ \mathit{grain}\ x)\ \&$
$(\forall x.\ \mathit{animal}\ x \longrightarrow$
$((\forall y.\ \mathit{plant}\ y \longrightarrow \mathit{eats}\ x\ y)\ \vee$
$(\forall y.\ \mathit{animal}\ y\ \&\ \mathit{smaller\text{-}than}\ y\ x\ \&$
$(\exists z.\ \mathit{plant}\ z\ \&\ \mathit{eats}\ y\ z) \longrightarrow \mathit{eats}\ x\ y)))\ \&$
$(\forall x\ y.\ \mathit{bird}\ y\ \&\ (\mathit{snail}\ x \vee \mathit{caterpillar}\ x) \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{fox}\ y \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{fox}\ x\ \&\ \mathit{wolf}\ y \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{wolf}\ x\ \&\ (\mathit{fox}\ y \vee \mathit{grain}\ y) \longrightarrow \sim \mathit{eats}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{caterpillar}\ y \longrightarrow \mathit{eats}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{snail}\ y \longrightarrow \sim \mathit{eats}\ x\ y)\ \&$
$(\forall x.\ (\mathit{caterpillar}\ x \vee \mathit{snail}\ x) \longrightarrow (\exists y.\ \mathit{plant}\ y\ \&\ \mathit{eats}\ x\ y))$
$\longrightarrow (\exists x\ y.\ \mathit{animal}\ x\ \&\ \mathit{animal}\ y\ \&\ (\exists z.\ \mathit{grain}\ z\ \&\ \mathit{eats}\ y\ z\ \&\ \mathit{eats}\ x\ y))$
by $(\mathit{tactic}\langle\!\langle \mathit{Meson.safe\text{-}best\text{-}meson\text{-}tac}\ @\{\mathit{claset}\}\ 1\rangle\!\rangle)$
— Nearly twice as fast as *meson*, which performs iterative deepening rather than best-first search

The Los problem. Circulated by John Harrison

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

$(\forall x\ y.\ P\ x\ y \longrightarrow P\ y\ x)\ \&$
$(\forall x\ y.\ P\ x\ y \mid Q\ x\ y)$
$\longrightarrow (\forall x\ y.\ P\ x\ y) \mid (\forall x\ y.\ Q\ x\ y)$
by *meson*

A similar example, suggested by Johannes Schumann and credited to Pelletier

lemma $(\forall x\ y\ z.\ P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z) \longrightarrow$
$(\forall x\ y\ z.\ Q\ x\ y \longrightarrow Q\ y\ z \longrightarrow Q\ x\ z) \longrightarrow$
$(\forall x\ y.\ Q\ x\ y \longrightarrow Q\ y\ x) \longrightarrow\ (\forall x\ y.\ P\ x\ y \mid Q\ x\ y) \longrightarrow$
$(\forall x\ y.\ P\ x\ y) \mid (\forall x\ y.\ Q\ x\ y)$
by *meson*

Problem 50. What has this to do with equality?

lemma $(\forall x.\ P\ a\ x \mid (\forall y.\ P\ x\ y)) \longrightarrow (\exists x.\ \forall y.\ P\ x\ y)$
by *blast*

Problem 54: NOT PROVED

lemma $(\forall y{::}'a.\ \exists z.\ \forall x.\ F\ x\ z = (x{=}y)) \longrightarrow$
$\sim (\exists w.\ \forall x.\ F\ x\ w = (\forall u.\ F\ x\ u \longrightarrow (\exists y.\ F\ y\ u\ \&\ \sim (\exists z.\ F\ z\ u\ \&\ F\ z\ y))))$
oops

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988). *meson* cannot report who killed Agatha.

lemma *lives agatha & lives butler & lives charles &*
(killed agatha agatha | killed butler agatha | killed charles agatha) &
(∀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 $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z \longrightarrow P\ x\ z) \longrightarrow P\ (f\ a\ b)\ (f\ a\ c)$
by *blast*

Problem 58: Challenge found on info-hol

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

Problem 59

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

by *blast*

Problem 60

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

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ \text{--->}\ p(f\ x))\ \text{--->}\ p(f(f\ x)))\ =$
$(\forall x.\ (\sim\ p\ a\ |\ p\ x\ |\ p(f(f\ x)))\ \&$
$(\sim\ p\ a\ |\ \sim\ p(f\ x)\ |\ p(f(f\ x))))$
by *blast*

* 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: $\forall x\ y.\ \ T(i\ x\ y)\ \&\ T\ x\ \text{--->}\ T\ y$
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

36 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*

36.1 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*

36.2 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*

36.3 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

36.4 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* $\geq$ *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* ⟹ ∃ *A*. (∀ *x* ∈ *A*. *P x*) ∧ (∃ *y*. *y* ∈ *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

37 Meson test cases

theory *Meson-Test*
imports *Main*
begin

WARNING: there are many potential conflicts between variables used below and constants declared in HOL!

hide *const subset member quotient*

Test data for the MESON proof procedure (Excludes the equality problems 51, 52, 56, 58)

37.1 Interactive examples

ML ⟪
writelnProblem 25;
Goal $(\exists x.\ P\ x)\ \&\ (\forall x.\ L\ x\ \text{--->}\ \sim\ (M\ x\ \&\ R\ x))\ \&\ (\forall x.\ P\ x\ \text{--->}\ (M\ x\ \&\ L\ x))\ \&\ ((\forall x.\ P\ x\ \text{--->}\ Q\ x)\ |\ (\exists x.\ P\ x\ \&\ R\ x))\ \text{--->}\ (\exists x.\ Q\ x\ \&\ P\ x)$;
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 $((\exists x.\ p\ x) = (\exists x.\ q\ x))\ \&\ (\forall x.\ \forall y.\ p\ x\ \&\ q\ y\ \text{--->}\ (r\ x = s\ y))\ \text{--->}\ ((\forall x.\ p\ x\ \text{--->}\ r\ x) = (\forall x.\ q\ x\ \text{--->}\ s\ x))$;
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 (∀x. ∀y. q x y = (∀z. p z x = (p z y::bool))) --> (∀x. (∀y. q x y = (q y
x::bool)));
by (rtac ccontr 1);
val [prem43] = gethyps 1;
val nnf43 = Meson.make-nnf prem43;
val xsko43 = Meson.skolemize nnf43;
by (cut-facts-tac [xsko43] 1 THEN REPEAT (etac exE 1));
val [-,sko43] = gethyps 1;
val clauses43 = Meson.make-clauses [sko43];   (*6*)
val horns43 = Meson.make-horns clauses43;     (*16*)
val go43::- = Meson.gocls clauses43;
⟩⟩
```

ML ⟨⟨

```
Goal False;
by (rtac go43 1);
by (Meson.best-prolog-tac Meson.size-of-subgoals horns43);   (*1.6 secs*)
⟩⟩
```

MORE and MUCH HARDER test data for the MESON proof procedure (courtesy John Harrison).

abbreviation *EQU001-0-ax equal* ≡ (∀*X*. *equal*(*X*::'*a*,*X*)) &
  (∀*Y X*. *equal*(*X*::'*a*,*Y*) --> *equal*(*Y*::'*a*,*X*)) &
  (∀*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 ≡
  (∀*X Y*. *sum*(*X*::'*a*,*Y*,*add*(*X*::'*a*,*Y*))) &
  (∀*X Y*. *product*(*X*::'*a*,*Y*,*multiply*(*X*::'*a*,*Y*))) &
  (∀*Y X Z*. *sum*(*X*::'*a*,*Y*,*Z*) --> *sum*(*Y*::'*a*,*X*,*Z*)) &
  (∀*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 &
(~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 &
(~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 &
(~sum(x::'a,multiplicative-identity,multiplicative-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 &
(~product(x::'a,additive-identity,additive-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 &
(~equal(INVERSE(additive-identity),multiplicative-identity)) −−> False
by *meson*

abbreviation *CAT003-0-ax f1 compos codomain domain equal there-exists equivalent* ≡
$(\forall\, Y\ X.\ equivalent(X::'a, Y) \longrightarrow there\text{-}exists(X))$ &
$(\forall\, X\ Y.\ equivalent(X::'a, Y) \longrightarrow equal(X::'a, Y))$ &
$(\forall\, X\ Y.\ there\text{-}exists(X)$ & $equal(X::'a, Y) \longrightarrow equivalent(X::'a, Y))$ &
$(\forall\, X.\ there\text{-}exists(domain(X)) \longrightarrow there\text{-}exists(X))$ &
$(\forall\, X.\ there\text{-}exists(codomain(X)) \longrightarrow there\text{-}exists(X))$ &
$(\forall\, Y\ X.\ there\text{-}exists(compos(X::'a, Y)) \longrightarrow there\text{-}exists(domain(X)))$ &

(∀ X Y. there-exists(compos(X::'a,Y)) −−> equal(domain(X),codomain(Y)))
&
(∀ X Y. there-exists(domain(X)) & equal(domain(X),codomain(Y)) −−> there-exists(compos(X::'a,Y)))
&
(∀ X Y Z. equal(compos(X::'a,compos(Y::'a,Z)),compos(compos(X::'a,Y),Z)))
&
(∀ X. equal(compos(X::'a,domain(X)),X)) &
(∀ X. equal(compos(codomain(X),X),X)) &
(∀ X Y. equivalent(X::'a,Y) −−> there-exists(Y)) &
(∀ X Y. there-exists(X) & there-exists(Y) & equal(X::'a,Y) −−> equivalent(X::'a,Y))
&
(∀ Y X. there-exists(compos(X::'a,Y)) −−> there-exists(codomain(X))) &
(∀ X Y. there-exists(f1(X::'a,Y)) | equal(X::'a,Y)) &
(∀ X Y. equal(X::'a,f1(X::'a,Y)) | equal(Y::'a,f1(X::'a,Y)) | equal(X::'a,Y))
&
(∀ X Y. equal(X::'a,f1(X::'a,Y)) & equal(Y::'a,f1(X::'a,Y)) −−> equal(X::'a,Y))

abbreviation *CAT003-0-eq f1 compos codomain domain equivalent there-exists equal* ≡
(∀ X Y. equal(X::'a,Y) & there-exists(X) −−> there-exists(Y)) &
(∀ X Y Z. equal(X::'a,Y) & equivalent(X::'a,Z) −−> equivalent(Y::'a,Z)) &
(∀ X Z Y. equal(X::'a,Y) & equivalent(Z::'a,X) −−> equivalent(Z::'a,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 Z. equal(X::'a,Y) −−> equal(compos(X::'a,Z),compos(Y::'a,Z))) &
(∀ X Z Y. equal(X::'a,Y) −−> equal(compos(Z::'a,X),compos(Z::'a,Y))) &
(∀ 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)))

lemma *CAT001-3*:
EQU001-0-ax equal &
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
(*there-exists*(*compos*(a::'a,b))) &
(∀ Y X Z. equal(compos(compos(a::'a,b),X),Y) & equal(compos(compos(a::'a,b),Z),Y)
−−> equal(X::'a,Z)) &
(*there-exists*(*compos*(b::'a,h))) &
(*equal*(*compos*(b::'a,h),*compos*(b::'a,g))) &
(~*equal*(h::'a,g)) −−> *False*
by *meson*

lemma *CAT003-3*:
EQU001-0-ax equal &
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
(*there-exists*(*compos*(a::'a,b))) &
(∀ 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* ≡
$(\forall X\ Y.\ defined(X::'a,Y) --> product(X::'a,Y,compos(X::'a,Y)))$ &
$(\forall Z\ X\ Y.\ product(X::'a,Y,Z) --> defined(X::'a,Y))$ &
$(\forall X\ Xy\ Y\ Z.\ product(X::'a,Y,Xy)\ \&\ defined(Xy::'a,Z) --> defined(Y::'a,Z))$ &
$(\forall Y\ Xy\ Z\ X\ Yz.\ product(X::'a,Y,Xy)\ \&\ product(Y::'a,Z,Yz)\ \&\ defined(Xy::'a,Z) --> defined(X::'a,Yz))$ &
$(\forall Xy\ Y\ Z\ X\ Yz\ Xyz.\ product(X::'a,Y,Xy)\ \&\ product(Xy::'a,Z,Xyz)\ \&\ product(Y::'a,Z,Yz) --> product(X::'a,Yz,Xyz))$ &
$(\forall Z\ Yz\ X\ Y.\ product(Y::'a,Z,Yz)\ \&\ defined(X::'a,Yz) --> defined(X::'a,Y))$ &
$(\forall Y\ X\ Yz\ Xy\ Z.\ product(Y::'a,Z,Yz)\ \&\ product(X::'a,Y,Xy)\ \&\ defined(X::'a,Yz) --> defined(Xy::'a,Z))$ &
$(\forall Yz\ X\ Y\ Xy\ Z\ Xyz.\ product(Y::'a,Z,Yz)\ \&\ product(X::'a,Yz,Xyz)\ \&\ product(X::'a,Y,Xy) --> product(Xy::'a,Z,Xyz))$ &
$(\forall Y\ X\ Z.\ defined(X::'a,Y)\ \&\ defined(Y::'a,Z)\ \&\ identity\text{-}map(Y) --> defined(X::'a,Z))$ &
$(\forall X.\ identity\text{-}map(domain(X)))$ &
$(\forall X.\ identity\text{-}map(codomain(X)))$ &
$(\forall X.\ defined(X::'a,domain(X)))$ &
$(\forall X.\ defined(codomain(X),X))$ &
$(\forall X.\ product(X::'a,domain(X),X))$ &
$(\forall X.\ product(codomain(X),X,X))$ &
$(\forall X\ Y.\ defined(X::'a,Y)\ \&\ identity\text{-}map(X) --> product(X::'a,Y,Y))$ &
$(\forall Y\ X.\ defined(X::'a,Y)\ \&\ identity\text{-}map(Y) --> product(X::'a,Y,X))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)\ \&\ product(X::'a,Y,W) --> equal(Z::'a,W))$

abbreviation *CAT001-0-eq compos defined identity-map codomain domain product equal* ≡
$(\forall X\ Y\ Z\ W.\ equal(X::'a,Y)\ \&\ product(X::'a,Z,W) --> product(Y::'a,Z,W))$ &
$(\forall X\ Z\ Y\ W.\ equal(X::'a,Y)\ \&\ product(Z::'a,X,W) --> product(Z::'a,Y,W))$ &
$(\forall X\ Z\ W\ Y.\ equal(X::'a,Y)\ \&\ product(Z::'a,W,X) --> product(Z::'a,W,Y))$ &
$(\forall X\ Y.\ equal(X::'a,Y) --> equal(domain(X),domain(Y)))$ &
$(\forall X\ Y.\ equal(X::'a,Y) --> equal(codomain(X),codomain(Y)))$ &
$(\forall X\ Y.\ equal(X::'a,Y)\ \&\ identity\text{-}map(X) --> identity\text{-}map(Y))$ &
$(\forall X\ Y\ Z.\ equal(X::'a,Y)\ \&\ defined(X::'a,Z) --> defined(Y::'a,Z))$ &
$(\forall X\ Z\ Y.\ equal(X::'a,Y)\ \&\ defined(Z::'a,X) --> defined(Z::'a,Y))$ &
$(\forall X\ Z\ Y.\ equal(X::'a,Y) --> equal(compos(Z::'a,X),compos(Z::'a,Y)))$ &
$(\forall X\ Y\ Z.\ equal(X::'a,Y) --> equal(compos(X::'a,Z),compos(Y::'a,Z)))$

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*))) &
(∀ *D F' E*. *equal*(*D*::'*a*,*E*) −−> *equal*(*apply*(*F'*::'*a*,*D*),*apply*(*F'*::'*a*,*E*))) &
(∀ *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*(
&
(∀ *Y*. ~*equal*(*Y*::'*a*,*apply*(*combinator*::'*a*,*Y*))) −−> *False*
by *meson*

lemma *COL023-1*:
EQU001-0-ax equal &

(∀ X Y Z. equal(apply(apply(apply(b::'a,X),Y),Z),apply(X::'a,apply(Y::'a,Z))))
&
(∀ X Y Z. equal(apply(apply(apply(n::'a,X),Y),Z),apply(apply(apply(X::'a,Z),Y),Z)))
&
(∀ A B C. equal(A::'a,B) −−> equal(apply(A::'a,C),apply(B::'a,C))) &
(∀ D F' E. equal(D::'a,E) −−> equal(apply(F'::'a,D),apply(F'::'a,E))) &
(∀ Y. ~equal(Y::'a,apply(combinator::'a,Y))) −−> False
by *meson*

lemma *COL032-1*:
EQU001-0-ax equal &
(∀ X. equal(apply(m::'a,X),apply(X::'a,X))) &
(∀ Y X Z. equal(apply(apply(apply(q::'a,X),Y),Z),apply(Y::'a,apply(X::'a,Z))))
&
(∀ A B C. equal(A::'a,B) −−> equal(apply(A::'a,C),apply(B::'a,C))) &
(∀ D F' E. equal(D::'a,E) −−> equal(apply(F'::'a,D),apply(F'::'a,E))) &
(∀ G H. equal(G::'a,H) −−> equal(f(G),f(H))) &
(∀ Y. ~equal(apply(Y::'a,f(Y)),apply(f(Y),apply(Y::'a,f(Y))))) −−> False
by *meson*

lemma *COL052-2*:
EQU001-0-ax equal &
(∀ X Y W. equal(response(compos(X::'a,Y),W),response(X::'a,response(Y::'a,W))))
&
(∀ X Y. agreeable(X) −−> equal(response(X::'a,common-bird(Y)),response(Y::'a,common-bird(Y))))
&
(∀ Z X. equal(response(X::'a,Z),response(compatible(X),Z)) −−> agreeable(X))
&
(∀ A B. equal(A::'a,B) −−> equal(common-bird(A),common-bird(B))) &
(∀ C D. equal(C::'a,D) −−> equal(compatible(C),compatible(D))) &
(∀ Q R. equal(Q::'a,R) & agreeable(Q) −−> agreeable(R)) &
(∀ A B C. equal(A::'a,B) −−> equal(compos(A::'a,C),compos(B::'a,C))) &
(∀ D F' E. equal(D::'a,E) −−> equal(compos(F'::'a,D),compos(F'::'a,E))) &
(∀ G H I'. equal(G::'a,H) −−> equal(response(G::'a,I'),response(H::'a,I'))) &
(∀ J L K'. equal(J::'a,K') −−> equal(response(L::'a,J),response(L::'a,K'))) &
(agreeable(c)) &
(~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*)) &

$(\forall\ Y\ Z\ X\ W.\ \mathit{halts3}(X::'a,Y,Z)\ \&\ \mathit{outputs}(X::'a,W)\ -->\ \mathit{halts3\text{-}outputs}(X::'a,Y,Z,W))$
$\&$
$(\forall\ Y\ X.\ \mathit{program\text{-}not\text{-}halts2}(X::'a,Y)\ -->\ \mathit{program}(X))\ \&$
$(\forall\ X\ Y.\ \sim(\mathit{program\text{-}not\text{-}halts2}(X::'a,Y)\ \&\ \mathit{halts2}(X::'a,Y)))\ \&$
$(\forall\ X\ Y.\ \mathit{program}(X)\ -->\ \mathit{program\text{-}not\text{-}halts2}(X::'a,Y)\ |\ \mathit{halts2}(X::'a,Y))\ \&$
$(\forall\ W\ X\ Y.\ \mathit{halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{halts2}(X::'a,Y))\ \&$
$(\forall\ Y\ X\ W.\ \mathit{halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{outputs}(X::'a,W))\ \&$
$(\forall\ Y\ X\ W.\ \mathit{halts2}(X::'a,Y)\ \&\ \mathit{outputs}(X::'a,W)\ -->\ \mathit{halts2\text{-}outputs}(X::'a,Y,W))$
$\&$
$(\forall\ X\ W\ Y\ Z.\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a,Y,Z,W)\ -->\ \mathit{program\text{-}halts2}(Y::'a,Z))$
$\&$
$(\forall\ X\ Y\ Z\ W.\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a,Y,Z,W)\ -->\ \mathit{halts3\text{-}outputs}(X::'a,Y,Z,W))$
$\&$
$(\forall\ X\ Y\ Z\ W.\ \mathit{program\text{-}halts2}(Y::'a,Z)\ \&\ \mathit{halts3\text{-}outputs}(X::'a,Y,Z,W)\ -->$
$\mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a,Y,Z,W))\ \&$
$(\forall\ X\ W\ Y\ Z.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a,Y,Z,W)\ -->\ \mathit{program\text{-}not\text{-}halts2}(Y::'a,Z))$
$\&$
$(\forall\ X\ Y\ Z\ W.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a,Y,Z,W)\ -->\ \mathit{halts3\text{-}outputs}(X::'a,Y,Z,W))$
$\&$
$(\forall\ X\ Y\ Z\ W.\ \mathit{program\text{-}not\text{-}halts2}(Y::'a,Z)\ \&\ \mathit{halts3\text{-}outputs}(X::'a,Y,Z,W)\ -->$
$\mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(X::'a,Y,Z,W))\ \&$
$(\forall\ X\ W\ Y.\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{program\text{-}halts2}(Y::'a,Y))$
$\&$
$(\forall\ X\ Y\ W.\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{halts2\text{-}outputs}(X::'a,Y,W))$
$\&$
$(\forall\ X\ Y\ W.\ \mathit{program\text{-}halts2}(Y::'a,Y)\ \&\ \mathit{halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a$
$\&$
$(\forall\ X\ W\ Y.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{program\text{-}not\text{-}halts2}(Y::'a,Y))$
$\&$
$(\forall\ X\ Y\ W.\ \mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a,Y,W)\ -->\ \mathit{halts2\text{-}outputs}(X::'a,Y,W))$
$\&$
$(\forall\ X\ Y\ W.\ \mathit{program\text{-}not\text{-}halts2}(Y::'a,Y)\ \&\ \mathit{halts2\text{-}outputs}(X::'a,Y,W)\ -->$
$\mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(X::'a,Y,W))\ \&$
$(\forall\ X.\ \mathit{algorithm\text{-}program\text{-}decides}(X)\ -->\ \mathit{program\text{-}program\text{-}decides}(c1))\ \&$
$(\forall\ W\ Y\ Z.\ \mathit{program\text{-}program\text{-}decides}(W)\ -->\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,Y,Z,\mathit{good}))$
$\&$
$(\forall\ W\ Y\ Z.\ \mathit{program\text{-}program\text{-}decides}(W)\ -->\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,Y,Z,\mathit{bad}))$
$\&$
$(\forall\ W.\ \mathit{program}(W)\ \&\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,\mathit{f3}(W),\mathit{f3}(W),\mathit{good})$
$\&\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,\mathit{f3}(W),\mathit{f3}(W),\mathit{bad})\ -->\ \mathit{program}(c2))$
$\&$
$(\forall\ W\ Y.\ \mathit{program}(W)\ \&\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,\mathit{f3}(W),\mathit{f3}(W),\mathit{good})$
$\&\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,\mathit{f3}(W),\mathit{f3}(W),\mathit{bad})\ -->\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(c2::'a,Y,g$
$\&$
$(\forall\ W\ Y.\ \mathit{program}(W)\ \&\ \mathit{program\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,\mathit{f3}(W),\mathit{f3}(W),\mathit{good})$
$\&\ \mathit{program\text{-}not\text{-}halts2\text{-}halts3\text{-}outputs}(W::'a,\mathit{f3}(W),\mathit{f3}(W),\mathit{bad})\ -->\ \mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(c2::'a,$
$\&$
$(\forall\ V.\ \mathit{program}(V)\ \&\ \mathit{program\text{-}halts2\text{-}halts2\text{-}outputs}(V::'a,\mathit{f4}(V),\mathit{good})\ \&\ \mathit{program\text{-}not\text{-}halts2\text{-}halts2\text{-}outputs}(V$
$-->\ \mathit{program}(c3))\ \&$

(∀ 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*) & *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*)) &
(∀ *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*) −−> *equidis-*
tant(*Y*::′*a*,*V*,*W*,*Z*)) &
(∀ *X V Y W Z*. *equal*(*X*::′*a*,*Y*) & *equidistant*(*V*::′*a*,*X*,*W*,*Z*) −−> *equidis-*
tant(*V*::′*a*,*Y*,*W*,*Z*)) &
(∀ *X V W Y Z*. *equal*(*X*::′*a*,*Y*) & *equidistant*(*V*::′*a*,*W*,*X*,*Z*) −−> *equidis-*

$tant(V::'a,W,Y,Z))$ &
$(\forall X\ V\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,Z,X)\ -->\ equidis-$
$tant(V::'a,W,Z,Y))$ &
$(\forall X\ Y\ V1\ V2\ V3\ V4.\ equal(X::'a,Y) --> equal(outer\text{-}pasch(X::'a,V1,V2,V3,V4),outer\text{-}pasch(Y::'a,V1,V$
&
$(\forall X\ V1\ Y\ V2\ V3\ V4.\ equal(X::'a,Y) --> equal(outer\text{-}pasch(V1::'a,X,V2,V3,V4),outer\text{-}pasch(V1::'a,Y,V$
&
$(\forall X\ V1\ V2\ Y\ V3\ V4.\ equal(X::'a,Y) --> equal(outer\text{-}pasch(V1::'a,V2,X,V3,V4),outer\text{-}pasch(V1::'a,V2,$
&
$(\forall X\ V1\ V2\ V3\ Y\ V4.\ equal(X::'a,Y) --> equal(outer\text{-}pasch(V1::'a,V2,V3,X,V4),outer\text{-}pasch(V1::'a,V2,$
&
$(\forall X\ V1\ V2\ V3\ V4\ Y.\ equal(X::'a,Y) --> equal(outer\text{-}pasch(V1::'a,V2,V3,V4,X),outer\text{-}pasch(V1::'a,V2,$
&
$(\forall A\ B\ C\ D\ E\ F'.\ equal(A::'a,B) --> equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F')))$
&
$(\forall G\ I'\ H\ J\ K'\ L.\ equal(G::'a,H) --> equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L)))$
&
$(\forall M\ O'\ P\ N\ Q\ R.\ equal(M::'a,N) --> equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R)))$
&
$(\forall S'\ U\ V\ W\ T'\ X.\ equal(S'::'a,T') --> equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X)))$
&
$(\forall Y\ A1\ B1\ C1\ D1\ Z.\ equal(Y::'a,Z) --> equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z)$
&
$(\forall E1\ F1\ G1\ H1\ I1\ J1.\ equal(E1::'a,F1) --> equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1,$
&
$(\forall K1\ M1\ L1\ N1\ O1\ P1.\ equal(K1::'a,L1) --> equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1,$
&
$(\forall Q1\ S1\ T1\ R1\ U1\ V1.\ equal(Q1::'a,R1) --> equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U$
&
$(\forall W1\ Y1\ Z1\ A2\ X1\ B2.\ equal(W1::'a,X1) --> equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2$
&
$(\forall C2\ E2\ F2\ G2\ H2\ D2.\ equal(C2::'a,D2) --> equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H$
&
$(\forall X\ Y\ V1\ V2\ V3.\ equal(X::'a,Y) --> equal(extension(X::'a,V1,V2,V3),extension(Y::'a,V1,V2,V3)))$
&
$(\forall X\ V1\ Y\ V2\ V3.\ equal(X::'a,Y) --> equal(extension(V1::'a,X,V2,V3),extension(V1::'a,Y,V2,V3)))$
&
$(\forall X\ V1\ V2\ Y\ V3.\ equal(X::'a,Y) --> equal(extension(V1::'a,V2,X,V3),extension(V1::'a,V2,Y,V3)))$
&
$(\forall X\ V1\ V2\ V3\ Y.\ equal(X::'a,Y) --> equal(extension(V1::'a,V2,V3,X),extension(V1::'a,V2,V3,Y)))$
&
$(\forall X\ Y\ V1\ V2\ V3\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(X::'a,V1,V2,V3,V4,V5),continuous(Y::'a,$
&
$(\forall X\ V1\ Y\ V2\ V3\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,X,V2,V3,V4,V5),continuous(V1::'$
&
$(\forall X\ V1\ V2\ Y\ V3\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,V2,X,V3,V4,V5),continuous(V1::'$
&
$(\forall X\ V1\ V2\ V3\ Y\ V4\ V5.\ equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,X,V4,V5),continuous(V1::'$
&

(∀ 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 *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 &
(~*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* ≡
(∀ Y X. *equidistant*(X::'a,Y,Y,X)) &
(∀ X Y Z V V2 W. *equidistant*(X::'a,Y,Z,V) & *equidistant*(X::'a,Y,V2,W) −−> *equidistant*(Z::'a,V,V2,W)) &
(∀ Z X Y. *equidistant*(X::'a,Y,Z,Z) −−> *equal*(X::'a,Y)) &
(∀ 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)) &
(∀ 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. *between*(X::'a,Y,X) −−> *equal*(X::'a,Y)) &
(∀ U V W X Y. *between*(U::'a,V,W) & *between*(Y::'a,X,W) −−> *between*(V::'a,*inner-pasch*(U::'a,V,W,X
&
(∀ V W X Y U. *between*(U::'a,V,W) & *between*(Y::'a,X,W) −−> *between*(X::'a,*inner-pasch*(U::'a,V,W,X,
&
(~*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)) &
(∀ 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) &

$between(U::'a,V,X)$ & $between(V::'a,W,X)$ −−> $between(V1::'a,continuous(U::'a,V,V1,W,X,X1),X1))$
&
$(\forall\ U\ V\ V1\ W\ X\ X1.\ equidistant(U::'a,V,U,V1)$ & $equidistant(U::'a,X,U,X1)$ &
$between(U::'a,V,X)$ & $between(V::'a,W,X)$ −−> $equidistant(U::'a,W,U,continuous(U::'a,V,V1,W,X,X1))$
&
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $between(X::'a,W,Z)$ −−> $between(Y::'a,W,Z))$
&
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $between(W::'a,X,Z)$ −−> $between(W::'a,Y,Z))$
&
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $between(W::'a,Z,X)$ −−> $between(W::'a,Z,Y))$
&
$(\forall\ X\ Y\ V\ W\ Z.\ equal(X::'a,Y)$ & $equidistant(X::'a,V,W,Z)$ −−> $equidistant(Y::'a,V,W,Z))$ &
$(\forall\ X\ V\ Y\ W\ Z.\ equal(X::'a,Y)$ & $equidistant(V::'a,X,W,Z)$ −−> $equidistant(V::'a,Y,W,Z))$ &
$(\forall\ X\ V\ W\ Y\ Z.\ equal(X::'a,Y)$ & $equidistant(V::'a,W,X,Z)$ −−> $equidistant(V::'a,W,Y,Z))$ &
$(\forall\ X\ V\ W\ Z\ Y.\ equal(X::'a,Y)$ & $equidistant(V::'a,W,Z,X)$ −−> $equidistant(V::'a,W,Z,Y))$ &
$(\forall\ X\ Y\ V1\ V2\ V3\ V4.\ equal(X::'a,Y)$ −−> $equal(inner\text{-}pasch(X::'a,V1,V2,V3,V4),inner\text{-}pasch(Y::'a,V1,V$
&
$(\forall\ X\ V1\ Y\ V2\ V3\ V4.\ equal(X::'a,Y)$ −−> $equal(inner\text{-}pasch(V1::'a,X,V2,V3,V4),inner\text{-}pasch(V1::'a,Y,V$
&
$(\forall\ X\ V1\ V2\ Y\ V3\ V4.\ equal(X::'a,Y)$ −−> $equal(inner\text{-}pasch(V1::'a,V2,X,V3,V4),inner\text{-}pasch(V1::'a,V2,$
&
$(\forall\ X\ V1\ V2\ V3\ Y\ V4.\ equal(X::'a,Y)$ −−> $equal(inner\text{-}pasch(V1::'a,V2,V3,X,V4),inner\text{-}pasch(V1::'a,V2,$
&
$(\forall\ X\ V1\ V2\ V3\ V4\ Y.\ equal(X::'a,Y)$ −−> $equal(inner\text{-}pasch(V1::'a,V2,V3,V4,X),inner\text{-}pasch(V1::'a,V2,$
&
$(\forall\ A\ B\ C\ D\ E\ F'.\ equal(A::'a,B)$ −−> $equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F')))$
&
$(\forall\ G\ I'\ H\ J\ K'\ L.\ equal(G::'a,H)$ −−> $equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L)))$
&
$(\forall\ M\ O'\ P\ N\ Q\ R.\ equal(M::'a,N)$ −−> $equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R)))$
&
$(\forall\ S'\ U\ V\ W\ T'\ X.\ equal(S'::'a,T')$ −−> $equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X)))$
&
$(\forall\ Y\ A1\ B1\ C1\ D1\ Z.\ equal(Y::'a,Z)$ −−> $equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z)$
&
$(\forall\ E1\ F1\ G1\ H1\ I1\ J1.\ equal(E1::'a,F1)$ −−> $equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1,$
&
$(\forall\ K1\ M1\ L1\ N1\ O1\ P1.\ equal(K1::'a,L1)$ −−> $equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1,$
&
$(\forall\ Q1\ S1\ T1\ R1\ U1\ V1.\ equal(Q1::'a,R1)$ −−> $equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U$
&
$(\forall\ W1\ Y1\ Z1\ A2\ X1\ B2.\ equal(W1::'a,X1)$ −−> $equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2$
&
$(\forall\ C2\ E2\ F2\ G2\ H2\ D2.\ equal(C2::'a,D2)$ −−> $equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H$
&

(∀ 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*))
&
(∀ *V W U X*. *trapezoid*(*U*::'*a*,*V*,*W*,*X*) −−> *parallel*(*V*::'*a*,*W*,*U*,*X*)) &
(∀ *U V X Y*. *parallel*(*U*::'*a*,*V*,*X*,*Y*) −−> *eq*(*X*::'*a*,*V*,*U*,*V*,*X*,*Y*)) &
(*trapezoid*(*a*::'*a*,*b*,*c*,*d*)) &
(~*eq*(*a*::'*a*,*c*,*b*,*c*,*a*,*d*)) −−> *False*
by *meson*

abbreviation *GRP003-0-ax equal multiply INVERSE identity product* ≡
(∀ *X*. *product*(*identity*::'*a*,*X*,*X*)) &
(∀ *X*. *product*(*X*::'*a*,*identity*,*X*)) &

$(\forall X.\ product(INVERSE(X),X,identity))$ &
$(\forall X.\ product(X::'a,INVERSE(X),identity))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W) \longrightarrow equal(Z::'a,W))$
&
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
$\longrightarrow product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
$\longrightarrow product(U::'a,Z,W))$

abbreviation *GRP003-0-eq product multiply INVERSE equal* $\equiv$
$(\forall X\ Y.\ equal(X::'a,Y) \longrightarrow equal(INVERSE(X),INVERSE(Y)))$ &
$(\forall X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall X\ W\ Y.\ equal(X::'a,Y) \longrightarrow equal(multiply(W::'a,X),multiply(W::'a,Y)))$
&
$(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z) \longrightarrow product(Y::'a,W,Z))$
&
$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z) \longrightarrow product(W::'a,Y,Z))$
&
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X) \longrightarrow product(W::'a,Z,Y))$

lemma *GRP001-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall X.\ product(X::'a,X,identity))$ &
$(product(a::'a,b,c))$ &
$(\sim product(b::'a,a,c)) \longrightarrow False$
oops

lemma *GRP008-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A\ B.\ equal(A::'a,B) \longrightarrow equal(h(A),h(B)))$ &
$(\forall C\ D.\ equal(C::'a,D) \longrightarrow equal(j(C),j(D)))$ &
$(\forall A\ B.\ equal(A::'a,B)$ & $q(A) \longrightarrow q(B))$ &
$(\forall B\ A\ C.\ q(A)$ & $product(A::'a,B,C) \longrightarrow product(B::'a,A,C))$ &
$(\forall A.\ product(j(A),A,h(A)) \mid product(A::'a,j(A),h(A)) \mid q(A))$ &
$(\forall A.\ product(j(A),A,h(A))$ & $product(A::'a,j(A),h(A)) \longrightarrow q(A))$ &
$(\sim q(identity)) \longrightarrow False$
by *meson*

lemma *GRP013-1*:
EQU001-0-ax equal &

GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A.\ product(A::'a,A,identity))$ &
$(product(a::'a,b,c))$ &
$(product(INVERSE(a),INVERSE(b),d))$ &
$(\forall A\ C\ B.\ product(INVERSE(A),INVERSE(B),C) \longrightarrow product(A::'a,C,B))$ &
$(\sim product(c::'a,d,identity)) \longrightarrow False$
oops

lemma *GRP037-3*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A\ B\ C.\ subgroup\text{-}member(A)$ & $subgroup\text{-}member(B)$ & $product(A::'a,INVERSE(B),C)$
$\longrightarrow subgroup\text{-}member(C))$ &
$(\forall A\ B.\ equal(A::'a,B)$ & $subgroup\text{-}member(A) \longrightarrow subgroup\text{-}member(B))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(Gidentity::'a,A,A))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(A::'a,Gidentity,A))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(A::'a,Ginverse(A),Gidentity))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow product(Ginverse(A),A,Gidentity))$ &
$(\forall A.\ subgroup\text{-}member(A) \longrightarrow subgroup\text{-}member(Ginverse(A)))$ &
$(\forall A\ B.\ equal(A::'a,B) \longrightarrow equal(Ginverse(A),Ginverse(B)))$ &
$(\forall A\ C\ D\ B.\ product(A::'a,B,C)$ & $product(A::'a,D,C) \longrightarrow equal(D::'a,B))$ &
$(\forall B\ C\ D\ A.\ product(A::'a,B,C)$ & $product(D::'a,B,C) \longrightarrow equal(D::'a,A))$ &
$(subgroup\text{-}member(a))$ &
$(subgroup\text{-}member(Gidentity))$ &
$(\sim equal(INVERSE(a),Ginverse(a))) \longrightarrow False$
by *meson*

lemma *GRP031-2*:
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W) \longrightarrow equal(Z::'a,W))$
&
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
$\longrightarrow product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
$\longrightarrow product(U::'a,Z,W))$ &
$(\forall A.\ product(A::'a,INVERSE(A),identity))$ &
$(\forall A.\ product(A::'a,identity,A))$ &
$(\forall A.\ \sim product(A::'a,a,identity)) \longrightarrow False$
by *meson*

lemma *GRP034-4*:
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X.\ product(identity::'a,X,X))$ &
$(\forall X.\ product(X::'a,identity,X))$ &

$(\forall X.\ product(X::'a,INVERSE(X),identity))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
$--> product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
$--> product(U::'a,Z,W))$ &
$(\forall B\ A\ C.\ subgroup\text{-}member(A)$ & $subgroup\text{-}member(B)$ & $product(B::'a,INVERSE(A),C)$
$--> subgroup\text{-}member(C))$ &
$(subgroup\text{-}member(a))$ &
$(\sim subgroup\text{-}member(INVERSE(a))) --> False$
by *meson*

lemma *GRP047-2*:
$(\forall X.\ product(identity::'a,X,X))$ &
$(\forall X.\ product(INVERSE(X),X,identity))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W) --> equal(Z::'a,W))$
&
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
$--> product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
$--> product(U::'a,Z,W))$ &
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X) --> product(W::'a,Z,Y))$
&
$(equal(a::'a,b))$ &
$(\sim equal(multiply(c::'a,a),multiply(c::'a,b))) --> False$
by *meson*

lemma *GRP130-1-002*:
$(group\text{-}element(e\text{-}1))$ &
$(group\text{-}element(e\text{-}2))$ &
$(\sim equal(e\text{-}1::'a,e\text{-}2))$ &
$(\sim equal(e\text{-}2::'a,e\text{-}1))$ &
$(\forall X\ Y.\ group\text{-}element(X)$ & $group\text{-}element(Y) --> product(X::'a,Y,e\text{-}1)$ |
$product(X::'a,Y,e\text{-}2))$ &
$(\forall X\ Y\ W\ Z.\ product(X::'a,Y,W)$ & $product(X::'a,Y,Z) --> equal(W::'a,Z))$
&
$(\forall X\ Y\ W\ Z.\ product(X::'a,W,Y)$ & $product(X::'a,Z,Y) --> equal(W::'a,Z))$
&
$(\forall Y\ X\ W\ Z.\ product(W::'a,Y,X)$ & $product(Z::'a,Y,X) --> equal(W::'a,Z))$
&
$(\forall Z1\ Z2\ Y\ X.\ product(X::'a,Y,Z1)$ & $product(X::'a,Z1,Z2) --> product(Z2::'a,Y,X))$
$--> False$
oops

abbreviation *GRP004-0-ax INVERSE identity multiply equal* $\equiv$
$(\forall X.\ equal(multiply(identity::'a,X),X))$ &
$(\forall X.\ equal(multiply(INVERSE(X),X),identity))$ &

(∀ 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-equal(*V4*::'*a*,*V5*)) &
(∀ *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*)) &
(∀ *X Y*. *quotient*(*X*::'*a*,*Y*,*Divide*(*X*::'*a*,*Y*))) &
(∀ *X Y Z W*. *quotient*(*X*::'*a*,*Y*,*Z*) & *quotient*(*X*::'*a*,*Y*,*W*) −−> *equal*(*Z*::'*a*,*W*))
&
(∀ *X Y W Z*. *equal*(*X*::'*a*,*Y*) & *quotient*(*X*::'*a*,*W*,*Z*) −−> *quotient*(*Y*::'*a*,*W*,*Z*))
&

$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ quotient(W::'a,X,Z) \longrightarrow quotient(W::'a,Y,Z))$
&
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ quotient(W::'a,Z,X) \longrightarrow quotient(W::'a,Z,Y))$
&
$(\forall X\ Z\ Y.\ equal(X::'a,Y)\ \&\ mless\text{-}equal(Z::'a,X) \longrightarrow mless\text{-}equal(Z::'a,Y))\ \&$
$(\forall X\ Y\ Z.\ equal(X::'a,Y)\ \&\ mless\text{-}equal(X::'a,Z) \longrightarrow mless\text{-}equal(Y::'a,Z))\ \&$
$(\forall X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(Divide(X::'a,W),Divide(Y::'a,W)))\ \&$
$(\forall X\ W\ Y.\ equal(X::'a,Y) \longrightarrow equal(Divide(W::'a,X),Divide(W::'a,Y)))\ \&$
$(\forall X.\ quotient(X::'a,identity,Zero))\ \&$
$(\forall X.\ quotient(Zero::'a,X,Zero))\ \&$
$(\forall X.\ quotient(X::'a,X,Zero))\ \&$
$(\forall X.\ quotient(X::'a,Zero,X))\ \&$
$(\forall Y\ X\ Z.\ mless\text{-}equal(X::'a,Y)\ \&\ mless\text{-}equal(Y::'a,Z) \longrightarrow mless\text{-}equal(X::'a,Z))$
&
$(\forall W1\ X\ Z\ W2\ Y.\ quotient(X::'a,Y,W1)\ \&\ mless\text{-}equal(W1::'a,Z)\ \&\ quotient(X::'a,Z,W2)$
$\longrightarrow mless\text{-}equal(W2::'a,Y))\ \&$
$(mless\text{-}equal(x::'a,y))\ \&$
$(quotient(z::'a,y,zQy))\ \&$
$(quotient(z::'a,x,zQx))\ \&$
$(\sim mless\text{-}equal(zQy::'a,zQx)) \longrightarrow False$
oops

lemma *HEN008-4*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
$(\forall X.\ equal(Divide(X::'a,identity),Zero))\ \&$
$(\forall X.\ equal(Divide(Zero::'a,X),Zero))\ \&$
$(\forall X.\ equal(Divide(X::'a,X),Zero))\ \&$
$(equal(Divide(a::'a,Zero),a))\ \&$
$(\forall Y\ X\ Z.\ mless\text{-}equal(X::'a,Y)\ \&\ mless\text{-}equal(Y::'a,Z) \longrightarrow mless\text{-}equal(X::'a,Z))$
&
$(\forall X\ Z\ Y.\ mless\text{-}equal(Divide(X::'a,Y),Z) \longrightarrow mless\text{-}equal(Divide(X::'a,Z),Y))$
&
$(\forall Y\ Z\ X.\ mless\text{-}equal(X::'a,Y) \longrightarrow mless\text{-}equal(Divide(Z::'a,Y),Divide(Z::'a,X)))$
&
$(mless\text{-}equal(a::'a,b))\ \&$
$(\sim mless\text{-}equal(Divide(a::'a,c),Divide(b::'a,c))) \longrightarrow False$
oops

lemma *HEN009-5*:
EQU001-0-ax equal &
$(\forall Y\ X.\ equal(Divide(Divide(X::'a,Y),X),Zero))\ \&$
$(\forall X\ Y\ Z.\ equal(Divide(Divide(Divide(X::'a,Z),Divide(Y::'a,Z)),Divide(Divide(X::'a,Y),Z)),Zero))$
&
$(\forall X.\ equal(Divide(Zero::'a,X),Zero))\ \&$
$(\forall X\ Y.\ equal(Divide(X::'a,Y),Zero)\ \&\ equal(Divide(Y::'a,X),Zero) \longrightarrow equal(X::'a,Y))$

&
(∀ X. equal(Divide(X::'a,identity),Zero)) &
(∀ 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))) &
(∀ Y X Z. equal(Divide(X::'a,Y),Zero) & equal(Divide(Y::'a,Z),Zero) −−>
equal(Divide(X::'a,Z),Zero)) &
(∀ X Z Y. equal(Divide(Divide(X::'a,Y),Z),Zero) −−> equal(Divide(Divide(X::'a,Z),Y),Zero))
&
(∀ Y Z X. equal(Divide(X::'a,Y),Zero) −−> equal(Divide(Divide(Z::'a,Y),Divide(Z::'a,X)),Zero))
&
(~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)) &
(~equal(b::'a,d)) −−> 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 &
(~mless-equal(a::'a,a)) −−> False
oops

lemma LCL010-1:
(∀ X Y. is-a-theorem(equivalent(X::'a,Y)) & is-a-theorem(X) −−> is-a-theorem(Y))
&
(∀ X Z Y. is-a-theorem(equivalent(equivalent(X::'a,Y),equivalent(equivalent(X::'a,Z),equivalent(Z::'a,Y)))))
&
(~is-a-theorem(equivalent(equivalent(a::'a,b),equivalent(equivalent(c::'a,b),equivalent(a::'a,c)))))
−−> False
by meson

lemma LCL077-2:
(∀ 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,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*:
$(\forall\, X\ Y.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(X::'a,Y))\ \&\ \mathit{is\text{-}a\text{-}theorem}(X)\ \text{--}\!>\ \mathit{is\text{-}a\text{-}theorem}(Y))$
&
$(\forall\, Y\ Z\ U\ X.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(\mathit{implies}(\mathit{implies}(X::'a,Y),Z),\mathit{implies}(\mathit{implies}(Z::'a,X),\mathit{implies}(U::'a,X)))))$
&
$(\sim \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(a::'a,\mathit{implies}(b::'a,a))))\ \text{--}\!>\ \mathit{False}$
by *meson*

lemma *LCL111-1*:
$(\forall\, X\ Y.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(X,Y))\ \&\ \mathit{is\text{-}a\text{-}theorem}(X)\ \text{--}\!>\ \mathit{is\text{-}a\text{-}theorem}(Y))$
&
$(\forall\, Y\ X.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(X,\mathit{implies}(Y,X))))\ \&$
$(\forall\, Y\ X\ Z.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(\mathit{implies}(X,Y),\mathit{implies}(\mathit{implies}(Y,Z),\mathit{implies}(X,Z)))))$
&
$(\forall\, Y\ X.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(\mathit{implies}(\mathit{implies}(X,Y),Y),\mathit{implies}(\mathit{implies}(Y,X),X))))$
&
$(\forall\, Y\ X.\ \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(\mathit{implies}(\mathit{not}(X),\mathit{not}(Y)),\mathit{implies}(Y,X))))\ \&$
$(\sim \mathit{is\text{-}a\text{-}theorem}(\mathit{implies}(\mathit{implies}(a,b),\mathit{implies}(\mathit{implies}(c,a),\mathit{implies}(c,b)))))\ \text{--}\!>\ \mathit{False}$
by *meson*

lemma *LCL143-1*:
$(\forall\, X.\ \mathit{equal}(X,X))\ \&$
$(\forall\, Y\ X.\ \mathit{equal}(X,Y)\ \text{--}\!>\ \mathit{equal}(Y,X))\ \&$
$(\forall\, Y\ X\ Z.\ \mathit{equal}(X,Y)\ \&\ \mathit{equal}(Y,Z)\ \text{--}\!>\ \mathit{equal}(X,Z))\ \&$
$(\forall\, X.\ \mathit{equal}(\mathit{implies}(\mathit{true},X),X))\ \&$
$(\forall\, Y\ X\ Z.\ \mathit{equal}(\mathit{implies}(\mathit{implies}(X,Y),\mathit{implies}(\mathit{implies}(Y,Z),\mathit{implies}(X,Z))),\mathit{true}))$
&
$(\forall\, Y\ X.\ \mathit{equal}(\mathit{implies}(\mathit{implies}(X,Y),Y),\mathit{implies}(\mathit{implies}(Y,X),X)))\ \&$
$(\forall\, Y\ X.\ \mathit{equal}(\mathit{implies}(\mathit{implies}(\mathit{not}(X),\mathit{not}(Y)),\mathit{implies}(Y,X)),\mathit{true}))\ \&$
$(\forall\, A\ B\ C.\ \mathit{equal}(A,B)\ \text{--}\!>\ \mathit{equal}(\mathit{implies}(A,C),\mathit{implies}(B,C)))\ \&$
$(\forall\, D\ F'\ E.\ \mathit{equal}(D,E)\ \text{--}\!>\ \mathit{equal}(\mathit{implies}(F',D),\mathit{implies}(F',E)))\ \&$
$(\forall\, G\ H.\ \mathit{equal}(G,H)\ \text{--}\!>\ \mathit{equal}(\mathit{not}(G),\mathit{not}(H)))\ \&$
$(\forall\, X\ Y.\ \mathit{equal}(\mathit{big\text{-}V}(X,Y),\mathit{implies}(\mathit{implies}(X,Y),Y)))\ \&$
$(\forall\, X\ Y.\ \mathit{equal}(\mathit{big\text{-}hat}(X,Y),\mathit{not}(\mathit{big\text{-}V}(\mathit{not}(X),\mathit{not}(Y)))))\ \&$
$(\forall\, X\ Y.\ \mathit{ordered}(X,Y)\ \text{--}\!>\ \mathit{equal}(\mathit{implies}(X,Y),\mathit{true}))\ \&$
$(\forall\, X\ Y.\ \mathit{equal}(\mathit{implies}(X,Y),\mathit{true})\ \text{--}\!>\ \mathit{ordered}(X,Y))\ \&$
$(\forall\, A\ B\ C.\ \mathit{equal}(A,B)\ \text{--}\!>\ \mathit{equal}(\mathit{big\text{-}V}(A,C),\mathit{big\text{-}V}(B,C)))\ \&$
$(\forall\, D\ F'\ E.\ \mathit{equal}(D,E)\ \text{--}\!>\ \mathit{equal}(\mathit{big\text{-}V}(F',D),\mathit{big\text{-}V}(F',E)))\ \&$
$(\forall\, G\ H\ I'.\ \mathit{equal}(G,H)\ \text{--}\!>\ \mathit{equal}(\mathit{big\text{-}hat}(G,I'),\mathit{big\text{-}hat}(H,I')))\ \&$
$(\forall\, J\ L\ K'.\ \mathit{equal}(J,K')\ \text{--}\!>\ \mathit{equal}(\mathit{big\text{-}hat}(L,J),\mathit{big\text{-}hat}(L,K')))\ \&$
$(\forall\, M\ N\ O'.\ \mathit{equal}(M,N)\ \&\ \mathit{ordered}(M,O')\ \text{--}\!>\ \mathit{ordered}(N,O'))\ \&$
$(\forall\, P\ R\ Q.\ \mathit{equal}(P,Q)\ \&\ \mathit{ordered}(R,P)\ \text{--}\!>\ \mathit{ordered}(R,Q))\ \&$
$(\mathit{ordered}(x,y))\ \&$
$(\sim \mathit{ordered}(\mathit{implies}(z,x),\mathit{implies}(z,y)))\ \text{--}\!>\ \mathit{False}$
by *meson*

lemma *LCL182-1*:
$(\forall A.\ axiom(or(not(or(A,A)),A))) \ \&$
$(\forall B\ A.\ axiom(or(not(A),or(B,A)))) \ \&$
$(\forall B\ A.\ axiom(or(not(or(A,B)),or(B,A)))) \ \&$
$(\forall B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C))))) \ \&$
$(\forall A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B))))) \ \&$
$(\forall X.\ axiom(X) \longrightarrow theorem(X)) \ \&$
$(\forall X\ Y.\ axiom(or(not(Y),X)) \ \& \ theorem(Y) \longrightarrow theorem(X)) \ \&$
$(\forall X\ Y\ Z.\ axiom(or(not(X),Y)) \ \& \ theorem(or(not(Y),Z)) \longrightarrow theorem(or(not(X),Z)))$
$\&$
$(\sim theorem(or(not(or(not(p),q)),or(not(not(q)),not(p))))) \longrightarrow False$
by *meson*

lemma *LCL200-1*:
$(\forall A.\ axiom(or(not(or(A,A)),A))) \ \&$
$(\forall B\ A.\ axiom(or(not(A),or(B,A)))) \ \&$
$(\forall B\ A.\ axiom(or(not(or(A,B)),or(B,A)))) \ \&$
$(\forall B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C))))) \ \&$
$(\forall A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B))))) \ \&$
$(\forall X.\ axiom(X) \longrightarrow theorem(X)) \ \&$
$(\forall X\ Y.\ axiom(or(not(Y),X)) \ \& \ theorem(Y) \longrightarrow theorem(X)) \ \&$
$(\forall X\ Y\ Z.\ axiom(or(not(X),Y)) \ \& \ theorem(or(not(Y),Z)) \longrightarrow theorem(or(not(X),Z)))$
$\&$
$(\sim theorem(or(not(not(or(p,q))),not(q)))) \longrightarrow False$
by *meson*

lemma *LCL215-1*:
$(\forall A.\ axiom(or(not(or(A,A)),A))) \ \&$
$(\forall B\ A.\ axiom(or(not(A),or(B,A)))) \ \&$
$(\forall B\ A.\ axiom(or(not(or(A,B)),or(B,A)))) \ \&$
$(\forall B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C))))) \ \&$
$(\forall A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B))))) \ \&$
$(\forall X.\ axiom(X) \longrightarrow theorem(X)) \ \&$
$(\forall X\ Y.\ axiom(or(not(Y),X)) \ \& \ theorem(Y) \longrightarrow theorem(X)) \ \&$
$(\forall X\ Y\ Z.\ axiom(or(not(X),Y)) \ \& \ theorem(or(not(Y),Z)) \longrightarrow theorem(or(not(X),Z)))$
$\&$
$(\sim theorem(or(not(or(not(p),q)),or(not(or(p,q)),q)))) \longrightarrow False$
by *meson*

lemma *LCL230-2*:
$(q \longrightarrow p \mid r) \ \&$
$(\sim p) \ \&$
$(q) \ \&$
$(\sim r) \longrightarrow False$

by *meson*

lemma *LDA003-1*:
EQU001-0-ax equal &
(∀ *Y X Z*. *equal*(*f*(*X*::'*a*,*f*(*Y*::'*a*,*Z*)),*f*(*f*(*X*::'*a*,*Y*),*f*(*X*::'*a*,*Z*)))) &
(∀ *X Y*. *left*(*X*::'*a*,*f*(*X*::'*a*,*Y*))) &
(∀ *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*))) &
(∀ *A B C*. *equal*(*A*::'*a*,*B*) −−> *equal*(*f*(*A*::'*a*,*C*),*f*(*B*::'*a*,*C*))) &
(∀ *D F' E*. *equal*(*D*::'*a*,*E*) −−> *equal*(*f*(*F'*::'*a*,*D*),*f*(*F'*::'*a*,*E*))) &
(∀ *G H I'*. *equal*(*G*::'*a*,*H*) & *left*(*G*::'*a*,*I'*) −−> *left*(*H*::'*a*,*I'*)) &
(∀ *J L K'*. *equal*(*J*::'*a*,*K'*) & *left*(*L*::'*a*,*J*) −−> *left*(*L*::'*a*,*K'*)) &
(~*left*(*num3*::'*a*,*u*)) −−> *False*
oops

lemma *MSC002-1*:
(*at*(*something*::'*a*,*here*,*now*)) &
(∀ *Place Situation*. *hand-at*(*Place*::'*a*,*Situation*) −−> *hand-at*(*Place*::'*a*,*let-go*(*Situation*)))
&
(∀ *Place Another-place Situation*. *hand-at*(*Place*::'*a*,*Situation*) −−> *hand-at*(*Another-place*::'*a*,*go*(*Another-pl*
&
(∀ *Thing Situation*. ~*held*(*Thing*::'*a*,*let-go*(*Situation*))) &
(∀ *Situation Thing*. *at*(*Thing*::'*a*,*here*,*Situation*) −−> *red*(*Thing*)) &
(∀ *Thing Place Situation*. *at*(*Thing*::'*a*,*Place*,*Situation*) −−> *at*(*Thing*::'*a*,*Place*,*let-go*(*Situation*)))
&
(∀ *Thing Place Situation*. *at*(*Thing*::'*a*,*Place*,*Situation*) −−> *at*(*Thing*::'*a*,*Place*,*pick-up*(*Situation*)))
&
(∀ *Thing Place Situation*. *at*(*Thing*::'*a*,*Place*,*Situation*) −−> *grabbed*(*Thing*::'*a*,*pick-up*(*go*(*Place*::'*a*,*let-go*(*Si*
&
(∀ *Thing Situation*. *red*(*Thing*) & *put*(*Thing*::'*a*,*there*,*Situation*) −−> *answer*(*Situation*))
&
(∀ *Place Thing Another-place Situation*. *at*(*Thing*::'*a*,*Place*,*Situation*) & *grabbed*(*Thing*::'*a*,*Situation*)
−−> *put*(*Thing*::'*a*,*Another-place*,*go*(*Another-place*::'*a*,*Situation*))) &
(∀ *Thing Place Another-place Situation*. *at*(*Thing*::'*a*,*Place*,*Situation*) −−> *held*(*Thing*::'*a*,*Situation*)
| *at*(*Thing*::'*a*,*Place*,*go*(*Another-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-*
−−> *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-of*
−−> *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-*
−−> *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*))
&
(∀ *X Y*. *value*(*X*::'*a*,*truth*) & *value*(*Y*::'*a*,*falsity*) −−> *value*(*xor*(*X*::'*a*, *Y*),*truth*))
&
(∀ *X Y*. *value*(*X*::'*a*,*falsity*) & *value*(*Y*::'*a*,*truth*) −−> *value*(*xor*(*X*::'*a*, *Y*),*truth*))
&
(∀ *X Y*. *value*(*X*::'*a*,*falsity*) & *value*(*Y*::'*a*,*falsity*) −−> *value*(*xor*(*X*::'*a*, *Y*),*falsity*))
&
(∀ *Value*. ~*value*(*xor*(*xor*(*xor*(*xor*(*truth*::'*a*,*falsity*),*falsity*),*truth*),*falsity*), *Value*))
−−> *False*
by *meson*

lemma *MSC006-1*:
$(\forall\ Y\ X\ Z.\ p(X::'a,Y)\ \&\ p(Y::'a,Z)\ -->\ p(X::'a,Z))\ \&$
$(\forall\ Y\ X\ Z.\ q(X::'a,Y)\ \&\ q(Y::'a,Z)\ -->\ q(X::'a,Z))\ \&$
$(\forall\ Y\ X.\ q(X::'a,Y)\ -->\ q(Y::'a,X))\ \&$
$(\forall\ X\ Y.\ p(X::'a,Y)\ |\ q(X::'a,Y))\ \&$
$(\sim p(a::'a,b))\ \&$
$(\sim q(c::'a,d))\ -->\ False$
by *meson*

lemma *NUM001-1*:
$(\forall\ A.\ equal(A::'a,A))\ \&$
$(\forall\ B\ A\ C.\ equal(A::'a,B)\ \&\ equal(B::'a,C)\ -->\ equal(A::'a,C))\ \&$
$(\forall\ B\ A.\ equal(add(A::'a,B),add(B::'a,A)))\ \&$
$(\forall\ A\ B\ C.\ equal(add(A::'a,add(B::'a,C)),add(add(A::'a,B),C)))\ \&$
$(\forall\ B\ A.\ equal(subtract(add(A::'a,B),B),A))\ \&$
$(\forall\ A\ B.\ equal(A::'a,subtract(add(A::'a,B),B)))\ \&$
$(\forall\ A\ C\ B.\ equal(add(subtract(A::'a,B),C),subtract(add(A::'a,C),B)))\ \&$
$(\forall\ A\ C\ B.\ equal(subtract(add(A::'a,B),C),add(subtract(A::'a,C),B)))\ \&$
$(\forall\ A\ C\ B\ D.\ equal(A::'a,B)\ \&\ equal(C::'a,add(A::'a,D))\ -->\ equal(C::'a,add(B::'a,D)))$
$\&$
$(\forall\ A\ C\ D\ B.\ equal(A::'a,B)\ \&\ equal(C::'a,add(D::'a,A))\ -->\ equal(C::'a,add(D::'a,B)))$
$\&$
$(\forall\ A\ C\ B\ D.\ equal(A::'a,B)\ \&\ equal(C::'a,subtract(A::'a,D))\ -->\ equal(C::'a,subtract(B::'a,D)))$
$\&$
$(\forall\ A\ C\ D\ B.\ equal(A::'a,B)\ \&\ equal(C::'a,subtract(D::'a,A))\ -->\ equal(C::'a,subtract(D::'a,B)))$
$\&$
$(\sim equal(add(add(a::'a,b),c),add(a::'a,add(b::'a,c))))\ -->\ False$
by *meson*

abbreviation *NUM001-0-ax multiply successor num0 add equal* $\equiv$
$(\forall\ A.\ equal(add(A::'a,num0),A))\ \&$
$(\forall\ A\ B.\ equal(add(A::'a,successor(B)),successor(add(A::'a,B))))\ \&$
$(\forall\ A.\ equal(multiply(A::'a,num0),num0))\ \&$
$(\forall\ B\ A.\ equal(multiply(A::'a,successor(B)),add(multiply(A::'a,B),A)))\ \&$
$(\forall\ A\ B.\ equal(successor(A),successor(B))\ -->\ equal(A::'a,B))\ \&$
$(\forall\ A\ B.\ equal(A::'a,B)\ -->\ equal(successor(A),successor(B)))$

abbreviation *NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless*
$\equiv$
$(\forall\ A\ C\ B.\ mless(A::'a,B)\ \&\ mless(C::'a,A)\ -->\ mless(C::'a,B))\ \&$
$(\forall\ A\ B\ C.\ equal(add(successor(A),B),C)\ -->\ mless(B::'a,C))\ \&$
$(\forall\ A\ B.\ mless(A::'a,B)\ -->\ equal(add(successor(predecessor\text{-}of\text{-}1st\text{-}minus\text{-}2nd(B::'a,A)),A),B))$

abbreviation *NUM001-2-ax equal mless divides* $\equiv$
$(\forall\ A\ B.\ divides(A::'a,B)\ -->\ mless(A::'a,B)\ |\ equal(A::'a,B))\ \&$
$(\forall\ A\ B.\ mless(A::'a,B)\ -->\ divides(A::'a,B))\ \&$

$(\forall\ A\ B.\ equal(A::'a,B)\ --{>}\ divides(A::'a,B))$

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 &
$(mless(b::'a,c))$ &
$(\sim mless(b::'a,a))$ &
$(divides(c::'a,a))$ &
$(\forall A.\ \sim equal(successor(A),num0))\ --{>}\ False$
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 &
$(\forall B\ A.\ equal(add(A::'a,B),add(B::'a,A)))$ &
$(\forall B\ A\ C.\ equal(add(A::'a,B),add(C::'a,B))\ --{>}\ equal(A::'a,C))$ &
$(mless(a::'a,a))$ &
$(\forall A.\ \sim equal(successor(A),num0))\ --{>}\ False$
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 $\equiv$
$(\forall X\ U\ Y.\ subclass(X::'a,Y)\ \&\ member(U::'a,X)\ --{>}\ member(U::'a,Y))$ &
$(\forall X\ Y.\ member(not\text{-}subclass\text{-}element(X::'a,Y),X)\ |\ subclass(X::'a,Y))$ &
$(\forall X\ Y.\ member(not\text{-}subclass\text{-}element(X::'a,Y),Y)\ --{>}\ subclass(X::'a,Y))$ &
$(\forall X.\ subclass(X::'a,universal\text{-}class))$ &
$(\forall X\ Y.\ equal(X::'a,Y)\ --{>}\ subclass(X::'a,Y))$ &
$(\forall Y\ X.\ equal(X::'a,Y)\ --{>}\ subclass(Y::'a,X))$ &
$(\forall X\ Y.\ subclass(X::'a,Y)\ \&\ subclass(Y::'a,X)\ --{>}\ equal(X::'a,Y))$ &
$(\forall X\ U\ Y.\ member(U::'a,unordered\text{-}pair(X::'a,Y))\ --{>}\ equal(U::'a,X)\ |\ equal(U::'a,Y))$
&
$(\forall X\ Y.\ member(X::'a,universal\text{-}class)\ --{>}\ member(X::'a,unordered\text{-}pair(X::'a,Y)))$
&
$(\forall X\ Y.\ member(Y::'a,universal\text{-}class)\ --{>}\ member(Y::'a,unordered\text{-}pair(X::'a,Y)))$
&
$(\forall X\ Y.\ member(unordered\text{-}pair(X::'a,Y),universal\text{-}class))$ &
$(\forall X.\ equal(unordered\text{-}pair(X::'a,X),singleton(X)))$ &

$(\forall\, X\ Y.\ equal(unordered\text{-}pair(singleton(X),unordered\text{-}pair(X::'a,singleton(Y))),ordered\text{-}pair(X::'a,Y)))$
&
$(\forall\, V\ Y\ U\ X.\ member(ordered\text{-}pair(U::'a,V),cross\text{-}product(X::'a,Y)) --> mem-$
$ber(U::'a,X))$ &
$(\forall\, U\ X\ V\ Y.\ member(ordered\text{-}pair(U::'a,V),cross\text{-}product(X::'a,Y)) --> mem-$
$ber(V::'a,Y))$ &
$(\forall\, U\ V\ X\ Y.\ member(U::'a,X)$ & $member(V::'a,Y) --> member(ordered\text{-}pair(U::'a,V),cross\text{-}product(X::'a$
&
$(\forall\, X\ Y\ Z.\ member(Z::'a,cross\text{-}product(X::'a,Y)) --> equal(ordered\text{-}pair(first(Z),second(Z)),Z))$
&
$(subclass(element\text{-}relation::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),element\text{-}relation) --> member(X::'a,Y))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))$
& $member(X::'a,Y) --> member(ordered\text{-}pair(X::'a,Y),element\text{-}relation))$ &
$(\forall\, Y\ Z\ X.\ member(Z::'a,intersection(X::'a,Y)) --> member(Z::'a,X))$ &
$(\forall\, X\ Z\ Y.\ member(Z::'a,intersection(X::'a,Y)) --> member(Z::'a,Y))$ &
$(\forall\, Z\ X\ Y.\ member(Z::'a,X)$ & $member(Z::'a,Y) --> member(Z::'a,intersection(X::'a,Y)))$
&
$(\forall\, Z\ X.\ \sim(member(Z::'a,complement(X))$ & $member(Z::'a,X)))$ &
$(\forall\, Z\ X.\ member(Z::'a,universal\text{-}class) --> member(Z::'a,complement(X)) \mid$
$member(Z::'a,X))$ &
$(\forall\, X\ Y.\ equal(complement(intersection(complement(X),complement(Y))),union(X::'a,Y)))$
&
$(\forall\, X\ Y.\ equal(intersection(complement(intersection(X::'a,Y)),complement(intersection(complement(X),com$
&
$(\forall\, Xr\ X\ Y.\ equal(intersection(Xr::'a,cross\text{-}product(X::'a,Y)),restrct(Xr::'a,X,Y)))$
&
$(\forall\, Xr\ X\ Y.\ equal(intersection(cross\text{-}product(X::'a,Y),Xr),restrct(Xr::'a,X,Y)))$
&
$(\forall\, Z\ X.\ \sim(equal(restrct(X::'a,singleton(Z),universal\text{-}class),null\text{-}class)$ & $mem-$
$ber(Z::'a,domain\text{-}of(X))))$ &
$(\forall\, Z\ X.\ member(Z::'a,universal\text{-}class) --> equal(restrct(X::'a,singleton(Z),universal\text{-}class),null\text{-}class)$
$\mid member(Z::'a,domain\text{-}of(X)))$ &
$(\forall\, X.\ subclass(rot(X),cross\text{-}product(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),universal\text{-}class)))$
&
$(\forall\, V\ W\ U\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),rot(X)) --> mem-$
$ber(ordered\text{-}pair(ordered\text{-}pair(V::'a,W),U),X))$ &
$(\forall\, U\ V\ W\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(V::'a,W),U),X)$ & $member(ordered\text{-}pair(ordered\text{-}pair(U::'a$
$--> member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),rot(X)))$ &
$(\forall\, X.\ subclass(flip(X),cross\text{-}product(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),universal\text{-}class)))$
&
$(\forall\, V\ U\ W\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),flip(X)) --> mem-$
$ber(ordered\text{-}pair(ordered\text{-}pair(V::'a,U),W),X))$ &
$(\forall\, U\ V\ W\ X.\ member(ordered\text{-}pair(ordered\text{-}pair(V::'a,U),W),X)$ & $member(ordered\text{-}pair(ordered\text{-}pair(U::'a$
$--> member(ordered\text{-}pair(ordered\text{-}pair(U::'a,V),W),flip(X)))$ &
$(\forall\, Y.\ equal(domain\text{-}of(flip(cross\text{-}product(Y::'a,universal\text{-}class))),INVERSE(Y)))$
&

$(\forall\, Z.\ equal(domain\text{-}of(INVERSE(Z)),range\text{-}of(Z)))\ \&$
$(\forall\, Z\ X\ Y.\ equal(first(not\text{-}subclass\text{-}element(restrct(Z::'a,X,singleton(Y)),null\text{-}class)),domain(Z::'a,X,Y)))$
$\&$
$(\forall\, Z\ X\ Y.\ equal(second(not\text{-}subclass\text{-}element(restrct(Z::'a,singleton(X),Y),null\text{-}class)),rng(Z::'a,X,Y)))$
$\&$
$(\forall\, Xr\ X.\ equal(range\text{-}of(restrct(Xr::'a,X,universal\text{-}class)),image'(Xr::'a,X)))\ \&$
$(\forall\, X.\ equal(union(X::'a,singleton(X)),successor(X)))\ \&$
$(subclass(successor\text{-}relation::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
$\&$
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),successor\text{-}relation) \longrightarrow equal(successor(X),Y))$
$\&$
$(\forall\, X\ Y.\ equal(successor(X),Y)\ \&\ member(ordered\text{-}pair(X::'a,Y),cross\text{-}product(universal\text{-}class::'a,universal\text{-}$
$\longrightarrow member(ordered\text{-}pair(X::'a,Y),successor\text{-}relation))\ \&$
$(\forall\, X.\ inductive(X) \longrightarrow member(null\text{-}class::'a,X))\ \&$
$(\forall\, X.\ inductive(X) \longrightarrow subclass(image'(successor\text{-}relation::'a,X),X))\ \&$
$(\forall\, X.\ member(null\text{-}class::'a,X)\ \&\ subclass(image'(successor\text{-}relation::'a,X),X)$
$\longrightarrow inductive(X))\ \&$
$(inductive(omega))\ \&$
$(\forall\, Y.\ inductive(Y) \longrightarrow subclass(omega::'a,Y))\ \&$
$(member(omega::'a,universal\text{-}class))\ \&$
$(\forall\, X.\ equal(domain\text{-}of(restrct(element\text{-}relation::'a,universal\text{-}class,X)),sum\text{-}class(X)))$
$\&$
$(\forall\, X.\ member(X::'a,universal\text{-}class) \longrightarrow member(sum\text{-}class(X),universal\text{-}class))$
$\&$
$(\forall\, X.\ equal(complement(image'(element\text{-}relation::'a,complement(X))),powerClass(X)))$
$\&$
$(\forall\, U.\ member(U::'a,universal\text{-}class) \longrightarrow member(powerClass(U),universal\text{-}class))$
$\&$
$(\forall\, Yr\ Xr.\ subclass(compos(Yr::'a,Xr),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
$\&$
$(\forall\, Z\ Yr\ Xr\ Y.\ member(ordered\text{-}pair(Y::'a,Z),compos(Yr::'a,Xr)) \longrightarrow mem-$
$ber(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y)))))\ \&$
$(\forall\, Y\ Z\ Yr\ Xr.\ member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y))))\ \&\ mem-$
$ber(ordered\text{-}pair(Y::'a,Z),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)) \longrightarrow mem-$
$ber(ordered\text{-}pair(Y::'a,Z),compos(Yr::'a,Xr)))\ \&$
$(\forall\, X.\ single\text{-}valued\text{-}class(X) \longrightarrow subclass(compos(X::'a,INVERSE(X)),identity\text{-}relation))$
$\&$
$(\forall\, X.\ subclass(compos(X::'a,INVERSE(X)),identity\text{-}relation) \longrightarrow single\text{-}valued\text{-}class(X))$
$\&$
$(\forall\, Xf.\ function(Xf) \longrightarrow subclass(Xf::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
$\&$
$(\forall\, Xf.\ function(Xf) \longrightarrow subclass(compos(Xf::'a,INVERSE(Xf)),identity\text{-}relation))$
$\&$
$(\forall\, Xf.\ subclass(Xf::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))\ \&\ sub-$
$class(compos(Xf::'a,INVERSE(Xf)),identity\text{-}relation) \longrightarrow function(Xf))\ \&$
$(\forall\, Xf\ X.\ function(Xf)\ \&\ member(X::'a,universal\text{-}class) \longrightarrow member(image'(Xf::'a,X),universal\text{-}class))$
$\&$
$(\forall\, X.\ equal(X::'a,null\text{-}class) \mid member(regular(X),X))\ \&$
$(\forall\, X.\ equal(X::'a,null\text{-}class) \mid equal(intersection(X::'a,regular(X)),null\text{-}class))\ \&$

$(\forall\ Xf\ Y.\ equal(sum\text{-}class(image'(Xf::'a,singleton(Y))),apply(Xf::'a,Y)))\ \&$
$(function(choice))\ \&$
$(\forall\ Y.\ member(Y::'a,universal\text{-}class) \dashrightarrow equal(Y::'a,null\text{-}class) \mid member(apply(choice::'a,Y),Y))$
$\&$
$(\forall\ Xf.\ one\text{-}to\text{-}one(Xf) \dashrightarrow function(Xf))\ \&$
$(\forall\ Xf.\ one\text{-}to\text{-}one(Xf) \dashrightarrow function(INVERSE(Xf)))\ \&$
$(\forall\ Xf.\ function(INVERSE(Xf))\ \&\ function(Xf) \dashrightarrow one\text{-}to\text{-}one(Xf))\ \&$
$(equal(intersection(cross\text{-}product(universal\text{-}class::'a,universal\text{-}class),intersection(cross\text{-}product(universal\text{-}clas$
$\&$
$(equal(intersection(INVERSE(subset\text{-}relation),subset\text{-}relation),identity\text{-}relation))$
$\&$
$(\forall\ Xr.\ equal(complement(domain\text{-}of(intersection(Xr::'a,identity\text{-}relation))),diagonalise(Xr)))$
$\&$
$(\forall\ X.\ equal(intersection(domain\text{-}of(X),diagonalise(compos(INVERSE(element\text{-}relation),X))),cantor(X)))$
$\&$
$(\forall\ Xf.\ operation(Xf) \dashrightarrow function(Xf))\ \&$
$(\forall\ Xf.\ operation(Xf) \dashrightarrow equal(cross\text{-}product(domain\text{-}of(domain\text{-}of(Xf)),domain\text{-}of(domain\text{-}of(Xf))),dom$
$\&$
$(\forall\ Xf.\ operation(Xf) \dashrightarrow subclass(range\text{-}of(Xf),domain\text{-}of(domain\text{-}of(Xf))))$
$\&$
$(\forall\ Xf.\ function(Xf)\ \&\ equal(cross\text{-}product(domain\text{-}of(domain\text{-}of(Xf)),domain\text{-}of(domain\text{-}of(Xf))),domain\text{-}$
$\&\ subclass(range\text{-}of(Xf),domain\text{-}of(domain\text{-}of(Xf))) \dashrightarrow operation(Xf))\ \&$
$(\forall\ Xf1\ Xf2\ Xh.\ compatible(Xh::'a,Xf1,Xf2) \dashrightarrow function(Xh))\ \&$
$(\forall\ Xf2\ Xf1\ Xh.\ compatible(Xh::'a,Xf1,Xf2) \dashrightarrow equal(domain\text{-}of(domain\text{-}of(Xf1)),domain\text{-}of(Xh)))$
$\&$
$(\forall\ Xf1\ Xh\ Xf2.\ compatible(Xh::'a,Xf1,Xf2) \dashrightarrow subclass(range\text{-}of(Xh),domain\text{-}of(domain\text{-}of(Xf2))))$
$\&$
$(\forall\ Xh\ Xh1\ Xf1\ Xf2.\ function(Xh)\ \&\ equal(domain\text{-}of(domain\text{-}of(Xf1)),domain\text{-}of(Xh))$
$\&\ subclass(range\text{-}of(Xh),domain\text{-}of(domain\text{-}of(Xf2))) \dashrightarrow compatible(Xh1::'a,Xf1,Xf2))$
$\&$
$(\forall\ Xh\ Xf2\ Xf1.\ homomorphism(Xh::'a,Xf1,Xf2) \dashrightarrow operation(Xf1))\ \&$
$(\forall\ Xh\ Xf1\ Xf2.\ homomorphism(Xh::'a,Xf1,Xf2) \dashrightarrow operation(Xf2))\ \&$
$(\forall\ Xh\ Xf1\ Xf2.\ homomorphism(Xh::'a,Xf1,Xf2) \dashrightarrow compatible(Xh::'a,Xf1,Xf2))$
$\&$
$(\forall\ Xf2\ Xh\ Xf1\ X\ Y.\ homomorphism(Xh::'a,Xf1,Xf2)\ \&\ member(ordered\text{-}pair(X::'a,Y),domain\text{-}of(Xf1))$
$\dashrightarrow equal(apply(Xf2::'a,ordered\text{-}pair(apply(Xh::'a,X),apply(Xh::'a,Y))),apply(Xh::'a,apply(Xf1::'a,ordered-$
$\&$
$(\forall\ Xh\ Xf1\ Xf2.\ operation(Xf1)\ \&\ operation(Xf2)\ \&\ compatible(Xh::'a,Xf1,Xf2)$
$\dashrightarrow member(ordered\text{-}pair(not\text{-}homomorphism1(Xh::'a,Xf1,Xf2),not\text{-}homomorphism2(Xh::'a,Xf1,Xf2)),dom$
$\mid homomorphism(Xh::'a,Xf1,Xf2))\ \&$
$(\forall\ Xh\ Xf1\ Xf2.\ operation(Xf1)\ \&\ operation(Xf2)\ \&\ compatible(Xh::'a,Xf1,Xf2)$
$\&\ equal(apply(Xf2::'a,ordered\text{-}pair(apply(Xh::'a,not\text{-}homomorphism1(Xh::'a,Xf1,Xf2)),apply(Xh::'a,not\text{-}hom$
$\dashrightarrow homomorphism(Xh::'a,Xf1,Xf2))$

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*

$$
\begin{array}{l}
\quad \mathit{first\ domain\text{-}of\ domain\ difference\ diagonalise\ cross\text{-}product\ compos} \\
\quad \mathit{complement\ cantor\ apply\ equal} \equiv \\
\ (\forall\, D\ E\ F'.\ \mathit{equal}(D::'a,E) \longrightarrow \mathit{equal}(\mathit{apply}(D::'a,F'),\mathit{apply}(E::'a,F'))) \ \&\ \\
\ (\forall\, G\ I'\ H.\ \mathit{equal}(G::'a,H) \longrightarrow \mathit{equal}(\mathit{apply}(I'::'a,G),\mathit{apply}(I'::'a,H))) \ \&\ \\
\ (\forall\, J\ K'.\ \mathit{equal}(J::'a,K') \longrightarrow \mathit{equal}(\mathit{cantor}(J),\mathit{cantor}(K'))) \ \&\ \\
\ (\forall\, L\ M.\ \mathit{equal}(L::'a,M) \longrightarrow \mathit{equal}(\mathit{complement}(L),\mathit{complement}(M))) \ \&\ \\
\ (\forall\, N\ O'\ P.\ \mathit{equal}(N::'a,O') \longrightarrow \mathit{equal}(\mathit{compos}(N::'a,P),\mathit{compos}(O'::'a,P))) \ \&\ \\
\ (\forall\, Q\ S'\ R.\ \mathit{equal}(Q::'a,R) \longrightarrow \mathit{equal}(\mathit{compos}(S'::'a,Q),\mathit{compos}(S'::'a,R))) \ \&\ \\
(\forall\, T'\ U\ V.\ \mathit{equal}(T'::'a,U) \longrightarrow \mathit{equal}(\mathit{cross\text{-}product}(T'::'a,V),\mathit{cross\text{-}product}(U::'a,V))) \\
\&\ \\
\ (\forall\, W\ Y\ X.\ \mathit{equal}(W::'a,X) \longrightarrow \mathit{equal}(\mathit{cross\text{-}product}(Y::'a,W),\mathit{cross\text{-}product}(Y::'a,X))) \\
\&\ \\
\ (\forall\, Z\ A1.\ \mathit{equal}(Z::'a,A1) \longrightarrow \mathit{equal}(\mathit{diagonalise}(Z),\mathit{diagonalise}(A1))) \ \&\ \\
\ (\forall\, B1\ C1\ D1.\ \mathit{equal}(B1::'a,C1) \longrightarrow \mathit{equal}(\mathit{difference}(B1::'a,D1),\mathit{difference}(C1::'a,D1))) \\
\&\ \\
\ (\forall\, E1\ G1\ F1.\ \mathit{equal}(E1::'a,F1) \longrightarrow \mathit{equal}(\mathit{difference}(G1::'a,E1),\mathit{difference}(G1::'a,F1))) \\
\&\ \\
\ (\forall\, H1\ I1\ J1\ K1.\ \mathit{equal}(H1::'a,I1) \longrightarrow \mathit{equal}(\mathit{domain}(H1::'a,J1,K1),\mathit{domain}(I1::'a,J1,K1))) \\
\&\ \\
\ (\forall\, L1\ N1\ M1\ O1.\ \mathit{equal}(L1::'a,M1) \longrightarrow \mathit{equal}(\mathit{domain}(N1::'a,L1,O1),\mathit{domain}(N1::'a,M1,O1))) \\
\&\ \\
\ (\forall\, P1\ R1\ S1\ Q1.\ \mathit{equal}(P1::'a,Q1) \longrightarrow \mathit{equal}(\mathit{domain}(R1::'a,S1,P1),\mathit{domain}(R1::'a,S1,Q1))) \\
\&\ \\
\ (\forall\, T1\ U1.\ \mathit{equal}(T1::'a,U1) \longrightarrow \mathit{equal}(\mathit{domain\text{-}of}(T1),\mathit{domain\text{-}of}(U1))) \ \&\ \\
\ (\forall\, V1\ W1.\ \mathit{equal}(V1::'a,W1) \longrightarrow \mathit{equal}(\mathit{first}(V1),\mathit{first}(W1))) \ \&\ \\
\ (\forall\, X1\ Y1.\ \mathit{equal}(X1::'a,Y1) \longrightarrow \mathit{equal}(\mathit{flip}(X1),\mathit{flip}(Y1))) \ \&\ \\
\ (\forall\, Z1\ A2\ B2.\ \mathit{equal}(Z1::'a,A2) \longrightarrow \mathit{equal}(\mathit{image'}(Z1::'a,B2),\mathit{image'}(A2::'a,B2))) \\
\&\ \\
\ (\forall\, C2\ E2\ D2.\ \mathit{equal}(C2::'a,D2) \longrightarrow \mathit{equal}(\mathit{image'}(E2::'a,C2),\mathit{image'}(E2::'a,D2))) \\
\&\ \\
\ (\forall\, F2\ G2\ H2.\ \mathit{equal}(F2::'a,G2) \longrightarrow \mathit{equal}(\mathit{intersection}(F2::'a,H2),\mathit{intersection}(G2::'a,H2))) \\
\&\ \\
\ (\forall\, I2\ K2\ J2.\ \mathit{equal}(I2::'a,J2) \longrightarrow \mathit{equal}(\mathit{intersection}(K2::'a,I2),\mathit{intersection}(K2::'a,J2))) \\
\&\ \\
\ (\forall\, L2\ M2.\ \mathit{equal}(L2::'a,M2) \longrightarrow \mathit{equal}(\mathit{INVERSE}(L2),\mathit{INVERSE}(M2))) \ \&\ \\
\ (\forall\, N2\ O2\ P2\ Q2.\ \mathit{equal}(N2::'a,O2) \longrightarrow \mathit{equal}(\mathit{not\text{-}homomorphism1}(N2::'a,P2,Q2),\mathit{not\text{-}homomorphism1}(O2 \\
\&\ \\
\ (\forall\, R2\ T2\ S2\ U2.\ \mathit{equal}(R2::'a,S2) \longrightarrow \mathit{equal}(\mathit{not\text{-}homomorphism1}(T2::'a,R2,U2),\mathit{not\text{-}homomorphism1}(T2: \\
\&\ \\
\ (\forall\, V2\ X2\ Y2\ W2.\ \mathit{equal}(V2::'a,W2) \longrightarrow \mathit{equal}(\mathit{not\text{-}homomorphism1}(X2::'a,Y2,V2),\mathit{not\text{-}homomorphism1}(X \\
\&\ \\
\ (\forall\, Z2\ A3\ B3\ C3.\ \mathit{equal}(Z2::'a,A3) \longrightarrow \mathit{equal}(\mathit{not\text{-}homomorphism2}(Z2::'a,B3,C3),\mathit{not\text{-}homomorphism2}(A3: \\
\&\ \\
\ (\forall\, D3\ F3\ E3\ G3.\ \mathit{equal}(D3::'a,E3) \longrightarrow \mathit{equal}(\mathit{not\text{-}homomorphism2}(F3::'a,D3,G3),\mathit{not\text{-}homomorphism2}(F3 \\
\&\ \\
\ (\forall\, H3\ J3\ K3\ I3.\ \mathit{equal}(H3::'a,I3) \longrightarrow \mathit{equal}(\mathit{not\text{-}homomorphism2}(J3::'a,K3,H3),\mathit{not\text{-}homomorphism2}(J3::' \\
\&\ \\
\ (\forall\, L3\ M3\ N3.\ \mathit{equal}(L3::'a,M3) \longrightarrow \mathit{equal}(\mathit{not\text{-}subclass\text{-}element}(L3::'a,N3),\mathit{not\text{-}subclass\text{-}element}(M3::'a,N3 \\
\&\
\end{array}
$$

$(\forall\ O3\ Q3\ P3.\ equal(O3::'a,P3) \longrightarrow equal(not\text{-}subclass\text{-}element(Q3::'a,O3),not\text{-}subclass\text{-}element(Q3::'a,P3$
$\&$
$(\forall\ R3\ S3\ T3.\ equal(R3::'a,S3) \longrightarrow equal(ordered\text{-}pair(R3::'a,T3),ordered\text{-}pair(S3::'a,T3)))$
$\&$
$(\forall\ U3\ W3\ V3.\ equal(U3::'a,V3) \longrightarrow equal(ordered\text{-}pair(W3::'a,U3),ordered\text{-}pair(W3::'a,V3)))$
$\&$
$(\forall\ X3\ Y3.\ equal(X3::'a,Y3) \longrightarrow equal(powerClass(X3),powerClass(Y3))) \ \&$
$(\forall\ Z3\ A4\ B4\ C4.\ equal(Z3::'a,A4) \longrightarrow equal(rng(Z3::'a,B4,C4),rng(A4::'a,B4,C4)))$
$\&$
$(\forall\ D4\ F4\ E4\ G4.\ equal(D4::'a,E4) \longrightarrow equal(rng(F4::'a,D4,G4),rng(F4::'a,E4,G4)))$
$\&$
$(\forall\ H4\ J4\ K4\ I4.\ equal(H4::'a,I4) \longrightarrow equal(rng(J4::'a,K4,H4),rng(J4::'a,K4,I4)))$
$\&$
$(\forall\ L4\ M4.\ equal(L4::'a,M4) \longrightarrow equal(range\text{-}of(L4),range\text{-}of(M4))) \ \&$
$(\forall\ N4\ O4.\ equal(N4::'a,O4) \longrightarrow equal(regular(N4),regular(O4))) \ \&$
$(\forall\ P4\ Q4\ R4\ S4.\ equal(P4::'a,Q4) \longrightarrow equal(restrct(P4::'a,R4,S4),restrct(Q4::'a,R4,S4)))$
$\&$
$(\forall\ T4\ V4\ U4\ W4.\ equal(T4::'a,U4) \longrightarrow equal(restrct(V4::'a,T4,W4),restrct(V4::'a,U4,W4)))$
$\&$
$(\forall\ X4\ Z4\ A5\ Y4.\ equal(X4::'a,Y4) \longrightarrow equal(restrct(Z4::'a,A5,X4),restrct(Z4::'a,A5,Y4)))$
$\&$
$(\forall\ B5\ C5.\ equal(B5::'a,C5) \longrightarrow equal(rot(B5),rot(C5))) \ \&$
$(\forall\ D5\ E5.\ equal(D5::'a,E5) \longrightarrow equal(second(D5),second(E5))) \ \&$
$(\forall\ F5\ G5.\ equal(F5::'a,G5) \longrightarrow equal(singleton(F5),singleton(G5))) \ \&$
$(\forall\ H5\ I5.\ equal(H5::'a,I5) \longrightarrow equal(successor(H5),successor(I5))) \ \&$
$(\forall\ J5\ K5.\ equal(J5::'a,K5) \longrightarrow equal(sum\text{-}class(J5),sum\text{-}class(K5))) \ \&$
$(\forall\ L5\ M5\ N5.\ equal(L5::'a,M5) \longrightarrow equal(union(L5::'a,N5),union(M5::'a,N5)))$
$\&$
$(\forall\ O5\ Q5\ P5.\ equal(O5::'a,P5) \longrightarrow equal(union(Q5::'a,O5),union(Q5::'a,P5)))$
$\&$
$(\forall\ R5\ S5\ T5.\ equal(R5::'a,S5) \longrightarrow equal(unordered\text{-}pair(R5::'a,T5),unordered\text{-}pair(S5::'a,T5)))$
$\&$
$(\forall\ U5\ W5\ V5.\ equal(U5::'a,V5) \longrightarrow equal(unordered\text{-}pair(W5::'a,U5),unordered\text{-}pair(W5::'a,V5)))$
$\&$
$(\forall\ X5\ Y5\ Z5\ A6.\ equal(X5::'a,Y5) \ \&\ compatible(X5::'a,Z5,A6) \longrightarrow compati$
$ble(Y5::'a,Z5,A6)) \ \&$
$(\forall\ B6\ D6\ C6\ E6.\ equal(B6::'a,C6) \ \&\ compatible(D6::'a,B6,E6) \longrightarrow compati$
$ble(D6::'a,C6,E6)) \ \&$
$(\forall\ F6\ H6\ I6\ G6.\ equal(F6::'a,G6) \ \&\ compatible(H6::'a,I6,F6) \longrightarrow compati$
$ble(H6::'a,I6,G6)) \ \&$
$(\forall\ J6\ K6.\ equal(J6::'a,K6) \ \&\ function(J6) \longrightarrow function(K6)) \ \&$
$(\forall\ L6\ M6\ N6\ O6.\ equal(L6::'a,M6) \ \&\ homomorphism(L6::'a,N6,O6) \longrightarrow ho$
$momorphism(M6::'a,N6,O6)) \ \&$
$(\forall\ P6\ R6\ Q6\ S6.\ equal(P6::'a,Q6) \ \&\ homomorphism(R6::'a,P6,S6) \longrightarrow ho$
$momorphism(R6::'a,Q6,S6)) \ \&$
$(\forall\ T6\ V6\ W6\ U6.\ equal(T6::'a,U6) \ \&\ homomorphism(V6::'a,W6,T6) \longrightarrow ho$
$momorphism(V6::'a,W6,U6)) \ \&$
$(\forall\ X6\ Y6.\ equal(X6::'a,Y6) \ \&\ inductive(X6) \longrightarrow inductive(Y6)) \ \&$
$(\forall\ Z6\ A7\ B7.\ equal(Z6::'a,A7) \ \&\ member(Z6::'a,B7) \longrightarrow member(A7::'a,B7))$

&
 (∀ 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)) −−> irreflex-
ive(X::'a,Y)) &
(∀ Y X. connected(X::'a,Y) −−> subclass(cross-product(Y::'a,Y),union(identity-relation::'a,symmetrizatio
&
(∀ 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))
&

$(\forall\, Xr\ Y.\ equal(restrct(intersection(Xr::'a,INVERSE(Xr)),Y,Y),null\text{-}class)$ $--> asymmetric(Xr::'a,Y))$ &
$(\forall\, Xr\ Y\ Z.\ equal(segment(Xr::'a,Y,Z),domain\text{-}of(restrct(Xr::'a,Y,singleton(Z)))))$ &
$(\forall\, X\ Y.\ well\text{-}ordering(X::'a,Y)$ $--> connected(X::'a,Y))$ &
$(\forall\, Y\ Xr\ U.\ well\text{-}ordering(Xr::'a,Y)$ & $subclass(U::'a,Y)$ $--> equal(U::'a,null\text{-}class)$ | $member(least(Xr::'a,U),U))$ &
$(\forall\, Y\ V\ Xr\ U.\ well\text{-}ordering(Xr::'a,Y)$ & $subclass(U::'a,Y)$ & $member(V::'a,U)$ $--> member(least(Xr::'a,U),U))$ &
$(\forall\, Y\ Xr\ U.\ well\text{-}ordering(Xr::'a,Y)$ & $subclass(U::'a,Y)$ $--> equal(segment(Xr::'a,U,least(Xr::'a,U)),null\text{-}$ &
$(\forall\, Y\ V\ U\ Xr.\ \sim(well\text{-}ordering(Xr::'a,Y)$ & $subclass(U::'a,Y)$ & $member(V::'a,U)$ & $member(ordered\text{-}pair(V::'a,least(Xr::'a,U)),Xr)))$ &
$(\forall\, Xr\ Y.\ connected(Xr::'a,Y)$ & $equal(not\text{-}well\text{-}ordering(Xr::'a,Y),null\text{-}class)$ $--> well\text{-}ordering(Xr::'a,Y))$ &
$(\forall\, Xr\ Y.\ connected(Xr::'a,Y)$ $--> subclass(not\text{-}well\text{-}ordering(Xr::'a,Y),Y)$ | $well\text{-}ordering(Xr::'a,Y))$ &
$(\forall\, V\ Xr\ Y.\ member(V::'a,not\text{-}well\text{-}ordering(Xr::'a,Y))$ & $equal(segment(Xr::'a,not\text{-}well\text{-}ordering(Xr::'a,Y),$ & $connected(Xr::'a,Y)$ $--> well\text{-}ordering(Xr::'a,Y))$ &
$(\forall\, Xr\ Y\ Z.\ section(Xr::'a,Y,Z)$ $--> subclass(Y::'a,Z))$ &
$(\forall\, Xr\ Z\ Y.\ section(Xr::'a,Y,Z)$ $--> subclass(domain\text{-}of(restrct(Xr::'a,Z,Y)),Y))$ &
$(\forall\, Xr\ Y\ Z.\ subclass(Y::'a,Z)$ & $subclass(domain\text{-}of(restrct(Xr::'a,Z,Y)),Y)$ $--> section(Xr::'a,Y,Z))$ &
$(\forall\, X.\ member(X::'a,ordinal\text{-}numbers)$ $--> well\text{-}ordering(element\text{-}relation::'a,X))$ &
$(\forall\, X.\ member(X::'a,ordinal\text{-}numbers)$ $--> subclass(sum\text{-}class(X),X))$ &
$(\forall\, X.\ well\text{-}ordering(element\text{-}relation::'a,X)$ & $subclass(sum\text{-}class(X),X)$ & $member(X::'a,universal\text{-}class)$ $--> member(X::'a,ordinal\text{-}numbers))$ &
$(\forall\, X.\ well\text{-}ordering(element\text{-}relation::'a,X)$ & $subclass(sum\text{-}class(X),X)$ $--> member(X::'a,ordinal\text{-}numbers)$ | $equal(X::'a,ordinal\text{-}numbers))$ &
$(equal(union(singleton(null\text{-}class),image'(successor\text{-}relation::'a,ordinal\text{-}numbers)),kind\text{-}1\text{-}ordinals))$ &
$(equal(intersection(complement(kind\text{-}1\text{-}ordinals),ordinal\text{-}numbers),limit\text{-}ordinals))$ &
$(\forall\, X.\ subclass(rest\text{-}of(X),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$ &
$(\forall\, V\ U\ X.\ member(ordered\text{-}pair(U::'a,V),rest\text{-}of(X))$ $--> member(U::'a,domain\text{-}of(X)))$ &
$(\forall\, X\ U\ V.\ member(ordered\text{-}pair(U::'a,V),rest\text{-}of(X))$ $--> equal(restrct(X::'a,U,universal\text{-}class),V))$ &
$(\forall\, U\ V\ X.\ member(U::'a,domain\text{-}of(X))$ & $equal(restrct(X::'a,U,universal\text{-}class),V)$ $--> member(ordered\text{-}pair(U::'a,V),rest\text{-}of(X)))$ &
$(subclass(rest\text{-}relation::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$ &
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),rest\text{-}relation)$ $--> equal(rest\text{-}of(X),Y))$ &
$(\forall\, X.\ member(X::'a,universal\text{-}class)$ $--> member(ordered\text{-}pair(X::'a,rest\text{-}of(X)),rest\text{-}relation))$ &
$(\forall\, X\ Z.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z))$ $--> function(Z))$ &
$(\forall\, Z\ X.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z))$ $--> function(X))$ &

$(\forall\, Z\ X.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z)) \longrightarrow member(domain\text{-}of(X),ordinal\text{-}numbers))$
&
$(\forall\, Z\ X.\ member(X::'a,recursion\text{-}equation\text{-}functions(Z)) \longrightarrow equal(compos(Z::'a,rest\text{-}of(X)),X))$
&
$(\forall\, X\ Z.\ function(Z)\ \&\ function(X)\ \&\ member(domain\text{-}of(X),ordinal\text{-}numbers)\ \&$
$equal(compos(Z::'a,rest\text{-}of(X)),X) \longrightarrow member(X::'a,recursion\text{-}equation\text{-}functions(Z)))$
&
$(subclass(union\text{-}of\text{-}range\text{-}map::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}class)))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),union\text{-}of\text{-}range\text{-}map) \longrightarrow equal(sum\text{-}class(range\text{-}of(X)),Y))$
&
$(\forall\, X\ Y.\ member(ordered\text{-}pair(X::'a,Y),cross\text{-}product(universal\text{-}class::'a,universal\text{-}class))$
$\&\ equal(sum\text{-}class(range\text{-}of(X)),Y) \longrightarrow member(ordered\text{-}pair(X::'a,Y),union\text{-}of\text{-}range\text{-}map))$
&
$(\forall\, X\ Y.\ equal(apply(recursion(X::'a,successor\text{-}relation,union\text{-}of\text{-}range\text{-}map),Y),ordinal\text{-}add(X::'a,Y)))$
&
$(\forall\, X\ Y.\ equal(recursion(null\text{-}class::'a,apply(add\text{-}relation::'a,X),union\text{-}of\text{-}range\text{-}map),ordinal\text{-}multiply(X::'a,$
&
$(\forall\, X.\ member(X::'a,omega) \longrightarrow equal(integer\text{-}of(X),X))\ \&$
$(\forall\, X.\ member(X::'a,omega) \mid equal(integer\text{-}of(X),null\text{-}class))$

abbreviation *NUM004-0-eq well-ordering transitive section irreflexive connected asymmetric symmetrization-of segment rest-of recursion-equation-functions recursion ordinal-multiply ordinal-add not-well-ordering least integer-of equal* $\equiv$
$(\forall\, D\ E.\ equal(D::'a,E) \longrightarrow equal(integer\text{-}of(D),integer\text{-}of(E)))\ \&$
$(\forall\, F'\ G\ H.\ equal(F'::'a,G) \longrightarrow equal(least(F'::'a,H),least(G::'a,H)))\ \&$
$(\forall\, I'\ K'\ J.\ equal(I'::'a,J) \longrightarrow equal(least(K'::'a,I'),least(K'::'a,J)))\ \&$
$(\forall\, L\ M\ N.\ equal(L::'a,M) \longrightarrow equal(not\text{-}well\text{-}ordering(L::'a,N),not\text{-}well\text{-}ordering(M::'a,N)))$
&
$(\forall\, O'\ Q\ P.\ equal(O'::'a,P) \longrightarrow equal(not\text{-}well\text{-}ordering(Q::'a,O'),not\text{-}well\text{-}ordering(Q::'a,P)))$
&
$(\forall\, R\ S'\ T'.\ equal(R::'a,S') \longrightarrow equal(ordinal\text{-}add(R::'a,T'),ordinal\text{-}add(S'::'a,T')))$
&
$(\forall\, U\ W\ V.\ equal(U::'a,V) \longrightarrow equal(ordinal\text{-}add(W::'a,U),ordinal\text{-}add(W::'a,V)))$
&
$(\forall\, X\ Y\ Z.\ equal(X::'a,Y) \longrightarrow equal(ordinal\text{-}multiply(X::'a,Z),ordinal\text{-}multiply(Y::'a,Z)))$
&
$(\forall\, A1\ C1\ B1.\ equal(A1::'a,B1) \longrightarrow equal(ordinal\text{-}multiply(C1::'a,A1),ordinal\text{-}multiply(C1::'a,B1)))$
&
$(\forall\, F1\ G1\ H1\ I1.\ equal(F1::'a,G1) \longrightarrow equal(recursion(F1::'a,H1,I1),recursion(G1::'a,H1,I1)))$
&
$(\forall\, J1\ L1\ K1\ M1.\ equal(J1::'a,K1) \longrightarrow equal(recursion(L1::'a,J1,M1),recursion(L1::'a,K1,M1)))$
&
$(\forall\, N1\ P1\ Q1\ O1.\ equal(N1::'a,O1) \longrightarrow equal(recursion(P1::'a,Q1,N1),recursion(P1::'a,Q1,O1)))$
&
$(\forall\, R1\ S1.\ equal(R1::'a,S1) \longrightarrow equal(recursion\text{-}equation\text{-}functions(R1),recursion\text{-}equation\text{-}functions(S1)))$
&
$(\forall\, T1\ U1.\ equal(T1::'a,U1) \longrightarrow equal(rest\text{-}of(T1),rest\text{-}of(U1)))\ \&$

$(\forall\ V1\ W1\ X1\ Y1.\ equal(V1::'a,W1) \longrightarrow equal(segment(V1::'a,X1,Y1),segment(W1::'a,X1,Y1)))$
&
$(\forall\ Z1\ B2\ A2\ C2.\ equal(Z1::'a,A2) \longrightarrow equal(segment(B2::'a,Z1,C2),segment(B2::'a,A2,C2)))$
&
$(\forall\ D2\ F2\ G2\ E2.\ equal(D2::'a,E2) \longrightarrow equal(segment(F2::'a,G2,D2),segment(F2::'a,G2,E2)))$
&
$(\forall\ H2\ I2.\ equal(H2::'a,I2) \longrightarrow equal(symmetrization\text{-}of(H2),symmetrization\text{-}of(I2)))$
&
$(\forall\ J2\ K2\ L2.\ equal(J2::'a,K2)\ \&\ asymmetric(J2::'a,L2) \longrightarrow asymmetric(K2::'a,L2))$
&
$(\forall\ M2\ O2\ N2.\ equal(M2::'a,N2)\ \&\ asymmetric(O2::'a,M2) \longrightarrow asymmetric(O2::'a,N2))$
&
$(\forall\ P2\ Q2\ R2.\ equal(P2::'a,Q2)\ \&\ connected(P2::'a,R2) \longrightarrow connected(Q2::'a,R2))$
&
$(\forall\ S2\ U2\ T2.\ equal(S2::'a,T2)\ \&\ connected(U2::'a,S2) \longrightarrow connected(U2::'a,T2))$
&
$(\forall\ V2\ W2\ X2.\ equal(V2::'a,W2)\ \&\ irreflexive(V2::'a,X2) \longrightarrow irreflexive(W2::'a,X2))$
&
$(\forall\ Y2\ A3\ Z2.\ equal(Y2::'a,Z2)\ \&\ irreflexive(A3::'a,Y2) \longrightarrow irreflexive(A3::'a,Z2))$
&
$(\forall\ B3\ C3\ D3\ E3.\ equal(B3::'a,C3)\ \&\ section(B3::'a,D3,E3) \longrightarrow section(C3::'a,D3,E3))$
&
$(\forall\ F3\ H3\ G3\ I3.\ equal(F3::'a,G3)\ \&\ section(H3::'a,F3,I3) \longrightarrow section(H3::'a,G3,I3))$
&
$(\forall\ J3\ L3\ M3\ K3.\ equal(J3::'a,K3)\ \&\ section(L3::'a,M3,J3) \longrightarrow section(L3::'a,M3,K3))$
&
$(\forall\ N3\ O3\ P3.\ equal(N3::'a,O3)\ \&\ transitive(N3::'a,P3) \longrightarrow transitive(O3::'a,P3))$
&
$(\forall\ Q3\ S3\ R3.\ equal(Q3::'a,R3)\ \&\ transitive(S3::'a,Q3) \longrightarrow transitive(S3::'a,R3))$
&
$(\forall\ T3\ U3\ V3.\ equal(T3::'a,U3)\ \&\ well\text{-}ordering(T3::'a,V3) \longrightarrow well\text{-}ordering(U3::'a,V3))$
&
$(\forall\ W3\ Y3\ X3.\ equal(W3::'a,X3)\ \&\ well\text{-}ordering(Y3::'a,W3) \longrightarrow well\text{-}ordering(Y3::'a,X3))$

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*)) &

$(\forall\, S'.\ \sim at(a::'a,S')) \longrightarrow False$
by *meson*

abbreviation *PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds* $\equiv$
$(\forall\, X\ Y\ State.\ holds(X::'a,State)\ \&\ holds(Y::'a,State) \longrightarrow holds(and'(X::'a,Y),State))$
&
$(\forall\, State\ X.\ holds(EMPTY::'a,State)\ \&\ holds(clear(X),State)\ \&\ differ(X::'a,table)$
$\longrightarrow holds(holding(X),do(pickup(X),State)))\ \&$
$(\forall\, Y\ X\ State.\ holds(on(X::'a,Y),State)\ \&\ holds(clear(X),State)\ \&\ holds(EMPTY::'a,State)$
$\longrightarrow holds(clear(Y),do(pickup(X),State)))\ \&$
$(\forall\, Y\ State\ X\ Z.\ holds(on(X::'a,Y),State)\ \&\ differ(X::'a,Z) \longrightarrow holds(on(X::'a,Y),do(pickup(Z),State)))$
&
$(\forall\, State\ X\ Z.\ holds(clear(X),State)\ \&\ differ(X::'a,Z) \longrightarrow holds(clear(X),do(pickup(Z),State)))$
&
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State) \longrightarrow holds(EMPTY::'a,do(putdown(X::'a,Y$
&
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State) \longrightarrow holds(on(X::'a,Y),do(putdown(X::'a,Y$
&
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State) \longrightarrow holds(clear(X),do(putdown(X::'a,Y),St$
&
$(\forall\, Z\ W\ X\ Y\ State.\ holds(on(X::'a,Y),State) \longrightarrow holds(on(X::'a,Y),do(putdown(Z::'a,W),State)))$
&
$(\forall\, X\ State\ Z\ Y.\ holds(clear(Z),State)\ \&\ differ(Z::'a,Y) \longrightarrow holds(clear(Z),do(putdown(X::'a,Y),State)))$

abbreviation *PLA001-1-ax EMPTY clear s0 on holds table d c b a differ* $\equiv$
$(\forall\, Y\ X.\ differ(Y::'a,X) \longrightarrow differ(X::'a,Y))\ \&$
$(differ(a::'a,b))\ \&$
$(differ(a::'a,c))\ \&$
$(differ(a::'a,d))\ \&$
$(differ(a::'a,table))\ \&$
$(differ(b::'a,c))\ \&$
$(differ(b::'a,d))\ \&$
$(differ(b::'a,table))\ \&$
$(differ(c::'a,d))\ \&$
$(differ(c::'a,table))\ \&$
$(differ(d::'a,table))\ \&$
$(holds(on(a::'a,table),s0))\ \&$
$(holds(on(b::'a,table),s0))\ \&$
$(holds(on(c::'a,d),s0))\ \&$
$(holds(on(d::'a,table),s0))\ \&$
$(holds(clear(a),s0))\ \&$
$(holds(clear(b),s0))\ \&$
$(holds(clear(c),s0))\ \&$
$(holds(EMPTY::'a,s0))\ \&$
$(\forall\, State.\ holds(clear(table),State))$

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*))
&
(∀ *Y X*. *mless-or-equal*(*X*::′*a*,*Y*) | *mless-or-equal*(*Y*::′*a*,*X*)) &
(∀ *X Y*. *equal*(*X*::′*a*,*Y*) −−> *mless-or-equal*(*X*::′*a*,*Y*)) &
(∀ *X Y Z*. *equal*(*X*::′*a*,*Y*) & *mless-or-equal*(*X*::′*a*,*Z*) −−> *mless-or-equal*(*Y*::′*a*,*Z*))
&
(∀ *X Z Y*. *equal*(*X*::′*a*,*Y*) & *mless-or-equal*(*Z*::′*a*,*X*) −−> *mless-or-equal*(*Z*::′*a*,*Y*))
&
(*q1*(*a*::′*a*,*b*,*c*)) &
(∀ *W*. ~(*q4*(*a*::′*a*,*b*,*W*) & *mless-or-equal*(*a*::′*a*,*W*) & *mless-or-equal*(*b*::′*a*,*W*) &

mless-or-equal(*W*::'*a*,*a*))) &
(∀ *W*. ~(*q4*(*a*::'*a*,*b*,*W*) & *mless-or-equal*(*a*::'*a*,*W*) & *mless-or-equal*(*b*::'*a*,*W*) &
mless-or-equal(*W*::'*a*,*b*))) −−> *False*
by *meson*

abbreviation *SWV001-1-ax mless-THAN successor predecessor equal* ≡
(∀ *X*. *equal*(*predecessor*(*successor*(*X*)),*X*)) &
(∀ *X*. *equal*(*successor*(*predecessor*(*X*)),*X*)) &
(∀ *X Y*. *equal*(*predecessor*(*X*),*predecessor*(*Y*)) −−> *equal*(*X*::'*a*,*Y*)) &
(∀ *X Y*. *equal*(*successor*(*X*),*successor*(*Y*)) −−> *equal*(*X*::'*a*,*Y*)) &
(∀ *X*. *mless-THAN*(*predecessor*(*X*),*X*)) &
(∀ *X*. *mless-THAN*(*X*::'*a*,*successor*(*X*))) &
(∀ *X Y Z*. *mless-THAN*(*X*::'*a*,*Y*) & *mless-THAN*(*Y*::'*a*,*Z*) −−> *mless-THAN*(*X*::'*a*,*Z*))
&
(∀ *X Y*. *mless-THAN*(*X*::'*a*,*Y*) | *mless-THAN*(*Y*::'*a*,*X*) | *equal*(*X*::'*a*,*Y*)) &
(∀ *X*. ~*mless-THAN*(*X*::'*a*,*X*)) &
(∀ *Y X*. ~(*mless-THAN*(*X*::'*a*,*Y*) & *mless-THAN*(*Y*::'*a*,*X*))) &
(∀ *Y X Z*. *equal*(*X*::'*a*,*Y*) & *mless-THAN*(*X*::'*a*,*Z*) −−> *mless-THAN*(*Y*::'*a*,*Z*))
&
(∀ *Y Z X*. *equal*(*X*::'*a*,*Y*) & *mless-THAN*(*Z*::'*a*,*X*) −−> *mless-THAN*(*Z*::'*a*,*Y*))

abbreviation *SWV001-0-eq a successor predecessor equal* ≡
(∀ *X Y*. *equal*(*X*::'*a*,*Y*) −−> *equal*(*predecessor*(*X*),*predecessor*(*Y*))) &
(∀ *X Y*. *equal*(*X*::'*a*,*Y*) −−> *equal*(*successor*(*X*),*successor*(*Y*))) &
(∀ *X Y*. *equal*(*X*::'*a*,*Y*) −−> *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 &
(~*mless-THAN*(*n*::'*a*,*j*)) &
(*mless-THAN*(*k*::'*a*,*j*)) &
(~*mless-THAN*(*k*::'*a*,*i*)) &
(*mless-THAN*(*i*::'*a*,*n*)) &
(*mless-THAN*(*a*(*j*),*a*(*k*))) &
(∀ *X*. *mless-THAN*(*X*::'*a*,*j*) & *mless-THAN*(*a*(*X*),*a*(*k*)) −−> *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*)))))
&
(*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*) −−> *an-*
swer(*yes*)) &
(∀ *X Y*. *ask-1-if-2*(*X*::'*a*,*Y*) & *a-truth*(*truthteller*(*X*)) −−> *a-truth*(*Y*) | *an-*
swer(*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)) &
(∀ 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)

& $product(V3::'a,X,V4) \longrightarrow sum(V1::'a,V2,V4))$ &
$(\forall\ Y\ Z\ V1\ V2\ V3\ X\ V4.\ product(Y::'a,X,V1)$ & $product(Z::'a,X,V2)$ & $sum(Y::'a,Z,V3)$
& $sum(V1::'a,V2,V4) \longrightarrow product(V3::'a,X,V4))$ &
$(\forall\ X\ Y\ U\ V.\ sum(X::'a,Y,U)$ & $sum(X::'a,Y,V) \longrightarrow equal(U::'a,V))$ &
$(\forall\ X\ Y\ U\ V.\ product(X::'a,Y,U)$ & $product(X::'a,Y,V) \longrightarrow equal(U::'a,V))$
&
$(\forall\ X\ Y.\ equal(X::'a,Y) \longrightarrow equal(additive\text{-}inverse(X),additive\text{-}inverse(Y)))$ &
$(\forall\ X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(add(X::'a,W),add(Y::'a,W)))$ &
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $sum(X::'a,W,Z) \longrightarrow sum(Y::'a,W,Z))$ &
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $sum(W::'a,X,Z) \longrightarrow sum(W::'a,Y,Z))$ &
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $sum(W::'a,Z,X) \longrightarrow sum(W::'a,Z,Y))$ &
$(\forall\ X\ Y\ W.\ equal(X::'a,Y) \longrightarrow equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall\ X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z) \longrightarrow product(Y::'a,W,Z))$
&
$(\forall\ X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z) \longrightarrow product(W::'a,Y,Z))$
&
$(\forall\ X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X) \longrightarrow product(W::'a,Z,Y))$

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
&
$(\sim product(a::'a,additive\text{-}identity,additive\text{-}identity)) \longrightarrow False$
oops

lemma *RNG011-5*:
EQU001-0-ax equal &
$(\forall\ A\ B\ C.\ equal(A::'a,B) \longrightarrow equal(add(A::'a,C),add(B::'a,C)))$ &
$(\forall\ D\ F'\ E.\ equal(D::'a,E) \longrightarrow equal(add(F'::'a,D),add(F'::'a,E)))$ &
$(\forall\ G\ H.\ equal(G::'a,H) \longrightarrow equal(additive\text{-}inverse(G),additive\text{-}inverse(H)))$ &
$(\forall\ I'\ J\ K'.\ equal(I'::'a,J) \longrightarrow equal(multiply(I'::'a,K'),multiply(J::'a,K')))$ &
$(\forall\ L\ N\ M.\ equal(L::'a,M) \longrightarrow equal(multiply(N::'a,L),multiply(N::'a,M)))$ &
$(\forall\ A\ B\ C\ D.\ equal(A::'a,B) \longrightarrow equal(associator(A::'a,C,D),associator(B::'a,C,D)))$
&
$(\forall\ E\ G\ F'\ H.\ equal(E::'a,F') \longrightarrow equal(associator(G::'a,E,H),associator(G::'a,F',H)))$
&
$(\forall\ I'\ K'\ L\ J.\ equal(I'::'a,J) \longrightarrow equal(associator(K'::'a,L,I'),associator(K'::'a,L,J)))$
&
$(\forall\ M\ N\ O'.\ equal(M::'a,N) \longrightarrow equal(commutator(M::'a,O'),commutator(N::'a,O')))$
&

$(\forall\, P\ R\ Q.\ equal(P::'a,Q) \longrightarrow equal(commutator(R::'a,P),commutator(R::'a,Q)))$
&
$(\forall\, Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z))))$ &
$(\forall\, X.\ equal(add(X::'a,additive\text{-}identity),X))$ &
$(\forall\, X.\ equal(add(additive\text{-}identity::'a,X),X))$ &
$(\forall\, X.\ equal(add(X::'a,additive\text{-}inverse(X)),additive\text{-}identity))$ &
$(\forall\, X.\ equal(add(additive\text{-}inverse(X),X),additive\text{-}identity))$ &
$(equal(additive\text{-}inverse(additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X\ Y.\ equal(add(X::'a,add(additive\text{-}inverse(X),Y)),Y))$ &
$(\forall\, X\ Y.\ equal(additive\text{-}inverse(add(X::'a,Y)),add(additive\text{-}inverse(X),additive\text{-}inverse(Y))))$
&
$(\forall\, X.\ equal(additive\text{-}inverse(additive\text{-}inverse(X)),X))$ &
$(\forall\, X.\ equal(multiply(X::'a,additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X.\ equal(multiply(additive\text{-}identity::'a,X),additive\text{-}identity))$ &
$(\forall\, X\ Y.\ equal(multiply(additive\text{-}inverse(X),additive\text{-}inverse(Y)),multiply(X::'a,Y)))$
&
$(\forall\, X\ Y.\ equal(multiply(X::'a,additive\text{-}inverse(Y)),additive\text{-}inverse(multiply(X::'a,Y))))$
&
$(\forall\, X\ Y.\ equal(multiply(additive\text{-}inverse(X),Y),additive\text{-}inverse(multiply(X::'a,Y))))$
&
$(\forall\, Y\ X\ Z.\ equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))$
&
$(\forall\, X\ Y\ Z.\ equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))$
&
$(\forall\, X\ Y.\ equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))$
&
$(\forall\, X\ Y\ Z.\ equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive\text{-}inverse(multiply(X::'a,m$
&
$(\forall\, X\ Y.\ equal(commutator(X::'a,Y),add(multiply(Y::'a,X),additive\text{-}inverse(multiply(X::'a,Y)))))$
&
$(\forall\, X\ Y.\ equal(multiply(multiply(associator(X::'a,X,Y),X),associator(X::'a,X,Y)),additive\text{-}identity))$
&
$(\sim equal(multiply(multiply(associator(a::'a,a,b),a),associator(a::'a,a,b)),additive\text{-}identity))$
$\longrightarrow False$
by *meson*

lemma *RNG023-6*:
EQU001-0-ax equal &
$(\forall\, Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(X::'a,add(Y::'a,Z)),add(add(X::'a,Y),Z)))$ &
$(\forall\, X.\ equal(add(additive\text{-}identity::'a,X),X))$ &
$(\forall\, X.\ equal(add(X::'a,additive\text{-}identity),X))$ &
$(\forall\, X.\ equal(multiply(additive\text{-}identity::'a,X),additive\text{-}identity))$ &
$(\forall\, X.\ equal(multiply(X::'a,additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X.\ equal(add(additive\text{-}inverse(X),X),additive\text{-}identity))$ &
$(\forall\, X.\ equal(add(X::'a,additive\text{-}inverse(X)),additive\text{-}identity))$ &
$(\forall\, Y\ X\ Z.\ equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))$

&
(∀ 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))
&
(∀ Subset Superset. member(member-of-1-not-of-2(Subset::′a,Superset),Superset)
−−> subset(Subset::′a,Superset)) &
(∀ Subset Superset. equal-sets(Subset::′a,Superset) −−> subset(Subset::′a,Superset))
&
(∀ Subset Superset. equal-sets(Superset::′a,Subset) −−> subset(Subset::′a,Superset))
&
(∀ Set2 Set1. subset(Set1::′a,Set2) & subset(Set2::′a,Set1) −−> equal-sets(Set2::′a,Set1))
&
(∀ Set2 Intersection Element Set1. intersection(Set1::′a,Set2,Intersection) & mem-

ber(*Element*::'*a*,*Intersection*) −−> *member*(*Element*::'*a*,*Set1*)) &
(∀ *Set1 Intersection Element Set2*. *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) & *member*(*Element*::'*a*,*Intersection*) −−> *member*(*Element*::'*a*,*Set2*)) &
(∀ *Set2 Set1 Element Intersection*. *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) & *member*(*Element*::'*a*,*Set2*) & *member*(*Element*::'*a*,*Set1*) −−> *member*(*Element*::'*a*,*Intersection*))
&
(∀ *Set2 Intersection Set1*. *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Intersection*) | *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) | *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set1*))
&
(∀ *Set1 Intersection Set2*. *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Intersection*) | *intersection*(*Set1*::'*a*,*Set2*,*Intersection*) | *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set2*))
&
(∀ *Set1 Set2 Intersection*. *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Intersection*) & *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set2*) & *member*(*h*(*Set1*::'*a*,*Set2*,*Intersection*),*Set1*) −−> *intersection*(*Set1*::'*a*,*Set2*,*Intersection*)) &
(*intersection*(*a*::'*a*,*b*,*aIb*)) &
(*intersection*(*b*::'*a*,*c*,*bIc*)) &
(*intersection*(*a*::'*a*,*bIc*,*aIbIc*)) &
(~*intersection*(*aIb*::'*a*,*c*,*aIbIc*)) −−> *False*
oops

lemma *SET009-1*:
(∀ *Subset Element Superset*. *member*(*Element*::'*a*,*Subset*) & *ssubset*(*Subset*::'*a*,*Superset*) −−> *member*(*Element*::'*a*,*Superset*)) &
(∀ *Superset Subset*. *ssubset*(*Subset*::'*a*,*Superset*) | *member*(*member-of-1-not-of-2*(*Subset*::'*a*,*Superset*),*Subset*)
&
(∀ *Subset Superset*. *member*(*member-of-1-not-of-2*(*Subset*::'*a*,*Superset*),*Superset*) −−> *ssubset*(*Subset*::'*a*,*Superset*)) &
(∀ *Subset Superset*. *equal-sets*(*Subset*::'*a*,*Superset*) −−> *ssubset*(*Subset*::'*a*,*Superset*))
&
(∀ *Subset Superset*. *equal-sets*(*Superset*::'*a*,*Subset*) −−> *ssubset*(*Subset*::'*a*,*Superset*))
&
(∀ *Set2 Set1*. *ssubset*(*Set1*::'*a*,*Set2*) & *ssubset*(*Set2*::'*a*,*Set1*) −−> *equal-sets*(*Set2*::'*a*,*Set1*))
&
(∀ *Set2 Difference Element Set1*. *difference*(*Set1*::'*a*,*Set2*,*Difference*) & *member*(*Element*::'*a*,*Difference*) −−> *member*(*Element*::'*a*,*Set1*)) &
(∀ *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*))))
&
(∀ *Z X*. *member*(*Z*::'*a*,*second*(*X*)) −−> *member*(*Z*::'*a*,*f7*(*Z*::'*a*,*X*))) &
(∀ *X U Z V*. *little-set*(*U*) & *little-set*(*V*) & *equal*(*X*::'*a*,*ordered-pair*(*U*::'*a*,*V*))
& *member*(*Z*::'*a*,*V*) −−> *member*(*Z*::'*a*,*second*(*X*))) &
(∀ *Z*. *member*(*Z*::'*a*,*estin*) −−> *ordered-pair-predicate*(*Z*)) &
(∀ *Z*. *member*(*Z*::'*a*,*estin*) −−> *member*(*first*(*Z*),*second*(*Z*))) &
(∀ *Z*. *little-set*(*Z*) & *ordered-pair-predicate*(*Z*) & *member*(*first*(*Z*),*second*(*Z*))
−−> *member*(*Z*::'*a*,*estin*)) &
(∀ *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*)) &

$(\forall\, X\ Z\ Y.\ member(Z::'a,X)\ \&\ member(Z::'a,Y)\ -->\ member(Z::'a,intersection(X::'a,Y)))$
$\&$
$(\forall\, Z\ X.\ \sim(member(Z::'a,complement(X))\ \&\ member(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ little\text{-}set(Z)\ -->\ member(Z::'a,complement(X))\ |\ member(Z::'a,X))\ \&$
$(\forall\, X\ Y.\ equal(union(X::'a,Y),complement(intersection(complement(X),complement(Y)))))$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,domain\text{-}of(X))\ -->\ ordered\text{-}pair\text{-}predicate(f8(Z::'a,X)))$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,domain\text{-}of(X))\ -->\ member(f8(Z::'a,X),X))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,domain\text{-}of(X))\ -->\ equal(Z::'a,first(f8(Z::'a,X))))\ \&$
$(\forall\, X\ Z\ Xp.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Xp)\ \&\ member(Xp::'a,X)\ \&$
$equal(Z::'a,first(Xp))\ -->\ member(Z::'a,domain\text{-}of(X)))\ \&$
$(\forall\, X\ Y\ Z.\ member(Z::'a,cross\text{-}product(X::'a,Y))\ -->\ ordered\text{-}pair\text{-}predicate(Z))$
$\&$
$(\forall\, Y\ Z\ X.\ member(Z::'a,cross\text{-}product(X::'a,Y))\ -->\ member(first(Z),X))\ \&$
$(\forall\, X\ Z\ Y.\ member(Z::'a,cross\text{-}product(X::'a,Y))\ -->\ member(second(Z),Y))$
$\&$
$(\forall\, X\ Z\ Y.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Z)\ \&\ member(first(Z),X)\ \&$
$member(second(Z),Y)\ -->\ member(Z::'a,cross\text{-}product(X::'a,Y)))\ \&$
$(\forall\, X\ Z.\ member(Z::'a,inv1\ X)\ -->\ ordered\text{-}pair\text{-}predicate(Z))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,inv1\ X)\ -->\ member(ordered\text{-}pair(second(Z),first(Z)),X))$
$\&$
$(\forall\, Z\ X.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Z)\ \&\ member(ordered\text{-}pair(second(Z),first(Z)),X)$
$-->\ member(Z::'a,inv1\ X))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ little\text{-}set(f9(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ little\text{-}set(f10(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ little\text{-}set(f11(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ equal(Z::'a,ordered\text{-}pair(f9(Z::'a,X),ordered\text{-}pair(f10(Z::'a,X),f11$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,rot\text{-}right(X))\ -->\ member(ordered\text{-}pair(f10(Z::'a,X),ordered\text{-}pair(f11(Z::'a,X),f9(Z$
$\&$
$(\forall\, Z\ V\ W\ U\ X.\ little\text{-}set(Z)\ \&\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ little\text{-}set(W)\ \&$
$equal(Z::'a,ordered\text{-}pair(U::'a,ordered\text{-}pair(V::'a,W)))\ \&\ member(ordered\text{-}pair(V::'a,ordered\text{-}pair(W::'a,U))$
$-->\ member(Z::'a,rot\text{-}right(X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ little\text{-}set(f12(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ little\text{-}set(f13(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ little\text{-}set(f14(Z::'a,X)))\ \&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ equal(Z::'a,ordered\text{-}pair(f12(Z::'a,X),ordered\text{-}pair(f13(Z::'a,X$
$\&$
$(\forall\, Z\ X.\ member(Z::'a,flip\text{-}range\text{-}of(X))\ -->\ member(ordered\text{-}pair(f12(Z::'a,X),ordered\text{-}pair(f14(Z::'a,X),f$
$\&$
$(\forall\, Z\ U\ W\ V\ X.\ little\text{-}set(Z)\ \&\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ little\text{-}set(W)\ \&$
$equal(Z::'a,ordered\text{-}pair(U::'a,ordered\text{-}pair(V::'a,W)))\ \&\ member(ordered\text{-}pair(U::'a,ordered\text{-}pair(W::'a,V))$
$-->\ member(Z::'a,flip\text{-}range\text{-}of(X)))\ \&$
$(\forall\, X.\ equal(successor(X),union(X::'a,singleton\text{-}set(X))))\ \&$
$(\forall\, Z.\ \sim member(Z::'a,empty\text{-}set))\ \&$
$(\forall\, Z.\ little\text{-}set(Z)\ -->\ member(Z::'a,universal\text{-}set))\ \&$
$(little\text{-}set(infinity))\ \&$
$(member(empty\text{-}set::'a,infinity))\ \&$

$(\forall X.\ member(X::'a,infinity) \longrightarrow member(successor(X),infinity))\ \&$
$(\forall Z\ X.\ member(Z::'a,sigma(X)) \longrightarrow member(f16(Z::'a,X),X))\ \&$
$(\forall Z\ X.\ member(Z::'a,sigma(X)) \longrightarrow member(Z::'a,f16(Z::'a,X)))\ \&$
$(\forall X\ Z\ Y.\ member(Y::'a,X)\ \&\ member(Z::'a,Y) \longrightarrow member(Z::'a,sigma(X)))$
$\&$
$(\forall U.\ little\text{-}set(U) \longrightarrow little\text{-}set(sigma(U)))\ \&$
$(\forall X\ U\ Y.\ ssubset(X::'a,Y)\ \&\ member(U::'a,X) \longrightarrow member(U::'a,Y))\ \&$
$(\forall Y\ X.\ ssubset(X::'a,Y) \mid member(f17(X::'a,Y),X))\ \&$
$(\forall X\ Y.\ member(f17(X::'a,Y),Y) \longrightarrow ssubset(X::'a,Y))\ \&$
$(\forall X\ Y.\ proper\text{-}subset(X::'a,Y) \longrightarrow ssubset(X::'a,Y))\ \&$
$(\forall X\ Y.\ \sim(proper\text{-}subset(X::'a,Y)\ \&\ equal(X::'a,Y)))\ \&$
$(\forall X\ Y.\ ssubset(X::'a,Y) \longrightarrow proper\text{-}subset(X::'a,Y) \mid equal(X::'a,Y))\ \&$
$(\forall Z\ X.\ member(Z::'a,powerset(X)) \longrightarrow ssubset(Z::'a,X))\ \&$
$(\forall Z\ X.\ little\text{-}set(Z)\ \&\ ssubset(Z::'a,X) \longrightarrow member(Z::'a,powerset(X)))\ \&$
$(\forall U.\ little\text{-}set(U) \longrightarrow little\text{-}set(powerset(U)))\ \&$
$(\forall Z\ X.\ relation(Z)\ \&\ member(X::'a,Z) \longrightarrow ordered\text{-}pair\text{-}predicate(X))\ \&$
$(\forall Z.\ relation(Z) \mid member(f18(Z),Z))\ \&$
$(\forall Z.\ ordered\text{-}pair\text{-}predicate(f18(Z)) \longrightarrow relation(Z))\ \&$
$(\forall U\ X\ V\ W.\ single\text{-}valued\text{-}set(X)\ \&\ little\text{-}set(U)\ \&\ little\text{-}set(V)\ \&\ little\text{-}set(W)$
$\&\ member(ordered\text{-}pair(U::'a,V),X)\ \&\ member(ordered\text{-}pair(U::'a,W),X) \longrightarrow$
$equal(V::'a,W))\ \&$
$(\forall X.\ single\text{-}valued\text{-}set(X) \mid little\text{-}set(f19(X)))\ \&$
$(\forall X.\ single\text{-}valued\text{-}set(X) \mid little\text{-}set(f20(X)))\ \&$
$(\forall X.\ single\text{-}valued\text{-}set(X) \mid little\text{-}set(f21(X)))\ \&$
$(\forall X.\ single\text{-}valued\text{-}set(X) \mid member(ordered\text{-}pair(f19(X),f20(X)),X))\ \&$
$(\forall X.\ single\text{-}valued\text{-}set(X) \mid member(ordered\text{-}pair(f19(X),f21(X)),X))\ \&$
$(\forall X.\ equal(f20(X),f21(X)) \longrightarrow single\text{-}valued\text{-}set(X))\ \&$
$(\forall Xf.\ function(Xf) \longrightarrow relation(Xf))\ \&$
$(\forall Xf.\ function(Xf) \longrightarrow single\text{-}valued\text{-}set(Xf))\ \&$
$(\forall Xf.\ relation(Xf)\ \&\ single\text{-}valued\text{-}set(Xf) \longrightarrow function(Xf))\ \&$
$(\forall Z\ X\ Xf.\ member(Z::'a,image'(X::'a,Xf)) \longrightarrow ordered\text{-}pair\text{-}predicate(f22(Z::'a,X,Xf)))$
$\&$
$(\forall Z\ X\ Xf.\ member(Z::'a,image'(X::'a,Xf)) \longrightarrow member(f22(Z::'a,X,Xf),Xf))$
$\&$
$(\forall Z\ Xf\ X.\ member(Z::'a,image'(X::'a,Xf)) \longrightarrow member(first(f22(Z::'a,X,Xf)),X))$
$\&$
$(\forall X\ Xf\ Z.\ member(Z::'a,image'(X::'a,Xf)) \longrightarrow equal(second(f22(Z::'a,X,Xf)),Z))$
$\&$
$(\forall Xf\ X\ Y\ Z.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Y)\ \&\ member(Y::'a,Xf)\ \&$
$member(first(Y),X)\ \&\ equal(second(Y),Z) \longrightarrow member(Z::'a,image'(X::'a,Xf)))$
$\&$
$(\forall X\ Xf.\ little\text{-}set(X)\ \&\ function(Xf) \longrightarrow little\text{-}set(image'(X::'a,Xf)))\ \&$
$(\forall X\ U\ Y.\ \sim(disjoint(X::'a,Y)\ \&\ member(U::'a,X)\ \&\ member(U::'a,Y)))\ \&$
$(\forall Y\ X.\ disjoint(X::'a,Y) \mid member(f23(X::'a,Y),X))\ \&$
$(\forall X\ Y.\ disjoint(X::'a,Y) \mid member(f23(X::'a,Y),Y))\ \&$
$(\forall X.\ equal(X::'a,empty\text{-}set) \mid member(f24(X),X))\ \&$
$(\forall X.\ equal(X::'a,empty\text{-}set) \mid disjoint(f24(X),X))\ \&$
$(function(f25))\ \&$
$(\forall X.\ little\text{-}set(X) \longrightarrow equal(X::'a,empty\text{-}set) \mid member(f26(X),X))\ \&$

$(\forall X.\ little\text{-}set(X) \longrightarrow equal(X::'a,empty\text{-}set) \mid member(ordered\text{-}pair(X::'a,f26(X)),f25))$
&
$(\forall Z\ X.\ member(Z::'a,range\text{-}of(X)) \longrightarrow ordered\text{-}pair\text{-}predicate(f27(Z::'a,X)))$
&
$(\forall Z\ X.\ member(Z::'a,range\text{-}of(X)) \longrightarrow member(f27(Z::'a,X),X))$ &
$(\forall Z\ X.\ member(Z::'a,range\text{-}of(X)) \longrightarrow equal(Z::'a,second(f27(Z::'a,X))))$ &
$(\forall X\ Z\ Xp.\ little\text{-}set(Z)$ & $ordered\text{-}pair\text{-}predicate(Xp)$ & $member(Xp::'a,X)$ &
$equal(Z::'a,second(Xp)) \longrightarrow member(Z::'a,range\text{-}of(X)))$ &
$(\forall Z.\ member(Z::'a,identity\text{-}relation) \longrightarrow ordered\text{-}pair\text{-}predicate(Z))$ &
$(\forall Z.\ member(Z::'a,identity\text{-}relation) \longrightarrow equal(first(Z),second(Z)))$ &
$(\forall Z.\ little\text{-}set(Z)$ & $ordered\text{-}pair\text{-}predicate(Z)$ & $equal(first(Z),second(Z)) \longrightarrow$
$member(Z::'a,identity\text{-}relation))$ &
$(\forall X\ Y.\ equal(restrct(X::'a,Y),intersection(X::'a,cross\text{-}product(Y::'a,universal\text{-}set))))$
&
$(\forall Xf.\ one\text{-}to\text{-}one\text{-}function(Xf) \longrightarrow function(Xf))$ &
$(\forall Xf.\ one\text{-}to\text{-}one\text{-}function(Xf) \longrightarrow function(inv1\ Xf))$ &
$(\forall Xf.\ function(Xf)$ & $function(inv1\ Xf) \longrightarrow one\text{-}to\text{-}one\text{-}function(Xf))$ &
$(\forall Z\ Xf\ Y.\ member(Z::'a,apply(Xf::'a,Y)) \longrightarrow ordered\text{-}pair\text{-}predicate(f28(Z::'a,Xf,Y)))$
&
$(\forall Z\ Y\ Xf.\ member(Z::'a,apply(Xf::'a,Y)) \longrightarrow member(f28(Z::'a,Xf,Y),Xf))$
&
$(\forall Z\ Xf\ Y.\ member(Z::'a,apply(Xf::'a,Y)) \longrightarrow equal(first(f28(Z::'a,Xf,Y)),Y))$
&
$(\forall Z\ Xf\ Y.\ member(Z::'a,apply(Xf::'a,Y)) \longrightarrow member(Z::'a,second(f28(Z::'a,Xf,Y))))$
&
$(\forall Xf\ Y\ Z\ W.\ ordered\text{-}pair\text{-}predicate(W)$ & $member(W::'a,Xf)$ & $equal(first(W),Y)$
& $member(Z::'a,second(W)) \longrightarrow member(Z::'a,apply(Xf::'a,Y)))$ &
$(\forall Xf\ X\ Y.\ equal(apply\text{-}to\text{-}two\text{-}arguments(Xf::'a,X,Y),apply(Xf::'a,ordered\text{-}pair(X::'a,Y))))$
&
$(\forall X\ Y\ Xf.\ maps(Xf::'a,X,Y) \longrightarrow function(Xf))$ &
$(\forall Y\ Xf\ X.\ maps(Xf::'a,X,Y) \longrightarrow equal(domain\text{-}of(Xf),X))$ &
$(\forall X\ Xf\ Y.\ maps(Xf::'a,X,Y) \longrightarrow ssubset(range\text{-}of(Xf),Y))$ &
$(\forall X\ Xf\ Y.\ function(Xf)$ & $equal(domain\text{-}of(Xf),X)$ & $ssubset(range\text{-}of(Xf),Y)$
$\longrightarrow maps(Xf::'a,X,Y))$ &
$(\forall Xf\ Xs.\ closed(Xs::'a,Xf) \longrightarrow little\text{-}set(Xs))$ &
$(\forall Xs\ Xf.\ closed(Xs::'a,Xf) \longrightarrow little\text{-}set(Xf))$ &
$(\forall Xf\ Xs.\ closed(Xs::'a,Xf) \longrightarrow maps(Xf::'a,cross\text{-}product(Xs::'a,Xs),Xs))$ &
$(\forall Xf\ Xs.\ little\text{-}set(Xs)$ & $little\text{-}set(Xf)$ & $maps(Xf::'a,cross\text{-}product(Xs::'a,Xs),Xs)$
$\longrightarrow closed(Xs::'a,Xf))$ &
$(\forall Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow little\text{-}set(f29(Z::'a,Xf,Xg)))$
&
$(\forall Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow little\text{-}set(f30(Z::'a,Xf,Xg)))$
&
$(\forall Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow little\text{-}set(f31(Z::'a,Xf,Xg)))$
&
$(\forall Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow equal(Z::'a,ordered\text{-}pair(f29(Z::'a,Xf,Xg),f30(Z::'a$
&
$(\forall Z\ Xg\ Xf.\ member(Z::'a,composition(Xf::'a,Xg)) \longrightarrow member(ordered\text{-}pair(f29(Z::'a,Xf,Xg),f31(Z::'a,X$
&

(∀ 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) & mem-
ber(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) & mem-
ber(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′))) &
(∀ J L K′. equal(J::'a,K′) −−> equal(f10(L::'a,J),f10(L::'a,K′))) &
(∀ M N O′. equal(M::'a,N) −−> equal(f11(M::'a,O′),f11(N::'a,O′))) &
(∀ P R Q. equal(P::'a,Q) −−> equal(f11(R::'a,P),f11(R::'a,Q))) &
(∀ S′ T′ U. equal(S′::'a,T′) −−> equal(f12(S′::'a,U),f12(T′::'a,U))) &
(∀ V X W. equal(V::'a,W) −−> equal(f12(X::'a,V),f12(X::'a,W))) &
(∀ Y Z A1. equal(Y::'a,Z) −−> equal(f13(Y::'a,A1),f13(Z::'a,A1))) &
(∀ B1 D1 C1. equal(B1::'a,C1) −−> equal(f13(D1::'a,B1),f13(D1::'a,C1))) &
(∀ E1 F1 G1. equal(E1::'a,F1) −−> equal(f14(E1::'a,G1),f14(F1::'a,G1))) &
(∀ H1 J1 I1. equal(H1::'a,I1) −−> equal(f14(J1::'a,H1),f14(J1::'a,I1))) &

$$
\begin{array}{l}
(\forall\, K1\ L1\ M1.\ equal(K1::'a,L1) \longrightarrow equal(f16(K1::'a,M1),f16(L1::'a,M1))) \ \&\\
(\forall\, N1\ P1\ O1.\ equal(N1::'a,O1) \longrightarrow equal(f16(P1::'a,N1),f16(P1::'a,O1))) \ \&\\
(\forall\, Q1\ R1\ S1.\ equal(Q1::'a,R1) \longrightarrow equal(f17(Q1::'a,S1),f17(R1::'a,S1))) \ \&\\
(\forall\, T1\ V1\ U1.\ equal(T1::'a,U1) \longrightarrow equal(f17(V1::'a,T1),f17(V1::'a,U1))) \ \&\\
(\forall\, W1\ X1.\ equal(W1::'a,X1) \longrightarrow equal(f18(W1),f18(X1))) \ \&\\
(\forall\, Y1\ Z1.\ equal(Y1::'a,Z1) \longrightarrow equal(f19(Y1),f19(Z1))) \ \&\\
(\forall\, C2\ D2.\ equal(C2::'a,D2) \longrightarrow equal(f20(C2),f20(D2))) \ \&\\
(\forall\, E2\ F2.\ equal(E2::'a,F2) \longrightarrow equal(f21(E2),f21(F2))) \ \&\\
(\forall\, G2\ H2\ I2\ J2.\ equal(G2::'a,H2) \longrightarrow equal(f22(G2::'a,I2,J2),f22(H2::'a,I2,J2)))\\
\&\\
(\forall\, K2\ M2\ L2\ N2.\ equal(K2::'a,L2) \longrightarrow equal(f22(M2::'a,K2,N2),f22(M2::'a,L2,N2)))\\
\&\\
(\forall\, O2\ Q2\ R2\ P2.\ equal(O2::'a,P2) \longrightarrow equal(f22(Q2::'a,R2,O2),f22(Q2::'a,R2,P2)))\\
\&\\
(\forall\, S2\ T2\ U2.\ equal(S2::'a,T2) \longrightarrow equal(f23(S2::'a,U2),f23(T2::'a,U2))) \ \&\\
(\forall\, V2\ X2\ W2.\ equal(V2::'a,W2) \longrightarrow equal(f23(X2::'a,V2),f23(X2::'a,W2)))\\
\&\\
(\forall\, Y2\ Z2.\ equal(Y2::'a,Z2) \longrightarrow equal(f24(Y2),f24(Z2))) \ \&\\
(\forall\, A3\ B3.\ equal(A3::'a,B3) \longrightarrow equal(f26(A3),f26(B3))) \ \&\\
(\forall\, C3\ D3\ E3.\ equal(C3::'a,D3) \longrightarrow equal(f27(C3::'a,E3),f27(D3::'a,E3))) \ \&\\
(\forall\, F3\ H3\ G3.\ equal(F3::'a,G3) \longrightarrow equal(f27(H3::'a,F3),f27(H3::'a,G3))) \ \&\\
(\forall\, I3\ J3\ K3\ L3.\ equal(I3::'a,J3) \longrightarrow equal(f28(I3::'a,K3,L3),f28(J3::'a,K3,L3)))\\
\&\\
(\forall\, M3\ O3\ N3\ P3.\ equal(M3::'a,N3) \longrightarrow equal(f28(O3::'a,M3,P3),f28(O3::'a,N3,P3)))\\
\&\\
(\forall\, Q3\ S3\ T3\ R3.\ equal(Q3::'a,R3) \longrightarrow equal(f28(S3::'a,T3,Q3),f28(S3::'a,T3,R3)))\\
\&\\
(\forall\, U3\ V3\ W3\ X3.\ equal(U3::'a,V3) \longrightarrow equal(f29(U3::'a,W3,X3),f29(V3::'a,W3,X3)))\\
\&\\
(\forall\, Y3\ A4\ Z3\ B4.\ equal(Y3::'a,Z3) \longrightarrow equal(f29(A4::'a,Y3,B4),f29(A4::'a,Z3,B4)))\\
\&\\
(\forall\, C4\ E4\ F4\ D4.\ equal(C4::'a,D4) \longrightarrow equal(f29(E4::'a,F4,C4),f29(E4::'a,F4,D4)))\\
\&\\
(\forall\, I4\ J4\ K4\ L4.\ equal(I4::'a,J4) \longrightarrow equal(f30(I4::'a,K4,L4),f30(J4::'a,K4,L4)))\\
\&\\
(\forall\, M4\ O4\ N4\ P4.\ equal(M4::'a,N4) \longrightarrow equal(f30(O4::'a,M4,P4),f30(O4::'a,N4,P4)))\\
\&\\
(\forall\, Q4\ S4\ T4\ R4.\ equal(Q4::'a,R4) \longrightarrow equal(f30(S4::'a,T4,Q4),f30(S4::'a,T4,R4)))\\
\&\\
(\forall\, U4\ V4\ W4\ X4.\ equal(U4::'a,V4) \longrightarrow equal(f31(U4::'a,W4,X4),f31(V4::'a,W4,X4)))\\
\&\\
(\forall\, Y4\ A5\ Z4\ B5.\ equal(Y4::'a,Z4) \longrightarrow equal(f31(A5::'a,Y4,B5),f31(A5::'a,Z4,B5)))\\
\&\\
(\forall\, C5\ E5\ F5\ D5.\ equal(C5::'a,D5) \longrightarrow equal(f31(E5::'a,F5,C5),f31(E5::'a,F5,D5)))\\
\&\\
(\forall\, G5\ H5\ I5\ J5\ K5\ L5.\ equal(G5::'a,H5) \longrightarrow 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) \longrightarrow equal(f32(O5::'a,M5,P5,Q5,R5),f32(O5::'a,N5,P5,Q5,R5)\\
\&
\end{array}
$$

$$(\forall\, S5\ U5\ V5\ T5\ W5\ X5.\ equal(S5::'a,T5) \longrightarrow equal(f32(U5::'a,V5,S5,W5,X5),f32(U5::'a,V5,T5,W5,X5)$$
$$\&$$
$$(\forall\, Y5\ A6\ B6\ C6\ Z5\ D6.\ equal(Y5::'a,Z5) \longrightarrow equal(f32(A6::'a,B6,C6,Y5,D6),f32(A6::'a,B6,C6,Z5,D6)))$$
$$\&$$
$$(\forall\, E6\ G6\ H6\ I6\ J6\ F6.\ equal(E6::'a,F6) \longrightarrow equal(f32(G6::'a,H6,I6,J6,E6),f32(G6::'a,H6,I6,J6,F6)))$$
$$\&$$
$$(\forall\, K6\ L6\ M6\ N6\ O6\ P6.\ equal(K6::'a,L6) \longrightarrow equal(f33(K6::'a,M6,N6,O6,P6),f33(L6::'a,M6,N6,O6,P6)$$
$$\&$$
$$(\forall\, Q6\ S6\ R6\ T6\ U6\ V6.\ equal(Q6::'a,R6) \longrightarrow equal(f33(S6::'a,Q6,T6,U6,V6),f33(S6::'a,R6,T6,U6,V6)))$$
$$\&$$
$$(\forall\, W6\ Y6\ Z6\ X6\ A7\ B7.\ equal(W6::'a,X6) \longrightarrow equal(f33(Y6::'a,Z6,W6,A7,B7),f33(Y6::'a,Z6,X6,A7,B7)$$
$$\&$$
$$(\forall\, C7\ E7\ F7\ G7\ D7\ H7.\ equal(C7::'a,D7) \longrightarrow equal(f33(E7::'a,F7,G7,C7,H7),f33(E7::'a,F7,G7,D7,H7))$$
$$\&$$
$$(\forall\, I7\ K7\ L7\ M7\ N7\ J7.\ equal(I7::'a,J7) \longrightarrow equal(f33(K7::'a,L7,M7,N7,I7),f33(K7::'a,L7,M7,N7,J7)))$$
$$\&$$
$$(\forall\, A\ B\ C.\ equal(A::'a,B) \longrightarrow equal(apply(A::'a,C),apply(B::'a,C))) \ \&$$
$$(\forall\, D\ F'\ E.\ equal(D::'a,E) \longrightarrow equal(apply(F'::'a,D),apply(F'::'a,E))) \ \&$$
$$(\forall\, G\ H\ I'\ J.\ equal(G::'a,H) \longrightarrow equal(apply\text{-}to\text{-}two\text{-}arguments(G::'a,I',J),apply\text{-}to\text{-}two\text{-}arguments(H::'a,I',$$
$$\&$$
$$(\forall\, K'\ M\ L\ N.\ equal(K'::'a,L) \longrightarrow equal(apply\text{-}to\text{-}two\text{-}arguments(M::'a,K',N),apply\text{-}to\text{-}two\text{-}arguments(M::'a$$
$$\&$$
$$(\forall\, O'\ Q\ R\ P.\ equal(O'::'a,P) \longrightarrow equal(apply\text{-}to\text{-}two\text{-}arguments(Q::'a,R,O'),apply\text{-}to\text{-}two\text{-}arguments(Q::'a,$$
$$\&$$
$$(\forall\, S'\ T'.\ equal(S'::'a,T') \longrightarrow equal(complement(S'),complement(T'))) \ \&$$
$$(\forall\, U\ V\ W.\ equal(U::'a,V) \longrightarrow equal(composition(U::'a,W),composition(V::'a,W)))$$
$$\&$$
$$(\forall\, X\ Z\ Y.\ equal(X::'a,Y) \longrightarrow equal(composition(Z::'a,X),composition(Z::'a,Y)))$$
$$\&$$
$$(\forall\, A1\ B1.\ equal(A1::'a,B1) \longrightarrow equal(inv1\ A1,inv1\ B1)) \ \&$$
$$(\forall\, C1\ D1\ E1.\ equal(C1::'a,D1) \longrightarrow equal(cross\text{-}product(C1::'a,E1),cross\text{-}product(D1::'a,E1)))$$
$$\&$$
$$(\forall\, F1\ H1\ G1.\ equal(F1::'a,G1) \longrightarrow equal(cross\text{-}product(H1::'a,F1),cross\text{-}product(H1::'a,G1)))$$
$$\&$$
$$(\forall\, I1\ J1.\ equal(I1::'a,J1) \longrightarrow equal(domain\text{-}of(I1),domain\text{-}of(J1))) \ \&$$
$$(\forall\, I10\ J10.\ equal(I10::'a,J10) \longrightarrow equal(first(I10),first(J10))) \ \&$$
$$(\forall\, Q10\ R10.\ equal(Q10::'a,R10) \longrightarrow equal(flip\text{-}range\text{-}of(Q10),flip\text{-}range\text{-}of(R10)))$$
$$\&$$
$$(\forall\, S10\ T10\ U10.\ equal(S10::'a,T10) \longrightarrow equal(image'(S10::'a,U10),image'(T10::'a,U10)))$$
$$\&$$
$$(\forall\, V10\ X10\ W10.\ equal(V10::'a,W10) \longrightarrow equal(image'(X10::'a,V10),image'(X10::'a,W10)))$$
$$\&$$
$$(\forall\, Y10\ Z10\ A11.\ equal(Y10::'a,Z10) \longrightarrow equal(intersection(Y10::'a,A11),intersection(Z10::'a,A11)))$$
$$\&$$
$$(\forall\, B11\ D11\ C11.\ equal(B11::'a,C11) \longrightarrow equal(intersection(D11::'a,B11),intersection(D11::'a,C11)))$$
$$\&$$
$$(\forall\, E11\ F11\ G11.\ equal(E11::'a,F11) \longrightarrow equal(non\text{-}ordered\text{-}pair(E11::'a,G11),non\text{-}ordered\text{-}pair(F11::'a,G$$
$$\&$$
$$(\forall\, H11\ J11\ I11.\ equal(H11::'a,I11) \longrightarrow equal(non\text{-}ordered\text{-}pair(J11::'a,H11),non\text{-}ordered\text{-}pair(J11::'a,I11)$$

$$
\begin{array}{l}
\&\\
\ (\forall\ K11\ L11\ M11.\ equal(K11::'a,L11)\ -->\ equal(ordered\text{-}pair(K11::'a,M11),ordered\text{-}pair(L11::'a,M11)))\\
\&\\
\ (\forall\ N11\ P11\ O11.\ equal(N11::'a,O11)\ -->\ equal(ordered\text{-}pair(P11::'a,N11),ordered\text{-}pair(P11::'a,O11)))\\
\&\\
\ \ (\forall\ Q11\ R11.\ equal(Q11::'a,R11)\ -->\ equal(powerset(Q11),powerset(R11)))\ \&\\
\ \ (\forall\ S11\ T11.\ equal(S11::'a,T11)\ -->\ equal(range\text{-}of(S11),range\text{-}of(T11)))\ \&\\
\ (\forall\ U11\ V11\ W11.\ equal(U11::'a,V11)\ -->\ equal(restrct(U11::'a,W11),restrct(V11::'a,W11)))\\
\&\\
\ (\forall\ X11\ Z11\ Y11.\ equal(X11::'a,Y11)\ -->\ equal(restrct(Z11::'a,X11),restrct(Z11::'a,Y11)))\\
\&\\
\ \ (\forall\ A12\ B12.\ equal(A12::'a,B12)\ -->\ equal(rot\text{-}right(A12),rot\text{-}right(B12)))\ \&\\
\ \ (\forall\ C12\ D12.\ equal(C12::'a,D12)\ -->\ equal(second(C12),second(D12)))\ \&\\
\ \ (\forall\ K12\ L12.\ equal(K12::'a,L12)\ -->\ equal(sigma(K12),sigma(L12)))\ \&\\
\ (\forall\ M12\ N12.\ equal(M12::'a,N12)\ -->\ equal(singleton\text{-}set(M12),singleton\text{-}set(N12)))\\
\&\\
\ \ (\forall\ O12\ P12.\ equal(O12::'a,P12)\ -->\ equal(successor(O12),successor(P12)))\ \&\\
\ (\forall\ Q12\ R12\ S12.\ equal(Q12::'a,R12)\ -->\ equal(union(Q12::'a,S12),union(R12::'a,S12)))\\
\&\\
\ (\forall\ T12\ V12\ U12.\ equal(T12::'a,U12)\ -->\ equal(union(V12::'a,T12),union(V12::'a,U12)))\\
\&\\
\ (\forall\ W12\ X12\ Y12.\ equal(W12::'a,X12)\ \&\ closed(W12::'a,Y12)\ -->\ closed(X12::'a,Y12))\\
\&\\
\ (\forall\ Z12\ B13\ A13.\ equal(Z12::'a,A13)\ \&\ closed(B13::'a,Z12)\ -->\ closed(B13::'a,A13))\\
\&\\
\ (\forall\ C13\ D13\ E13.\ equal(C13::'a,D13)\ \&\ disjoint(C13::'a,E13)\ -->\ disjoint(D13::'a,E13))\\
\&\\
\ (\forall\ F13\ H13\ G13.\ equal(F13::'a,G13)\ \&\ disjoint(H13::'a,F13)\ -->\ disjoint(H13::'a,G13))\\
\&\\
\ \ (\forall\ I13\ J13.\ equal(I13::'a,J13)\ \&\ function(I13)\ -->\ function(J13))\ \&\\
\ (\forall\ K13\ L13\ M13\ N13\ O13\ P13.\ equal(K13::'a,L13)\ \&\ homomorphism(K13::'a,M13,N13,O13,P13)\\
-->\ homomorphism(L13::'a,M13,N13,O13,P13))\ \&\\
\ (\forall\ Q13\ S13\ R13\ T13\ U13\ V13.\ equal(Q13::'a,R13)\ \&\ homomorphism(S13::'a,Q13,T13,U13,V13)\\
-->\ homomorphism(S13::'a,R13,T13,U13,V13))\ \&\\
\ (\forall\ W13\ Y13\ Z13\ X13\ A14\ B14.\ equal(W13::'a,X13)\ \&\ homomorphism(Y13::'a,Z13,W13,A14,B14)\\
-->\ homomorphism(Y13::'a,Z13,X13,A14,B14))\ \&\\
\ (\forall\ C14\ E14\ F14\ G14\ D14\ H14.\ equal(C14::'a,D14)\ \&\ homomorphism(E14::'a,F14,G14,C14,H14)\\
-->\ homomorphism(E14::'a,F14,G14,D14,H14))\ \&\\
\ (\forall\ I14\ K14\ L14\ M14\ N14\ J14.\ equal(I14::'a,J14)\ \&\ homomorphism(K14::'a,L14,M14,N14,I14)\\
-->\ homomorphism(K14::'a,L14,M14,N14,J14))\ \&\\
\ \ (\forall\ O14\ P14.\ equal(O14::'a,P14)\ \&\ little\text{-}set(O14)\ -->\ little\text{-}set(P14))\ \&\\
\ (\forall\ Q14\ R14\ S14\ T14.\ equal(Q14::'a,R14)\ \&\ maps(Q14::'a,S14,T14)\ -->\ maps(R14::'a,S14,T14))\\
\&\\
\ \ \ (\forall\ U14\ \ W14\ \ V14\ \ X14.\ \ equal(U14::'a,V14)\ \ \&\ \ maps(W14::'a,U14,X14)\ \ -->\\
maps(W14::'a,V14,X14))\ \&\\
\ (\forall\ Y14\ A15\ B15\ Z14.\ equal(Y14::'a,Z14)\ \&\ maps(A15::'a,B15,Y14)\ -->\ maps(A15::'a,B15,Z14))\\
\&\\
\ (\forall\ C15\ D15\ E15.\ equal(C15::'a,D15)\ \&\ member(C15::'a,E15)\ -->\ member(D15::'a,E15))\\
\&
\end{array}
$$

(∀ 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*:
(∀ A. p(A::'a,a) | p(A::'a,f(A))) &
(∀ A. p(A::'a,a) | p(f(A),A)) &
(∀ A B. ~(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*) | *equal*(*c*::'*a*,*d*)) &
(*equal*(*a*::'*a*,*c*) | *equal*(*b*::'*a*,*d*)) &
(~*equal*(*a*::'*a*,*d*)) &
(~*equal*(*b*::'*a*,*c*)) −−> *False*
by *meson*

lemma *SYN349-1*:
(∀ *X Y*. *f*(*w*(*X*),*g*(*X*::'*a*,*Y*)) −−> *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*))) &
(∀ *X Y*. *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*)) −−> *f*(*w*(*X*),*g*(*X*::'*a*,*Y*))) &
(∀ *Y X*. *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*)) & *f*(*Y*::'*a*,*g*(*X*::'*a*,*Y*)) −−> *f*(*g*(*X*::'*a*,*Y*),*Y*) |
f(*g*(*X*::'*a*,*Y*),*w*(*X*))) &
(∀ *Y X*. *f*(*g*(*X*::'*a*,*Y*),*Y*) & *f*(*Y*::'*a*,*g*(*X*::'*a*,*Y*)) −−> *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*)) |
f(*g*(*X*::'*a*,*Y*),*w*(*X*))) &
(∀ *Y X*. *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*)))
&
(∀ *Y X*. *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*))) &
(∀ *Y X*. *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*)) & *f*(*g*(*X*::'*a*,*Y*),*w*(*X*)) −−> *f*(*g*(*X*::'*a*,*Y*),*Y*) |
f(*Y*::'*a*,*g*(*X*::'*a*,*Y*))) &
(∀ *Y X*. *f*(*g*(*X*::'*a*,*Y*),*Y*) & *f*(*g*(*X*::'*a*,*Y*),*w*(*X*)) −−> *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*)) |
f(*Y*::'*a*,*g*(*X*::'*a*,*Y*))) &
(∀ *Y X*. *f*(*Y*::'*a*,*g*(*X*::'*a*,*Y*)) & *f*(*g*(*X*::'*a*,*Y*),*w*(*X*)) −−> *f*(*X*::'*a*,*g*(*X*::'*a*,*Y*)) |
f(*g*(*X*::'*a*,*Y*),*Y*)) &
(∀ *Y X*. ~(*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*)) &
(∀ *X Y*. *f*(*X*::'*a*,*Y*) −−> *f*(*b*::'*a*,*z*(*X*::'*a*,*Y*)) | *f*(*Y*::'*a*,*z*(*X*::'*a*,*Y*))) &
(∀ *X Y*. *f*(*X*::'*a*,*Y*) | *f*(*z*(*X*::'*a*,*Y*),*z*(*X*::'*a*,*Y*))) &
(∀ *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*))) &
(∀ *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*)))
&
(∀ *X Y*. ~(*f*(*X*::'*a*,*Y*) & *f*(*X*::'*a*,*z*(*X*::'*a*,*Y*)) & *f*(*Y*::'*a*,*z*(*X*::'*a*,*Y*)))) &
(∀ *X Y*. *f*(*X*::'*a*,*Y*) −−> *f*(*X*::'*a*,*z*(*X*::'*a*,*Y*)) | *f*(*Y*::'*a*,*z*(*X*::'*a*,*Y*))) −−> *False*
by *meson*

lemma *TOP001-2*:
(∀ *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*))
&
(∀ *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*)) &
(∀ *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*)) &
(∀ *X*. *subset-sets*(*X*::'*a*,*X*)) &
(∀ *X U Y*. *subset-sets*(*X*::'*a*,*Y*) & *element-of-set*(*U*::'*a*,*X*) −−> *element-of-set*(*U*::'*a*,*Y*))
&
(∀ *X Y*. *equal-sets*(*X*::'*a*,*Y*) −−> *subset-sets*(*X*::'*a*,*Y*)) &
(∀ *Y X*. *subset-sets*(*X*::'*a*,*Y*) | *element-of-set*(*in-1st-set*(*X*::'*a*,*Y*),*X*)) &
(∀ *X Y*. *element-of-set*(*in-1st-set*(*X*::'*a*,*Y*),*Y*) −−> *subset-sets*(*X*::'*a*,*Y*)) &
(*basis*(*cx*::'*a*,*f*)) &
(~*subset-sets*(*union-of-members*(*top-of-basis*(*f*)),*cx*)) −−> *False*
by *meson*

lemma *TOP002-2*:
(∀ *Vf U*. *element-of-collection*(*U*::'*a*,*top-of-basis*(*Vf*)) | *element-of-set*(*f11*(*Vf*::'*a*,*U*),*U*))
&
(∀ *X*. ~*element-of-set*(*X*::'*a*,*empty-set*)) &
(~*element-of-collection*(*empty-set*::'*a*,*top-of-basis*(*f*))) −−> *False*
by *meson*

lemma *TOP004-1*:
(∀ *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*))
&
(∀ *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 U Va*. *element-of-set*(*U*::'*a*,*intersection-of-members*(*Vf*)) & *element-of-collection*(*Va*::'*a*,*Vf*)
−−> *element-of-set*(*U*::'*a*,*Va*)) &
(∀ *U Vf*. *element-of-set*(*U*::'*a*,*intersection-of-members*(*Vf*)) | *element-of-collection*(*f2*(*Vf*::'*a*,*U*),*Vf*))
&
(∀ *Vf U*. *element-of-set*(*U*::'*a*,*f2*(*Vf*::'*a*,*U*)) −−> *element-of-set*(*U*::'*a*,*intersection-of-members*(*Vf*)))
&
(∀ *Vt X*. *topological-space*(*X*::'*a*,*Vt*) −−> *equal-sets*(*union-of-members*(*Vt*),*X*))
&
(∀ *X Vt*. *topological-space*(*X*::'*a*,*Vt*) −−> *element-of-collection*(*empty-set*::'*a*,*Vt*))
&
(∀ *X Vt*. *topological-space*(*X*::'*a*,*Vt*) −−> *element-of-collection*(*X*::'*a*,*Vt*)) &
(∀ *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*))
&
(∀ *X Vf Vt*. *topological-space*(*X*::'*a*,*Vt*) & *subset-collections*(*Vf*::'*a*,*Vt*) −−>
element-of-collection(*union-of-members*(*Vf*),*Vt*)) &
(∀ *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*)

$$
\begin{array}{l}
\mid \mathit{subset\text{-}collections}(\mathit{f5}(X::'a,Vt),Vt)) \;\&\\
\;(\forall\, X\; Vt.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vt),X)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{empty\text{-}set}::'a,Vt)\\
\&\;\mathit{element\text{-}of\text{-}collection}(X::'a,Vt)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{union\text{-}of\text{-}members}(\mathit{f5}(X::'a,Vt)),Vt)\\
\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt)\mid \mathit{element\text{-}of\text{-}collection}(\mathit{f3}(X::'a,Vt),Vt))\;\&\\
\;(\forall\, X\; Vt.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vt),X)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{empty\text{-}set}::'a,Vt)\\
\&\;\mathit{element\text{-}of\text{-}collection}(X::'a,Vt)\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt)\mid \mathit{element\text{-}of\text{-}collection}(\mathit{f4}(X::'a,Vt),Vt)\\
\mid \mathit{subset\text{-}collections}(\mathit{f5}(X::'a,Vt),Vt))\;\&\\
\;(\forall\, X\; Vt.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vt),X)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{empty\text{-}set}::'a,Vt)\\
\&\;\mathit{element\text{-}of\text{-}collection}(X::'a,Vt)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{union\text{-}of\text{-}members}(\mathit{f5}(X::'a,Vt)),Vt)\\
\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt)\mid \mathit{element\text{-}of\text{-}collection}(\mathit{f4}(X::'a,Vt),Vt))\;\&\\
\;(\forall\, X\; Vt.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vt),X)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{empty\text{-}set}::'a,Vt)\\
\&\;\mathit{element\text{-}of\text{-}collection}(X::'a,Vt)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{intersection\text{-}of\text{-}sets}(\mathit{f3}(X::'a,Vt),\mathit{f4}(X::'a,Vt)),Vt)\\
\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt)\mid \mathit{subset\text{-}collections}(\mathit{f5}(X::'a,Vt),Vt))\;\&\\
\;(\forall\, X\; Vt.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vt),X)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{empty\text{-}set}::'a,Vt)\\
\&\;\mathit{element\text{-}of\text{-}collection}(X::'a,Vt)\;\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{intersection\text{-}of\text{-}sets}(\mathit{f3}(X::'a,Vt),\mathit{f4}(X::'a,Vt)),Vt)\\
\&\;\mathit{element\text{-}of\text{-}collection}(\mathit{union\text{-}of\text{-}members}(\mathit{f5}(X::'a,Vt)),Vt)\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt))\\
\&\\
\;\;(\forall\, U\; X\; Vt.\; \mathit{open}(U::'a,X,Vt)\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt))\;\&\\
\;\;(\forall\, X\; U\; Vt.\; \mathit{open}(U::'a,X,Vt)\longrightarrow \mathit{element\text{-}of\text{-}collection}(U::'a,Vt))\;\&\\
\;\;(\forall\, X\; U\; Vt.\; \mathit{topological\text{-}space}(X::'a,Vt)\;\&\;\mathit{element\text{-}of\text{-}collection}(U::'a,Vt)\longrightarrow\\
\mathit{open}(U::'a,X,Vt))\;\&\\
\;(\forall\, U\; X\; Vt.\; \mathit{closed}(U::'a,X,Vt)\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt))\;\&\\
\;(\forall\, U\, X\; Vt.\; \mathit{closed}(U::'a,X,Vt)\longrightarrow \mathit{open}(\mathit{relative\text{-}complement\text{-}sets}(U::'a,X),X,Vt))\\
\&\\
\;(\forall\, U\, X\; Vt.\; \mathit{topological\text{-}space}(X::'a,Vt)\;\&\;\mathit{open}(\mathit{relative\text{-}complement\text{-}sets}(U::'a,X),X,Vt)\\
\longrightarrow \mathit{closed}(U::'a,X,Vt))\;\&\\
\;(\forall\, Vs\; X\; Vt.\; \mathit{finer}(Vt::'a,Vs,X)\longrightarrow \mathit{topological\text{-}space}(X::'a,Vt))\;\&\\
\;(\forall\, Vt\; X\; Vs.\; \mathit{finer}(Vt::'a,Vs,X)\longrightarrow \mathit{topological\text{-}space}(X::'a,Vs))\;\&\\
\;(\forall\, X\; Vs\; Vt.\; \mathit{finer}(Vt::'a,Vs,X)\longrightarrow \mathit{subset\text{-}collections}(Vs::'a,Vt))\;\&\\
\;(\forall\, X\; Vs\; Vt.\; \mathit{topological\text{-}space}(X::'a,Vt)\;\&\;\mathit{topological\text{-}space}(X::'a,Vs)\;\&\;\mathit{subset\text{-}collections}(Vs::'a,Vt)\\
\longrightarrow \mathit{finer}(Vt::'a,Vs,X))\;\&\\
\;(\forall\, Vf\; X.\; \mathit{basis}(X::'a,Vf)\longrightarrow \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vf),X))\;\&\\
\;(\forall\, X\; Vf\; Y\; Vb1\; Vb2.\; \mathit{basis}(X::'a,Vf)\;\&\;\mathit{element\text{-}of\text{-}set}(Y::'a,X)\;\&\;\mathit{element\text{-}of\text{-}collection}(Vb1::'a,Vf)\\
\&\;\mathit{element\text{-}of\text{-}collection}(Vb2::'a,Vf)\;\&\;\mathit{element\text{-}of\text{-}set}(Y::'a,\mathit{intersection\text{-}of\text{-}sets}(Vb1::'a,Vb2))\\
\longrightarrow \mathit{element\text{-}of\text{-}set}(Y::'a,\mathit{f6}(X::'a,Vf,Y,Vb1,Vb2)))\;\&\\
\;(\forall\, X\; Y\; Vb1\; Vb2\; Vf.\; \mathit{basis}(X::'a,Vf)\;\&\;\mathit{element\text{-}of\text{-}set}(Y::'a,X)\;\&\;\mathit{element\text{-}of\text{-}collection}(Vb1::'a,Vf)\\
\&\;\mathit{element\text{-}of\text{-}collection}(Vb2::'a,Vf)\;\&\;\mathit{element\text{-}of\text{-}set}(Y::'a,\mathit{intersection\text{-}of\text{-}sets}(Vb1::'a,Vb2))\\
\longrightarrow \mathit{element\text{-}of\text{-}collection}(\mathit{f6}(X::'a,Vf,Y,Vb1,Vb2),Vf))\;\&\\
\;(\forall\, X\; Vf\; Y\; Vb1\; Vb2.\; \mathit{basis}(X::'a,Vf)\;\&\;\mathit{element\text{-}of\text{-}set}(Y::'a,X)\;\&\;\mathit{element\text{-}of\text{-}collection}(Vb1::'a,Vf)\\
\&\;\mathit{element\text{-}of\text{-}collection}(Vb2::'a,Vf)\;\&\;\mathit{element\text{-}of\text{-}set}(Y::'a,\mathit{intersection\text{-}of\text{-}sets}(Vb1::'a,Vb2))\\
\longrightarrow \mathit{subset\text{-}sets}(\mathit{f6}(X::'a,Vf,Y,Vb1,Vb2),\mathit{intersection\text{-}of\text{-}sets}(Vb1::'a,Vb2)))\;\&\\
\;(\forall\, Vf\, X.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vf),X)\longrightarrow \mathit{basis}(X::'a,Vf)\mid \mathit{element\text{-}of\text{-}set}(\mathit{f7}(X::'a,Vf),X))\\
\&\\
\;(\forall\, X\; Vf.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vf),X)\longrightarrow \mathit{basis}(X::'a,Vf)\mid \mathit{element\text{-}of\text{-}collection}(\mathit{f8}(X::'a,Vf),Vf)\\
\&\\
\;(\forall\, X\; Vf.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vf),X)\longrightarrow \mathit{basis}(X::'a,Vf)\mid \mathit{element\text{-}of\text{-}collection}(\mathit{f9}(X::'a,Vf),Vf)\\
\&\\
\;(\forall\, X\; Vf.\; \mathit{equal\text{-}sets}(\mathit{union\text{-}of\text{-}members}(Vf),X)\longrightarrow \mathit{basis}(X::'a,Vf)\mid \mathit{element\text{-}of\text{-}set}(\mathit{f7}(X::'a,Vf),\mathit{intersectio}\\
\&
\end{array}
$$

$(\forall\ Uu9\ X\ Vf.\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X)\ \&\ element\text{-}of\text{-}set(f7(X::'a,Vf),Uu9)$
$\&\ element\text{-}of\text{-}collection(Uu9::'a,Vf)\ \&\ subset\text{-}sets(Uu9::'a,intersection\text{-}of\text{-}sets(f8(X::'a,Vf),f9(X::'a,Vf)))$
$--> basis(X::'a,Vf))\ \&$
$(\forall\ Vf\ U\ X.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$--> element\text{-}of\text{-}set(X::'a,f10(Vf::'a,U,X)))\ \&$
$(\forall\ U\ X\ Vf.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$--> element\text{-}of\text{-}collection(f10(Vf::'a,U,X),Vf))\ \&$
$(\forall\ Vf\ X\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ \&\ element\text{-}of\text{-}set(X::'a,U)$
$--> subset\text{-}sets(f10(Vf::'a,U,X),U))\ \&$
$(\forall\ Vf\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf))\ |\ element\text{-}of\text{-}set(f11(Vf::'a,U),U))$
$\&$
$(\forall\ Vf\ Uu11\ U.\ element\text{-}of\text{-}set(f11(Vf::'a,U),Uu11)\ \&\ element\text{-}of\text{-}collection(Uu11::'a,Vf)$
$\&\ subset\text{-}sets(Uu11::'a,U)\ --> element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(Vf)))\ \&$
$(\forall\ U\ Y\ X\ Vt.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y))\ -->$
$topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ U\ Vt\ Y\ X.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y))\ -->$
$subset\text{-}sets(Y::'a,X))\ \&$
$(\forall\ X\ Y\ U\ Vt.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y))\ -->$
$element\text{-}of\text{-}collection(f12(X::'a,Vt,Y,U),Vt))\ \&$
$(\forall\ X\ Vt\ Y\ U.\ element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a,Vt,Y))\ -->$
$equal\text{-}sets(U::'a,intersection\text{-}of\text{-}sets(Y::'a,f12(X::'a,Vt,Y,U))))\ \&$
$(\forall\ X\ Vt\ U\ Y\ Uu12.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ \&\ element\text{-}of\text{-}collection(Uu12::'a,Vt)$
$\&\ equal\text{-}sets(U::'a,intersection\text{-}of\text{-}sets(Y::'a,Uu12))\ --> element\text{-}of\text{-}collection(U::'a,subspace\text{-}topology(X::'a$
$\&$
$(\forall\ U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt))\ --> topological\text{-}space(X::'a,Vt))$
$\&$
$(\forall\ U\ Vt\ Y\ X.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt))\ --> subset\text{-}sets(Y::'a,X))$
$\&$
$(\forall\ Y\ X\ Vt\ U.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt))\ --> element\text{-}of\text{-}set(U::'a,f13(Y::'a,X,Vt,U)))$
$\&$
$(\forall\ X\ Vt\ U\ Y.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt))\ --> subset\text{-}sets(f13(Y::'a,X,Vt,U),Y))$
$\&$
$(\forall\ Y\ U\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt))\ --> open(f13(Y::'a,X,Vt,U),X,Vt))$
$\&$
$(\forall\ U\ Y\ Uu13\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ \&\ element\text{-}of\text{-}set(U::'a,Uu13)$
$\&\ subset\text{-}sets(Uu13::'a,Y)\ \&\ open(Uu13::'a,X,Vt)\ --> element\text{-}of\text{-}set(U::'a,interior(Y::'a,X,Vt)))$
$\&$
$(\forall\ U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt))\ --> topological\text{-}space(X::'a,Vt))$
$\&$
$(\forall\ U\ Vt\ Y\ X.\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt))\ --> subset\text{-}sets(Y::'a,X))$
$\&$
$(\forall\ Y\ X\ Vt\ U\ V.\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt))\ \&\ subset\text{-}sets(Y::'a,V)$
$\&\ closed(V::'a,X,Vt)\ --> element\text{-}of\text{-}set(U::'a,V))\ \&$
$(\forall\ Y\ X\ Vt\ U.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ --> element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,$
$|\ subset\text{-}sets(Y::'a,f14(Y::'a,X,Vt,U)))\ \&$
$(\forall\ Y\ U\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ --> element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,$
$|\ closed(f14(Y::'a,X,Vt,U),X,Vt))\ \&$
$(\forall\ Y\ X\ Vt\ U.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ \&\ element\text{-}of\text{-}set(U::'a,f14(Y::'a,X,Vt,U))$
$--> element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt)))\ \&$

$(\forall\ U\ Y\ X\ Vt.\ neighborhood(U::'a,Y,X,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Y\ U\ X\ Vt.\ neighborhood(U::'a,Y,X,Vt)\ -->\ open(U::'a,X,Vt))\ \&$
$(\forall\ X\ Vt\ Y\ U.\ neighborhood(U::'a,Y,X,Vt)\ -->\ element\text{-}of\text{-}set(Y::'a,U))\ \&$
$(\forall\ X\ Vt\ Y\ U.\ topological\text{-}space(X::'a,Vt)\ \&\ open(U::'a,X,Vt)\ \&\ element\text{-}of\text{-}set(Y::'a,U)$
$-->\ neighborhood(U::'a,Y,X,Vt))\ \&$
$(\forall\ Z\ Y\ X\ Vt.\ limit\text{-}point(Z::'a,Y,X,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Z\ Vt\ Y\ X.\ limit\text{-}point(Z::'a,Y,X,Vt)\ -->\ subset\text{-}sets(Y::'a,X))\ \&$
$(\forall\ Z\ X\ Vt\ U\ Y.\ limit\text{-}point(Z::'a,Y,X,Vt)\ \&\ neighborhood(U::'a,Z,X,Vt)\ -->$
$element\text{-}of\text{-}set(f15(Z::'a,Y,X,Vt,U),intersection\text{-}of\text{-}sets(U::'a,Y)))\ \&$
$(\forall\ Y\ X\ Vt\ U\ Z.\ \sim(limit\text{-}point(Z::'a,Y,X,Vt)\ \&\ neighborhood(U::'a,Z,X,Vt)\ \&$
$eq\text{-}p(f15(Z::'a,Y,X,Vt,U),Z)))\ \&$
$(\forall\ Y\ Z\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ -->\ limit\text{-}point(Z::'a,Y,X,Vt)$
$|\ neighborhood(f16(Z::'a,Y,X,Vt),Z,X,Vt))\ \&$
$(\forall\ X\ Vt\ Y\ Uu16\ Z.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Y::'a,X)\ \&\ element\text{-}of\text{-}set(Uu16::'a,intersection$
$-->\ limit\text{-}point(Z::'a,Y,X,Vt)\ |\ eq\text{-}p(Uu16::'a,Z))\ \&$
$(\forall\ U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,boundary(Y::'a,X,Vt))\ -->\ topological\text{-}space(X::'a,Vt))$
$\&$
$(\forall\ U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,boundary(Y::'a,X,Vt))\ -->\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt)))$
$\&$
$(\forall\ U\ Y\ X\ Vt.\ element\text{-}of\text{-}set(U::'a,boundary(Y::'a,X,Vt))\ -->\ element\text{-}of\text{-}set(U::'a,closure(relative\text{-}comple$
$\&$
$(\forall\ U\ Y\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ element\text{-}of\text{-}set(U::'a,closure(Y::'a,X,Vt))$
$\&\ element\text{-}of\text{-}set(U::'a,closure(relative\text{-}complement\text{-}sets(Y::'a,X),X,Vt))\ -->\ element\text{-}of\text{-}set(U::'a,boundar$
$\&$
$(\forall\ X\ Vt.\ hausdorff(X::'a,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ X\text{-}2\ X\text{-}1\ X\ Vt.\ hausdorff(X::'a,Vt)\ \&\ element\text{-}of\text{-}set(X\text{-}1::'a,X)\ \&\ element\text{-}of\text{-}set(X\text{-}2::'a,X)$
$-->\ eq\text{-}p(X\text{-}1::'a,X\text{-}2)\ |\ neighborhood(f17(X::'a,Vt,X\text{-}1,X\text{-}2),X\text{-}1,X,Vt))\ \&$
$(\forall\ X\text{-}1\ X\text{-}2\ X\ Vt.\ hausdorff(X::'a,Vt)\ \&\ element\text{-}of\text{-}set(X\text{-}1::'a,X)\ \&\ element\text{-}of\text{-}set(X\text{-}2::'a,X)$
$-->\ eq\text{-}p(X\text{-}1::'a,X\text{-}2)\ |\ neighborhood(f18(X::'a,Vt,X\text{-}1,X\text{-}2),X\text{-}2,X,Vt))\ \&$
$(\forall\ X\ Vt\ X\text{-}1\ X\text{-}2.\ hausdorff(X::'a,Vt)\ \&\ element\text{-}of\text{-}set(X\text{-}1::'a,X)\ \&\ element\text{-}of\text{-}set(X\text{-}2::'a,X)$
$-->\ eq\text{-}p(X\text{-}1::'a,X\text{-}2)\ |\ disjoint\text{-}s(f17(X::'a,Vt,X\text{-}1,X\text{-}2),f18(X::'a,Vt,X\text{-}1,X\text{-}2)))$
$\&$
$(\forall\ Vt\ X.\ topological\text{-}space(X::'a,Vt)\ -->\ hausdorff(X::'a,Vt)\ |\ element\text{-}of\text{-}set(f19(X::'a,Vt),X))$
$\&$
$(\forall\ Vt\ X.\ topological\text{-}space(X::'a,Vt)\ -->\ hausdorff(X::'a,Vt)\ |\ element\text{-}of\text{-}set(f20(X::'a,Vt),X))$
$\&$
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ eq\text{-}p(f19(X::'a,Vt),f20(X::'a,Vt))\ -->$
$hausdorff(X::'a,Vt))\ \&$
$(\forall\ X\ Vt\ Uu19\ Uu20.\ topological\text{-}space(X::'a,Vt)\ \&\ neighborhood(Uu19::'a,f19(X::'a,Vt),X,Vt)$
$\&\ neighborhood(Uu20::'a,f20(X::'a,Vt),X,Vt)\ \&\ disjoint\text{-}s(Uu19::'a,Uu20)\ -->$
$hausdorff(X::'a,Vt))\ \&$
$(\forall\ Va1\ Va2\ X\ Vt.\ separation(Va1::'a,Va2,X,Vt)\ -->\ topological\text{-}space(X::'a,Vt))$
$\&$
$(\forall\ Va2\ X\ Vt\ Va1.\ \sim(separation(Va1::'a,Va2,X,Vt)\ \&\ equal\text{-}sets(Va1::'a,empty\text{-}set)))$
$\&$
$(\forall\ Va1\ X\ Vt\ Va2.\ \sim(separation(Va1::'a,Va2,X,Vt)\ \&\ equal\text{-}sets(Va2::'a,empty\text{-}set)))$
$\&$
$(\forall\ Va2\ X\ Va1\ Vt.\ separation(Va1::'a,Va2,X,Vt)\ -->\ element\text{-}of\text{-}collection(Va1::'a,Vt))$
$\&$

$(\forall\ Va1\ X\ Va2\ Vt.\ separation(Va1::'a,Va2,X,Vt)\ -->\ element\text{-}of\text{-}collection(Va2::'a,Vt))$
&
$(\forall\ Vt\ Va1\ Va2\ X.\ separation(Va1::'a,Va2,X,Vt)\ -->\ equal\text{-}sets(union\text{-}of\text{-}sets(Va1::'a,Va2),X))$
&
$(\forall\ X\ Vt\ Va1\ Va2.\ separation(Va1::'a,Va2,X,Vt)\ -->\ disjoint\text{-}s(Va1::'a,Va2))$
&
$(\forall\ Vt\ X\ Va1\ Va2.\ topological\text{-}space(X::'a,Vt)\ \&\ element\text{-}of\text{-}collection(Va1::'a,Vt)$
$\&\ element\text{-}of\text{-}collection(Va2::'a,Vt)\ \&\ equal\text{-}sets(union\text{-}of\text{-}sets(Va1::'a,Va2),X)\ \&$
$disjoint\text{-}s(Va1::'a,Va2)\ -->\ separation(Va1::'a,Va2,X,Vt)\ |\ equal\text{-}sets(Va1::'a,empty\text{-}set)$
$|\ equal\text{-}sets(Va2::'a,empty\text{-}set))\ \&$
$(\forall\ X\ Vt.\ connected\text{-}space(X::'a,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Va1\ Va2\ X\ Vt.\ \sim(connected\text{-}space(X::'a,Vt)\ \&\ separation(Va1::'a,Va2,X,Vt)))$
&
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ -->\ connected\text{-}space(X::'a,Vt)\ |\ separation(f21(X::'a,Vt),f22(X::'a,Vt),X,Vt))\ \&$
$(\forall\ Va\ X\ Vt.\ connected\text{-}set(Va::'a,X,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Vt\ Va\ X.\ connected\text{-}set(Va::'a,X,Vt)\ -->\ subset\text{-}sets(Va::'a,X))\ \&$
$(\forall\ X\ Vt\ Va.\ connected\text{-}set(Va::'a,X,Vt)\ -->\ connected\text{-}space(Va::'a,subspace\text{-}topology(X::'a,Vt,Va)))$
&
$(\forall\ X\ Vt\ Va.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Va::'a,X)\ \&\ connected\text{-}space(Va::'a,subspace\text{-}topology(X$
$-->\ connected\text{-}set(Va::'a,X,Vt))\ \&$
$(\forall\ Vf\ X\ Vt.\ open\text{-}covering(Vf::'a,X,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ X\ Vf\ Vt.\ open\text{-}covering(Vf::'a,X,Vt)\ -->\ subset\text{-}collections(Vf::'a,Vt))\ \&$
$(\forall\ Vt\ Vf\ X.\ open\text{-}covering(Vf::'a,X,Vt)\ -->\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X))$
&
$(\forall\ Vt\ Vf\ X.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}collections(Vf::'a,Vt)\ \&\ equal\text{-}sets(union\text{-}of\text{-}members(Vf),X$
$-->\ open\text{-}covering(Vf::'a,X,Vt))\ \&$
$(\forall\ X\ Vt.\ compact\text{-}space(X::'a,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ X\ Vt\ Vf1.\ compact\text{-}space(X::'a,Vt)\ \&\ open\text{-}covering(Vf1::'a,X,Vt)\ -->\ finite'(f23(X::'a,Vt,Vf1)))\ \&$
$(\forall\ X\ Vt\ Vf1.\ compact\text{-}space(X::'a,Vt)\ \&\ open\text{-}covering(Vf1::'a,X,Vt)\ -->\ subset\text{-}collections(f23(X::'a,Vt,V$
&
$(\forall\ Vf1\ X\ Vt.\ compact\text{-}space(X::'a,Vt)\ \&\ open\text{-}covering(Vf1::'a,X,Vt)\ -->\ open\text{-}covering(f23(X::'a,Vt,Vf1)$
&
$(\forall\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ -->\ compact\text{-}space(X::'a,Vt)\ |\ open\text{-}covering(f24(X::'a,Vt),X,Vt))$
&
$(\forall\ Uu24\ X\ Vt.\ topological\text{-}space(X::'a,Vt)\ \&\ finite'(Uu24)\ \&\ subset\text{-}collections(Uu24::'a,f24(X::'a,Vt))$
$\&\ open\text{-}covering(Uu24::'a,X,Vt)\ -->\ compact\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Va\ X\ Vt.\ compact\text{-}set(Va::'a,X,Vt)\ -->\ topological\text{-}space(X::'a,Vt))\ \&$
$(\forall\ Vt\ Va\ X.\ compact\text{-}set(Va::'a,X,Vt)\ -->\ subset\text{-}sets(Va::'a,X))\ \&$
$(\forall\ X\ Vt\ Va.\ compact\text{-}set(Va::'a,X,Vt)\ -->\ compact\text{-}space(Va::'a,subspace\text{-}topology(X::'a,Vt,Va)))$
&
$(\forall\ X\ Vt\ Va.\ topological\text{-}space(X::'a,Vt)\ \&\ subset\text{-}sets(Va::'a,X)\ \&\ compact\text{-}space(Va::'a,subspace\text{-}topology(X:$
$-->\ compact\text{-}set(Va::'a,X,Vt))\ \&$
$(basis(cx::'a,f))\ \&$
$(\forall\ U.\ element\text{-}of\text{-}collection(U::'a,top\text{-}of\text{-}basis(f)))\ \&$
$(\forall\ V.\ element\text{-}of\text{-}collection(V::'a,top\text{-}of\text{-}basis(f)))\ \&$
$(\forall\ U\ V.\ \sim element\text{-}of\text{-}collection(intersection\text{-}of\text{-}sets(U::'a,V),top\text{-}of\text{-}basis(f)))\ -->$
$False$

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

38 Examples and regression tests for automated termination proofs

theory *Termination*
imports *Main Multiset*
begin

The *fun* command uses the method *lexicographic-order* by default.

38.1 Trivial examples

fun *identity* :: *nat* ⇒ *nat*
where
identity n = *n*

fun *yaSuc* :: *nat* ⇒ *nat*
where
yaSuc 0 = *0*
| *yaSuc* (*Suc n*) = *Suc* (*yaSuc n*)

38.2 Examples on natural numbers

```
fun bin :: (nat * nat) ⇒ nat
where
  bin (0, 0) = 1
| bin (Suc n, 0) = 0
| bin (0, Suc m) = 0
| bin (Suc n, Suc m) = bin (n, m) + bin (Suc n, m)


fun t :: (nat * nat) ⇒ nat
where
  t (0,n) = 0
| t (n,0) = 0
| t (Suc n, Suc m) = (if (n mod 2 = 0) then (t (Suc n, m)) else (t (n, Suc m)))


fun k :: (nat * nat) * (nat * nat) ⇒ nat
where
  k ((0,0),(0,0)) = 0
| k ((Suc z, y), (u,v)) = k((z, y), (u, v))
| k ((0, Suc y), (u,v)) = k((1, y), (u, v))
| k ((0,0), (Suc u, v)) = k((1, 1), (u, v))
| k ((0,0), (0, Suc v)) = k((1,1), (1,v))


fun gcd2 :: nat ⇒ nat ⇒ nat
where
  gcd2 x 0 = x
| gcd2 0 y = y
| gcd2 (Suc x) (Suc y) = (if x < y then gcd2 (Suc x) (y − x)
                                   else gcd2 (x − y) (Suc y))

fun ack :: (nat * nat) ⇒ nat
where
  ack (0, m) = Suc m
| ack (Suc n, 0) = ack(n, 1)
| ack (Suc n, Suc m) = ack (n, ack (Suc n, m))


fun greedy :: nat * nat * nat * nat * nat => nat
where
  greedy (Suc a, Suc b, Suc c, Suc d, Suc e) =
  (if (a < 10) then greedy (Suc a, Suc b, c, d + 2, Suc e) else
  (if (a < 20) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 30) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 40) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 50) then greedy (Suc a, b, Suc c, d, Suc e) else
  (if (a < 60) then greedy (a, Suc b, Suc c, d, Suc e) else
  (if (a < 70) then greedy (a, Suc b, Suc c, d, Suc e) else
```

```
  (if (a < 80) then greedy (a, Suc b, Suc c, d, Suc e) else
  (if (a < 90) then greedy (Suc a, Suc b, Suc c, d, e) else
  greedy (Suc a, Suc b, Suc c, d, e)))))))))
| greedy (a, b, c, d, e) = 0


fun blowup :: nat => nat => nat => nat => nat => nat => nat => nat =>
nat => nat
where
  blowup 0 0 0 0 0 0 0 0 0 = 0
| blowup 0 0 0 0 0 0 0 0 (Suc i) = Suc (blowup i i i i i i i i i)
| blowup 0 0 0 0 0 0 0 (Suc h) i = Suc (blowup h h h h h h h h i)
| blowup 0 0 0 0 0 0 (Suc g) h i = Suc (blowup g g g g g g g h i)
| blowup 0 0 0 0 0 (Suc f) g h i = Suc (blowup f f f f f f g h i)
| blowup 0 0 0 0 (Suc e) f g h i = Suc (blowup e e e e e f g h i)
| blowup 0 0 0 (Suc d) e f g h i = Suc (blowup d d d d e f g h i)
| blowup 0 0 (Suc c) d e f g h i = Suc (blowup c c c d e f g h i)
| blowup 0 (Suc b) c d e f g h i = Suc (blowup b b c d e f g h i)
| blowup (Suc a) b c d e f g h i = Suc (blowup a b c d e f g h i)
```

38.3 Simple examples with other datatypes than nat, e.g. trees and lists

```
datatype tree = Node | Branch tree tree

fun g-tree :: tree * tree ⇒ tree
where
  g-tree (Node, Node) = Node
| g-tree (Node, Branch a b) = Branch Node (g-tree (a,b))
| g-tree (Branch a b, Node) = Branch (g-tree (a,Node)) b
| g-tree (Branch a b, Branch c d) = Branch (g-tree (a,c)) (g-tree (b,d))


fun acklist :: 'a list * 'a list ⇒ 'a list
where
  acklist ([], m) = ((hd m)#m)
|  acklist (n#ns, []) = acklist (ns, [n])
|  acklist ((n#ns), (m#ms)) = acklist (ns, acklist ((n#ns), ms))
```

38.4 Examples with mutual recursion

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

```
fun sizechange-f :: 'a list => 'a list => 'a list and
sizechange-g :: 'a list => 'a list => 'a list => 'a list
where
  sizechange-f i x = (if i=[] then x else sizechange-g (tl i) x i)
| sizechange-g a b c = sizechange-f a (b @ c)

fun
  pedal :: nat => nat => nat => nat
and
  coast :: nat => nat => nat => nat
where
  pedal 0 m c = c
| pedal n 0 c = c
| pedal n m c =
    (if n < m then coast (n - 1) (m - 1) (c + m)
              else pedal (n - 1) m (c + m))

| coast n m c =
    (if n < m then coast n (m - 1) (c + n)
              else pedal n m (c + n))
```

38.5 Refined analysis: The *sizechange* method

Unsolvable for *lexicographic-order*

```
function fun1 :: nat * nat ⇒ nat
where
  fun1 (0,0) = 1
| fun1 (0, Suc b) = 0
| fun1 (Suc a, 0) = 0
| fun1 (Suc a, Suc b) = fun1 (b, a)
by pat-completeness auto
termination by sizechange
```

lexicographic-order can do the following, but it is much slower.

```
function
  prod :: nat => nat => nat => nat and
  eprod :: nat => nat => nat => nat and
  oprod :: nat => nat => nat => nat
where
  prod x y z = (if y mod 2 = 0 then eprod x y z else oprod x y z)
| oprod x y z = eprod x (y - 1) (z+x)
| eprod x y z = (if y=0 then z else prod (2*x) (y div 2) z)
by pat-completeness auto
termination by sizechange
```

Permutations of arguments:

```
function perm :: nat ⇒ nat ⇒ nat ⇒ nat
where
```

perm m n r = (if r > 0 then perm m (r − 1) n
else if n > 0 then perm r (n − 1) m
else m)
by *auto*
termination by *sizechange*

Artificial examples and regression tests:

function
fun2 :: nat ⇒ nat ⇒ nat ⇒ nat
where
fun2 x y z =
(if x > 1000 ∧ z > 0 then
fun2 (min x y) y (z − 1)
else if y > 0 ∧ x > 100 then
fun2 x (y − 1) (2 ∗ z)
else if z > 0 then
fun2 (min y (z − 1)) x x
else
0
)
by *pat-completeness auto*
termination by *sizechange* — requires Multiset

end

39 Coherent Logic Problems

theory *Coherent* **imports** *Main* **begin**

39.1 Equivalence of two versions of Pappus' Axiom

no-notation
comp (**infixl** *o 55*) **and**
rel-comp (**infixr** *O 75*)

lemma *p1p2*:
assumes
col a b c l ∧ col d e f m
col b f g n ∧ col c e g o
col b d h p ∧ col a e h q
col c d i r ∧ col a f i s
el n o ⟹ goal
el p q ⟹ goal
el s r ⟹ goal
⋀A. el A A ⟹ pl g A ⟹ pl h A ⟹ pl i A ⟹ goal
⋀A B C D. col A B C D ⟹ pl A D
⋀A B C D. col A B C D ⟹ pl B D
⋀A B C D. col A B C D ⟹ pl C D

$\bigwedge A\ B.\ pl\ A\ B \Longrightarrow ep\ A\ A$
$\bigwedge A\ B.\ ep\ A\ B \Longrightarrow ep\ B\ A$
$\bigwedge A\ B\ C.\ ep\ A\ B \Longrightarrow ep\ B\ C \Longrightarrow ep\ A\ C$
$\bigwedge A\ B.\ pl\ A\ B \Longrightarrow el\ B\ B$
$\bigwedge A\ B.\ el\ A\ B \Longrightarrow el\ B\ A$
$\bigwedge A\ B\ C.\ el\ A\ B \Longrightarrow el\ B\ C \Longrightarrow el\ A\ C$
$\bigwedge A\ B\ C.\ ep\ A\ B \Longrightarrow pl\ B\ C \Longrightarrow pl\ A\ C$
$\bigwedge A\ B\ C.\ pl\ A\ B \Longrightarrow el\ B\ C \Longrightarrow pl\ A\ C$
$\bigwedge A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ L\ M\ N\ O\ P\ Q.$
$col\ A\ B\ C\ D \Longrightarrow col\ E\ F\ G\ H \Longrightarrow col\ B\ G\ I\ J \Longrightarrow col\ C\ F\ I\ K \Longrightarrow$
$col\ B\ E\ L\ M \Longrightarrow col\ A\ F\ L\ N \Longrightarrow col\ C\ E\ O\ P \Longrightarrow col\ A\ G\ O\ Q \Longrightarrow$
$(\exists\ R.\ col\ I\ L\ O\ R) \vee pl\ A\ H \vee pl\ B\ H \vee pl\ C\ H \vee pl\ E\ D \vee pl\ F\ D \vee pl\ G\ D$
$\bigwedge A\ B\ C\ D.\ pl\ A\ B \Longrightarrow pl\ A\ C \Longrightarrow pl\ D\ B \Longrightarrow pl\ D\ C \Longrightarrow ep\ A\ D \vee el\ B\ C$
$\bigwedge A\ B.\ ep\ A\ A \Longrightarrow ep\ B\ B \Longrightarrow \exists\, C.\ pl\ A\ C \wedge pl\ B\ C$
shows *goal* **using** *assms*
by *coherent*

lemma *p2p1*:
assumes
$col\ a\ b\ c\ l \wedge col\ d\ e\ f\ m$
$col\ b\ f\ g\ n \wedge col\ c\ e\ g\ o$
$col\ b\ d\ h\ p \wedge col\ a\ e\ h\ q$
$col\ c\ d\ i\ r \wedge col\ a\ f\ i\ s$
$pl\ a\ m \Longrightarrow goal$
$pl\ b\ m \Longrightarrow goal$
$pl\ c\ m \Longrightarrow goal$
$pl\ d\ l \Longrightarrow goal$
$pl\ e\ l \Longrightarrow goal$
$pl\ f\ l \Longrightarrow goal$
$\bigwedge A.\ pl\ g\ A \Longrightarrow pl\ h\ A \Longrightarrow pl\ i\ A \Longrightarrow goal$
$\bigwedge A\ B\ C\ D.\ col\ A\ B\ C\ D \Longrightarrow pl\ A\ D$
$\bigwedge A\ B\ C\ D.\ col\ A\ B\ C\ D \Longrightarrow pl\ B\ D$
$\bigwedge A\ B\ C\ D.\ col\ A\ B\ C\ D \Longrightarrow pl\ C\ D$
$\bigwedge A\ B.\ pl\ A\ B \Longrightarrow ep\ A\ A$
$\bigwedge A\ B.\ ep\ A\ B \Longrightarrow ep\ B\ A$
$\bigwedge A\ B\ C.\ ep\ A\ B \Longrightarrow ep\ B\ C \Longrightarrow ep\ A\ C$
$\bigwedge A\ B.\ pl\ A\ B \Longrightarrow el\ B\ B$
$\bigwedge A\ B.\ el\ A\ B \Longrightarrow el\ B\ A$
$\bigwedge A\ B\ C.\ el\ A\ B \Longrightarrow el\ B\ C \Longrightarrow el\ A\ C$
$\bigwedge A\ B\ C.\ ep\ A\ B \Longrightarrow pl\ B\ C \Longrightarrow pl\ A\ C$
$\bigwedge A\ B\ C.\ pl\ A\ B \Longrightarrow el\ B\ C \Longrightarrow pl\ A\ C$
$\bigwedge A\ B\ C\ D\ E\ F\ G\ H\ I\ J\ K\ L\ M\ N\ O\ P\ Q.$
$col\ A\ B\ C\ J \Longrightarrow col\ D\ E\ F\ K \Longrightarrow col\ B\ F\ G\ L \Longrightarrow col\ C\ E\ G\ M \Longrightarrow$
$col\ B\ D\ H\ N \Longrightarrow col\ A\ E\ H\ O \Longrightarrow col\ C\ D\ I\ P \Longrightarrow col\ A\ F\ I\ Q \Longrightarrow$
$(\exists\ R.\ col\ G\ H\ I\ R) \vee el\ L\ M \vee el\ N\ O \vee el\ P\ Q$
$\bigwedge A\ B\ C\ D.\ pl\ C\ A \Longrightarrow pl\ C\ B \Longrightarrow pl\ D\ A \Longrightarrow pl\ D\ B \Longrightarrow ep\ C\ D \vee el\ A\ B$
$\bigwedge A\ B\ C.\ ep\ A\ A \Longrightarrow ep\ B\ B \Longrightarrow \exists\, C.\ pl\ A\ C \wedge pl\ B\ C$
shows *goal* **using** *assms*
by *coherent*

39.2 Preservation of the Diamond Property under reflexive closure

lemma *diamond*:
assumes
reflexive-rewrite a b reflexive-rewrite a c
$\bigwedge A$. *reflexive-rewrite b A* $\Longrightarrow$ *reflexive-rewrite c A* $\Longrightarrow$ *goal*
$\bigwedge A$. *equalish A A*
$\bigwedge A\ B$. *equalish A B* $\Longrightarrow$ *equalish B A*
$\bigwedge A\ B\ C$. *equalish A B* $\Longrightarrow$ *reflexive-rewrite B C* $\Longrightarrow$ *reflexive-rewrite A C*
$\bigwedge A\ B$. *equalish A B* $\Longrightarrow$ *reflexive-rewrite A B*
$\bigwedge A\ B$. *rewrite A B* $\Longrightarrow$ *reflexive-rewrite A B*
$\bigwedge A\ B$. *reflexive-rewrite A B* $\Longrightarrow$ *equalish A B* $\vee$ *rewrite A B*
$\bigwedge A\ B\ C$. *rewrite A B* $\Longrightarrow$ *rewrite A C* $\Longrightarrow$ $\exists D$. *rewrite B D* $\wedge$ *rewrite C D*
shows *goal* **using** *assms*
by *coherent*

end

40 Some examples for Presburger Arithmetic

theory *PresburgerEx*
imports *Presburger*
begin

lemma $\bigwedge m\ n\ ja\ ia.\ [\![\neg\ m \leq j;\ \neg\ (n::nat) \leq i;\ (e::nat) \neq 0;\ Suc\ j \leq ja]\!] \Longrightarrow \exists m.$
$\forall ja\ ia.\ m \leq ja \longrightarrow (if\ j = ja \wedge i = ia\ then\ e\ else\ 0) = 0$ **by** *presburger*
lemma $(0::nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
by *presburger*
lemma $(0::nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
by *presburger*

theorem $(\forall (y::int).\ 3\ dvd\ y) ==> \forall (x::int).\ b < x\ --> a \leq x$
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 $\forall (x::nat).\ \exists (y::nat).\ (0::nat) \leq 5\ --> y = 5 + x$
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 $\forall (x{::}int)\ y.\ x < y \longrightarrow 2 * x + 1 < 2 * y$
by *presburger*

theorem $\forall (x{::}int)\ y.\ 2 * x + 1 \neq 2 * y$
by *presburger*

theorem $\exists (x{::}int)\ y.\ 0 < x\ \&\ 0 \leq y\ \&\ 3 * x - 5 * y = 1$
by *presburger*

theorem $\sim (\exists (x{::}int)\ (y{::}int)\ (z{::}int).\ 4{*}x + (-6{::}int){*}y = 1)$
by *presburger*

theorem $\forall (x{::}int).\ b < x \longrightarrow a \leq x$
apply (*presburger elim*)
oops

theorem $\sim (\exists (x{::}int).\ False)$
by *presburger*

theorem $\forall (x{::}int).\ (a{::}int) < 3 * x \longrightarrow b < 3 * x$
apply (*presburger elim*)
oops

theorem $\forall (x{::}int).\ (2\ dvd\ x) \longrightarrow (\exists (y{::}int).\ x = 2{*}y)$
by *presburger*

theorem $\forall (x{::}int).\ (2\ dvd\ x) \longrightarrow (\exists (y{::}int).\ x = 2{*}y)$
by *presburger*

theorem $\forall (x{::}int).\ (2\ dvd\ x) = (\exists (y{::}int).\ x = 2{*}y)$
by *presburger*

theorem $\forall (x{::}int).\ ((2\ dvd\ x) = (\forall (y{::}int).\ x \neq 2{*}y + 1))$
by *presburger*

theorem $\sim (\forall (x{::}int).$
$\quad ((2\ dvd\ x) = (\forall (y{::}int).\ x \neq 2{*}y{+}1) \mid$
$\quad (\exists (q{::}int)\ (u{::}int)\ i.\ 3{*}i + 2{*}q - u < 17)$
$\quad \longrightarrow 0 < x \mid ((\sim 3\ dvd\ x)\ \&(x + 8 = 0))))$
by *presburger*

theorem $\sim (\forall (i{::}int).\ 4 \leq i \longrightarrow (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$

by *presburger*

theorem $\forall (i::int).\ 8 \leq i \longrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
by *presburger*

theorem $\exists (j::int).\ \forall\, i.\ j \leq i \longrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
by *presburger*

theorem $\sim\ (\forall\, j\ (i::int).\ j \leq i \longrightarrow (\exists\, x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
by *presburger*

Slow: about 5 seconds on a 1.6GHz machine.

theorem $(\exists\, m::nat.\ n = 2 * m) \longrightarrow (n + 1)\ div\ 2 = n\ div\ 2$
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

41 Generic reflection and reification

theory *Reflection*
imports *Main*
uses *reify-data.ML* (*reflection.ML*)
begin

setup $\langle\!\langle$ *Reify-Data.setup* $\rangle\!\rangle$

lemma *ext2*: $(\forall\, x.\ f\ x = g\ x) \Longrightarrow f = g$
by (*blast intro*: *ext*)

use *reflection.ML*

method-setup *reify* = $\langle\!\langle$
Attrib.thms −−

```
    Scan.option (Scan.lift (Args.$$$ () |-- Args.term --| Scan.lift (Args.$$$ )))
>>
  (fn (eqs, to) => fn ctxt => 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);
in
  thms --
  Scan.optional (keyword rulesN |-- thms) [] --
  Scan.option (keyword onlyN |-- Args.term) >>
  (fn ((eqs,ths),to) => fn ctxt =>
    let
      val (ceqs,cths) = Reify-Data.get ctxt
      val corr-thms = ths@cths
      val raw-eqs = eqs@ceqs
   in SIMPLE-METHOD' (Reflection.reflection-tac ctxt corr-thms raw-eqs to) end)
end
\⟩ reflection

end
```

42 Examples for generic reflection and reification

theory *ReflectionEx*
imports *Reflection*
begin

This theory presents two methods: reify and reflection

Consider an HOL type 'a, the structure of which is not recongnisable on the theory level. This is the case of bool, arithmetical terms such as int, real etc ... In order to implement a simplification on terms of type 'a we often need its structure. Traditionnaly such simplifications are written in ML, proofs are synthesized. An other strategy is to declare an HOL-datatype tau and an HOL function (the interpretation) that maps elements of tau to elements of 'a. The functionality of *reify* is to compute a term s::tau, which is the representant of t. For this it needs equations for the interpretation.

NB: All the interpretations supported by *reify* must have the type $'b\ list \Rightarrow tau \Rightarrow 'a$. The method *reify* can also be told which subterm of the current

subgoal should be reified. The general call for *reify* is: *reify eqs* (t), where *eqs* are the defining equations of the interpretation and (t) is an optional parameter which specifies the subterm to which reification should be applied to. If (t) is abscent, *reify* tries to reify the whole subgoal.

The method reflection uses *reify* and has a very similar signature: *reflection corr-thm eqs* (t). Here again *eqs* and (t) are as described above and *corr-thm* is a thorem proving $I\ vs\ (f\ t) = I\ vs\ t$. We assume that I is the interpretation and f is some useful and executable simplification of type $tau \Rightarrow tau$. The method *reflection* applies reification and hence the theorem $t = I\ xs\ s$ and hence using *corr-thm* derives $t = I\ xs\ (f\ s)$. It then uses normalization by evaluation to prove $f\ s = s'$ which almost finishes the proof of $t = t'$ where $I\ xs\ s' = t'$.

Example 1 : Propositional formulae and NNF.

The type *fm* represents simple propositional formulae:

```
datatype form = TrueF | FalseF | Less nat nat |
              And form form | Or form form | Neg form | ExQ form

fun interp :: form ⇒ ('a::ord) list ⇒ bool where
interp TrueF e = True |
interp FalseF e = False |
interp (Less i j) e = (e!i < e!j) |
interp (And f1 f2) e = (interp f1 e & interp f2 e) |
interp (Or f1 f2) e = (interp f1 e | interp f2 e) |
interp (Neg f) e = (~ interp f e) |
interp (ExQ f) e = (EX x. interp f (x#e))

lemmas interp-reify-eqs = interp.simps
declare interp-reify-eqs[reify]

lemma EX x. x < y & x < z
  apply (reify )
  oops

datatype fm = And fm fm | Or fm fm | Imp fm fm | Iff fm fm | NOT fm | At
nat

consts Ifm :: fm ⇒ bool list ⇒ bool
primrec
  Ifm (At n) vs = vs!n
  Ifm (And p q) vs = (Ifm p vs ∧ Ifm q vs)
  Ifm (Or p q) vs = (Ifm p vs ∨ Ifm q vs)
  Ifm (Imp p q) vs = (Ifm p vs ⟶ Ifm q vs)
  Ifm (Iff p q) vs = (Ifm p vs = Ifm q vs)
  Ifm (NOT p) vs = (¬ (Ifm p vs))

lemma Q ⟶ (D & F & ((~ D) & (~ F)))
```

apply (*reify Ifm.simps*)
oops

Method *reify* maps a bool to an fm. For this it needs the semantics of fm, i.e. the rewrite rules in *Ifm.simps*.

lemma $Q \longrightarrow (D \ \& \ F \ \& \ ((\sim D) \ \& \ (\sim F)))$
apply (*reify Ifm.simps* $(((\sim D) \ \& \ (\sim F))))$
oops

Let's perform NNF. This is a version that tends to generate disjunctions

consts *fmsize* :: $fm \Rightarrow nat$
primrec
$fmsize\ (At\ n) = 1$
$fmsize\ (NOT\ p) = 1 + fmsize\ p$
$fmsize\ (And\ p\ q) = 1 + fmsize\ p + fmsize\ q$
$fmsize\ (Or\ p\ q) = 1 + fmsize\ p + fmsize\ q$
$fmsize\ (Imp\ p\ q) = 2 + fmsize\ p + fmsize\ q$
$fmsize\ (Iff\ p\ q) = 2 + 2* fmsize\ p + 2* fmsize\ q$

consts *nnf* :: $fm \Rightarrow fm$
recdef *nnf measure fmsize*
$nnf\ (At\ n) = At\ n$
$nnf\ (And\ p\ q) = And\ (nnf\ p)\ (nnf\ q)$
$nnf\ (Or\ p\ q) = Or\ (nnf\ p)\ (nnf\ q)$
$nnf\ (Imp\ p\ q) = Or\ (nnf\ (NOT\ p))\ (nnf\ q)$
$nnf\ (Iff\ p\ q) = Or\ (And\ (nnf\ p)\ (nnf\ q))\ (And\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q)))$
$nnf\ (NOT\ (And\ p\ q)) = Or\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q))$
$nnf\ (NOT\ (Or\ p\ q)) = And\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q))$
$nnf\ (NOT\ (Imp\ p\ q)) = And\ (nnf\ p)\ (nnf\ (NOT\ q))$
$nnf\ (NOT\ (Iff\ p\ q)) = Or\ (And\ (nnf\ p)\ (nnf\ (NOT\ q)))\ (And\ (nnf\ (NOT\ p))$
$(nnf\ q))$
$nnf\ (NOT\ (NOT\ p)) = nnf\ p$
$nnf\ (NOT\ p) = NOT\ p$

The correctness theorem of nnf: it preserves the semantics of fm

lemma *nnf*[*reflection*]: $Ifm\ (nnf\ p)\ vs = Ifm\ p\ vs$
by (*induct p rule*: *nnf.induct*) *auto*

Now let's perform NNF using our *nnf* function defined above. First to the whole subgoal.

lemma $(\neg\ (A = B)) \wedge (B \longrightarrow (A \neq (B \mid C \wedge (B \longrightarrow A \mid D)))) \longrightarrow A \vee B \wedge D$
apply (*reflection Ifm.simps*)
oops

Now we specify on which subterm it should be applied

lemma $(\neg\ (A = B)) \wedge (B \longrightarrow (A \neq (B \mid C \wedge (B \longrightarrow A \mid D)))) \longrightarrow A \vee B \wedge D$
apply (*reflection Ifm.simps only*: $(B \mid C \wedge (B \longrightarrow A \mid D)))$
oops

The type *num* reflects linear expressions over natural number

datatype *num* = *C nat* | *Add num num* | *Mul nat num* | *Var nat* | *CN nat nat num*

This is just technical to make recursive definitions easier.

consts *num-size* :: *num* ⇒ *nat*
primrec
num-size (*C c*) = *1*
num-size (*Var n*) = *1*
num-size (*Add a b*) = *1* + *num-size a* + *num-size b*
num-size (*Mul c a*) = *1* + *num-size a*
num-size (*CN n c a*) = *4* + *num-size a*

The semantics of num

consts *Inum*:: *num* ⇒ *nat list* ⇒ *nat*
primrec
Inum-C : *Inum* (*C i*) *vs* = *i*
Inum-Var: *Inum* (*Var n*) *vs* = *vs*!*n*
Inum-Add: *Inum* (*Add s t*) *vs* = *Inum s vs* + *Inum t vs*
Inum-Mul: *Inum* (*Mul c t*) *vs* = *c* ∗ *Inum t vs*
Inum-CN : *Inum* (*CN n c t*) *vs* = *c*∗(*vs*!*n*) + *Inum t vs*

Let's reify some nat expressions ...

lemma *4* ∗ (*2*∗*x* + (*y*::*nat*)) + *f a* ≠ *0*
apply (*reify Inum.simps* (*4* ∗ (*2*∗*x* + (*y*::*nat*)) + *f a*))
oops

We're in a bad situation!! x, y and f a have been recongnized as a constants, which is correct but does not correspond to our intuition of the constructor C. It should encapsulate constants, i.e. numbers, i.e. numerals.

So let's leave the *Inum-C* equation at the end and see what happens ...

lemma *4* ∗ (*2*∗*x* + (*y*::*nat*)) ≠ *0*
apply (*reify Inum-Var Inum-Add Inum-Mul Inum-CN Inum-C* (*4* ∗ (*2*∗*x* + (*y*::*nat*))))
oops

Hmmm let's specialize *Inum-C* with numerals.

lemma *Inum-number*: *Inum* (*C* (*number-of t*)) *vs* = *number-of t* **by** *simp*
lemmas *Inum-eqs* = *Inum-Var Inum-Add Inum-Mul Inum-CN Inum-number*

Second attempt

lemma *1* ∗ (*2*∗*x* + (*y*::*nat*)) ≠ *0*
apply (*reify Inum-eqs* (*1* ∗ (*2*∗*x* + (*y*::*nat*))))
oops

That was fine, so let's try another one ...

lemma *1 * (2* x + (y::nat) + 0 + 1) ≠ 0*
  apply (*reify Inum-eqs (1 * (2*x + (y::nat) + 0 + 1))*)
oops

Oh!! 0 is not a variable ... Oh! 0 is not a *number-of* ... thing. The same for 1. So let's add those equations too

lemma *Inum-01*: *Inum (C 0) vs = 0 Inum (C 1) vs = 1 Inum (C(Suc n)) vs = Suc n*
  by *simp+*

lemmas *Inum-eqs'= Inum-eqs Inum-01*

Third attempt:

lemma *1 * (2*x + (y::nat) + 0 + 1) ≠ 0*
  apply (*reify Inum-eqs' (1 * (2*x + (y::nat) + 0 + 1))*)
oops

Okay, let's try reflection. Some simplifications on num follow. You can skim until the main theorem *linum*

consts *lin-add* :: *num × num ⇒ num*
recdef *lin-add measure* (*λ(x,y). ((size x) + (size y))*)
  lin-add (CN n1 c1 r1,CN n2 c2 r2) =
  (if n1=n2 then
  (let c = c1 + c2
  in (if c=0 then lin-add(r1,r2) else CN n1 c (lin-add (r1,r2))))
  else if n1 ≤ n2 then (CN n1 c1 (lin-add (r1,CN n2 c2 r2)))
  else (CN n2 c2 (lin-add (CN n1 c1 r1,r2))))
  lin-add (CN n1 c1 r1,t) = CN n1 c1 (lin-add (r1, t))
  lin-add (t,CN n2 c2 r2) = CN n2 c2 (lin-add (t,r2))
  lin-add (C b1, C b2) = C (b1+b2)
  lin-add (a,b) = Add a b
lemma *lin-add*: *Inum (lin-add (t,s)) bs = Inum (Add t s) bs*
apply (*induct t s rule*: *lin-add.induct*, *simp-all add*: *Let-def*)
apply (*case-tac c1+c2 = 0,case-tac n1 ≤ n2*, *simp-all*)
by (*case-tac n1 = n2*, *simp-all add*: *algebra-simps*)

consts *lin-mul* :: *num ⇒ nat ⇒ num*
recdef *lin-mul measure size*
  *lin-mul (C j) = (λ i. C (i*j))*
  *lin-mul (CN n c a) = (λ i. if i=0 then (C 0) else CN n (i*c) (lin-mul a i))*
  lin-mul t = (λ i. Mul i t)

lemma *lin-mul*: *Inum (lin-mul t i) bs = Inum (Mul i t) bs*
by (*induct t arbitrary*: *i rule*: *lin-mul.induct*, *auto simp add*: *algebra-simps*)

consts *linum*:: *num ⇒ num*
recdef *linum measure num-size*
  linum (C b) = C b

linum (Var n) = CN n 1 (C 0)
linum (Add t s) = lin-add (linum t, linum s)
linum (Mul c t) = lin-mul (linum t) c
linum (CN n c t) = lin-add (linum (Mul c (Var n)),linum t)

lemma *linum[reflection] : Inum (linum t) bs = Inum t bs*
by (*induct t rule*: *linum.induct*, *simp-all add*: *lin-mul lin-add*)

Now we can use linum to simplify nat terms using reflection

lemma *(Suc (Suc 1)) * (x + (Suc 1)*y) = 3*x + 6*y*
apply (*reflection Inum-eqs' only*: *(Suc (Suc 1)) * (x + (Suc 1)*y)*)
oops

Let's lift this to formulae and see what happens

datatype *aform = Lt num num | Eq num num | Ge num num | NEq num num*
|
Conj aform aform | Disj aform aform | NEG aform | T | F
consts *linaformsize*:: *aform ⇒ nat*
recdef *linaformsize measure size*
linaformsize T = 1
linaformsize F = 1
linaformsize (Lt a b) = 1
linaformsize (Ge a b) = 1
linaformsize (Eq a b) = 1
linaformsize (NEq a b) = 1
linaformsize (NEG p) = 2 + linaformsize p
linaformsize (Conj p q) = 1 + linaformsize p + linaformsize q
linaformsize (Disj p q) = 1 + linaformsize p + linaformsize q

consts *is-aform* :: *aform => nat list => bool*
primrec
is-aform T vs = True
is-aform F vs = False
is-aform (Lt a b) vs = (Inum a vs < Inum b vs)
is-aform (Eq a b) vs = (Inum a vs = Inum b vs)
is-aform (Ge a b) vs = (Inum a vs ≥ Inum b vs)
is-aform (NEq a b) vs = (Inum a vs ≠ Inum b vs)
is-aform (NEG p) vs = (¬ (is-aform p vs))
is-aform (Conj p q) vs = (is-aform p vs ∧ is-aform q vs)
is-aform (Disj p q) vs = (is-aform p vs ∨ is-aform q vs)

Let's reify and do reflection

lemma *(3::nat)*x + t < 0 ∧ (2 * x + y ≠ 17)*
apply (*reify Inum-eqs' is-aform.simps*)
oops

Note that reification handles several interpretations at the same time

lemma *(3::nat)*x + t < 0 & x*x + t*x + 3 + 1 = z*t*4*z | x + x + 1 < 0*

apply (*reflection Inum-eqs′ is-aform.simps only*:*x* + *x* + *1*)
oops

For reflection we now define a simple transformation on aform: NNF + linum on atoms

consts *linaform*:: *aform* ⇒ *aform*
recdef *linaform measure linaformsize*
linaform (*Lt s t*) = *Lt* (*linum s*) (*linum t*)
linaform (*Eq s t*) = *Eq* (*linum s*) (*linum t*)
linaform (*Ge s t*) = *Ge* (*linum s*) (*linum t*)
linaform (*NEq s t*) = *NEq* (*linum s*) (*linum t*)
linaform (*Conj p q*) = *Conj* (*linaform p*) (*linaform q*)
linaform (*Disj p q*) = *Disj* (*linaform p*) (*linaform q*)
linaform (*NEG T*) = *F*
linaform (*NEG F*) = *T*
linaform (*NEG* (*Lt a b*)) = *Ge a b*
linaform (*NEG* (*Ge a b*)) = *Lt a b*
linaform (*NEG* (*Eq a b*)) = *NEq a b*
linaform (*NEG* (*NEq a b*)) = *Eq a b*
linaform (*NEG* (*NEG p*)) = *linaform p*
linaform (*NEG* (*Conj p q*)) = *Disj* (*linaform* (*NEG p*)) (*linaform* (*NEG q*))
linaform (*NEG* (*Disj p q*)) = *Conj* (*linaform* (*NEG p*)) (*linaform* (*NEG q*))
linaform p = *p*

lemma *linaform*: *is-aform* (*linaform p*) *vs* = *is-aform p vs*
by (*induct p rule*: *linaform.induct*) (*auto simp add*: *linum*)

lemma (((*Suc*(*Suc* (*Suc 0*)))∗((*x*::*nat*) + (*Suc* (*Suc 0*)))) + (*Suc* (*Suc* (*Suc 0*)))
∗ ((*Suc*(*Suc* (*Suc 0*)))∗((*x*::*nat*) + (*Suc* (*Suc 0*))))< *0*) ∧ (*Suc 0* + *Suc 0*< *0*)
apply (*reflection Inum-eqs′ is-aform.simps rules*: *linaform*)
oops

declare *linaform*[*reflection*]
lemma (((*Suc*(*Suc* (*Suc 0*)))∗((*x*::*nat*) + (*Suc* (*Suc 0*)))) + (*Suc* (*Suc* (*Suc 0*)))
∗ ((*Suc*(*Suc* (*Suc 0*)))∗((*x*::*nat*) + (*Suc* (*Suc 0*))))< *0*) ∧ (*Suc 0* + *Suc 0*< *0*)
apply (*reflection Inum-eqs′ is-aform.simps*)
oops

We now give an example where Interpretaions have 0 or more than only one envornement of different types and show that automatic reification also deals with binding

datatype *rb* = *BC bool*| *BAnd rb rb* | *BOr rb rb*
consts *Irb* :: *rb* ⇒ *bool*
primrec
Irb (*BC p*) = *p*
Irb (*BAnd s t*) = (*Irb s* ∧ *Irb t*)
Irb (*BOr s t*) = (*Irb s* ∨ *Irb t*)

lemma $A \wedge (B \vee D \wedge B) \wedge A \wedge (B \vee D \wedge B) \vee A \wedge (B \vee D \wedge B) \vee A \wedge (B \vee D \wedge B)$
apply (*reify Irb.simps*)
oops

datatype *rint* = *IC int*| *IVar nat* | *IAdd rint rint* | *IMult rint rint* | *INeg rint* | *ISub rint rint*
consts *Irint* :: *rint* ⇒ *int list* ⇒ *int*
primrec
Irint-Var: *Irint* (*IVar n*) *vs* = *vs*!*n*
Irint-Neg: *Irint* (*INeg t*) *vs* = − *Irint t vs*
Irint-Add: *Irint* (*IAdd s t*) *vs* = *Irint s vs* + *Irint t vs*
Irint-Sub: *Irint* (*ISub s t*) *vs* = *Irint s vs* − *Irint t vs*
Irint-Mult: *Irint* (*IMult s t*) *vs* = *Irint s vs* ∗ *Irint t vs*
Irint-C: *Irint* (*IC i*) *vs* = *i*
lemma *Irint-C0*: *Irint* (*IC 0*) *vs* = *0*
by *simp*
lemma *Irint-C1*: *Irint* (*IC 1*) *vs* = *1*
by *simp*
lemma *Irint-Cnumberof*: *Irint* (*IC* (*number-of x*)) *vs* = *number-of x*
by *simp*
lemmas *Irint-simps* = *Irint-Var Irint-Neg Irint-Add Irint-Sub Irint-Mult Irint-C0 Irint-C1 Irint-Cnumberof*
lemma (*3*::*int*) ∗ *x* + *y*∗*y* − *9* + (− *z*) = *0*
apply (*reify Irint-simps* ((*3*::*int*) ∗ *x* + *y*∗*y* − *9* + (− *z*)))
oops
datatype *rlist* = *LVar nat*| *LEmpty*| *LCons rint rlist* | *LAppend rlist rlist*
consts *Irlist* :: *rlist* ⇒ *int list* ⇒ (*int list*) *list* ⇒ (*int list*)
primrec
Irlist (*LEmpty*) *is vs* = []
Irlist (*LVar n*) *is vs* = *vs*!*n*
Irlist (*LCons i t*) *is vs* = ((*Irint i is*)#(*Irlist t is vs*))
Irlist (*LAppend s t*) *is vs* = (*Irlist s is vs*) @ (*Irlist t is vs*)
lemma [(*1*::*int*)] = []
apply (*reify Irlist.simps Irint-simps* ([*1*]:: *int list*))
oops

lemma ([(*3*::*int*) ∗ *x* + *y*∗*y* − *9* + (− *z*)] @ []) @ *xs* = [*y*∗*y* − *z* − *9* + (*3*::*int*) ∗ *x*]
apply (*reify Irlist.simps Irint-simps* (([(*3*::*int*) ∗ *x* + *y*∗*y* − *9* + (− *z*)] @ []) @ *xs*))
oops

datatype *rnat* = *NC nat*| *NVar nat*| *NSuc rnat* | *NAdd rnat rnat* | *NMult rnat rnat* | *NNeg rnat* | *NSub rnat rnat* | *Nlgth rlist*
consts *Irnat* :: *rnat* ⇒ *int list* ⇒ (*int list*) *list* ⇒ *nat list* ⇒ *nat*
primrec
Irnat-Suc: *Irnat* (*NSuc t*) *is ls vs* = *Suc* (*Irnat t is ls vs*)

Irnat-Var: *Irnat* (*NVar n*) *is ls vs* = *vs*!*n*
Irnat-Neg: *Irnat* (*NNeg t*) *is ls vs* = *0*
Irnat-Add: *Irnat* (*NAdd s t*) *is ls vs* = *Irnat s is ls vs* + *Irnat t is ls vs*
Irnat-Sub: *Irnat* (*NSub s t*) *is ls vs* = *Irnat s is ls vs* − *Irnat t is ls vs*
Irnat-Mult: *Irnat* (*NMult s t*) *is ls vs* = *Irnat s is ls vs* ∗ *Irnat t is ls vs*
Irnat-lgth: *Irnat* (*Nlgth rxs*) *is ls vs* = *length* (*Irlist rxs is ls*)
Irnat-C: *Irnat* (*NC i*) *is ls vs* = *i*
lemma *Irnat-C0*: *Irnat* (*NC 0*) *is ls vs* = *0*
by *simp*
lemma *Irnat-C1*: *Irnat* (*NC 1*) *is ls vs* = *1*
by *simp*
lemma *Irnat-Cnumberof*: *Irnat* (*NC* (*number-of x*)) *is ls vs* = *number-of x*
by *simp*
lemmas *Irnat-simps* = *Irnat-Suc Irnat-Var Irnat-Neg Irnat-Add Irnat-Sub Irnat-Mult Irnat-lgth*
Irnat-C0 Irnat-C1 Irnat-Cnumberof
lemma (*Suc n*) ∗ *length* ((*[*(*3*::*int*) ∗ *x* + *y*∗*y* − *9* + (− *z*)*]* @ *[]*) @ *xs*) = *length xs*
apply (*reify Irnat-simps Irlist.simps Irint-simps* ((*Suc n*) ∗*length* ((*[*(*3*::*int*) ∗ *x* + *y*∗*y* − *9* + (− *z*)*]* @ *[]*) @ *xs*)))
oops
datatype *rifm* = *RT* | *RF* | *RVar nat*
| *RNLT rnat rnat* | *RNILT rnat rint* | *RNEQ rnat rnat*
|*RAnd rifm rifm* | *ROr rifm rifm* | *RImp rifm rifm*| *RIff rifm rifm*
| *RNEX rifm* | *RIEX rifm*| *RLEX rifm* | *RNALL rifm* | *RIALL rifm*| *RLALL rifm*
| *RBEX rifm* | *RBALL rifm*

consts *Irifm* :: *rifm* ⇒ *bool list* ⇒ *int list* ⇒ (*int list*) *list* ⇒ *nat list* ⇒ *bool*
primrec
Irifm RT ps is ls ns = *True*
Irifm RF ps is ls ns = *False*
Irifm (*RVar n*) *ps is ls ns* = *ps*!*n*
Irifm (*RNLT s t*) *ps is ls ns* = (*Irnat s is ls ns* < *Irnat t is ls ns*)
Irifm (*RNILT s t*) *ps is ls ns* = (*int* (*Irnat s is ls ns*) < *Irint t is*)
Irifm (*RNEQ s t*) *ps is ls ns* = (*Irnat s is ls ns* = *Irnat t is ls ns*)
Irifm (*RAnd p q*) *ps is ls ns* = (*Irifm p ps is ls ns* ∧ *Irifm q ps is ls ns*)
Irifm (*ROr p q*) *ps is ls ns* = (*Irifm p ps is ls ns* ∨ *Irifm q ps is ls ns*)
Irifm (*RImp p q*) *ps is ls ns* = (*Irifm p ps is ls ns* ⟶ *Irifm q ps is ls ns*)
Irifm (*RIff p q*) *ps is ls ns* = (*Irifm p ps is ls ns* = *Irifm q ps is ls ns*)
Irifm (*RNEX p*) *ps is ls ns* = (∃ *x*. *Irifm p ps is ls* (*x*#*ns*))
Irifm (*RIEX p*) *ps is ls ns* = (∃ *x*. *Irifm p ps* (*x*#*is*) *ls ns*)
Irifm (*RLEX p*) *ps is ls ns* = (∃ *x*. *Irifm p ps is* (*x*#*ls*) *ns*)
Irifm (*RBEX p*) *ps is ls ns* = (∃ *x*. *Irifm p* (*x*#*ps*) *is ls ns*)
Irifm (*RNALL p*) *ps is ls ns* = (∀ *x*. *Irifm p ps is ls* (*x*#*ns*))
Irifm (*RIALL p*) *ps is ls ns* = (∀ *x*. *Irifm p ps* (*x*#*is*) *ls ns*)
Irifm (*RLALL p*) *ps is ls ns* = (∀ *x*. *Irifm p ps is* (*x*#*ls*) *ns*)
Irifm (*RBALL p*) *ps is ls ns* = (∀ *x*. *Irifm p* (*x*#*ps*) *is ls ns*)

lemma $\forall x.\ \exists n.\ ((Suc\ n) * length\ (([(3::int) * x + (f\ t)*y - 9 + (-\ z)] @ []) @ xs) = length\ xs) \wedge m < 5*n - length\ (xs @ [2,3,4,x*z + 8 - y]) \longrightarrow (\exists p.\ \forall q.\ p \wedge q \longrightarrow r)$
apply (*reify Irifm.simps Irnat-simps Irlist.simps Irint-simps*)
oops

datatype *prod* = *Zero* | *One* | *Var nat* | *Mul prod prod*
| *Pw prod nat* | *PNM nat nat prod*
consts *Iprod* :: *prod* ⇒ ('*a*::{*ordered-idom*,*recpower*}) *list* ⇒'*a*
primrec
Iprod Zero vs = *0*
Iprod One vs = *1*
Iprod (*Var n*) *vs* = *vs*!*n*
Iprod (*Mul a b*) *vs* = (*Iprod a vs* ∗ *Iprod b vs*)
Iprod (*Pw a n*) *vs* = ((*Iprod a vs*) ^ *n*)
Iprod (*PNM n k t*) *vs* = (*vs* ! *n*)^*k* ∗ *Iprod t vs*
consts *prodmul*:: *prod* × *prod* ⇒ *prod*
datatype *sgn* = *Pos prod* | *Neg prod* | *ZeroEq prod* | *NZeroEq prod* | *Tr* | *F*
| *Or sgn sgn* | *And sgn sgn*

consts *Isgn* :: *sgn* ⇒ ('*a*::{*ordered-idom*, *recpower*}) *list* ⇒*bool*
primrec
Isgn Tr vs = *True*
Isgn F vs = *False*
Isgn (*ZeroEq t*) *vs* = (*Iprod t vs* = *0*)
Isgn (*NZeroEq t*) *vs* = (*Iprod t vs* ≠ *0*)
Isgn (*Pos t*) *vs* = (*Iprod t vs* > *0*)
Isgn (*Neg t*) *vs* = (*Iprod t vs* < *0*)
Isgn (*And p q*) *vs* = (*Isgn p vs* ∧ *Isgn q vs*)
Isgn (*Or p q*) *vs* = (*Isgn p vs* ∨ *Isgn q vs*)

lemmas *eqs* = *Isgn.simps Iprod.simps*

lemma (*x*::'*a*::{*ordered-idom*, *recpower*})^*4* ∗ *y* ∗ *z* ∗ *y*^*2* ∗ *z*^*23* > *0*
apply (*reify eqs*)
oops

end

43 Square roots of primes are irrational

theory *Sqrt*
imports *Complex-Main Primes*
begin

The square root of any prime number (including *2*) is irrational.

```
theorem sqrt-prime-irrational:
  assumes prime p
  shows sqrt (real p) ∉ ℚ
proof
  from ‹prime p› have p: 1 < p by (simp add: prime-def)
  assume sqrt (real p) ∈ ℚ
  then obtain m n where
      n: n ≠ 0 and sqrt-rat: |sqrt (real p)| = real m / real n
    and gcd: gcd m n = 1 by (rule Rats-abs-nat-div-natE)
  have eq: m^2 = p * n^2
  proof -
    from n and sqrt-rat have real m = |sqrt (real p)| * real n by simp
    then have real (m^2) = (sqrt (real p))^2 * real (n^2)
      by (auto simp add: power2-eq-square)
    also have (sqrt (real p))^2 = real p by simp
    also have ... * real (n^2) = real (p * n^2) by simp
    finally show ?thesis ..
  qed
  have p dvd m ∧ p dvd n
  proof
    from eq have p dvd m^2 ..
    with ‹prime p› show p dvd m by (rule prime-dvd-power-two)
    then obtain k where m = p * k ..
    with eq have p * n^2 = p^2 * k^2 by (auto simp add: power2-eq-square mult-ac)
    with p have n^2 = p * k^2 by (simp add: power2-eq-square)
    then have p dvd n^2 ..
    with ‹prime p› show p dvd n by (rule prime-dvd-power-two)
  qed
  then have p dvd gcd m n ..
  with gcd have p dvd 1 by simp
  then have p ≤ 1 by (simp add: dvd-imp-le)
  with p show False by simp
qed

corollary sqrt (real (2::nat)) ∉ ℚ
  by (rule sqrt-prime-irrational) (rule two-is-prime)
```

43.1 Variations

Here is an alternative version of the main proof, using mostly linear forward-reasoning. While this results in less top-down structure, it is probably closer to proofs seen in mathematics.

```
theorem
  assumes prime p
  shows sqrt (real p) ∉ ℚ
proof
  from ‹prime p› have p: 1 < p by (simp add: prime-def)
  assume sqrt (real p) ∈ ℚ
  then obtain m n where
```

```
    n: n ≠ 0 and sqrt-rat: |sqrt (real p)| = real m / real n
  and gcd: gcd m n = 1 by (rule Rats-abs-nat-div-natE)
 from n and sqrt-rat have real m = |sqrt (real p)| * real n by simp
 then have real (m^2) = (sqrt (real p))^2 * real (n^2)
  by (auto simp add: power2-eq-square)
 also have (sqrt (real p))^2 = real p by simp
 also have ... * real (n^2) = real (p * n^2) by simp
 finally have eq: m^2 = p * n^2 ..
 then have p dvd m^2 ..
 with ‹prime p› have dvd-m: p dvd m by (rule prime-dvd-power-two)
 then obtain k where m = p * k ..
 with eq have p * n^2 = p^2 * k^2 by (auto simp add: power2-eq-square mult-ac)
 with p have n^2 = p * k^2 by (simp add: power2-eq-square)
 then have p dvd n^2 ..
 with ‹prime p› have p dvd n by (rule prime-dvd-power-two)
 with dvd-m have p dvd gcd m n by (rule gcd-greatest)
 with gcd have p dvd 1 by simp
 then have p ≤ 1 by (simp add: dvd-imp-le)
 with p show False by simp
qed

end
```

44 Square roots of primes are irrational (script version)

```
theory Sqrt-Script
imports Complex-Main Primes
begin
```

Contrast this linear Isabelle/Isar script with Markus Wenzel's more mathematical version.

44.1 Preliminaries

```
lemma prime-nonzero: prime p ⟹ p ≠ 0
 by (force simp add: prime-def)

lemma prime-dvd-other-side:
   n * n = p * (k * k) ⟹ prime p ⟹ p dvd n
 apply (subgoal-tac p dvd n * n, blast dest: prime-dvd-mult)
 apply auto
 done

lemma reduction: prime p ⟹
   0 < k ⟹ k * k = p * (j * j) ⟹ k < p * j ∧ 0 < j
 apply (rule ccontr)
```

apply (*simp add*: *linorder-not-less*)
apply (*erule disjE*)
apply (*frule mult-le-mono*, *assumption*)
apply *auto*
apply (*force simp add*: *prime-def*)
done

lemma *rearrange*: $(j::nat) * (p * j) = k * k \Longrightarrow k * k = p * (j * j)$
by (*simp add*: *mult-ac*)

lemma *prime-not-square*:
$prime\ p \Longrightarrow (\bigwedge k.\ 0 < k \Longrightarrow m * m \neq p * (k * k))$
apply (*induct m rule*: *nat-less-induct*)
apply *clarify*
apply (*frule prime-dvd-other-side*, *assumption*)
apply (*erule dvdE*)
apply (*simp add*: *nat-mult-eq-cancel-disj prime-nonzero*)
apply (*blast dest*: *rearrange reduction*)
done

44.2 Main theorem

The square root of any prime number (including *2*) is irrational.

theorem *prime-sqrt-irrational*:
$prime\ p \Longrightarrow x * x = real\ p \Longrightarrow 0 \leq x \Longrightarrow x \notin \mathbb{Q}$
apply (*rule notI*)
apply (*erule Rats-abs-nat-div-natE*)
apply (*simp del*: *real-of-nat-mult*
add: *real-abs-def divide-eq-eq prime-not-square real-of-nat-mult* [*symmetric*])
done

lemmas *two-sqrt-irrational* =
prime-sqrt-irrational [*OF two-is-prime*]

end

45 Arithmetic Series for Reals

theory *Arithmetic-Series-Complex*
imports *Complex-Main*
begin

lemma *arith-series-real*:
$(2::real) * (\sum i \in \{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1) * d))$
proof −
have

((*1*::*real*) + *1*) * (∑ *i*∈{..<*n*}. *a* + *of-nat*(*i*)∗*d*) =
of-nat(*n*) * (*a* + (*a* + *of-nat*(*n* − *1*)∗*d*))
by (*rule arith-series-general*)
thus *?thesis* **by** *simp*
qed

end

46 Divergence of the Harmonic Series

theory *HarmonicSeries*
imports *Complex-Main*
begin

47 Abstract

The following document presents a proof of the Divergence of Harmonic Series theorem formalised in the Isabelle/Isar theorem proving system.

Theorem: The series $\sum_{n=1}^{\infty} \frac{1}{n}$ does not converge to any number.

Informal Proof: The informal proof is based on the following auxillary lemmas:

- aux: $\sum_{n=2^m-1}^{2^m} \frac{1}{n} \geq \frac{1}{2}$
- aux2: $\sum_{n=1}^{2^M} \frac{1}{n} = 1 + \sum_{m=1}^{M} \sum_{n=2^m-1}^{2^m} \frac{1}{n}$

From *aux* and *aux2* we can deduce that $\sum_{n=1}^{2^M} \frac{1}{n} \geq 1 + \frac{M}{2}$ for all M. Now for contradiction, assume that $\sum_{n=1}^{\infty} \frac{1}{n} = s$ for some s. Because $\forall n. \frac{1}{n} > 0$ all the partial sums in the series must be less than s. However with our deduction above we can choose $N > 2 * s - 2$ and thus $\sum_{n=1}^{2^N} \frac{1}{n} > s$. This leads to a contradiction and hence $\sum_{n=1}^{\infty} \frac{1}{n}$ is not summable. QED.

48 Formal Proof

lemma *two-pow-sub*:
0 < *m* ⟹ (*2*::*nat*) ^ *m* − *2*^(*m* − *1*) = *2*^(*m* − *1*)
by (*induct m*) *auto*

We first prove the following auxillary lemma. This lemma simply states that the finite sums: $\frac{1}{2}$, $\frac{1}{3} + \frac{1}{4}$, $\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}$ etc. are all greater than or equal to $\frac{1}{2}$. We do this by observing that each term in the sum is greater than or equal to the last term, e.g. $\frac{1}{3} > \frac{1}{4}$ and thus $\frac{1}{3} + \frac{1}{4} > \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$.

lemma *harmonic-aux*:

```
  ∀m>0. (∑n∈{(2::nat)^(m − 1)+1..2^m}. 1/real n) ≥ 1/2
  (is ∀m>0. (∑n∈(?S m). 1/real n) ≥ 1/2)
proof
  fix m::nat
  obtain tm where tmdef: tm = (2::nat)^m by simp
  {
    assume mgt0: 0 < m
    have ⋀x. x∈(?S m) ⟹ 1/(real x) ≥ 1/(real tm)
    proof −
      fix x::nat
      assume xs: x∈(?S m)
      have xgt0: x>0
      proof −
        from xs have
          x ≥ 2^(m − 1) + 1 by auto
        moreover with mgt0 have
          2^(m − 1) + 1 ≥ (1::nat) by auto
        ultimately have
          x ≥ 1 by (rule xtrans)
        thus ?thesis by simp
      qed
      moreover from xs have x ≤ 2^m by auto
      ultimately have
        inverse (real x) ≥ inverse (real ((2::nat)^m)) by simp
      moreover
      from xgt0 have real x ≠ 0 by simp
      then have
        inverse (real x) = 1 / (real x)
        by (rule nonzero-inverse-eq-divide)
      moreover from mgt0 have real tm ≠ 0 by (simp add: tmdef)
      then have
        inverse (real tm) = 1 / (real tm)
        by (rule nonzero-inverse-eq-divide)
      ultimately show
        1/(real x) ≥ 1/(real tm) by (auto simp add: tmdef)
    qed
    then have
      (∑n∈(?S m). 1 / real n) ≥ (∑n∈(?S m). 1/(real tm))
      by (rule setsum-mono)
    moreover have
      (∑n∈(?S m). 1/(real tm)) = 1/2
    proof −
      have
        (∑n∈(?S m). 1/(real tm)) =
         (1/(real tm))*(∑n∈(?S m). 1)
        by simp
      also have
        ... = ((1/(real tm)) * real (card (?S m)))
        by (simp add: real-of-card real-of-nat-def)
```

```
    also have
      ... = ((1/(real tm)) * real (tm - (2^(m - 1))))
      by (simp add: tmdef)
    also from mgt0 have
      ... = ((1/(real tm)) * real ((2::nat)^(m - 1)))
      by (auto simp: tmdef dest: two-pow-sub)
    also have
      ... = (real (2::nat))^(m - 1) / (real (2::nat))^m
      by (simp add: tmdef realpow-real-of-nat [symmetric])
    also from mgt0 have
      ... = (real (2::nat))^(m - 1) / (real (2::nat))^((m - 1) + 1)
      by auto
    also have ... = 1/2 by simp
    finally show ?thesis .
   qed
   ultimately have
    (∑ n∈(?S m). 1 / real n) ≥ 1/2
    by - (erule subst)
 }
 thus 0 < m ⟶ 1 / 2 ≤ (∑ n∈(?S m). 1 / real n) by simp
qed
```

We then show that the sum of a finite number of terms from the harmonic series can be regrouped in increasing powers of 2. For example: $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} = 1 + (\frac{1}{2}) + (\frac{1}{3} + \frac{1}{4}) + (\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8})$.

```
lemma harmonic-aux2 [rule-format]:
 0<M ⟹ (∑ n∈{1..(2::nat)^M}. 1/real n) =
  (1 + (∑ m∈{1..M}. ∑ n∈{(2::nat)^(m - 1)+1..2^m}. 1/real n))
 (is 0<M ⟹ ?LHS M = ?RHS M)
proof (induct M)
 case 0 show ?case by simp
next
 case (Suc M)
 have ant: 0 < Suc M by fact
 {
  have suc: ?LHS (Suc M) = ?RHS (Suc M)
  proof cases — show that LHS = c and RHS = c, and thus LHS = RHS
   assume mz: M=0
   {
    then have
     ?LHS (Suc M) = ?LHS 1 by simp
    also have
     ... = (∑ n∈{(1::nat)..2}. 1/real n) by simp
    also have
     ... = ((∑ n∈{Suc 1..2}. 1/real n) + 1/(real (1::nat)))
     by (subst setsum-head)
       (auto simp: atLeastSucAtMost-greaterThanAtMost)
    also have
     ... = ((∑ n∈{2..2::nat}. 1/real n) + 1/(real (1::nat)))
```

```
    by (simp add: nat-number)
  also have
    ... =  1/(real (2::nat)) + 1/(real (1::nat)) by simp
  finally have
    ?LHS (Suc M) = 1/2 + 1 by simp
 }
 moreover
 {
  from mz have
    ?RHS (Suc M) = ?RHS 1 by simp
  also have
    ... = (∑ n∈{((2::nat) ^ 0)+1..2 ^ 1}. 1/real n) + 1
    by simp
  also have
    ... = (∑ n∈{2::nat..2}. 1/real n) + 1
  proof −
    have (2::nat) ^ 0 = 1 by simp
    then have (2::nat) ^ 0+1 = 2 by simp
    moreover have (2::nat) ^ 1 = 2 by simp
    ultimately have {((2::nat) ^ 0)+1..2 ^ 1} = {2::nat..2} by auto
    thus ?thesis by simp
  qed
  also have
    ... = 1/2 + 1
    by simp
  finally have
    ?RHS (Suc M) = 1/2 + 1 by simp
 }
 ultimately show ?LHS (Suc M) = ?RHS (Suc M) by simp
next
 assume mnz: M≠0
 then have mgtz: M>0 by simp
 with Suc have suc:
  (?LHS M) = (?RHS M) by blast
 have
  (?LHS (Suc M)) =
   ((?LHS M) + (∑ n∈{(2::nat) ^ M+1..2 ^ (Suc M)}. 1 / real n))
 proof −
  have
    {1..(2::nat) ^ (Suc M)} =
     {1..(2::nat) ^ M}∪{(2::nat) ^ M+1..(2::nat) ^ (Suc M)}
    by auto
  moreover have
    {1..(2::nat) ^ M}∩{(2::nat) ^ M+1..(2::nat) ^ (Suc M)} = {}
    by auto
  moreover have
    finite {1..(2::nat) ^ M} and finite {(2::nat) ^ M+1..(2::nat) ^ (Suc M)}
    by auto
  ultimately show ?thesis
```

```
      by (auto intro: setsum-Un-disjoint)
    qed
    moreover
    {
     have
      (?RHS (Suc M)) =
       (1 + (∑ m∈{1..M}. ∑ n∈{(2::nat) ^ (m − 1)+1..2 ^ m}. 1/real n) +
       (∑ n∈{(2::nat) ^ (Suc M − 1)+1..2 ^ (Suc M)}. 1/real n)) by simp
     also have
      ... = (?RHS M) + (∑ n∈{(2::nat) ^ M+1..2 ^ (Suc M)}. 1/real n)
      by simp
     also from suc have
      ... = (?LHS M) +  (∑ n∈{(2::nat) ^ M+1..2 ^ (Suc M)}. 1/real n)
      by simp
     finally have
      (?RHS (Suc M)) = ... by simp
    }
    ultimately show ?LHS (Suc M) = ?RHS (Suc M) by simp
   qed
  }
  thus ?case by simp
qed
```

Using *harmonic-aux* and *harmonic-aux2* we now show that each group sum is greater than or equal to $\frac{1}{2}$ and thus the finite sum is bounded below by a value proportional to the number of elements we choose.

```
lemma harmonic-aux3 [rule-format]:
  shows ∀ (M::nat). (∑ n∈{1..(2::nat) ^ M}. 1 / real n) ≥ 1 + (real M)/2
  (is ∀ M. ?P M ≥ -)
proof (rule allI, cases)
  fix M::nat
  assume M=0
  then show ?P M ≥ 1 + (real M)/2 by simp
next
  fix M::nat
  assume M≠0
  then have M > 0 by simp
  then have
    (?P M) =
     (1 + (∑ m∈{1..M}. ∑ n∈{(2::nat) ^ (m − 1)+1..2 ^ m}. 1/real n))
    by (rule harmonic-aux2)
  also have
    ... ≥ (1 + (∑ m∈{1..M}. 1/2))
  proof −
    let ?f = (λx. 1/2)
    let ?g = (λx. (∑ n∈{(2::nat) ^ (x − 1)+1..2 ^ x}. 1/real n))
    from harmonic-aux have ⋀x. x∈{1..M} ⟹ ?f x ≤ ?g x by simp
   then have (∑ m∈{1..M}. ?g m) ≥ (∑ m∈{1..M}. ?f m) by (rule setsum-mono)
    thus ?thesis by simp
```

```
  qed
  finally have (?P M) ≥ (1 + (∑ m∈{1..M}. 1/2)) .
  moreover
  {
    have
      (∑ m∈{1..M}. (1::real)/2) = 1/2 * (∑ m∈{1..M}. 1)
      by auto
    also have
      ... = 1/2*(real (card {1..M}))
      by (simp only: real-of-card[symmetric])
    also have
      ... = 1/2*(real M) by simp
    also have
      ... = (real M)/2 by simp
    finally have (∑ m∈{1..M}. (1::real)/2) = (real M)/2 .
  }
  ultimately show (?P M) ≥ (1 + (real M)/2) by simp
qed
```

The final theorem shows that as we take more and more elements (see *harmonic-aux3*) we get an ever increasing sum. By assuming the sum converges, the lemma *series-pos-less* ($[\![summable\ ?f; \forall m{\geq}?n.\ 0 < ?f\ m]\!] \Longrightarrow setsum\ ?f\ \{0..{<}?n\} < suminf\ ?f$) states that each sum is bounded above by the series' limit. This contradicts our first statement and thus we prove that the harmonic series is divergent.

```
theorem DivergenceOfHarmonicSeries:
  shows ¬summable (λn. 1/real (Suc n))
  (is ¬summable ?f)
proof — by contradiction
  let ?s = suminf ?f — let ?s equal the sum of the harmonic series
  assume sf: summable ?f
  then obtain n::nat where ndef: n = nat ⌈2 * ?s⌉ by simp
  then have ngt: 1 + real n/2 > ?s
  proof −
    have ∀ n. 0 ≤ ?f n by simp
    with sf have ?s ≥ 0
      by − (rule suminf-0-le, simp-all)
    then have cgt0: ⌈2*?s⌉ ≥ 0 by simp

    from ndef have n = nat ⌈(2*?s)⌉ .
    then have real n = real (nat ⌈2*?s⌉) by simp
    with cgt0 have real n = real ⌈2*?s⌉
      by (auto dest: real-nat-eq-real)
    then have real n ≥ 2*(?s) by simp
    then have real n/2 ≥ (?s) by simp
    then show 1 + real n/2 > (?s) by simp
  qed

  obtain j where jdef: j = (2::nat) ^ n by simp
```

have $\forall m{\geq}j.\ 0 < ?f\ m$ **by** *simp*
with *sf* **have** $(\sum i{\in}\{0..{<}j\}.\ ?f\ i) < ?s$ **by** (*rule series-pos-less*)
then have $(\sum i{\in}\{1..{<}Suc\ j\}.\ 1/(real\ i)) < ?s$
apply −
apply (*subst*(*asm*) *setsum-shift-bounds-Suc-ivl* [*symmetric*])
by *simp*
with *jdef* **have**
$(\sum i{\in}\{1..{<}\ Suc\ ((2{::}nat)\ \hat{}\ n)\}.\ 1\ /\ (real\ i)) < ?s$ **by** *simp*
then have
$(\sum i{\in}\{1..(2{::}nat)\ \hat{}\ n\}.\ 1\ /\ (real\ i)) < ?s$
by (*simp only*: *atLeastLessThanSuc-atLeastAtMost*)
moreover from *harmonic-aux3* **have**
$(\sum i{\in}\{1..(2{::}nat)\ \hat{}\ n\}.\ 1\ /\ (real\ i)) \geq 1 + real\ n/2$ **by** *simp*
moreover from *ngt* **have** $1 + real\ n/2 > ?s$ **by** *simp*
ultimately show *False* **by** *simp*
qed

end

49 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

49.1 Examples and Test Cases

49.1.1 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

49.1.2 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

49.1.3 Equality

lemma $P = \mathit{True}$
refute
oops

lemma $P = \mathit{False}$
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

49.1.4 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

49.1.5 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-closure :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
trans-closure $P\ Q ==$ (*subset* $Q\ P$) & (*trans* P) & ($ALL\ R.$ *subset* $Q\ R \longrightarrow$ *trans* $R \longrightarrow$ *subset* $P\ R$)

lemma *trans-closure* $T\ P \longrightarrow (\exists\, x\ y.\ T\ x\ y)$
refute
oops

"The union of transitive closures is equal to the transitive closure of unions."

lemma $(\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow T\ x\ y) \longrightarrow$ *trans* $T \longrightarrow (\forall\, Q.\ (\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow Q\ x\ y) \longrightarrow$ *trans* $Q \longrightarrow$ *subset* $T\ Q)$
$\longrightarrow$ *trans-closure* $TP\ P$
$\longrightarrow$ *trans-closure* $TR\ R$
$\longrightarrow (T\ x\ y = (TP\ x\ y \mid TR\ x\ y))$
refute
oops

"Every surjective function is invertible."

lemma $(\forall\, y.\ \exists\, x.\ y = f\ x) \longrightarrow (\exists\, g.\ \forall\, x.\ g\ (f\ x) = x)$
refute
oops

"Every invertible function is surjective."

lemma $(\exists\, g.\ \forall\, x.\ g\ (f\ x) = x) \longrightarrow (\forall\, y.\ \exists\, x.\ y = f\ x)$
refute
oops

Every point is a fixed point of some function.

lemma $\exists f.\ f\ x = x$
refute [*maxsize=4*]
apply (*rule-tac* $x{=}\lambda x.\ x$ **in** *exI*)
apply *simp*
done

Axiom of Choice: first an incorrect version ...

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (EX!f.\ \forall\, x.\ P\ x\ (f\ x))$
refute
oops

... and now two correct ones

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (\exists f.\ \forall\, x.\ P\ x\ (f\ x))$
refute [*maxsize=4*]
apply (*simp add*: *choice*)

done

lemma $(\forall x.\ EX!y.\ P\ x\ y) \longrightarrow (EX!f.\ \forall x.\ P\ x\ (f\ x))$
  refute [*maxsize=2*]
  apply *auto*
    apply (*simp add*: *ex1-implies-ex choice*)
  apply (*fast intro*: *ext*)
done

49.1.6 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

49.1.7 Schematic variables

lemma *?P*
  refute
  apply *auto*
done

lemma $x = ?y$
  refute
  apply *auto*
done

49.1.8 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

49.1.9 Sets

lemma $P\ (A{::}'a\ \mathit{set})$
  refute
oops

lemma $P\ (A{::}'a\ \mathit{set}\ \mathit{set})$
  refute
oops

lemma $\{x.\ P\ x\} = \{y.\ P\ y\}$
  refute
  apply *simp*
done

lemma $x : \{x.\ P\ x\}$
  refute
oops

lemma $P\ \mathit{op}{:}$
  refute
oops

lemma $P\ (\mathit{op}{:}\ x)$
  refute
oops

lemma $P\ \mathit{Collect}$
  refute
oops

lemma $A\ \mathit{Un}\ B = A\ \mathit{Int}\ B$
  refute
oops

lemma $(A\ Int\ B)\ Un\ C = (A\ Un\ C)\ Int\ B$
refute
oops

lemma $Ball\ A\ P \longrightarrow Bex\ A\ P$
refute
oops

49.1.10 undefined

lemma *undefined*
refute
oops

lemma $P\ undefined$
refute
oops

lemma $undefined\ x$
refute
oops

lemma *undefined undefined*
refute
oops

49.1.11 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

49.1.12 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* ⟶ *P* (*Eps P*)
refute [*maxsize*=*3*]
apply (*auto simp add*: *someI*)
done

49.1.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of *'a*:

typedef *'a myTdef* = *insert* (*undefined*::*'a*) (*undefined*::*'a set*)
by *auto*

lemma (*x*::*'a myTdef*) = *y*
refute
oops

typedecl *myTdecl*

typedef *'a T-bij* = {(*f*::*'a*⇒*'a*). ∀ *y*. ∃!*x*. *f x* = *y*}
by *auto*

lemma *P* (*f*::(*myTdecl myTdef*) *T-bij*)
refute
oops

49.1.14 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 $P\ (\textit{unit-rec}\ u\ x)$
refute
oops

lemma $P\ (\textit{case}\ x\ \textit{of}\ ()\ \Rightarrow\ u)$
refute
oops

option

lemma $P\ (x{::}'a\ option)$
refute
oops

lemma $\forall x{::}'a\ option.\ P\ x$
refute
oops

lemma $P\ None$
refute
oops

lemma $P\ (Some\ x)$
refute
oops

lemma *option-rec* $n\ s\ None = n$
refute
apply *simp*
done

lemma *option-rec* $n\ s\ (Some\ x) = s\ x$
refute [*maxsize=4*]

apply *simp*
done

lemma P (*option-rec* n s x)
refute
oops

lemma P (*case* x *of None* $\Rightarrow$ n | *Some* u $\Rightarrow$ s u)
refute
oops

*

lemma P (x::'a*'b)
refute
oops

lemma $\forall x$::'a*'b. P x
refute
oops

lemma P (x, y)
refute
oops

lemma P (*fst* x)
refute
oops

lemma P (*snd* x)
refute
oops

lemma P *Pair*
refute
oops

lemma *prod-rec* p (a, b) = p a b
refute [*maxsize=2*]
apply *simp*
oops

lemma P (*prod-rec* p x)
refute
oops

lemma P (*case* x *of Pair* a b $\Rightarrow$ p a b)
refute
oops

\+

lemma $P\ (x::'a+'b)$
refute
oops

lemma $\forall x::'a+'b.\ P\ x$
refute
oops

lemma $P\ (Inl\ x)$
refute
oops

lemma $P\ (Inr\ x)$
refute
oops

lemma $P\ Inl$
refute
oops

lemma *sum-rec* $l\ r\ (Inl\ x) = l\ x$
refute [*maxsize=3*]
apply *simp*
done

lemma *sum-rec* $l\ r\ (Inr\ x) = r\ x$
refute [*maxsize=3*]
apply *simp*
done

lemma P (*sum-rec* $l\ r\ x$)
refute
oops

lemma P (*case* x *of Inl* $a \Rightarrow l\ a \mid Inr\ b \Rightarrow r\ b$)
refute
oops

Non-recursive datatypes

datatype $T1 = A \mid B$

lemma $P\ (x::T1)$
refute
oops

lemma $\forall x::T1.\ P\ x$
refute
oops

lemma *P A*
refute
oops

lemma *P B*
refute
oops

lemma *T1-rec a b A = a*
refute
apply *simp*
done

lemma *T1-rec a b B = b*
refute
apply *simp*
done

lemma *P (T1-rec a b x)*
refute
oops

lemma *P (case x of A ⇒ a | B ⇒ b)*
refute
oops

datatype *'a T2 = C T1 | D 'a*

lemma *P (x::'a T2)*
refute
oops

lemma *∀ x::'a T2. P x*
refute
oops

lemma *P D*
refute
oops

lemma *T2-rec c d (C x) = c x*
refute [*maxsize=4*]
apply *simp*
done

lemma *T2-rec c d (D x) = d x*
refute [*maxsize=4*]
apply *simp*
done

lemma *P (T2-rec c d x)*
refute
oops

lemma *P (case x of C u ⇒ c u | D v ⇒ 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 ⇒ 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 (λb. XOpt-rec-3 cx dx n1 s1 n2 s2 (x b))*
refute [*maxsize=1*]
apply *simp*
done

lemma *XOpt-rec-2 cx dx n1 s1 n2 s2 None = n1*
refute [*maxsize=2*]
apply *simp*
done

lemma *XOpt-rec-2 cx dx n1 s1 n2 s2 (Some x) = s1 x (XOpt-rec-1 cx dx n1 s1 n2 s2 x)*
refute [*maxsize=1*]
apply *simp*
done

lemma *XOpt-rec-3 cx dx n1 s1 n2 s2 None = n2*
refute [*maxsize=2*]
apply *simp*
done

lemma *XOpt-rec-3 cx dx n1 s1 n2 s2 (Some x) = s2 x (XOpt-rec-1 cx dx n1 s1 n2 s2 x)*
refute [*maxsize=1*]
apply *simp*
done

lemma *P (XOpt-rec-1 cx dx n1 s1 n2 s2 x)*
refute
oops

lemma *P (XOpt-rec-2 cx dx n1 s1 n2 s2 x)*
refute
oops

lemma *P (XOpt-rec-3 cx dx n1 s1 n2 s2 x)*
refute
oops

datatype *'a YOpt = CY ('a ⇒ 'a YOpt) option*

lemma *P (x::'a YOpt)*
refute
oops

lemma *P (CY None)*
refute
oops

lemma *P (CY (Some (λa. CY None)))*
refute
oops

lemma *YOpt-rec-1 cy n s (CY x) = cy x (YOpt-rec-2 cy n s x)*
refute [*maxsize=1*]
apply *simp*
done

lemma *YOpt-rec-2 cy n s None = n*
refute [*maxsize=2*]
apply *simp*
done

lemma *YOpt-rec-2 cy n s (Some x) = s x (λa. YOpt-rec-1 cy n s (x a))*
refute [*maxsize=1*]
apply *simp*
done

lemma *P (YOpt-rec-1 cy n s x)*
refute
oops

lemma *P (YOpt-rec-2 cy n s x)*
refute
oops

datatype *Trie = TR Trie list*

lemma *P (x::Trie)*
refute
oops

lemma *∀x::Trie. P x*
refute
oops

lemma *P (TR [TR []])*
refute
oops

lemma *Trie-rec-1 tr nil cons (TR x) = tr x (Trie-rec-2 tr nil cons x)*
refute [*maxsize=1*]
apply *simp*
done

lemma *Trie-rec-2 tr nil cons [] = nil*
refute [*maxsize=3*]

apply *simp*
done

lemma *Trie-rec-2 tr nil cons* (*x*#*xs*) = *cons x xs* (*Trie-rec-1 tr nil cons x*) (*Trie-rec-2 tr nil cons xs*)
refute [*maxsize=1*]
apply *simp*
done

lemma *P* (*Trie-rec-1 tr nil cons x*)
refute
oops

lemma *P* (*Trie-rec-2 tr nil cons x*)
refute
oops

datatype *InfTree* = *Leaf* | *Node nat* ⇒ *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 $\forall x::'a\ lambda.\ P\ x$
refute
oops

lemma $P\ (Lam\ (\lambda a.\ Var\ a))$
refute
oops

lemma *lambda-rec var app lam* $(Var\ x) = var\ x$
refute [*maxsize=1*]
apply *simp*
done

lemma *lambda-rec var app lam* $(App\ x\ y) = app\ x\ y$ (*lambda-rec var app lam x*) (*lambda-rec var app lam y*)
refute [*maxsize=1*]
apply *simp*
done

lemma *lambda-rec var app lam* $(Lam\ x) = lam\ x\ (\lambda a.$ *lambda-rec var app lam* $(x\ a))$
refute [*maxsize=1*]
apply *simp*
done

lemma P (*lambda-rec v a l x*)
refute
oops

Taken from "Inductive datatypes in HOL", p.8:

datatype $('a, 'b)\ T = C\ 'a \Rightarrow bool \mid D\ 'b\ list$
datatype $'c\ U = E\ ('c, 'c\ U)\ T$

lemma $P\ (x::'c\ U)$
refute
oops

lemma $\forall x::'c\ U.\ P\ x$
refute
oops

lemma $P\ (E\ (C\ (\lambda a.\ True)))$
refute
oops

lemma *U-rec-1 e c d nil cons* $(E\ x) = e\ x$ (*U-rec-2 e c d nil cons x*)
refute [*maxsize=1*]

apply *simp*
done

lemma *U-rec-2 e c d nil cons (C x) = c x*
refute [*maxsize=1*]
apply *simp*
done

lemma *U-rec-2 e c d nil cons (D x) = d x (U-rec-3 e c d nil cons x)*
refute [*maxsize=1*]
apply *simp*
done

lemma *U-rec-3 e c d nil cons [] = nil*
refute [*maxsize=2*]
apply *simp*
done

lemma *U-rec-3 e c d nil cons (x#xs) = cons x xs (U-rec-1 e c d nil cons x) (U-rec-3 e c d nil cons xs)*
refute [*maxsize=1*]
apply *simp*
done

lemma *P (U-rec-1 e c d nil cons x)*
refute
oops

lemma *P (U-rec-2 e c d nil cons x)*
refute
oops

lemma *P (U-rec-3 e c d nil cons x)*
refute
oops

49.1.15 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

49.1.16 Inductively defined sets

inductive-set *arbitrarySet* :: $'a\ set$
where
undefined : *arbitrarySet*

lemma x : *arbitrarySet*
refute
oops

inductive-set *evenCard* :: $'a\ set\ set$
where
$\{\}$: *evenCard*
| $[\![\ S$: *evenCard*; $x \notin S$; $y \notin S$; $x \neq y\ ]\!] \Longrightarrow S \cup \{x, y\}$: *evenCard*

lemma S : *evenCard*
refute
oops

inductive-set
even :: *nat set*
and *odd* :: *nat set*
where
0 : *even*
| n : *even* $\Longrightarrow$ *Suc* n : *odd*
| n : *odd* $\Longrightarrow$ *Suc* n : *even*

lemma n : *odd*

oops

consts $f :: 'a \Rightarrow 'a$

inductive-set
a-even :: $'a\ set$
and *a-odd* :: $'a\ set$
where
undefined : *a-even*
| x : *a-even* $\Longrightarrow$ $f\ x$: *a-odd*
| x : *a-odd* $\Longrightarrow$ $f\ x$: *a-even*

lemma x : *a-odd*
refute — finds a model of size 2, as expected
oops

49.1.17 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

49.1.18 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: $classD\text{-}const\ (classD\text{-}const\ x) = classD\text{-}const\ x$

lemma $P\ (x::'a::classD)$
  refute
oops

A type class with multiple superclasses:

axclass $classE < classC,\ classD$

lemma $P\ (x::'a::classE)$
  refute
oops

lemma P (x::$'a$::{*classB*, *classE*})
  refute
oops

OFCLASS:

lemma *OFCLASS*($'a$::*type*, *type-class*)
  refute — no countermodel exists
  apply *intro-classes*
done

lemma *OFCLASS*($'a$::*classC*, *type-class*)
  refute — no countermodel exists
  apply *intro-classes*
done

lemma *OFCLASS*($'a$, *classB-class*)
  refute — no countermodel exists
  apply *intro-classes*
  apply *simp*
done

lemma *OFCLASS*($'a$::*type*, *classC-class*)
  refute
oops

Overloading:

consts *inverse* :: $'a \Rightarrow 'a$

defs (**overloaded**)
  inverse-bool: *inverse* (*b*::*bool*) == ~ *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

50 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.

50.1 Lists

theorem *map g (map f xs) = map (g o f) xs*
quickcheck
oops

theorem *map g (map f xs) = map (f o g) xs*
quickcheck
oops

theorem *rev (xs @ ys) = rev ys @ rev xs*
quickcheck
oops

theorem *rev (xs @ ys) = rev xs @ rev ys*
quickcheck
oops

theorem *rev (rev xs) = xs*
quickcheck
oops

theorem *rev xs = xs*
quickcheck
oops

An example involving functions inside other data structures

primrec *app* :: *('a ⇒ 'a) list ⇒ 'a ⇒ 'a* **where**
app [] x = x
| *app (f # fs) x = app fs (f x)*

lemma *app (fs @ gs) x = app gs (app fs x)*
quickcheck
by (*induct fs arbitrary*: *x*) *simp-all*

lemma *app (fs @ gs) x = app fs (app gs x)*
quickcheck
oops

primrec *occurs* :: *'a* ⇒ *'a list* ⇒ *nat* **where**
occurs a [] = *0*
| *occurs a* (*x*#*xs*) = (*if* (*x*=*a*) *then Suc*(*occurs a xs*) *else occurs a xs*)

primrec *del1* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
del1 a [] = []
| *del1 a* (*x*#*xs*) = (*if* (*x*=*a*) *then xs else* (*x*#*del1 a xs*))

A lemma, you'd think to be true from our experience with delAll

lemma *Suc* (*occurs a* (*del1 a xs*)) = *occurs a xs*
— Wrong. Precondition needed.
quickcheck
oops

lemma *xs* ~= [] ⟶ *Suc* (*occurs a* (*del1 a xs*)) = *occurs a xs*
quickcheck
— Also wrong.
oops

lemma *0* < *occurs a xs* ⟶ *Suc* (*occurs a* (*del1 a xs*)) = *occurs a xs*
quickcheck
by (*induct xs*) *auto*

primrec *replace* :: *'a* ⇒ *'a* ⇒ *'a list* ⇒ *'a list* **where**
replace a b [] = []
| *replace a b* (*x*#*xs*) = (*if* (*x*=*a*) *then* (*b*#(*replace a b xs*))
else (*x*#(*replace a b xs*)))

lemma *occurs a xs* = *occurs b* (*replace a b xs*)
quickcheck
— Wrong. Precondition needed.
oops

lemma *occurs b xs* = *0* ∨ *a*=*b* ⟶ *occurs a xs* = *occurs b* (*replace a b xs*)
quickcheck
by (*induct xs*) *simp-all*

50.2 Trees

datatype *'a tree* = *Twig* | *Leaf 'a* | *Branch 'a tree 'a tree*

primrec *leaves* :: *'a tree* ⇒ *'a list* **where**
leaves Twig = []
| *leaves* (*Leaf a*) = [*a*]
| *leaves* (*Branch l r*) = (*leaves l*) @ (*leaves r*)

primrec *plant* :: *'a list* ⇒ *'a tree* **where**
plant [] = *Twig*
| *plant* (*x*#*xs*) = *Branch* (*Leaf x*) (*plant xs*)

primrec *mirror* :: *'a tree* ⇒ *'a tree* **where**
mirror (*Twig*) = *Twig*
| *mirror* (*Leaf a*) = *Leaf a*
| *mirror* (*Branch l r*) = *Branch* (*mirror r*) (*mirror l*)

theorem *plant* (*rev* (*leaves xt*)) = *mirror xt*
quickcheck
— Wrong!
oops

theorem *plant*((*leaves xt*) @ (*leaves yt*)) = *Branch xt yt*
quickcheck
— Wrong!
oops

datatype *'a ntree* = *Tip 'a* | *Node 'a 'a ntree 'a ntree*

primrec *inOrder* :: *'a ntree* ⇒ *'a list* **where**
inOrder (*Tip a*)= [*a*]
| *inOrder* (*Node f x y*) = (*inOrder x*)@[*f*]@(*inOrder y*)

primrec *root* :: *'a ntree* ⇒ *'a* **where**
root (*Tip a*) = *a*
| *root* (*Node f x y*) = *f*

theorem *hd* (*inOrder xt*) = *root xt*
quickcheck
— Wrong!
oops

end

51 A formalization of formal power series

theory *Formal-Power-Series*
imports *Main Fact Parity*
begin

51.1 The type of formal power series

typedef (**open**) *'a fps* = {*f* :: *nat* ⇒ *'a*. *True*}
morphisms *fps-nth Abs-fps*
by *simp*

notation *fps-nth* (**infixl** $ *75*)

lemma *expand-fps-eq*: *p* = *q* ⟷ (∀ *n*. *p* $ *n* = *q* $ *n*)

by (*simp add*: *fps-nth-inject* [*symmetric*] *expand-fun-eq*)

lemma *fps-ext*: $(\bigwedge n.\ p\ \$\ n = q\ \$\ n) \Longrightarrow p = q$
by (*simp add*: *expand-fps-eq*)

lemma *fps-nth-Abs-fps* [*simp*]: *Abs-fps f* $ *n* = *f n*
by (*simp add*: *Abs-fps-inverse*)

Definition of the basic elements 0 and 1 and the basic operations of addition, negation and multiplication

instantiation *fps* :: (*zero*) *zero*
begin

definition *fps-zero-def*:
0 = *Abs-fps* (λ*n*. *0*)

instance ..
end

lemma *fps-zero-nth* [*simp*]: *0* $ *n* = *0*
unfolding *fps-zero-def* **by** *simp*

instantiation *fps* :: ({*one*,*zero*}) *one*
begin

definition *fps-one-def*:
1 = *Abs-fps* (λ*n*. *if n* = *0 then 1 else 0*)

instance ..
end

lemma *fps-one-nth* [*simp*]: *1* $ *n* = (*if n* = *0 then 1 else 0*)
unfolding *fps-one-def* **by** *simp*

instantiation *fps* :: (*plus*) *plus*
begin

definition *fps-plus-def*:
op + = (λ*f g*. *Abs-fps* (λ*n*. *f* $ *n* + *g* $ *n*))

instance ..
end

lemma *fps-add-nth* [*simp*]: (*f* + *g*) $ *n* = *f* $ *n* + *g* $ *n*
unfolding *fps-plus-def* **by** *simp*

instantiation *fps* :: (*minus*) *minus*
begin

definition *fps-minus-def*:
  $op - = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ f\ \$\ n - g\ \$\ n))$

instance ..
end

lemma *fps-sub-nth* [*simp*]: $(f - g)\ \$\ n = f\ \$\ n - g\ \$\ n$
  unfolding *fps-minus-def* **by** *simp*

instantiation *fps* :: (*uminus*) *uminus*
begin

definition *fps-uminus-def*:
  $uminus = (\lambda f.\ Abs\text{-}fps\ (\lambda n.\ -\ (f\ \$\ n)))$

instance ..
end

lemma *fps-neg-nth* [*simp*]: $(-\ f)\ \$\ n = -\ (f\ \$\ n)$
  unfolding *fps-uminus-def* **by** *simp*

instantiation *fps* :: ({*comm-monoid-add*, *times*}) *times*
begin

definition *fps-times-def*:
  $op * = (\lambda f\ g.\ Abs\text{-}fps\ (\lambda n.\ \sum i{=}0..n.\ f\ \$\ i * g\ \$\ (n - i)))$

instance ..
end

lemma *fps-mult-nth*: $(f * g)\ \$\ n = (\sum i{=}0..n.\ f\$i * g\$(n - i))$
  unfolding *fps-times-def* **by** *simp*

declare *atLeastAtMost-iff*[*presburger*]
declare *Bex-def*[*presburger*]
declare *Ball-def*[*presburger*]

lemma *mult-delta-left*:
  fixes *x y* :: *'a*::*mult-zero*
  shows *(if b then x else 0) * y = (if b then x * y else 0)*
  by *simp*

lemma *mult-delta-right*:
  fixes *x y* :: *'a*::*mult-zero*
  shows *x * (if b then y else 0) = (if b then x * y else 0)*
  by *simp*

lemma *cond-value-iff*: *f (if b then x else y) = (if b then f x else f y)*
  by *auto*

lemma *cond-application-beta*: (*if b then f else g*) *x* = (*if b then f x else g x*)
by *auto*

51.2 Formal power series form a commutative ring with unity, if the range of sequences they represent is a commutative ring with unity

instance *fps* :: (*semigroup-add*) *semigroup-add*
proof
fix *a b c* :: *'a fps* **show** *a* + *b* + *c* = *a* + (*b* + *c*)
by (*simp add*: *fps-ext add-assoc*)
qed

instance *fps* :: (*ab-semigroup-add*) *ab-semigroup-add*
proof
fix *a b* :: *'a fps* **show** *a* + *b* = *b* + *a*
by (*simp add*: *fps-ext add-commute*)
qed

lemma *fps-mult-assoc-lemma*:
fixes *k* :: *nat* **and** *f* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *'a*::*comm-monoid-add*
shows (∑*j*=*0*..*k*. ∑*i*=*0*..*j*. *f i* (*j* − *i*) (*n* − *j*)) =
(∑*j*=*0*..*k*. ∑*i*=*0*..*k* − *j*. *f j i* (*n* − *j* − *i*))
proof (*induct k*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc k*) **thus** *?case*
by (*simp add*: *Suc-diff-le setsum-addf add-assoc*
cong: *strong-setsum-cong*)
qed

instance *fps* :: (*semiring-0*) *semigroup-mult*
proof
fix *a b c* :: *'a fps*
show (*a* ∗ *b*) ∗ *c* = *a* ∗ (*b* ∗ *c*)
proof (*rule fps-ext*)
fix *n* :: *nat*
have (∑*j*=*0*..*n*. ∑*i*=*0*..*j*. *a*$*i* ∗ *b*$(*j* − *i*) ∗ *c*$(*n* − *j*)) =
(∑*j*=*0*..*n*. ∑*i*=*0*..*n* − *j*. *a*$*j* ∗ *b*$*i* ∗ *c*$(*n* − *j* − *i*))
by (*rule fps-mult-assoc-lemma*)
thus ((*a* ∗ *b*) ∗ *c*) $ *n* = (*a* ∗ (*b* ∗ *c*)) $ *n*
by (*simp add*: *fps-mult-nth setsum-right-distrib*
setsum-left-distrib mult-assoc)
qed
qed

lemma *fps-mult-commute-lemma*:
fixes *n* :: *nat* **and** *f* :: *nat* ⇒ *nat* ⇒ *'a*::*comm-monoid-add*
shows (∑*i*=*0*..*n*. *f i* (*n* − *i*)) = (∑*i*=*0*..*n*. *f* (*n* − *i*) *i*)

proof (*rule setsum-reindex-cong*)
  show *inj-on* (λ*i*. *n* − *i*) {*0*..*n*}
    by (*rule inj-onI*) *simp*
  show {*0*..*n*} = (λ*i*. *n* − *i*) ‘ {*0*..*n*}
    by (*auto*, *rule-tac x*=*n* − *x* **in** *image-eqI*, *simp-all*)
next
  fix *i* **assume** *i* ∈ {*0*..*n*}
  hence *n* − (*n* − *i*) = *i* **by** *simp*
  thus *f* (*n* − *i*) *i* = *f* (*n* − *i*) (*n* − (*n* − *i*)) **by** *simp*
qed

instance *fps* :: (*comm-semiring-0*) *ab-semigroup-mult*
proof
  fix *a b* :: ′*a fps*
  show *a* ∗ *b* = *b* ∗ *a*
  proof (*rule fps-ext*)
    fix *n* :: *nat*
    have (∑ *i*=*0*..*n*. *a*$*i* ∗ *b*$(*n* − *i*)) = (∑ *i*=*0*..*n*. *a*$(*n* − *i*) ∗ *b*$*i*)
      by (*rule fps-mult-commute-lemma*)
    thus (*a* ∗ *b*) $ *n* = (*b* ∗ *a*) $ *n*
      by (*simp add*: *fps-mult-nth mult-commute*)
  qed
qed

instance *fps* :: (*monoid-add*) *monoid-add*
proof
  fix *a* :: ′*a fps* **show** *0* + *a* = *a*
    by (*simp add*: *fps-ext*)
next
  fix *a* :: ′*a fps* **show** *a* + *0* = *a*
    by (*simp add*: *fps-ext*)
qed

instance *fps* :: (*comm-monoid-add*) *comm-monoid-add*
proof
  fix *a* :: ′*a fps* **show** *0* + *a* = *a*
    by (*simp add*: *fps-ext*)
qed

instance *fps* :: (*semiring-1*) *monoid-mult*
proof
  fix *a* :: ′*a fps* **show** *1* ∗ *a* = *a*
    by (*simp add*: *fps-ext fps-mult-nth mult-delta-left setsum-delta*)
next
  fix *a* :: ′*a fps* **show** *a* ∗ *1* = *a*
    by (*simp add*: *fps-ext fps-mult-nth mult-delta-right setsum-delta*′)
qed

instance *fps* :: (*cancel-semigroup-add*) *cancel-semigroup-add*

proof
fix *a b c* :: *'a fps*
assume *a + b = a + c* **then show** *b = c*
by (*simp add*: *expand-fps-eq*)
next
fix *a b c* :: *'a fps*
assume *b + a = c + a* **then show** *b = c*
by (*simp add*: *expand-fps-eq*)
qed

instance *fps* :: (*cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
proof
fix *a b c* :: *'a fps*
assume *a + b = a + c* **then show** *b = c*
by (*simp add*: *expand-fps-eq*)
qed

instance *fps* :: (*cancel-comm-monoid-add*) *cancel-comm-monoid-add* **..**

instance *fps* :: (*group-add*) *group-add*
proof
fix *a* :: *'a fps* **show** *− a + a = 0*
by (*simp add*: *fps-ext*)
next
fix *a b* :: *'a fps* **show** *a − b = a + − b*
by (*simp add*: *fps-ext diff-minus*)
qed

instance *fps* :: (*ab-group-add*) *ab-group-add*
proof
fix *a* :: *'a fps*
show *− a + a = 0*
by (*simp add*: *fps-ext*)
next
fix *a b* :: *'a fps*
show *a − b = a + − b*
by (*simp add*: *fps-ext*)
qed

instance *fps* :: (*zero-neq-one*) *zero-neq-one*
by *default* (*simp add*: *expand-fps-eq*)

instance *fps* :: (*semiring-0*) *semiring*
proof
fix *a b c* :: *'a fps*
show *(a + b) ∗ c = a ∗ c + b ∗ c*
by (*simp add*: *expand-fps-eq fps-mult-nth left-distrib setsum-addf*)
next
fix *a b c* :: *'a fps*

show $a * (b + c) = a * b + a * c$
by (*simp add*: *expand-fps-eq fps-mult-nth right-distrib setsum-addf*)
qed

instance *fps* :: (*semiring-0*) *semiring-0*
proof
fix *a*:: *'a fps* **show** $0 * a = 0$
by (*simp add*: *fps-ext fps-mult-nth*)
next
fix *a*:: *'a fps* **show** $a * 0 = 0$
by (*simp add*: *fps-ext fps-mult-nth*)
qed

instance *fps* :: (*semiring-0-cancel*) *semiring-0-cancel* **..**

51.3 Selection of the nth power of the implicit variable in the infinite sum

lemma *fps-nonzero-nth*: $f \neq 0 \longleftrightarrow (\exists\ n.\ f\ \$n \neq 0)$
by (*simp add*: *expand-fps-eq*)

lemma *fps-nonzero-nth-minimal*:
$f \neq 0 \longleftrightarrow (\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m<n.\ f\ \$\ m = 0))$
proof
let *?n* = *LEAST n*. $f\ \$\ n \neq 0$
assume $f \neq 0$
then have $\exists n.\ f\ \$\ n \neq 0$
by (*simp add*: *fps-nonzero-nth*)
then have $f\ \$\ ?n \neq 0$
by (*rule LeastI-ex*)
moreover have $\forall m<?n.\ f\ \$\ m = 0$
by (*auto dest*: *not-less-Least*)
ultimately have $f\ \$\ ?n \neq 0 \wedge (\forall m<?n.\ f\ \$\ m = 0)$ **..**
then show $\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m<n.\ f\ \$\ m = 0)$ **..**
next
assume $\exists n.\ f\ \$\ n \neq 0 \wedge (\forall m<n.\ f\ \$\ m = 0)$
then show $f \neq 0$ **by** (*auto simp add*: *expand-fps-eq*)
qed

lemma *fps-eq-iff*: $f = g \longleftrightarrow (\forall n.\ f\ \$\ n = g\ \$n)$
by (*rule expand-fps-eq*)

lemma *fps-setsum-nth*: $(setsum\ f\ S)\ \$\ n = setsum\ (\lambda k.\ (f\ k)\ \$\ n)\ S$
proof (*cases finite S*)
assume ¬ *finite S* **then show** *?thesis* **by** *simp*
next
assume *finite S*
then show *?thesis* **by** (*induct set*: *finite*) *auto*
qed

51.4 Injection of the basic ring elements and multiplication by scalars

definition
fps-const c = Abs-fps (λn. if n = 0 then c else 0)

lemma *fps-nth-fps-const* [*simp*]: *fps-const c $ n = (if n = 0 then c else 0)*
unfolding *fps-const-def* **by** *simp*

lemma *fps-const-0-eq-0* [*simp*]: *fps-const 0 = 0*
by (*simp add*: *fps-ext*)

lemma *fps-const-1-eq-1* [*simp*]: *fps-const 1 = 1*
by (*simp add*: *fps-ext*)

lemma *fps-const-neg* [*simp*]: *− (fps-const (c::'a::ring)) = fps-const (− c)*
by (*simp add*: *fps-ext*)

lemma *fps-const-add* [*simp*]: *fps-const (c::'a::monoid-add) + fps-const d = fps-const (c + d)*
by (*simp add*: *fps-ext*)

lemma *fps-const-mult*[*simp*]: *fps-const (c::'a::ring) ∗ fps-const d = fps-const (c ∗ d)*
by (*simp add*: *fps-eq-iff fps-mult-nth setsum-0'*)

lemma *fps-const-add-left*: *fps-const (c::'a::monoid-add) + f = Abs-fps (λn. if n = 0 then c + f$0 else f$n)*
by (*simp add*: *fps-ext*)

lemma *fps-const-add-right*: *f + fps-const (c::'a::monoid-add) = Abs-fps (λn. if n = 0 then f$0 + c else f$n)*
by (*simp add*: *fps-ext*)

lemma *fps-const-mult-left*: *fps-const (c::'a::semiring-0) ∗ f = Abs-fps (λn. c ∗ f$n)*
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-left setsum-delta*)

lemma *fps-const-mult-right*: *f ∗ fps-const (c::'a::semiring-0) = Abs-fps (λn. f$n ∗ c)*
unfolding *fps-eq-iff fps-mult-nth*
by (*simp add*: *fps-const-def mult-delta-right setsum-delta'*)

lemma *fps-mult-left-const-nth* [*simp*]: *(fps-const (c::'a::semiring-1) ∗ f)$n = c∗ f$n*
by (*simp add*: *fps-mult-nth mult-delta-left setsum-delta*)

lemma *fps-mult-right-const-nth* [*simp*]: *(f ∗ fps-const (c::'a::semiring-1))$n = f$n ∗ c*

by (*simp add*: *fps-mult-nth mult-delta-right setsum-delta'*)

51.5 Formal power series form an integral domain

instance *fps* :: (*ring*) *ring* **..**

instance *fps* :: (*ring-1*) *ring-1*
by (*intro-classes*, *auto simp add*: *diff-minus left-distrib*)

instance *fps* :: (*comm-ring-1*) *comm-ring-1*
by (*intro-classes*, *auto simp add*: *diff-minus left-distrib*)

instance *fps* :: (*ring-no-zero-divisors*) *ring-no-zero-divisors*
proof
fix *a b* :: *'a fps*
assume *a0*: *a* ≠ *0* **and** *b0*: *b* ≠ *0*
then obtain *i j* **where** *i*: *a$i≠0* ∀ *k<i*. *a$k=0*
and *j*: *b$j* ≠*0* ∀ *k<j*. *b$k* =*0* **unfolding** *fps-nonzero-nth-minimal*
by *blast+*
have (*a* ∗ *b*) $ (*i+j*) = (∑ *k=0..i+j*. *a$k* ∗ *b$(i+j−k)*)
by (*rule fps-mult-nth*)
also have ... = (*a$i* ∗ *b$(i+j−i)*) + (∑ *k∈{0..i+j}−{i}*. *a$k* ∗ *b$(i+j−k)*)
by (*rule setsum-diff1'*) *simp-all*
also have (∑ *k∈{0..i+j}−{i}*. *a$k* ∗ *b$(i+j−k)*) = *0*
proof (*rule setsum-0' [rule-format]*)
fix *k* **assume** *k* ∈ {*0..i+j*} − {*i*}
then have *k* < *i* ∨ *i+j−k* < *j* **by** *auto*
then show *a$k* ∗ *b$(i+j−k)* = *0* **using** *i j* **by** *auto*
qed
also have *a$i* ∗ *b$(i+j−i)* + *0* = *a$i* ∗ *b$j* **by** *simp*
also have *a$i* ∗ *b$j* ≠ *0* **using** *i j* **by** *simp*
finally have (*a∗b*) $ (*i+j*) ≠ *0* **.**
then show *a∗b* ≠ *0* **unfolding** *fps-nonzero-nth* **by** *blast*
qed

instance *fps* :: (*idom*) *idom* **..**

instantiation *fps* :: (*comm-ring-1*) *number-ring*
begin
definition *number-of-fps-def*: (*number-of k*::*'a fps*) = *of-int k*

instance
by (*intro-classes*, *rule number-of-fps-def*)
end

51.6 Inverses of formal power series

declare *setsum-cong*[*fundef-cong*]

instantiation *fps* :: ({*comm-monoid-add*,*inverse*, *times*, *uminus*}) *inverse*
begin

fun *natfun-inverse*:: *'a fps* ⇒ *nat* ⇒ *'a* **where**
natfun-inverse f 0 = *inverse* (*f*$*0*)
| *natfun-inverse f n* = − *inverse* (*f*$*0*) ∗ *setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}

definition *fps-inverse-def*:
inverse f = (*if f*$*0* = *0 then 0 else Abs-fps* (*natfun-inverse f*))
definition *fps-divide-def*: *divide* = (λ(*f*::*'a fps*) *g*. *f* ∗ *inverse g*)
instance ..
end

lemma *fps-inverse-zero*[*simp*]:
inverse (*0* :: *'a*::{*comm-monoid-add*,*inverse*, *times*, *uminus*} *fps*) = *0*
by (*simp add*: *fps-ext fps-inverse-def*)

lemma *fps-inverse-one*[*simp*]: *inverse* (*1* :: *'a*::{*division-ring*,*zero-neq-one*} *fps*) = *1*
apply (*auto simp add*: *expand-fps-eq fps-inverse-def*)
by (*case-tac n*, *auto*)

instance *fps* :: ({*comm-monoid-add*,*inverse*, *times*, *uminus*}) *division-by-zero*
by *default* (*rule fps-inverse-zero*)

lemma *inverse-mult-eq-1*[*intro*]: **assumes** *f0*: *f*$*0* ≠ (*0*::*'a*::*field*)
shows *inverse f* ∗ *f* = *1*
proof−
have *c*: *inverse f* ∗ *f* = *f* ∗ *inverse f* **by** (*simp add*: *mult-commute*)
from *f0* **have** *ifn*: ⋀*n*. *inverse f* $ *n* = *natfun-inverse f n*
by (*simp add*: *fps-inverse-def*)
from *f0* **have** *th0*: (*inverse f* ∗ *f*) $ *0* = *1*
by (*simp add*: *fps-mult-nth fps-inverse-def*)
{**fix** *n*::*nat* **assume** *np*: *n* >*0*
from *np* **have** *eq*: {*0*..*n*} = {*0*} ∪ {*1* .. *n*} **by** *auto*
have *d*: {*0*} ∩ {*1* .. *n*} = {} **by** *auto*
have *f*: *finite* {*0*::*nat*} *finite* {*1*..*n*} **by** *auto*
from *f0 np* **have** *th0*: − (*inverse f*$*n*) =
(*setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}) / (*f*$*0*)
by (*cases n*, *simp*, *simp add*: *divide-inverse fps-inverse-def*)
from *th0*[*symmetric*, *unfolded nonzero-divide-eq-eq*[*OF f0*]]
have *th1*: *setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*} =
− (*f*$*0*) ∗ (*inverse f*)$*n*
by (*simp add*: *ring-simps*)
have (*f* ∗ *inverse f*) $ *n* = (∑ *i* = *0*..*n*. *f* $*i* ∗ *natfun-inverse f* (*n* − *i*))
unfolding *fps-mult-nth ifn* ..
also have ... = *f*$*0* ∗ *natfun-inverse f n*
+ (∑ *i* = *1*..*n*. *f*$*i* ∗ *natfun-inverse f* (*n*−*i*))

```
    unfolding setsum-Un-disjoint[OF f d, unfolded eq[symmetric]]
    by simp
  also have ... = 0 unfolding th1 ifn by simp
  finally have (inverse f * f)$n = 0 unfolding c . }
 with th0 show ?thesis by (simp add: fps-eq-iff)
qed

lemma fps-inverse-0-iff[simp]: (inverse f)$0 = (0::'a::division-ring) ⟷ f$0 =
0
 by (simp add: fps-inverse-def nonzero-imp-inverse-nonzero)

lemma fps-inverse-eq-0-iff[simp]: inverse f = (0:: ('a::field) fps) ⟷ f $0 = 0
proof−
 {assume f$0 = 0 hence inverse f = 0 by (simp add: fps-inverse-def)}
 moreover
 {assume h: inverse f = 0 and c: f $0 ≠ 0
  from inverse-mult-eq-1[OF c] h have False by simp}
 ultimately show ?thesis by blast
qed

lemma fps-inverse-idempotent[intro]: assumes f0: f$0 ≠ (0::'a::field)
 shows inverse (inverse f) = f
proof−
 from f0 have if0: inverse f $ 0 ≠ 0 by simp
 from inverse-mult-eq-1[OF f0] inverse-mult-eq-1[OF if0]
 have th0: inverse f * f = inverse f * inverse (inverse f)  by (simp add: mult-ac)
 then show ?thesis using f0 unfolding mult-cancel-left by simp
qed

lemma fps-inverse-unique: assumes f0: f$0 ≠ (0::'a::field) and fg: f*g = 1
 shows inverse f = g
proof−
 from inverse-mult-eq-1[OF f0] fg
 have th0: inverse f * f = g * f by (simp add: mult-ac)
 then show ?thesis using f0  unfolding mult-cancel-right
  by (auto simp add: expand-fps-eq)
qed

lemma fps-inverse-gp: inverse (Abs-fps(λn. (1::'a::field)))
 = Abs-fps (λn. if n= 0 then 1 else if n=1 then − 1 else 0)
 apply (rule fps-inverse-unique)
 apply simp
 apply (simp add: fps-eq-iff fps-mult-nth)
proof(clarsimp)
 fix n::nat assume n: n > 0
 let ?f = λi. if n = i then (1::'a) else if n − i = 1 then − 1 else 0
 let ?g = λi. if i = n then 1 else if i=n − 1 then − 1 else 0
 let ?h = λi. if i=n − 1 then − 1 else 0
 have th1: setsum ?f {0..n} = setsum ?g {0..n}
```

by (*rule setsum-cong2*) *auto*
have *th2*: *setsum ?g* {*0..n* − *1*} = *setsum ?h* {*0..n* − *1*}
using *n* **apply** − **by** (*rule setsum-cong2*) *auto*
have *eq*: {*0* .. *n*} = {*0*.. *n* − *1*} ∪ {*n*} **by** *auto*
from *n* **have** *d*: {*0*.. *n* − *1*} ∩ {*n*} = {} **by** *auto*
have *f*: *finite* {*0*.. *n* − *1*} *finite* {*n*} **by** *auto*
show *setsum ?f* {*0..n*} = *0*
unfolding *th1*
apply (*simp add*: *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *del*: *One-nat-def*)
unfolding *th2*
by(*simp add*: *setsum-delta*)
qed

51.7 Formal Derivatives, and the MacLaurin theorem around 0

definition *fps-deriv f* = *Abs-fps* (λ*n*. *of-nat* (*n* + *1*) ∗ *f* $ (*n* + *1*))

lemma *fps-deriv-nth*[*simp*]: *fps-deriv f* $ *n* = *of-nat* (*n* +*1*) ∗ *f* $ (*n*+*1*) **by** (*simp add*: *fps-deriv-def*)

lemma *fps-deriv-linear*[*simp*]: *fps-deriv* (*fps-const* (*a*::′*a*::*comm-semiring-1*) ∗ *f* + *fps-const b* ∗ *g*) = *fps-const a* ∗ *fps-deriv f* + *fps-const b* ∗ *fps-deriv g*
unfolding *fps-eq-iff fps-add-nth fps-const-mult-left fps-deriv-nth* **by** (*simp add*: *ring-simps*)

lemma *fps-deriv-mult*[*simp*]:
fixes *f* :: (′*a* :: *comm-ring-1*) *fps*
shows *fps-deriv* (*f* ∗ *g*) = *f* ∗ *fps-deriv g* + *fps-deriv f* ∗ *g*
proof−
let *?D* = *fps-deriv*
{**fix** *n*::*nat*
let *?Zn* = {*0* ..*n*}
let *?Zn1* = {*0* .. *n* + *1*}
let *?f* = λ*i*. *i* + *1*
have *fi*: *inj-on ?f* {*0..n*} **by** (*simp add*: *inj-on-def*)
have *eq*: {*1*.. *n*+*1*} = *?f* ‘ {*0..n*} **by** *auto*
let *?g* = λ*i*. *of-nat* (*i*+*1*) ∗ *g* $ (*i*+*1*) ∗ *f* $ (*n* − *i*) +
of-nat (*i*+*1*)∗ *f* $ (*i*+*1*) ∗ *g* $ (*n* − *i*)
let *?h* = λ*i*. *of-nat i* ∗ *g* $ *i* ∗ *f* $ ((*n*+*1*) − *i*) +
of-nat i∗ *f* $ *i* ∗ *g* $ ((*n* + *1*) − *i*)
{**fix** *k* **assume** *k*: *k* ∈ {*0..n*}
have *?h* (*k* + *1*) = *?g k* **using** *k* **by** *auto*}
note *th0* = *this*
have *eq*′: {*0..n* +*1*}− {*1* .. *n*+*1*} = {*0*} **by** *auto*
have *s0*: *setsum* (λ*i*. *of-nat i* ∗ *f* $ *i* ∗ *g* $ (*n* + *1* − *i*)) *?Zn1* = *setsum* (λ*i*. *of-nat* (*n* + *1* − *i*) ∗ *f* $ (*n* + *1* − *i*) ∗ *g* $ *i*) *?Zn1*
apply (*rule setsum-reindex-cong*[**where** *f*=λ*i*. *n* + *1* − *i*])

apply (*simp add*: *inj-on-def Ball-def*)
apply *presburger*
apply (*rule set-ext*)
apply (*presburger add*: *image-iff*)
by *simp*
have *s1*: *setsum* (λ*i*. *f* $ *i* ∗ *g* $ (*n* + *1* − *i*)) *?Zn1* = *setsum* (λ*i*. *f* $ (*n* + *1* − *i*) ∗ *g* $ *i*) *?Zn1*
apply (*rule setsum-reindex-cong*[**where** *f*=λ*i*. *n* + *1* − *i*])
apply (*simp add*: *inj-on-def Ball-def*)
apply *presburger*
apply (*rule set-ext*)
apply (*presburger add*: *image-iff*)
by *simp*
have (*f* ∗ *?D g* + *?D f* ∗ *g*)$*n* = (*?D g* ∗ *f* + *?D f* ∗ *g*)$*n* **by** (*simp only*: *mult-commute*)
also have ... = (∑ *i* = *0*..*n*. *?g i*)
by (*simp add*: *fps-mult-nth setsum-addf*[*symmetric*])
also have ... = *setsum ?h* {*1*..*n*+*1*}
using *th0 setsum-reindex-cong*[*OF fi eq*, *of ?g ?h*] **by** *auto*
also have ... = *setsum ?h* {*0*..*n*+*1*}
apply (*rule setsum-mono-zero-left*)
apply *simp*
apply (*simp add*: *subset-eq*)
unfolding *eq'*
by *simp*
also have ... = (*fps-deriv* (*f* ∗ *g*)) $ *n*
apply (*simp only*: *fps-deriv-nth fps-mult-nth setsum-addf*)
unfolding *s0 s1*
unfolding *setsum-addf*[*symmetric*] *setsum-right-distrib*
apply (*rule setsum-cong2*)
by (*auto simp add*: *of-nat-diff ring-simps*)
finally have (*f* ∗ *?D g* + *?D f* ∗ *g*) $ *n* = *?D* (*f*∗*g*) $ *n* **.}**
then show *?thesis* **unfolding** *fps-eq-iff* **by** *auto*
qed

lemma *fps-deriv-neg*[*simp*]: *fps-deriv* (− (*f*:: ('*a*:: *comm-ring-1*) *fps*)) = − (*fps-deriv f*)
by (*simp add*: *fps-eq-iff fps-deriv-def*)
lemma *fps-deriv-add*[*simp*]: *fps-deriv* ((*f*:: ('*a*::*comm-ring-1*) *fps*) + *g*) = *fps-deriv f* + *fps-deriv g*
using *fps-deriv-linear*[*of 1 f 1 g*] **by** *simp*

lemma *fps-deriv-sub*[*simp*]: *fps-deriv* ((*f*:: ('*a*::*comm-ring-1*) *fps*) − *g*) = *fps-deriv f* − *fps-deriv g*
unfolding *diff-minus* **by** *simp*

lemma *fps-deriv-const*[*simp*]: *fps-deriv* (*fps-const c*) = *0*
by (*simp add*: *fps-ext fps-deriv-def fps-const-def*)

lemma *fps-deriv-mult-const-left*[*simp*]: *fps-deriv* (*fps-const* (*c*::'*a*::*comm-ring-1*) ∗ *f*) = *fps-const c* ∗ *fps-deriv f*
by *simp*

lemma *fps-deriv-0*[*simp*]: *fps-deriv 0* = *0*
by (*simp add*: *fps-deriv-def fps-eq-iff*)

lemma *fps-deriv-1*[*simp*]: *fps-deriv 1* = *0*
by (*simp add*: *fps-deriv-def fps-eq-iff*)

lemma *fps-deriv-mult-const-right*[*simp*]: *fps-deriv* (*f* ∗ *fps-const* (*c*::'*a*::*comm-ring-1*)) = *fps-deriv f* ∗ *fps-const c*
by *simp*

lemma *fps-deriv-setsum*: *fps-deriv* (*setsum f S*) = *setsum* (λ*i*. *fps-deriv* (*f i* :: ('*a*::*comm-ring-1*) *fps*)) *S*
proof−
{assume ¬ *finite S* **hence** *?thesis* **by** *simp***}**
moreover
{assume *fS*: *finite S*
have *?thesis* **by** (*induct rule*: *finite-induct*[*OF fS*], *simp-all*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-deriv-eq-0-iff*[*simp*]: *fps-deriv f* = *0* ⟷ (*f* = *fps-const* (*f*$*0* :: '*a*::{*idom*,*semiring-char-0*}))
proof−
{assume *f*= *fps-const* (*f*$*0*) **hence** *fps-deriv f* = *fps-deriv* (*fps-const* (*f*$*0*)) **by** *simp*
hence *fps-deriv f* = *0* **by** *simp* **}**
moreover
{assume *z*: *fps-deriv f* = *0*
hence ∀ *n*. (*fps-deriv f*)$*n* = *0* **by** *simp*
hence ∀ *n*. *f*$(*n*+*1*) = *0* **by** (*simp del*: *of-nat-Suc of-nat-add One-nat-def*)
hence *f* = *fps-const* (*f*$*0*)
apply (*clarsimp simp add*: *fps-eq-iff fps-const-def*)
apply (*erule-tac x*=*n* − *1* **in** *allE*)
by *simp***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-deriv-eq-iff*:
fixes *f*:: ('*a*::{*idom*,*semiring-char-0*}) *fps*
shows *fps-deriv f* = *fps-deriv g* ⟷ (*f* = *fps-const*(*f*$*0* − *g*$*0*) + *g*)
proof−
have *fps-deriv f* = *fps-deriv g* ⟷ *fps-deriv* (*f* − *g*) = *0* **by** *simp*
also have ... ⟷ *f* − *g* = *fps-const* ((*f*−*g*)$*0*) **unfolding** *fps-deriv-eq-0-iff* **..**
finally show *?thesis* **by** (*simp add*: *ring-simps*)
qed

lemma *fps-deriv-eq-iff-ex*: (*fps-deriv f* = *fps-deriv g*) ⟷ (∃ (*c*::'*a*::{*idom*,*semiring-char-0*}). *f* = *fps-const c* + *g*)
apply *auto* **unfolding** *fps-deriv-eq-iff* **by** *blast*

fun *fps-nth-deriv* :: *nat* ⇒ ('*a*::*semiring-1*) *fps* ⇒ '*a fps* **where**
fps-nth-deriv 0 f = *f*
| *fps-nth-deriv* (*Suc n*) *f* = *fps-nth-deriv n* (*fps-deriv f*)

lemma *fps-nth-deriv-commute*: *fps-nth-deriv* (*Suc n*) *f* = *fps-deriv* (*fps-nth-deriv n f*)
by (*induct n arbitrary*: *f*, *auto*)

lemma *fps-nth-deriv-linear*[*simp*]: *fps-nth-deriv n* (*fps-const* (*a*::'*a*::*comm-semiring-1*) ∗ *f* + *fps-const b* ∗ *g*) = *fps-const a* ∗ *fps-nth-deriv n f* + *fps-const b* ∗ *fps-nth-deriv n g*
by (*induct n arbitrary*: *f g*, *auto simp add*: *fps-nth-deriv-commute*)

lemma *fps-nth-deriv-neg*[*simp*]: *fps-nth-deriv n* (− (*f*:: ('*a*:: *comm-ring-1*) *fps*)) = − (*fps-nth-deriv n f*)
by (*induct n arbitrary*: *f*, *simp-all*)

lemma *fps-nth-deriv-add*[*simp*]: *fps-nth-deriv n* ((*f*:: ('*a*::*comm-ring-1*) *fps*) + *g*) = *fps-nth-deriv n f* + *fps-nth-deriv n g*
using *fps-nth-deriv-linear*[*of n 1 f 1 g*] **by** *simp*

lemma *fps-nth-deriv-sub*[*simp*]: *fps-nth-deriv n* ((*f*:: ('*a*::*comm-ring-1*) *fps*) − *g*) = *fps-nth-deriv n f* − *fps-nth-deriv n g*
unfolding *diff-minus fps-nth-deriv-add* **by** *simp*

lemma *fps-nth-deriv-0*[*simp*]: *fps-nth-deriv n 0* = *0*
by (*induct n*, *simp-all*)

lemma *fps-nth-deriv-1*[*simp*]: *fps-nth-deriv n 1* = (*if n* = *0 then 1 else 0*)
by (*induct n*, *simp-all*)

lemma *fps-nth-deriv-const*[*simp*]: *fps-nth-deriv n* (*fps-const c*) = (*if n* = *0 then fps-const c else 0*)
by (*cases n*, *simp-all*)

lemma *fps-nth-deriv-mult-const-left*[*simp*]: *fps-nth-deriv n* (*fps-const* (*c*::'*a*::*comm-ring-1*) ∗ *f*) = *fps-const c* ∗ *fps-nth-deriv n f*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** *simp*

lemma *fps-nth-deriv-mult-const-right*[*simp*]: *fps-nth-deriv n* (*f* ∗ *fps-const* (*c*::'*a*::*comm-ring-1*)) = *fps-nth-deriv n f* ∗ *fps-const c*
using *fps-nth-deriv-linear*[*of n c f 0 0*] **by** (*simp add*: *mult-commute*)

lemma *fps-nth-deriv-setsum*: *fps-nth-deriv n* (*setsum f S*) = *setsum* (λ*i*. *fps-nth-deriv*

n (*f i* :: (*'a*::*comm-ring-1*) *fps*)) *S*
proof−
{assume ¬ *finite S* **hence** *?thesis* **by** *simp***}**
moreover
{assume *fS*: *finite S*
have *?thesis* **by** (*induct rule*: *finite-induct*[*OF fS*], *simp-all*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-deriv-maclauren-0*: (*fps-nth-deriv k* (*f*:: (*'a*::*comm-semiring-1*) *fps*)) $
0 = *of-nat* (*fact k*) ∗ *f*$(*k*)
by (*induct k arbitrary*: *f*) (*auto simp add*: *ring-simps of-nat-mult*)

51.8 Powers

instantiation *fps* :: (*semiring-1*) *power*
begin

fun *fps-pow* :: *nat* ⇒ *'a fps* ⇒ *'a fps* **where**
fps-pow 0 f = *1*
| *fps-pow* (*Suc n*) *f* = *f* ∗ *fps-pow n f*

definition *fps-power-def*: *power* (*f*::*'a fps*) *n* = *fps-pow n f*
instance ..
end

instantiation *fps* :: (*comm-ring-1*) *recpower*
begin
instance
apply (*intro-classes*)
by (*simp-all add*: *fps-power-def*)
end

lemma *fps-power-zeroth-eq-one*: *a*$*0* =*1* ⟹ *a*^*n* $ *0* = (*1*::*'a*::*semiring-1*)
by (*induct n*, *auto simp add*: *fps-power-def expand-fps-eq fps-mult-nth*)

lemma *fps-power-first-eq*: (*a*:: *'a*::*comm-ring-1 fps*)$*0* =*1* ⟹ *a*^*n* $ *1* = *of-nat*
n ∗ *a*$*1*
proof(*induct n*)
case *0* **thus** *?case* **by** (*simp add*: *fps-power-def*)
next
case (*Suc n*)
note *h* = *Suc.hyps*[*OF* ‹*a*$*0* = *1*›]
show *?case* **unfolding** *power-Suc fps-mult-nth*
using *h* ‹*a*$*0* = *1*› *fps-power-zeroth-eq-one*[*OF* ‹*a*$*0*=*1*›] **by** (*simp add*:
ring-simps)
qed

lemma *startsby-one-power*:*a* $ *0* = (*1*::*'a*::*comm-ring-1*) ⟹ *a*^*n* $ *0* = *1*

by (*induct n*, *auto simp add*: *fps-power-def fps-mult-nth*)

lemma *startsby-zero-power*:*a* $0 = (0::'a::comm-ring-1) \Longrightarrow n > 0 \Longrightarrow a\hat{}n \$0 = 0$
by (*induct n*, *auto simp add*: *fps-power-def fps-mult-nth*)

lemma *startsby-power*:*a* $0 = (v::'a::\{comm-ring-1, recpower\}) \Longrightarrow a\hat{}n \$0 = v\hat{}n$
by (*induct n*, *auto simp add*: *fps-power-def fps-mult-nth power-Suc*)

lemma *startsby-zero-power-iff*[*simp*]:
$a\hat{}n \$0 = (0::'a::\{idom, recpower\}) \longleftrightarrow (n \neq 0 \wedge a\$0 = 0)$
apply (*rule iffI*)
apply (*induct n*, *auto simp add*: *power-Suc fps-mult-nth*)
by (*rule startsby-zero-power*, *simp-all*)

lemma *startsby-zero-power-prefix*:
assumes *a0*: $a \$0 = (0::'a::idom)$
shows $\forall n < k.\ a\ \hat{}\ k \$ n = 0$
using *a0*
proof(*induct k rule*: *nat-less-induct*)
fix *k* **assume** *H*: $\forall m<k.\ a \$0 = 0 \longrightarrow (\forall n<m.\ a\ \hat{}\ m \$ n = 0)$ **and** *a0*: $a \$0 = (0::'a)$
let *?ths* = $\forall m<k.\ a\ \hat{}\ k \$ m = 0$
{assume $k = 0$ **then have** *?ths* **by** *simp***}**
moreover
{fix *l* **assume** *k*: $k = Suc\ l$
{fix *m* **assume** *mk*: $m < k$
{assume *m=0* **hence** $a\hat{}k \$ m = 0$ **using** *startsby-zero-power*[*of a k*] *k a0*
by *simp***}**
moreover
{assume *m0*: $m \neq 0$
have $a\ \hat{}k \$ m = (a\hat{}l * a) \$m$ **by** (*simp add*: *k power-Suc mult-commute*)
also have $\ldots = (\sum i = 0..m.\ a\ \hat{}\ l \$ i * a \$ (m - i))$ **by** (*simp add*: *fps-mult-nth*)
also have $\ldots = 0$ **apply** (*rule setsum-0'*)
apply *auto*
apply (*case-tac aa = m*)
using *a0*
apply *simp*
apply (*rule H*[*rule-format*])
using *a0 k mk* **by** *auto*
finally have $a\hat{}k \$ m = 0$ **.}**
ultimately have $a\hat{}k \$ m = 0$ **by** *blast***}**
hence *?ths* **by** *blast***}**
ultimately show *?ths* **by** (*cases k*, *auto*)
qed

lemma *startsby-zero-setsum-depends*:

assumes *a0*: $a\$0 = (0::'a::idom)$ **and** *kn*: $n \geq k$
shows $setsum\ (\lambda i.\ (a\ \hat{}\ i)\$k)\ \{0\ ..\ n\} = setsum\ (\lambda i.\ (a\ \hat{}\ i)\$k)\ \{0\ ..\ k\}$
apply (*rule setsum-mono-zero-right*)
using *kn* **apply** *auto*
apply (*rule startsby-zero-power-prefix*[*rule-format*, *OF a0*])
by *arith*

lemma *startsby-zero-power-nth-same*: **assumes** *a0*: $a\$0 = (0::'a::\{recpower, idom\})$
shows $a\hat{}n\ \$\ n = (a\$1)\ \hat{}\ n$
proof(*induct n*)
case *0* **thus** *?case* **by** (*simp add*: *power-0*)
next
case (*Suc n*)
have $a\ \hat{}\ Suc\ n\ \$\ (Suc\ n) = (a\hat{}n\ *\ a)\$(Suc\ n)$ **by** (*simp add*: *ring-simps power-Suc*)
also have $\ldots = setsum\ (\lambda i.\ a\hat{}n\$i\ *\ a\ \$\ (Suc\ n - i))\ \{0..\ Suc\ n\}$ **by** (*simp add*: *fps-mult-nth*)
also have $\ldots = setsum\ (\lambda i.\ a\hat{}n\$i\ *\ a\ \$\ (Suc\ n - i))\ \{n\ ..\ Suc\ n\}$
apply (*rule setsum-mono-zero-right*)
apply *simp*
apply *clarsimp*
apply *clarsimp*
apply (*rule startsby-zero-power-prefix*[*rule-format*, *OF a0*])
apply *arith*
done
also have $\ldots = a\hat{}n\ \$\ n\ *\ a\$1$ **using** *a0* **by** *simp*
finally show *?case* **using** *Suc.hyps* **by** (*simp add*: *power-Suc*)
qed

lemma *fps-inverse-power*:
fixes $a :: ('a::\{field, recpower\})\ fps$
shows $inverse\ (a\hat{}n) = inverse\ a\ \hat{}\ n$
proof−
{assume *a0*: $a\$0 = 0$
hence *eq*: $inverse\ a = 0$ **by** (*simp add*: *fps-inverse-def*)
{assume $n = 0$ **hence** *?thesis* **by** *simp***}**
moreover
{assume *n*: $n > 0$
from *startsby-zero-power*[*OF a0 n*] *eq a0 n* **have** *?thesis*
by (*simp add*: *fps-inverse-def*)**}**
ultimately have *?thesis* **by** *blast***}**
moreover
{assume *a0*: $a\$0 \neq 0$
have *?thesis*
apply (*rule fps-inverse-unique*)
apply (*simp add*: *a0*)
unfolding *power-mult-distrib*[*symmetric*]
apply (*rule ssubst*[**where** $t = a * inverse\ a$ **and** $s= 1$])
apply *simp-all*

```
    apply (subst mult-commute)
    by (rule inverse-mult-eq-1[OF a0])}
  ultimately show ?thesis by blast
qed

lemma fps-deriv-power: fps-deriv (a ^ n) = fps-const (of-nat n :: 'a:: comm-ring-1)
* fps-deriv a * a ^ (n - 1)
  apply (induct n, auto simp add: power-Suc ring-simps fps-const-add[symmetric]
simp del: fps-const-add)
  by (case-tac n, auto simp add: power-Suc ring-simps)

lemma fps-inverse-deriv:
  fixes a:: ('a :: field) fps
  assumes a0: a$0 ≠ 0
  shows fps-deriv (inverse a) = - fps-deriv a * inverse a ^ 2
proof-
  from inverse-mult-eq-1[OF a0]
  have fps-deriv (inverse a * a) = 0 by simp
  hence inverse a * fps-deriv a + fps-deriv (inverse a) * a = 0 by simp
  hence inverse a * (inverse a * fps-deriv a + fps-deriv (inverse a) * a) = 0 by
simp
  with inverse-mult-eq-1[OF a0]
  have inverse a ^ 2 * fps-deriv a + fps-deriv (inverse a) = 0
    unfolding power2-eq-square
    apply (simp add: ring-simps)
    by (simp add: mult-assoc[symmetric])
  hence inverse a ^ 2 * fps-deriv a + fps-deriv (inverse a) - fps-deriv a * inverse
a ^ 2 = 0 - fps-deriv a * inverse a ^ 2
    by simp
  then show fps-deriv (inverse a) = - fps-deriv a * inverse a ^ 2 by (simp add:
ring-simps)
qed

lemma fps-inverse-mult:
  fixes a::('a :: field) fps
  shows inverse (a * b) = inverse a * inverse b
proof-
  {assume a0: a$0 = 0 hence ab0: (a*b)$0 = 0 by (simp add: fps-mult-nth)
    from a0 ab0 have th: inverse a = 0 inverse (a*b) = 0 by simp-all
    have ?thesis unfolding th by simp}
  moreover
  {assume b0: b$0 = 0 hence ab0: (a*b)$0 = 0 by (simp add: fps-mult-nth)
    from b0 ab0 have th: inverse b = 0 inverse (a*b) = 0 by simp-all
    have ?thesis unfolding th by simp}
  moreover
  {assume a0: a$0 ≠ 0 and b0: b$0 ≠ 0
    from a0 b0 have ab0:(a*b) $ 0 ≠ 0 by (simp  add: fps-mult-nth)
    from inverse-mult-eq-1[OF ab0]
    have inverse (a*b) * (a*b) * inverse a * inverse b = 1 * inverse a * inverse
```

b **by** *simp*
then have *inverse* (*a*∗*b*) ∗ (*inverse a* ∗ *a*) ∗ (*inverse b* ∗ *b*) = *inverse a* ∗ *inverse b*
by (*simp add*: *ring-simps*)
then have *?thesis* **using** *inverse-mult-eq-1*[*OF a0*] *inverse-mult-eq-1*[*OF b0*]
by *simp*}
ultimately show *?thesis* **by** *blast*
qed

lemma *fps-inverse-deriv'*:
fixes *a*:: (*'a* :: *field*) *fps*
assumes *a0*: *a$0* ≠ *0*
shows *fps-deriv* (*inverse a*) = − *fps-deriv a* / *a* ^ *2*
using *fps-inverse-deriv*[*OF a0*]
unfolding *power2-eq-square fps-divide-def*
fps-inverse-mult **by** *simp*

lemma *inverse-mult-eq-1'*: **assumes** *f0*: *f$0* ≠ (*0*::*'a*::*field*)
shows *f* ∗ *inverse f* = *1*
by (*metis mult-commute inverse-mult-eq-1 f0*)

lemma *fps-divide-deriv*: **fixes** *a*:: (*'a* :: *field*) *fps*
assumes *a0*: *b$0* ≠ *0*
shows *fps-deriv* (*a* / *b*) = (*fps-deriv a* ∗ *b* − *a* ∗ *fps-deriv b*) / *b* ^ *2*
using *fps-inverse-deriv*[*OF a0*]
by (*simp add*: *fps-divide-def ring-simps power2-eq-square fps-inverse-mult inverse-mult-eq-1'*[*OF a0*])

51.9 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::*'a* :: {*recpower*, *comm-ring-1*})) ^ *n* = (*if even n then 1 else* − *1*)
by (*induct n*, *auto*)

definition *X* = *Abs-fps* (λ*n*. *if n* = *1 then 1 else 0*)

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps*(λ*n*. (*1*::*'a*::*field*)))
= *1* − *X*
by (*simp add*: *fps-inverse-gp fps-eq-iff X-def*)

lemma *X-mult-nth*[*simp*]: (*X* ∗ (*f* :: (*'a*::*semiring-1*) *fps*)) $*n* = (*if n* = *0 then 0 else f* $ (*n* − *1*))
proof−
{**assume** *n*: *n* ≠ *0*
have *fN*: *finite* {*0* .. *n*} **by** *simp*
have (*X* ∗ *f*) $*n* = (∑ *i* = *0*..*n*. *X* $ *i* ∗ *f* $ (*n* − *i*)) **by** (*simp add*: *fps-mult-nth*)
also have ... = *f* $ (*n* − *1*)
using *n* **by** (*simp add*: *X-def mult-delta-left setsum-delta* [*OF fN*])
finally have *?thesis* **using** *n* **by** *simp* }

```
  moreover
  {assume n: n=0 hence ?thesis by (simp add: fps-mult-nth X-def)}
  ultimately show ?thesis by blast
qed

lemma X-mult-right-nth[simp]: ((f :: ('a::comm-semiring-1) fps) * X) $n = (if n
= 0 then 0 else f $ (n - 1))
  by (metis X-mult-nth mult-commute)

lemma X-power-iff: X^k = Abs-fps (λn. if n = k then (1::'a::comm-ring-1) else
0)
proof(induct k)
  case 0 thus ?case by (simp add: X-def fps-power-def fps-eq-iff)
next
  case (Suc k)
  {fix m
    have (X^Suc k) $ m = (if m = 0 then (0::'a) else (X^k) $ (m - 1))
      by (simp add: power-Suc del: One-nat-def)
    then     have (X^Suc k) $ m = (if m = Suc k then (1::'a) else 0)
      using Suc.hyps by (auto cong del: if-weak-cong)}
  then show ?case by (simp add: fps-eq-iff)
qed

lemma X-power-mult-nth: (X^k * (f :: ('a::comm-ring-1) fps)) $n = (if n < k
then 0 else f $ (n - k))
  apply (induct k arbitrary: n)
  apply (simp)
  unfolding power-Suc mult-assoc
  by (case-tac n, auto)

lemma X-power-mult-right-nth: ((f :: ('a::comm-ring-1) fps) * X^k) $n = (if n <
k then 0 else f $ (n - k))
  by (metis X-power-mult-nth mult-commute)
lemma fps-deriv-X[simp]: fps-deriv X = 1
  by (simp add: fps-deriv-def X-def fps-eq-iff)

lemma fps-nth-deriv-X[simp]: fps-nth-deriv n X = (if n = 0 then X else if n=1
then 1 else 0)
  by (cases n, simp-all)

lemma X-nth[simp]: X$n = (if n = 1 then 1 else 0) by (simp add: X-def)
lemma X-power-nth[simp]: (X^k) $n = (if n = k then 1 else (0::'a::comm-ring-1))
  by (simp add: X-power-iff)

lemma fps-inverse-X-plus1:
  inverse (1 + X) = Abs-fps (λn. (- (1::'a::{recpower, field})) ^ n) (is - = ?r)
proof-
  have eq: (1 + X) * ?r = 1
    unfolding minus-one-power-iff
```

apply (*auto simp add*: *ring-simps fps-eq-iff*)
by *presburger+*
show *?thesis* **by** (*auto simp add*: *eq intro*: *fps-inverse-unique*)
qed

51.10 Integration

definition *fps-integral a a0* = *Abs-fps* (λ*n*. *if n* = *0 then a0 else* (*a*$(*n* − *1*) / *of-nat n*))

lemma *fps-deriv-fps-integral*: *fps-deriv* (*fps-integral a* (*a0* :: *'a* :: {*field*, *ring-char-0*})) = *a*
by (*simp add*: *fps-integral-def fps-deriv-def fps-eq-iff field-simps del*: *of-nat-Suc*)

lemma *fps-integral-linear*: *fps-integral* (*fps-const* (*a*::*'a*::{*field*, *ring-char-0*}) ∗ *f* + *fps-const b* ∗ *g*) (*a*∗*a0* + *b*∗*b0*) = *fps-const a* ∗ *fps-integral f a0* + *fps-const b* ∗ *fps-integral g b0* (**is** *?l* = *?r*)
proof−
have *fps-deriv ?l* = *fps-deriv ?r* **by** (*simp add*: *fps-deriv-fps-integral*)
moreover have *?l*$*0* = *?r*$*0* **by** (*simp add*: *fps-integral-def*)
ultimately show *?thesis*
unfolding *fps-deriv-eq-iff* **by** *auto*
qed

51.11 Composition of FPSs

definition *fps-compose* :: (*'a*::*semiring-1*) *fps* ⇒ *'a fps* ⇒ *'a fps* (**infixl** *oo 55*)
where
fps-compose-def: *a oo b* = *Abs-fps* (λ*n*. *setsum* (λ*i*. *a*$*i* ∗ (*b* ^*i*$*n*)) {*0*..*n*})

lemma *fps-compose-nth*: (*a oo b*)$*n* = *setsum* (λ*i*. *a*$*i* ∗ (*b* ^*i*$*n*)) {*0*..*n*} **by** (*simp add*: *fps-compose-def*)

lemma *fps-compose-X*[*simp*]: *a oo X* = (*a* :: (*'a* :: *comm-ring-1*) *fps*)
by (*simp add*: *fps-ext fps-compose-def mult-delta-right setsum-delta'*)

lemma *fps-const-compose*[*simp*]:
fps-const (*a*::*'a*::{*comm-ring-1*}) *oo b* = *fps-const* (*a*)
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum-delta*)

lemma *X-fps-compose-startby0*[*simp*]: *a*$*0* = *0* ⟹ *X oo a* = (*a* :: (*'a* :: *comm-ring-1*) *fps*)
by (*simp add*: *fps-eq-iff fps-compose-def mult-delta-left setsum-delta*
power-Suc not-le)

51.12 Rules from Herbert Wilf's Generatingfunctionology

51.12.1 Rule 1

lemma *fps-power-mult-eq-shift*:

*X^Suc k * Abs-fps (λn. a (n + Suc k)) = Abs-fps a − setsum (λi. fps-const (a i :: 'a:: field) * X^i) {0 .. k}* (**is** *?lhs = ?rhs*)
proof−
{fix *n*:: *nat*
have *?lhs $ n = (if n < Suc k then 0 else a n)*
unfolding *X-power-mult-nth* **by** *auto*
also have *... = ?rhs $ n*
proof(*induct k*)
case *0* **thus** *?case* **by** (*simp add*: *fps-setsum-nth power-Suc*)
next
case (*Suc k*)
note *th = Suc.hyps*[*symmetric*]
have (*Abs-fps a − setsum (λi. fps-const (a i :: 'a:: field) * X^i) {0 .. Suc k})$n = (Abs-fps a − setsum (λi. fps-const (a i :: 'a:: field) * X^i) {0 .. k} − fps-const (a (Suc k)) * X^ Suc k) $ n* **by** (*simp add*: *ring-simps*)
also have *... = (if n < Suc k then 0 else a n) − (fps-const (a (Suc k)) * X^ Suc k)$n*
using *th*
unfolding *fps-sub-nth* **by** *simp*
also have *... = (if n < Suc (Suc k) then 0 else a n)*
unfolding *X-power-mult-right-nth*
apply (*auto simp add*: *not-less fps-const-def*)
apply (*rule cong*[*of a a*, *OF refl*])
by *arith*
finally show *?case* **by** *simp*
qed
finally have *?lhs $ n = ?rhs $ n* **.}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

51.12.2 Rule 2

definition *XD = op * X o fps-deriv*

lemma *XD-add*[*simp*]:*XD (a + b) = XD a + XD (b :: ('a::comm-ring-1) fps)*
by (*simp add*: *XD-def ring-simps*)

lemma *XD-mult-const*[*simp*]:*XD (fps-const (c::'a::comm-ring-1) * a) = fps-const c * XD a*
by (*simp add*: *XD-def ring-simps*)

lemma *XD-linear*[*simp*]: *XD (fps-const c * a + fps-const d * b) = fps-const c * XD a + fps-const d * XD (b :: ('a::comm-ring-1) fps)*
by *simp*

lemma *XDN-linear*: *(XD^n) (fps-const c * a + fps-const d * b) = fps-const c * (XD^n) a + fps-const d * (XD^n) (b :: ('a::comm-ring-1) fps)*
by (*induct n*, *simp-all*)

lemma *fps-mult-X-deriv-shift*: $X* \ fps\text{-}deriv\ a = Abs\text{-}fps\ (\lambda n.\ of\text{-}nat\ n* \ a\$n)$ **by** (*simp add*: *fps-eq-iff*)

lemma *fps-mult-XD-shift*: $(XD\ \hat{}k)\ (a::('a::\{comm\text{-}ring\text{-}1, recpower, ring\text{-}char\text{-}0\})\ fps) = Abs\text{-}fps\ (\lambda n.\ (of\text{-}nat\ n\ \hat{}\ k) * a\$n)$
by (*induct k arbitrary*: *a*) (*simp-all add*: *power-Suc XD-def fps-eq-iff ring-simps del*: *One-nat-def*)

51.12.3 Rule 3 is trivial and is given by *fps-times-def*

51.12.4 Rule 5 — summation and "division" by (1 - X)

lemma *fps-divide-X-minus1-setsum-lemma*:
$a = ((1::('a::comm\text{-}ring\text{-}1)\ fps) - X) * Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
proof−
let *?X* $= X::('a::comm\text{-}ring\text{-}1)\ fps$
let *?sa* $= Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$
have *th0*: $\bigwedge i.\ (1 - (X::'a\ fps))\ \$\ i = (if\ i = 0\ then\ 1\ else\ if\ i = 1\ then\ -1\ else\ 0)$ **by** *simp*
{fix *n*:: *nat*
{assume *n=0* **hence** $a\$n = ((1 - ?X) * ?sa)\ \$\ n$
by (*simp add*: *fps-mult-nth*)**}**
moreover
{assume *n0*: $n \neq 0$
then have *u*: $\{0\} \cup (\{1\} \cup \{2..n\}) = \{0..n\}\ \{1\} \cup \{2..n\} = \{1..n\}$
$\{0..n-1\} \cup \{n\} = \{0..n\}$
apply (*simp-all add*: *expand-set-eq*) **by** *presburger+*
have *d*: $\{0\} \cap (\{1\} \cup \{2..n\}) = \{\}\ \{1\} \cap \{2..n\} = \{\}$
$\{0..n-1\} \cap \{n\} = \{\}$ **using** *n0*
by (*simp-all add*: *expand-set-eq*, *presburger+*)
have *f*: *finite* $\{0\}$ *finite* $\{1\}$ *finite* $\{2\ ..\ n\}$
finite $\{0\ ..\ n-1\}$ *finite* $\{n\}$ **by** *simp-all*
have $((1 - ?X) * ?sa)\ \$\ n = setsum\ (\lambda i.\ (1 - ?X)\$\ i * ?sa\ \$\ (n - i))\ \{0\ ..\ n\}$
by (*simp add*: *fps-mult-nth*)
also have $\ldots = a\$n$ **unfolding** *th0*
unfolding *setsum-Un-disjoint*[*OF f*(*1*) *finite-UnI*[*OF f*(*2*,*3*)] *d*(*1*), *unfolded u*(*1*)]
unfolding *setsum-Un-disjoint*[*OF f*(*2*) *f*(*3*) *d*(*2*)]
apply (*simp*)
unfolding *setsum-Un-disjoint*[*OF f*(*4*,*5*) *d*(*3*), *unfolded u*(*3*)]
by *simp*
finally have $a\$n = ((1 - ?X) * ?sa)\ \$\ n$ **by** *simp***}**
ultimately have $a\$n = ((1 - ?X) * ?sa)\ \$\ n$ **by** *blast***}**
then show *?thesis*
unfolding *fps-eq-iff* **by** *blast*
qed

lemma *fps-divide-X-minus1-setsum*:
$a\ /((1::('a::field)\ fps) - X) = Abs\text{-}fps\ (\lambda n.\ setsum\ (\lambda i.\ a\ \$\ i)\ \{0..n\})$

proof−
  let *?X = 1 − (X::('a::field) fps)*
  have *th0*: *?X $ 0 ≠ 0* **by** *simp*
  have *a /?X = ?X ∗ Abs-fps (λn::nat. setsum (op $ a) {0..n}) ∗ inverse ?X*
    using *fps-divide-X-minus1-setsum-lemma*[*of a*, *symmetric*] *th0*
    by (*simp add*: *fps-divide-def mult-assoc*)
  also have *... = (inverse ?X ∗ ?X) ∗ Abs-fps (λn::nat. setsum (op $ a) {0..n})*

    by (*simp add*: *mult-ac*)
  finally show *?thesis* **by** (*simp add*: *inverse-mult-eq-1*[*OF th0*])
qed

51.12.5 Rule 4 in its more general form: generalizes Rule 3 for an arbitrary finite product of FPS, also the relvant instance of powers of a FPS

definition *natpermute n k = {l:: nat list. length l = k ∧ foldl op + 0 l = n}*

lemma *natlist-trivial-1*: *natpermute n 1 = {[n]}*
  apply (*auto simp add*: *natpermute-def*)
  apply (*case-tac x*, *auto*)
  done

lemma *foldl-add-start0*:
  foldl op + x xs = x + foldl op + (0::nat) xs
  apply (*induct xs arbitrary*: *x*)
  apply *simp*
  unfolding *foldl.simps*
  apply *atomize*
  apply (*subgoal-tac ∀x::nat. foldl op + x xs = x + foldl op + (0::nat) xs*)
  apply (*erule-tac x=x + a* **in** *allE*)
  apply (*erule-tac x=a* **in** *allE*)
  apply *simp*
  apply *assumption*
  done

lemma *foldl-add-append*: *foldl op + (x::nat) (xs@ys) = foldl op + x xs + foldl op + 0 ys*
  apply (*induct ys arbitrary*: *x xs*)
  apply *auto*
  apply (*subst* (*2*) *foldl-add-start0*)
  apply *simp*
  apply (*subst* (*2*) *foldl-add-start0*)
  by *simp*

lemma *foldl-add-setsum*: *foldl op + (x::nat) xs = x + setsum (nth xs) {0..<length xs}*
proof(*induct xs arbitrary*: *x*)
  case *Nil* **thus** *?case* **by** *simp*

next
case (*Cons a as x*)
have *eq*: *setsum* (*op* ! (*a*#*as*)) {*1*..<*length* (*a*#*as*)} = *setsum* (*op* ! *as*) {*0*..<*length as*}
apply (*rule setsum-reindex-cong* [**where** *f*=*Suc*])
by (*simp-all add*: *inj-on-def*)
have *f*: *finite* {*0*} *finite* {*1* ..< *length* (*a*#*as*)} **by** *simp-all*
have *d*: {*0*} ∩ {*1*..<*length* (*a*#*as*)} = {} **by** *simp*
have *seq*: {*0*} ∪ {*1*..<*length*(*a*#*as*)} = {*0* ..<*length* (*a*#*as*)} **by** *auto*
have *foldl op* + *x* (*a*#*as*) = *x* + *foldl op* + *a as*
apply (*subst foldl-add-start0*) **by** *simp*
also have ... = *x* + *a* + *setsum* (*op* ! *as*) {*0*..<*length as*} **unfolding** *Cons.hyps* **by** *simp*
also have ... = *x* + *setsum* (*op* ! (*a*#*as*)) {*0*..<*length* (*a*#*as*)}
unfolding *eq*[*symmetric*]
unfolding *setsum-Un-disjoint*[*OF f d*, *unfolded seq*]
by *simp*
finally show *?case* **.**
qed

lemma *append-natpermute-less-eq*:
assumes *h*: *xs*@*ys* ∈ *natpermute n k* **shows** *foldl op* + *0 xs* ≤ *n* **and** *foldl op* + *0 ys* ≤ *n*
proof−
{from *h* **have** *foldl op* + *0* (*xs*@ *ys*) = *n* **by** (*simp add*: *natpermute-def*)
hence *foldl op* + *0 xs* + *foldl op* + *0 ys* = *n* **unfolding** *foldl-add-append* **.}**
note *th* = *this*
{from *th* **show** *foldl op* + *0 xs* ≤ *n* **by** *simp***}**
{from *th* **show** *foldl op* + *0 ys* ≤ *n* **by** *simp***}**
qed

lemma *natpermute-split*:
assumes *mn*: *h* ≤ *k*
shows *natpermute n k* = (⋃*m* ∈{*0*..*n*}. {*l1* @ *l2* |*l1 l2*. *l1* ∈ *natpermute m h* ∧ *l2* ∈ *natpermute* (*n* − *m*) (*k* − *h*)}) (**is** *?L* = *?R* **is** *?L* = (⋃*m* ∈{*0*..*n*}. *?S m*))
proof−
{fix *l* **assume** *l*: *l* ∈ *?R*
from *l* **obtain** *m xs ys* **where** *h*: *m* ∈ {*0*..*n*} **and** *xs*: *xs* ∈ *natpermute m h* **and** *ys*: *ys* ∈ *natpermute* (*n* − *m*) (*k* − *h*) **and** *leq*: *l* = *xs*@*ys* **by** *blast*
from *xs* **have** *xs'*: *foldl op* + *0 xs* = *m* **by** (*simp add*: *natpermute-def*)
from *ys* **have** *ys'*: *foldl op* + *0 ys* = *n* − *m* **by** (*simp add*: *natpermute-def*)
have *l* ∈ *?L* **using** *leq xs ys h*
apply *simp*
apply (*clarsimp simp add*: *natpermute-def simp del*: *foldl-append*)
apply (*simp add*: *foldl-add-append*[*unfolded foldl-append*])
unfolding *xs' ys'*
using *mn xs ys*
unfolding *natpermute-def* **by** *simp***}**

```
  moreover
  {fix l assume l: l ∈ natpermute n k
    let ?xs = take h l
    let ?ys = drop h l
    let ?m = foldl op + 0 ?xs
    from l have ls: foldl op + 0 (?xs @ ?ys) = n by (simp add: natpermute-def)
    have xs: ?xs ∈ natpermute ?m h using l mn by (simp add: natpermute-def)
   have ys: ?ys ∈ natpermute (n − ?m) (k − h) using l mn ls[unfolded foldl-add-append]
      by (simp add: natpermute-def)
    from ls have m: ?m ∈ {0..n}  unfolding foldl-add-append by simp
    from xs ys ls have l ∈ ?R
      apply auto
      apply (rule bexI[where x = ?m])
      apply (rule exI[where x = ?xs])
      apply (rule exI[where x = ?ys])
      using ls l unfolding foldl-add-append
      by (auto simp add: natpermute-def)}
  ultimately show ?thesis by blast
qed

lemma natpermute-0: natpermute n 0 = (if n = 0 then {[]} else {})
  by (auto simp add: natpermute-def)
lemma natpermute-0'[simp]: natpermute 0 k = (if k = 0 then {[]} else {replicate
k 0})
  apply (auto simp add: set-replicate-conv-if natpermute-def)
  apply (rule nth-equalityI)
  by simp-all

lemma natpermute-finite: finite (natpermute n k)
proof(induct k arbitrary: n)
  case 0 thus ?case
    apply (subst natpermute-split[of 0 0, simplified])
    by (simp add: natpermute-0)
next
  case (Suc k)
  then show ?case unfolding natpermute-split[of k Suc k, simplified]
    apply −
    apply (rule finite-UN-I)
    apply simp
    unfolding One-nat-def[symmetric] natlist-trivial-1
    apply simp
    unfolding image-Collect[symmetric]
    unfolding Collect-def mem-def
    apply (rule finite-imageI)
    apply blast
    done
qed

lemma natpermute-contain-maximal:
```

```
  {xs ∈ natpermute n (k+1). n ∈ set xs} = UNION {0 .. k} (λi. {(replicate (k+1)
0) [i:=n]})
  (is ?A = ?B)
proof−
  {fix xs assume H: xs ∈ natpermute n (k+1) and n: n ∈ set xs
    from n obtain i where i: i ∈ {0.. k} xs!i = n using H
      unfolding in-set-conv-nth by (auto simp add: less-Suc-eq-le natpermute-def)
    have eqs: ({0..k} − {i}) ∪ {i} = {0..k} using i by auto
    have f: finite({0..k} − {i}) finite {i} by auto
    have d: ({0..k} − {i}) ∩ {i} = {} using i by auto
    from H have n = setsum (nth xs) {0..k} apply (simp add: natpermute-def)
     unfolding foldl-add-setsum by (auto simp add: atLeastLessThanSuc-atLeastAtMost)
    also have ... = n + setsum (nth xs) ({0..k} − {i})
      unfolding setsum-Un-disjoint[OF f d, unfolded eqs] using i by simp
    finally have zxs: ∀ j∈ {0..k} − {i}. xs!j = 0 by auto
    from H have xsl: length xs = k+1 by (simp add: natpermute-def)
    from i have i': i < length (replicate (k+1) 0)   i < k+1
      unfolding length-replicate  by arith+
    have xs = replicate (k+1) 0 [i := n]
      apply (rule nth-equalityI)
      unfolding xsl length-list-update length-replicate
      apply simp
      apply clarify
      unfolding nth-list-update[OF i'(1)]
      using i zxs
      by (case-tac ia=i, auto simp del: replicate.simps)
    then have xs ∈ ?B using i by blast}
  moreover
  {fix i assume i: i ∈ {0..k}
    let ?xs = replicate (k+1) 0 [i:=n]
    have nxs: n ∈ set ?xs
      apply (rule set-update-memI) using i by simp
    have xsl: length ?xs = k+1 by (simp only: length-replicate length-list-update)
    have foldl op + 0 ?xs = setsum (nth ?xs) {0..<k+1}
      unfolding foldl-add-setsum add-0 length-replicate length-list-update ..
    also have ... = setsum (λj. if j = i then n else 0) {0..< k+1}
      apply (rule setsum-cong2) by (simp del: replicate.simps)
    also have ... = n using i by (simp add: setsum-delta)
    finally
   have ?xs ∈ natpermute n (k+1) using xsl unfolding natpermute-def Collect-def
mem-def
      by blast
    then have ?xs ∈ ?A  using nxs  by blast}
  ultimately show ?thesis by auto
qed


lemma fps-setprod-nth:
  fixes m :: nat and a :: nat ⇒ ('a::comm-ring-1) fps
```

shows (*setprod a* {*0* .. *m*})$*n* = *setsum* (λ*v*. *setprod* (λ*j*. (*a j*) $ (*v*!*j*)) {*0*..*m*}) (*natpermute n* (*m*+*1*))
(**is** *?P m n*)
proof(*induct m arbitrary*: *n rule*: *nat-less-induct*)
fix *m n* **assume** *H*: ∀ *m'* < *m*. ∀ *n*. *?P m' n*
{assume *m0*: *m* = *0*
hence *?P m n* **apply** *simp*
unfolding *natlist-trivial-1*[**where** *n* = *n*, *unfolded One-nat-def*] **by** *simp***}**
moreover
{fix *k* **assume** *k*: *m* = *Suc k*
have *km*: *k* < *m* **using** *k* **by** *arith*
have *u0*: {*0* .. *k*} ∪ {*m*} = {*0*..*m*} **using** *k* **apply** (*simp add*: *expand-set-eq*) **by** *presburger*
have *f0*: *finite* {*0* .. *k*} *finite* {*m*} **by** *auto*
have *d0*: {*0* .. *k*} ∩ {*m*} = {} **using** *k* **by** *auto*
have (*setprod a* {*0* .. *m*}) $ *n* = (*setprod a* {*0* .. *k*} ∗ *a m*) $ *n*
unfolding *setprod-Un-disjoint*[*OF f0 d0*, *unfolded u0*] **by** *simp*
also have ... = (∑ *i* = *0*..*n*. (∑ *v*∈*natpermute i* (*k* + *1*). ∏ *j*∈{*0*..*k*}. *a j* $ *v* ! *j*) ∗ *a m* $ (*n* − *i*))
unfolding *fps-mult-nth H*[*rule-format*, *OF km*] **..**
also have ... = (∑ *v*∈*natpermute n* (*m* + *1*). ∏ *j*∈{*0*..*m*}. *a j* $ *v* ! *j*)
apply (*simp add*: *k*)
unfolding *natpermute-split*[*of m m* + *1*, *simplified*, *of n*, *unfolded natlist-trivial-1*[*unfolded One-nat-def*] *k*]
apply (*subst setsum-UN-disjoint*)
apply *simp*
apply *simp*
unfolding *image-Collect*[*symmetric*]
apply *clarsimp*
apply (*rule finite-imageI*)
apply (*rule natpermute-finite*)
apply (*clarsimp simp add*: *expand-set-eq*)
apply *auto*
apply (*rule setsum-cong2*)
unfolding *setsum-left-distrib*
apply (*rule sym*)
apply (*rule-tac f*=λ*xs*. *xs* @[*n* − *x*] **in** *setsum-reindex-cong*)
apply (*simp add*: *inj-on-def*)
apply *auto*
unfolding *setprod-Un-disjoint*[*OF f0 d0*, *unfolded u0*, *unfolded k*]
apply (*clarsimp simp add*: *natpermute-def nth-append*)
done
finally have *?P m n* **.}**
ultimately show *?P m n* **by** (*cases m*, *auto*)
qed

The special form for powers

lemma *fps-power-nth-Suc*:
fixes *m* :: *nat* **and** *a* :: ('*a*::*comm-ring-1*) *fps*

shows $(a\ \hat{}\ Suc\ m)\$n = setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m\})\ (natpermute\ n\ (m+1))$
proof−
have f: *finite* $\{0\ ..m\}$ **by** *simp*
have *th0*: $a\hat{}Suc\ m = setprod\ (\lambda i.\ a)\ \{0..m\}$ **unfolding** *setprod-constant*[*OF f*, *of a*] **by** *simp*
show *?thesis* **unfolding** *th0 fps-setprod-nth* **..**
qed
lemma *fps-power-nth*:
fixes $m :: nat$ **and** $a :: ('a::comm\text{-}ring\text{-}1)\ fps$
shows $(a\ \hat{}m)\$n = (if\ m=0\ then\ 1\$n\ else\ setsum\ (\lambda v.\ setprod\ (\lambda j.\ a\ \$\ (v!j))\ \{0..m\ -\ 1\})\ (natpermute\ n\ m))$
by (*cases m*, *simp-all add*: *fps-power-nth-Suc del*: *power-Suc*)

lemma *fps-nth-power-0*:
fixes $m :: nat$ **and** $a :: ('a::\{comm\text{-}ring\text{-}1, recpower\})\ fps$
shows $(a\ \hat{}m)\$0 = (a\$0)\ \hat{}\ m$
proof−
{assume $m=0$ **hence** *?thesis* **by** *simp***}**
moreover
{fix n **assume** m: $m = Suc\ n$
have c: $m = card\ \{0..n\}$ **using** m **by** *simp*
have $(a\ \hat{}m)\$0 = setprod\ (\lambda i.\ a\$0)\ \{0..n\}$
by (*simp add*: *m fps-power-nth del*: *replicate.simps power-Suc*)
also have $\ldots = (a\$0)\ \hat{}\ m$
unfolding *c* **by** (*rule setprod-constant*, *simp*)
finally have *?thesis* **.}**
ultimately show *?thesis* **by** (*cases m*, *auto*)
qed

lemma *fps-compose-inj-right*:
assumes *a0*: $a\$0 = (0::'a::\{recpower,idom\})$
and *a1*: $a\$1 \neq 0$
shows $(b\ oo\ a = c\ oo\ a) \longleftrightarrow b = c$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof−
{assume *?rhs* **then have** *?lhs* **by** *simp***}**
moreover
{assume *h*: *?lhs*
{fix n **have** $b\$n = c\$n$
proof(*induct n rule*: *nat-less-induct*)
fix n **assume** H: $\forall m<n.\ b\$m = c\$m$
{assume *n0*: $n=0$
from h **have** $(b\ oo\ a)\$n = (c\ oo\ a)\$n$ **by** *simp*
hence $b\$n = c\$n$ **using** *n0* **by** (*simp add*: *fps-compose-nth*)**}**
moreover
{fix *n1* **assume** *n1*: $n = Suc\ n1$
have f: *finite* $\{0\ ..\ n1\}$ *finite* $\{n\}$ **by** *simp-all*
have *eq*: $\{0\ ..\ n1\} \cup \{n\} = \{0\ ..\ n\}$ **using** *n1* **by** *auto*
have d: $\{0\ ..\ n1\} \cap \{n\} = \{\}$ **using** *n1* **by** *auto*

have *seq*: $(\sum i = 0..n1.\ b\ \$\ i * a\ \hat{}\ i\ \$\ n) = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$\ n)$
apply (*rule setsum-cong2*)
using *H n1* **by** *auto*
have *th0*: $(b\ oo\ a)\ \$n = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$\ n) + b\$n * (a\$1)\hat{}n$
unfolding *fps-compose-nth setsum-Un-disjoint*[*OF f d*, *unfolded eq*] *seq*
using *startsby-zero-power-nth-same*[*OF a0*]
by *simp*
have *th1*: $(c\ oo\ a)\ \$n = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$\ n) + c\$n * (a\$1)\hat{}n$
unfolding *fps-compose-nth setsum-Un-disjoint*[*OF f d*, *unfolded eq*]
using *startsby-zero-power-nth-same*[*OF a0*]
by *simp*
from *h*[*unfolded fps-eq-iff*, *rule-format*, *of n*] *th0 th1 a1*
have $b\$n = c\$n$ **by** *auto*}
ultimately show $b\$n = c\$n$ **by** (*cases n*, *auto*)
qed}
then have *?rhs* **by** (*simp add*: *fps-eq-iff*)}
ultimately show *?thesis* **by** *blast*
qed

51.13 Radicals

declare *setprod-cong*[*fundef-cong*]
function *radical* :: $(nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow ('a::\{field, recpower\})\ fps \Rightarrow nat \Rightarrow 'a$ **where**
radical r 0 a 0 = 1
| *radical r 0 a (Suc n) = 0*
| *radical r (Suc k) a 0 = r (Suc k) (a$0)*
| *radical r (Suc k) a (Suc n) = (a$ Suc n − setsum (λxs. setprod (λj. radical r (Suc k) a (xs ! j)) {0..k}) {xs. xs ∈ natpermute (Suc n) (Suc k) ∧ Suc n ∉ set xs}) / (of-nat (Suc k) * (radical r (Suc k) a 0)^k)*
by *pat-completeness auto*

termination *radical*
proof
let *?R = measure* $(\lambda(r, k, a, n).\ n)$
{
show *wf ?R* **by** *auto*}
{**fix** *r k a n xs i*
assume *xs*: $xs \in \{xs \in natpermute\ (Suc\ n)\ (Suc\ k).\ Suc\ n \notin set\ xs\}$ **and** *i*: $i \in \{0..k\}$
{**assume** *c*: $Suc\ n \leq xs\ !\ i$
from *xs i* **have** $xs\ !i \neq Suc\ n$ **by** (*auto simp add*: *in-set-conv-nth natpermute-def*)
with *c* **have** *c'*: $Suc\ n < xs!i$ **by** *arith*
have *fths*: *finite* $\{0\ ..<\ i\}$ *finite* $\{i\}$ *finite* $\{i+1..<Suc\ k\}$ **by** *simp-all*
have *d*: $\{0\ ..<\ i\} \cap (\{i\} \cup \{i+1\ ..<\ Suc\ k\}) = \{\}\ \{i\} \cap \{i+1..<\ Suc\ k\} = \{\}$ **by** *auto*
have *eqs*: $\{0..<Suc\ k\} = \{0\ ..<\ i\} \cup (\{i\} \cup \{i+1\ ..<\ Suc\ k\})$ **using** *i* **by** *auto*

```
    from xs have Suc n = foldl op + 0 xs by (simp add: natpermute-def)
     also have ... = setsum (nth xs) {0..<Suc k} unfolding foldl-add-setsum
using xs
      by (simp add: natpermute-def)
    also have ... = xs!i + setsum (nth xs) {0..<i} + setsum (nth xs) {i+1..<Suc
k}
        unfolding eqs  setsum-Un-disjoint[OF fths(1) finite-UnI[OF fths(2,3)]
d(1)]
      unfolding setsum-Un-disjoint[OF fths(2) fths(3) d(2)]
      by simp
    finally have False using c' by simp}
  then show ((r,Suc k,a,xs!i), r, Suc k, a, Suc n) ∈ ?R
    apply auto by (metis not-less)}
 {fix r k a n
  show ((r,Suc k, a, 0),r, Suc k, a, Suc n) ∈ ?R by simp}
qed

definition fps-radical r n a = Abs-fps (radical r n a)

lemma fps-radical0[simp]: fps-radical r 0 a = 1
 apply (auto simp add: fps-eq-iff fps-radical-def)  by (case-tac n, auto)

lemma fps-radical-nth-0[simp]: fps-radical r n a $ 0 = (if n=0 then 1 else r n
(a$0))
 by (cases n, simp-all add: fps-radical-def)

lemma fps-radical-power-nth[simp]:
 assumes r: (r k (a$0)) ^ k = a$0
 shows fps-radical r k a ^ k $ 0 = (if k = 0 then 1 else a$0)
proof−
 {assume k=0 hence ?thesis by simp }
 moreover
 {fix h assume h: k = Suc h
  have fh: finite {0..h} by simp
  have eq1: fps-radical r k a ^ k $ 0 = (∏j∈{0..h}. fps-radical r k a $ (replicate
k 0) ! j)
    unfolding fps-power-nth h by simp
  also have ... = (∏j∈{0..h}. r k (a$0))
    apply (rule setprod-cong)
    apply simp
    using h
    apply (subgoal-tac replicate k (0::nat) ! x = 0)
    by (auto intro: nth-replicate simp del: replicate.simps)
  also have ... = a$0
    unfolding setprod-constant[OF fh] using r by (simp add: h)
  finally have ?thesis using h by simp}
 ultimately show ?thesis by (cases k, auto)
qed
```

lemma *natpermute-max-card*: **assumes** *n0*: $n \neq 0$
shows *card* $\{xs \in natpermute\ n\ (k+1).\ n \in set\ xs\} = k+1$
unfolding *natpermute-contain-maximal*
proof−
let *?A*= $\lambda i.\ \{replicate\ (k+1)\ 0[i := n]\}$
let *?K* = $\{0\ ..k\}$
have *fK*: *finite ?K* **by** *simp*
have *fAK*: $\forall i \in ?K.$ *finite* (*?A i*) **by** *auto*
have *d*: $\forall i \in ?K.\ \forall j \in ?K.\ i \neq j \longrightarrow \{replicate\ (k+1)\ 0[i := n]\} \cap \{replicate\ (k+1)\ 0[j := n]\} = \{\}$
proof(*clarify*)
fix *i j* **assume** *i*: $i \in ?K$ **and** *j*: $j \in ?K$ **and** *ij*: $i \neq j$
{assume *eq*: *replicate* $(k+1)$ *0* $[i:=n]$ = *replicate* $(k+1)$ *0* $[j:= n]$
have (*replicate* $(k+1)$ *0* $[i:=n]$! *i*) = *n* **using** *i* **by** (*simp del*: *replicate.simps*)
moreover
have (*replicate* $(k+1)$ *0* $[j:=n]$! *i*) = *0* **using** *i ij* **by** (*simp del*: *replicate.simps*)
ultimately have *False* **using** *eq n0* **by** (*simp del*: *replicate.simps*)**}**
then show $\{replicate\ (k+1)\ 0[i := n]\} \cap \{replicate\ (k+1)\ 0[j := n]\} = \{\}$
by *auto*
qed
from *card-UN-disjoint*[*OF fK fAK d*]
show *card* $(\bigcup i \in \{0..k\}.\ \{replicate\ (k+1)\ 0[i := n]\}) = k+1$ **by** *simp*
qed

lemma *power-radical*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes *r0*: (*r* (*Suc k*) (*a$0*)) ^ *Suc k* = *a$0* **and** *a0*: *a$0* ≠ *0*
shows (*fps-radical r* (*Suc k*) *a*) ^ (*Suc k*) = *a*
proof−
let *?r* = *fps-radical r* (*Suc k*) *a*
from *a0 r0* **have** *r00*: *r* (*Suc k*) (*a$0*) ≠ *0* **by** *auto*
{fix *z* **have** *?r* ^ *Suc k* $ *z* = *a$z*
proof(*induct z rule*: *nat-less-induct*)
fix *n* **assume** *H*: $\forall m<n.$ *?r* ^ *Suc k* $ *m* = *a$m*
{assume *n* = *0* **hence** *?r* ^ *Suc k* $ *n* = *a* $*n*
using *fps-radical-power-nth*[*of r Suc k a*, *OF r0*] **by** *simp***}**
moreover
{fix *n1* **assume** *n1*: *n* = *Suc n1*
have *fK*: *finite* $\{0..k\}$ **by** *simp*
have *nz*: *n* ≠ *0* **using** *n1* **by** *arith*
let *?Pnk* = *natpermute n* (*k* + *1*)
let *?Pnkn* = $\{xs \in ?Pnk.\ n \in set\ xs\}$
let *?Pnknn* = $\{xs \in ?Pnk.\ n \notin set\ xs\}$
have *eq*: *?Pnkn* ∪ *?Pnknn* = *?Pnk* **by** *blast*
have *d*: *?Pnkn* ∩ *?Pnknn* = {} **by** *blast*
have *f*: *finite ?Pnkn finite ?Pnknn*
using *finite-Un*[*of ?Pnkn ?Pnknn*, *unfolded eq*]
by (*metis natpermute-finite*)+

```
    let ?f = λv. ∏j∈{0..k}. ?r $ v ! j
   have setsum ?f ?Pnkn = setsum (λv. ?r $ n * r (Suc k) (a $ 0) ^ k) ?Pnkn
    proof(rule setsum-cong2)
      fix v assume v: v ∈ {xs ∈ natpermute n (k + 1). n ∈ set xs}
       let ?ths = (∏j∈{0..k}. fps-radical r (Suc k) a $ v ! j) = fps-radical r
(Suc k) a $ n * r (Suc k) (a $ 0) ^ k
      from v obtain i where i: i ∈ {0..k} v = replicate (k+1) 0 [i:= n]
        unfolding natpermute-contain-maximal by auto
      have (∏j∈{0..k}. fps-radical r (Suc k) a $ v ! j) = (∏j∈{0..k}. if j =
i then fps-radical r (Suc k) a $ n else r (Suc k) (a$0))
        apply (rule setprod-cong, simp)
        using i r0 by (simp del: replicate.simps)
      also have ... = (fps-radical r (Suc k) a $ n) * r (Suc k) (a$0) ^ k
        unfolding setprod-gen-delta[OF fK] using i r0 by simp
      finally show ?ths .
    qed
    then have setsum ?f ?Pnkn = of-nat (k+1) * ?r $ n * r (Suc k) (a $ 0)
^ k
      by (simp add: natpermute-max-card[OF nz, simplified])
    also have ... = a$n − setsum ?f ?Pnknn
      unfolding n1 using r00 a0 by (simp add: field-simps fps-radical-def del:
of-nat-Suc )
    finally have fn: setsum ?f ?Pnkn = a$n − setsum ?f ?Pnknn .
    have (?r ^ Suc k)$n = setsum ?f ?Pnkn + setsum ?f ?Pnknn
      unfolding fps-power-nth-Suc setsum-Un-disjoint[OF f d, unfolded eq] ..
    also have ... = a$n unfolding fn by simp
    finally have ?r ^ Suc k $ n = a $n .}
  ultimately show ?r ^ Suc k $ n = a $n by (cases n, auto)
 qed }
 then show ?thesis by (simp add: fps-eq-iff)
qed

lemma eq-divide-imp': assumes c0: (c::'a::field) ~= 0 and eq: a * c = b
 shows a = b / c
proof−
 from eq have a * c * inverse c = b * inverse c by simp
 hence a * (inverse c * c) = b/c by (simp only: field-simps divide-inverse)
 then show a = b/c unfolding field-inverse[OF c0] by simp
qed

lemma radical-unique:
 assumes r0: (r (Suc k) (b$0)) ^ Suc k = b$0
 and a0: r (Suc k) (b$0 ::'a::{field, ring-char-0, recpower}) = a$0 and b0: b$0
≠ 0
 shows a^(Suc k) = b ⟷ a = fps-radical r (Suc k) b
proof−
 let ?r = fps-radical r (Suc k) b
 have r00: r (Suc k) (b$0) ≠ 0 using b0 r0 by auto
 {assume H: a = ?r
```

```
  from H have a^Suc k = b using power-radical[of r k, OF r0 b0] by simp}
 moreover
 {assume H: a^Suc k = b

  have ceq: card {0..k} = Suc k by simp
  have fk: finite {0..k} by simp
  from a0 have a0r0: a$0 = ?r$0 by simp
  {fix n have a $ n = ?r $ n
   proof(induct n rule: nat-less-induct)
    fix n assume h: ∀ m<n. a$m = ?r $m
    {assume n = 0 hence a$n = ?r $n using a0 by simp }
    moreover
    {fix n1 assume n1: n = Suc n1
     have fK: finite {0..k} by simp
    have nz: n ≠ 0 using n1 by arith
    let ?Pnk = natpermute n (Suc k)
    let ?Pnkn = {xs ∈ ?Pnk. n ∈ set xs}
    let ?Pnknn = {xs ∈ ?Pnk. n ∉ set xs}
    have eq: ?Pnkn ∪ ?Pnknn = ?Pnk by blast
    have d: ?Pnkn ∩ ?Pnknn = {} by blast
    have f: finite ?Pnkn finite ?Pnknn
     using finite-Un[of ?Pnkn ?Pnknn, unfolded eq]
     by (metis natpermute-finite)+
    let ?f = λv. ∏j∈{0..k}. ?r $ v ! j
    let ?g = λv. ∏j∈{0..k}. a $ v ! j
    have setsum ?g ?Pnkn = setsum (λv. a $ n * (?r$0) ^k) ?Pnkn
    proof(rule setsum-cong2)
     fix v assume v: v ∈ {xs ∈ natpermute n (Suc k). n ∈ set xs}
     let ?ths = (∏j∈{0..k}. a $ v ! j) = a $ n * (?r$0) ^k
     from v obtain i where i: i ∈ {0..k} v = replicate (k+1) 0 [i:= n]
        unfolding Suc-plus1 natpermute-contain-maximal by (auto simp del:
replicate.simps)
      have (∏j∈{0..k}. a $ v ! j) = (∏j∈{0..k}. if j = i then a $ n else r
(Suc k) (b$0))
       apply (rule setprod-cong, simp)
       using i a0 by (simp del: replicate.simps)
     also have ... = a $ n * (?r $ 0) ^k
      unfolding  setprod-gen-delta[OF fK] using i by simp
     finally show ?ths .
    qed
    then have th0: setsum ?g ?Pnkn = of-nat (k+1) * a $ n * (?r $ 0) ^k
     by (simp add: natpermute-max-card[OF nz, simplified])
    have th1: setsum ?g ?Pnknn = setsum ?f ?Pnknn
    proof (rule setsum-cong2, rule setprod-cong, simp)
     fix xs i assume xs: xs ∈ ?Pnknn and i: i ∈ {0..k}
     {assume c: n ≤ xs ! i
           from xs i have xs !i ≠ n by (auto simp add: in-set-conv-nth
natpermute-def)
       with c have c': n < xs!i by arith
```

```
          have fths: finite {0 ..< i} finite {i} finite {i+1..<Suc k} by simp-all
          have d: {0 ..< i} ∩ ({i} ∪ {i+1 ..< Suc k}) = {} {i} ∩ {i+1..< Suc
k} = {} by auto
          have eqs: {0..<Suc k} = {0 ..< i} ∪ ({i} ∪ {i+1 ..< Suc k}) using i
by auto
          from xs have n = foldl op + 0 xs by (simp add: natpermute-def)
         also have ... = setsum (nth xs) {0..<Suc k} unfolding foldl-add-setsum
using xs
            by (simp add: natpermute-def)
           also have ... = xs!i + setsum (nth xs) {0..<i} + setsum (nth xs)
{i+1..<Suc k}
           unfolding eqs  setsum-Un-disjoint[OF fths(1) finite-UnI[OF fths(2,3)]
d(1)]
            unfolding setsum-Un-disjoint[OF fths(2) fths(3) d(2)]
            by simp
          finally have False using c' by simp}
        then have thn: xs!i < n by arith
        from h[rule-format, OF thn]
        show a$(xs !i) = ?r$(xs!i) .
      qed
      have th00: ⋀(x::'a). of-nat (Suc k) * (x * inverse (of-nat (Suc k))) = x
        by (simp add: field-simps del: of-nat-Suc)
      from H have b$n = a^Suc k $ n by (simp add: fps-eq-iff)
      also have a ^ Suc k$n = setsum ?g ?Pnkn + setsum ?g ?Pnknn
        unfolding fps-power-nth-Suc
        using setsum-Un-disjoint[OF f d, unfolded Suc-plus1[symmetric],
          unfolded eq, of ?g] by simp
       also have ... = of-nat (k+1) * a $ n * (?r $ 0) ^k + setsum ?f ?Pnknn
unfolding th0 th1 ..
      finally have of-nat (k+1) * a $ n * (?r $ 0) ^k = b$n - setsum ?f ?Pnknn
by simp
      then have a$n = (b$n - setsum ?f ?Pnknn) / (of-nat (k+1) * (?r $ 0) ^k)
        apply -
        apply (rule eq-divide-imp')
        using r00
        apply (simp del: of-nat-Suc)
        by (simp add: mult-ac)
      then have a$n = ?r $n
        apply (simp del: of-nat-Suc)
        unfolding fps-radical-def n1
        by (simp add: field-simps n1 th00 del: of-nat-Suc)}
       ultimately show a$n = ?r $ n by (cases n, auto)
     qed}
   then have a = ?r by (simp add: fps-eq-iff)}
  ultimately show ?thesis by blast
qed


lemma radical-power:
```

assumes *r0*: *r* (*Suc k*) ((*a*$*0*) ^ *Suc k*) = *a*$*0*
and *a0*: (*a*$*0* ::'*a*::{*field*, *ring-char-0*, *recpower*}) ≠ *0*
shows (*fps-radical r* (*Suc k*) (*a* ^ *Suc k*)) = *a*
proof−
let *?ak* = *a* ^ *Suc k*
have *ak0*: *?ak* $ *0* = (*a*$*0*) ^ *Suc k* **by** (*simp add*: *fps-nth-power-0 del*: *power-Suc*)
from *r0* **have** *th0*: *r* (*Suc k*) (*a* ^ *Suc k* $ *0*) ^ *Suc k* = *a* ^ *Suc k* $ *0* **using** *ak0* **by** *auto*
from *r0 ak0* **have** *th1*: *r* (*Suc k*) (*a* ^ *Suc k* $ *0*) = *a* $ *0* **by** *auto*
from *ak0 a0* **have** *ak00*: *?ak* $ *0* ≠*0* **by** *auto*
from *radical-unique*[*of r k ?ak a*, *OF th0 th1 ak00*] **show** *?thesis* **by** *metis*
qed

lemma *fps-deriv-radical*:
fixes *a*:: '*a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes *r0*: (*r* (*Suc k*) (*a*$*0*)) ^ *Suc k* = *a*$*0* **and** *a0*: *a*$*0* ≠ *0*
shows *fps-deriv* (*fps-radical r* (*Suc k*) *a*) = *fps-deriv a* / (*fps-const* (*of-nat* (*Suc k*)) ∗ (*fps-radical r* (*Suc k*) *a*) ^ *k*)
proof−
let *?r*= *fps-radical r* (*Suc k*) *a*
let *?w* = (*fps-const* (*of-nat* (*Suc k*)) ∗ *?r* ^ *k*)
from *a0 r0* **have** *r0'*: *r* (*Suc k*) (*a*$*0*) ≠ *0* **by** *auto*
from *r0'* **have** *w0*: *?w* $ *0* ≠ *0* **by** (*simp del*: *of-nat-Suc*)
note *th0* = *inverse-mult-eq-1*[*OF w0*]
let *?iw* = *inverse ?w*
from *power-radical*[*of r*, *OF r0 a0*]
have *fps-deriv* (*?r* ^ *Suc k*) = *fps-deriv a* **by** *simp*
hence *fps-deriv ?r* ∗ *?w* = *fps-deriv a*
by (*simp add*: *fps-deriv-power mult-ac del*: *power-Suc*)
hence *?iw* ∗ *fps-deriv ?r* ∗ *?w* = *?iw* ∗ *fps-deriv a* **by** *simp*
hence *fps-deriv ?r* ∗ (*?iw* ∗ *?w*) = *fps-deriv a* / *?w*
by (*simp add*: *fps-divide-def*)
then show *?thesis* **unfolding** *th0* **by** *simp*
qed

lemma *radical-mult-distrib*:
fixes *a*:: '*a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes
ra0: *r* (*k*) (*a* $ *0*) ^ *k* = *a* $ *0*
and *rb0*: *r* (*k*) (*b* $ *0*) ^ *k* = *b* $ *0*
and *r0'*: *r* (*k*) ((*a* ∗ *b*) $ *0*) = *r* (*k*) (*a* $ *0*) ∗ *r* (*k*) (*b* $ *0*)
and *a0*: *a*$*0* ≠ *0*
and *b0*: *b*$*0* ≠ *0*
shows *fps-radical r* (*k*) (*a*∗*b*) = *fps-radical r* (*k*) *a* ∗ *fps-radical r* (*k*) (*b*)
proof−
from *r0'* **have** *r0*: (*r* (*k*) ((*a*∗*b*)$*0*)) ^ *k* = (*a*∗*b*)$*0*
by (*simp add*: *fps-mult-nth ra0 rb0 power-mult-distrib*)
{assume *k*=*0* **hence** *?thesis* **by** *simp***}**
moreover

```
  {fix h assume k: k = Suc h
  let ?ra = fps-radical r (Suc h) a
  let ?rb = fps-radical r (Suc h) b
  have th0: r (Suc h) ((a * b) $ 0) = (fps-radical r (Suc h) a * fps-radical r (Suc
h) b) $ 0
    using r0' k by (simp add: fps-mult-nth)
  have ab0: (a*b) $ 0 ≠ 0 using a0 b0 by (simp add: fps-mult-nth)
  from radical-unique[of r h a*b fps-radical r (Suc h) a * fps-radical r (Suc h) b,
OF r0[unfolded k] th0 ab0, symmetric]
    power-radical[of r, OF ra0[unfolded k] a0] power-radical[of r, OF rb0[unfolded
k] b0] k
  have ?thesis by (auto simp add: power-mult-distrib simp del: power-Suc)}
ultimately show ?thesis by (cases k, auto)
qed

lemma radical-inverse:
  fixes a:: 'a ::{field, ring-char-0, recpower} fps
  assumes
  ra0: r (k) (a $ 0) ^ k = a $ 0
  and ria0: r (k) (inverse (a $ 0)) = inverse (r (k) (a $ 0))
  and r1: (r (k) 1) = 1
  and a0: a$0 ≠ 0
  shows fps-radical r (k) (inverse a) = inverse (fps-radical r (k) a)
proof−
  {assume k=0 then have ?thesis by simp}
  moreover
  {fix h assume k[simp]: k = Suc h
    let ?ra = fps-radical r (Suc h) a
    let ?ria = fps-radical r (Suc h) (inverse a)
    from ra0 a0 have th00: r (Suc h) (a$0) ≠ 0 by auto
    have ria0': r (Suc h) (inverse a $ 0) ^ Suc h = inverse a$0
    using ria0 ra0 a0
    by (simp add: fps-inverse-def  nonzero-power-inverse[OF th00, symmetric]
          del: power-Suc)
  from inverse-mult-eq-1[OF a0] have th0: a * inverse a = 1
    by (simp add: mult-commute)
  from radical-unique[where a=1 and b=1 and r=r and k=h, simplified, OF
r1[unfolded k]]
  have th01: fps-radical r (Suc h) 1 = 1 .
  have th1: r (Suc h) ((a * inverse a) $ 0) ^ Suc h = (a * inverse a) $ 0
    r (Suc h) ((a * inverse a) $ 0) =
r (Suc h) (a $ 0) * r (Suc h) (inverse a $ 0)
    using r1 unfolding th0  apply (simp-all add: ria0[symmetric])
    apply (simp add: fps-inverse-def a0)
    unfolding ria0[unfolded k]
    using th00 by simp
  from nonzero-imp-inverse-nonzero[OF a0] a0
  have th2: inverse a $ 0 ≠ 0 by (simp add: fps-inverse-def)
  from radical-mult-distrib[of r Suc h a inverse a, OF ra0[unfolded k] ria0' th1(2)
```

```
a0 th2]
  have th3: ?ra * ?ria = 1 unfolding th0 th01 by simp
  from th00 have ra0: ?ra $ 0 ≠ 0 by simp
  from fps-inverse-unique[OF ra0 th3] have ?thesis by simp}
ultimately show ?thesis by (cases k, auto)
qed

lemma fps-divide-inverse: (a::('a::field) fps) / b = a * inverse b
  by (simp add: fps-divide-def)

lemma radical-divide:
  fixes a:: 'a ::{field, ring-char-0, recpower} fps
  assumes
      ra0: r k (a $ 0) ^ k = a $ 0
  and rb0: r k (b $ 0) ^ k = b $ 0
  and r1: r k 1 = 1
  and rb0': r k (inverse (b $ 0)) = inverse (r k (b $ 0))
  and raib': r k (a$0 / (b$0)) = r k (a$0) / r k (b$0)
  and a0: a$0 ≠ 0
  and b0: b$0 ≠ 0
  shows fps-radical r k (a/b) = fps-radical r k a / fps-radical r k b
proof-
  from raib'
  have raib: r k (a$0 / (b$0)) = r k (a$0) * r k (inverse (b$0))
    by (simp add: divide-inverse rb0'[symmetric])

  {assume k=0 hence ?thesis by (simp add: fps-divide-def)}
  moreover
  {assume k0: k≠ 0
    from b0 k0 rb0 have rbn0: r k (b $0) ≠ 0
      by (auto simp add: power-0-left)

    from rb0 rb0' have rib0: (r k (inverse (b $ 0)))^k = inverse (b$0)
    by (simp add: nonzero-power-inverse[OF rbn0, symmetric])
  from rib0 have th0: r k (inverse b $ 0) ^ k = inverse b $ 0
    by (simp add:fps-inverse-def b0)
  from raib
  have th1: r k ((a * inverse b) $ 0) = r k (a $ 0) * r k (inverse b $ 0)
    by (simp add: divide-inverse fps-inverse-def  b0 fps-mult-nth)
  from nonzero-imp-inverse-nonzero[OF b0] b0 have th2: inverse b $ 0 ≠ 0
    by (simp add: fps-inverse-def)
  from radical-mult-distrib[of r k a inverse b, OF ra0 th0 th1 a0 th2]
  have th: fps-radical r k (a/b) = fps-radical r k a * fps-radical r k (inverse b)
    by (simp add: fps-divide-def)
  with radical-inverse[of r k b, OF rb0 rb0' r1 b0]
  have ?thesis by (simp add: fps-divide-def)}
ultimately show ?thesis by blast
qed
```

51.14 Derivative of composition

lemma *fps-compose-deriv*:
fixes *a*:: (*'a*::*idom*) *fps*
assumes *b0*: *b*$*0* = *0*
shows *fps-deriv* (*a oo b*) = ((*fps-deriv a*) *oo b*) * (*fps-deriv b*)
proof−
{fix *n*
have (*fps-deriv* (*a oo b*))$*n* = *setsum* (λ*i*. *a* $ *i* * (*fps-deriv* (*b*^*i*))$*n*) {*0*.. *Suc n*}
by (*simp add*: *fps-compose-def ring-simps setsum-right-distrib del*: *of-nat-Suc*)
also have ... = *setsum* (λ*i*. *a*$*i* * ((*fps-const* (*of-nat i*)) * (*fps-deriv b* * (*b*^(*i* − *1*))))$*n*) {*0*.. *Suc n*}
by (*simp add*: *ring-simps fps-deriv-power del*: *fps-mult-left-const-nth of-nat-Suc*)
also have ... = *setsum* (λ*i*. *of-nat i* * *a*$*i* * (((*b*^(*i* − *1*)) * *fps-deriv b*))$*n*) {*0*.. *Suc n*}
unfolding *fps-mult-left-const-nth* **by** (*simp add*: *ring-simps*)
also have ... = *setsum* (λ*i*. *of-nat i* * *a*$*i* * (*setsum* (λ*j*. (*b* ^ (*i* − *1*))$*j* * (*fps-deriv b*)$(*n* − *j*)) {*0*..*n*})) {*0*.. *Suc n*}
unfolding *fps-mult-nth* **..**
also have ... = *setsum* (λ*i*. *of-nat i* * *a*$*i* * (*setsum* (λ*j*. (*b* ^ (*i* − *1*))$*j* * (*fps-deriv b*)$(*n* − *j*)) {*0*..*n*})) {*1*.. *Suc n*}
apply (*rule setsum-mono-zero-right*)
apply (*auto simp add*: *mult-delta-left setsum-delta not-le*)
done
also have ... = *setsum* (λ*i*. *of-nat* (*i* + *1*) * *a*$(*i*+*1*) * (*setsum* (λ*j*. (*b* ^ *i*)$*j* * *of-nat* (*n* − *j* + *1*) * *b*$(*n* − *j* + *1*)) {*0*..*n*})) {*0*.. *n*}
unfolding *fps-deriv-nth*
apply (*rule setsum-reindex-cong*[**where** *f*=*Suc*])
by (*auto simp add*: *mult-assoc*)
finally have *th0*: (*fps-deriv* (*a oo b*))$*n* = *setsum* (λ*i*. *of-nat* (*i* + *1*) * *a*$(*i*+*1*) * (*setsum* (λ*j*. (*b* ^ *i*)$*j* * *of-nat* (*n* − *j* + *1*) * *b*$(*n* − *j* + *1*)) {*0*..*n*})) {*0*.. *n*} **.**

have (((*fps-deriv a*) *oo b*) * (*fps-deriv b*))$*n* = *setsum* (λ*i*. (*fps-deriv b*)$ (*n* − *i*) * ((*fps-deriv a*) *oo b*)$*i*) {*0*..*n*}
unfolding *fps-mult-nth* **by** (*simp add*: *mult-ac*)
also have ... = *setsum* (λ*i*. *setsum* (λ*j*. *of-nat* (*n* − *i* +*1*) * *b*$(*n* − *i* + *1*) * *of-nat* (*j* + *1*) * *a*$(*j*+*1*) * (*b*^*j*)$*i*) {*0*..*n*}) {*0*..*n*}
unfolding *fps-deriv-nth fps-compose-nth setsum-right-distrib mult-assoc*
apply (*rule setsum-cong2*)
apply (*rule setsum-mono-zero-left*)
apply (*simp-all add*: *subset-eq*)
apply *clarify*
apply (*subgoal-tac b*^*i*$*x* = *0*)
apply *simp*
apply (*rule startsby-zero-power-prefix*[*OF b0*, *rule-format*])
by *simp*
also have ... = *setsum* (λ*i*. *of-nat* (*i* + *1*) * *a*$(*i*+*1*) * (*setsum* (λ*j*. (*b* ^ *i*)$*j* * *of-nat* (*n* − *j* + *1*) * *b*$(*n* − *j* + *1*)) {*0*..*n*})) {*0*.. *n*}
unfolding *setsum-right-distrib*

```
    apply (subst setsum-commute)
    by ((rule setsum-cong2)+) simp
  finally have (fps-deriv (a oo b))$n = (((fps-deriv a) oo b) * (fps-deriv b)) $n
    unfolding th0 by simp}
then show ?thesis by (simp add: fps-eq-iff)
qed

lemma fps-mult-X-plus-1-nth:
  ((1+X)*a) $n = (if n = 0 then (a$n :: 'a::comm-ring-1) else a$n + a$(n - 1))
proof-
  {assume n = 0 hence ?thesis by (simp add: fps-mult-nth )}
  moreover
  {fix m assume m: n = Suc m
    have ((1+X)*a) $n = setsum (λi. (1+X)$i * a$(n-i)) {0..n}
      by (simp add: fps-mult-nth)
    also have ... = setsum (λi. (1+X)$i * a$(n-i)) {0.. 1}
      unfolding m
      apply (rule setsum-mono-zero-right)
      by (auto simp add: )
    also have ... = (if n = 0 then (a$n :: 'a::comm-ring-1) else a$n + a$(n -
1))
      unfolding m
      by (simp add: )
    finally have ?thesis .}
  ultimately show ?thesis by (cases n, auto)
qed
```

51.15 Finite FPS (i.e. polynomials) and X

```
lemma fps-poly-sum-X:
  assumes z: ∀ i > n. a$i = (0::'a::comm-ring-1)
  shows a = setsum (λi. fps-const (a$i) * X^i) {0..n} (is a = ?r)
proof-
  {fix i
    have a$i = ?r$i
      unfolding fps-setsum-nth fps-mult-left-const-nth X-power-nth
      by (simp add: mult-delta-right setsum-delta' z)
  }
  then show ?thesis unfolding fps-eq-iff by blast
qed
```

51.16 Compositional inverses

```
fun compinv :: 'a fps ⇒ nat ⇒ 'a::{recpower,field} where
  compinv a 0 = X$0
| compinv a (Suc n) = (X$ Suc n - setsum (λi. (compinv a i) * (a^i)$Suc n) {0
.. n}) / (a$1) ^ Suc n

definition fps-inv a = Abs-fps (compinv a)
```

lemma *fps-inv*: **assumes** *a0*: $a\$0 = 0$ **and** *a1*: $a\$1 \neq 0$
shows *fps-inv a oo a* = *X*
proof−
let *?i* = *fps-inv a oo a*
{fix *n*
have *?i* $\$n = X\$n$
proof(*induct n rule*: *nat-less-induct*)
fix *n* **assume** *h*: $\forall m<n.$ *?i*$\$m = X\$m$
{assume *n=0* **hence** *?i* $\$n = X\$n$ **using** *a0*
by (*simp add*: *fps-compose-nth fps-inv-def*)**}**
moreover
{fix *n1* **assume** *n1*: *n* = *Suc n1*
have *?i* $ *n* = *setsum* ($\lambda i.$ (*fps-inv a* $ *i*) ∗ (*a* ^ *i*)$*n*) {*0* .. *n1*} + *fps-inv a* $ *Suc n1* ∗ (*a* $ *1*) ^ *Suc n1*
by (*simp add*: *fps-compose-nth n1 startsby-zero-power-nth-same*[*OF a0*] *del*: *power-Suc*)
also have ... = *setsum* ($\lambda i.$ (*fps-inv a* $ *i*) ∗ (*a* ^ *i*)$*n*) {*0* .. *n1*} + (*X*$ *Suc n1* − *setsum* ($\lambda i.$ (*fps-inv a* $ *i*) ∗ (*a* ^ *i*)$*n*) {*0* .. *n1*})
using *a0 a1 n1* **by** (*simp add*: *fps-inv-def*)
also have ... = *X*$*n* **using** *n1* **by** *simp*
finally have *?i* $ *n* = *X*$*n* **.}**
ultimately show *?i* $ *n* = *X*$*n* **by** (*cases n*, *auto*)
qed}
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

fun *gcompinv* :: *'a fps* ⇒ *'a fps* ⇒ *nat* ⇒ *'a*::{*recpower*,*field*} **where**
gcompinv b a 0 = *b*$*0*
| *gcompinv b a* (*Suc n*) = (*b*$ *Suc n* − *setsum* ($\lambda i.$ (*gcompinv b a i*) ∗ (*a* ^ *i*)$*Suc n*) {*0* .. *n*}) / (*a*$*1*) ^ *Suc n*

definition *fps-ginv b a* = *Abs-fps* (*gcompinv b a*)

lemma *fps-ginv*: **assumes** *a0*: $a\$0 = 0$ **and** *a1*: $a\$1 \neq 0$
shows *fps-ginv b a oo a* = *b*
proof−
let *?i* = *fps-ginv b a oo a*
{fix *n*
have *?i* $\$n = b\$n$
proof(*induct n rule*: *nat-less-induct*)
fix *n* **assume** *h*: $\forall m<n.$ *?i*$\$m = b\$m$
{assume *n=0* **hence** *?i* $\$n = b\$n$ **using** *a0*
by (*simp add*: *fps-compose-nth fps-ginv-def*)**}**
moreover
{fix *n1* **assume** *n1*: *n* = *Suc n1*
have *?i* $ *n* = *setsum* ($\lambda i.$ (*fps-ginv b a* $ *i*) ∗ (*a* ^ *i*)$*n*) {*0* .. *n1*} + *fps-ginv b a* $ *Suc n1* ∗ (*a* $ *1*) ^ *Suc n1*
by (*simp add*: *fps-compose-nth n1 startsby-zero-power-nth-same*[*OF a0*]

```
            del: power-Suc)
      also have ... = setsum (λi. (fps-ginv b a $ i) * (a^i)$n) {0 .. n1} + (b$
Suc n1 − setsum (λi. (fps-ginv b a $ i) * (a^i)$n) {0 .. n1})
        using a0 a1 n1 by (simp add: fps-ginv-def)
      also have ... = b$n using n1 by simp
      finally have ?i $ n = b$n .}
    ultimately show ?i $ n = b$n by (cases n, auto)
  qed}
 then show ?thesis by (simp add: fps-eq-iff)
qed

lemma fps-inv-ginv: fps-inv = fps-ginv X
 apply (auto simp add: expand-fun-eq fps-eq-iff fps-inv-def fps-ginv-def)
 apply (induct-tac n rule: nat-less-induct, auto)
 apply (case-tac na)
 apply simp
 apply simp
 done

lemma fps-compose-1[simp]: 1 oo a = 1
 by (simp add: fps-eq-iff fps-compose-nth fps-power-def mult-delta-left setsum-delta)

lemma fps-compose-0[simp]: 0 oo a = 0
 by (simp add: fps-eq-iff fps-compose-nth)

lemma fps-pow-0: fps-pow n 0 = (if n = 0 then 1 else 0)
 by (induct n, simp-all)

lemma fps-compose-0-right[simp]: a oo 0 = fps-const (a$0)
 by (auto simp add: fps-eq-iff fps-compose-nth fps-power-def fps-pow-0 setsum-0')

lemma fps-compose-add-distrib: (a + b) oo c = (a oo c) + (b oo c)
 by (simp add: fps-eq-iff fps-compose-nth  ring-simps setsum-addf)

lemma fps-compose-setsum-distrib: (setsum f S) oo a = setsum (λi. f i oo a) S
proof−
 {assume ¬ finite S hence ?thesis by simp}
 moreover
 {assume fS: finite S
  have ?thesis
  proof(rule finite-induct[OF fS])
   show setsum f {} oo a = (∑i∈{}. f i oo a) by simp
  next
   fix x F assume fF: finite F and xF: x ∉ F and h: setsum f F oo a = setsum
(λi. f i oo a) F
   show setsum f (insert x F) oo a  = setsum (λi. f i oo a) (insert x F)
    using fF xF h by (simp add: fps-compose-add-distrib)
  qed}
 ultimately show ?thesis by blast
```

qed

lemma *convolution-eq*:
setsum (%*i*. *a* (*i* :: *nat*) * *b* (*n* − *i*)) {*0* .. *n*} = *setsum* (%(*i*,*j*). *a i* * *b j*) {(*i*,*j*). *i* <= *n* ∧ *j* ≤ *n* ∧ *i* + *j* = *n*}
apply (*rule setsum-reindex-cong*[**where** *f*=*fst*])
apply (*clarsimp simp add*: *inj-on-def*)
apply (*auto simp add*: *expand-set-eq image-iff*)
apply (*rule-tac x*= *x* **in** *exI*)
apply *clarsimp*
apply (*rule-tac x*=*n* − *x* **in** *exI*)
apply *arith*
done

lemma *product-composition-lemma*:
assumes *c0*: *c*$*0* = (*0*::'*a*::*idom*) **and** *d0*: *d*$*0* = *0*
shows ((*a oo c*) * (*b oo d*))$*n* = *setsum* (%(*k*,*m*). *a*$*k* * *b*$*m* * (*c* ^ *k* * *d* ^ *m*) $ *n*) {(*k*,*m*). *k* + *m* ≤ *n*} (**is** *?l* = *?r*)
proof−
let *?S* = {(*k*::*nat*, *m*::*nat*). *k* + *m* ≤ *n*}
have *s*: *?S* ⊆ {*0*..*n*} <*> {*0*..*n*} **by** (*auto simp add*: *subset-eq*)
have *f*: *finite* {(*k*::*nat*, *m*::*nat*). *k* + *m* ≤ *n*}
apply (*rule finite-subset*[*OF s*])
by *auto*
have *?r* = *setsum* (%*i*. *setsum* (%(*k*,*m*). *a*$*k* * (*c* ^ *k*)$*i* * *b*$*m* * (*d* ^ *m*) $ (*n* − *i*)) {(*k*,*m*). *k* + *m* ≤ *n*}) {*0*..*n*}
apply (*simp add*: *fps-mult-nth setsum-right-distrib*)
apply (*subst setsum-commute*)
apply (*rule setsum-cong2*)
by (*auto simp add*: *ring-simps*)
also have ... = *?l*
apply (*simp add*: *fps-mult-nth fps-compose-nth setsum-product*)
apply (*rule setsum-cong2*)
apply (*simp add*: *setsum-cartesian-product mult-assoc*)
apply (*rule setsum-mono-zero-right*[*OF f*])
apply (*simp add*: *subset-eq*) **apply** *presburger*
apply *clarsimp*
apply (*rule ccontr*)
apply (*clarsimp simp add*: *not-le*)
apply (*case-tac x* < *aa*)
apply *simp*
apply (*frule-tac startsby-zero-power-prefix*[*rule-format*, *OF c0*])
apply *blast*
apply *simp*
apply (*frule-tac startsby-zero-power-prefix*[*rule-format*, *OF d0*])
apply *blast*
done
finally show *?thesis* **by** *simp*
qed

lemma *product-composition-lemma'*:
assumes *c0*: *c$0 = (0::'a::idom)* **and** *d0*: *d$0 = 0*
shows *((a oo c) * (b oo d))$n = setsum (%k. setsum (%m. a$k * b$m * (c^k * d^m) $ n) {0..n}) {0..n}* (**is** *?l = ?r*)
unfolding *product-composition-lemma[OF c0 d0]*
unfolding *setsum-cartesian-product*
apply (*rule setsum-mono-zero-left*)
apply *simp*
apply (*clarsimp simp add*: *subset-eq*)
apply *clarsimp*
apply (*rule ccontr*)
apply (*subgoal-tac (c^aa * d^ba) $ n = 0*)
apply *simp*
unfolding *fps-mult-nth*
apply (*rule setsum-0'*)
apply (*clarsimp simp add*: *not-le*)
apply (*case-tac aaa < aa*)
apply (*rule startsby-zero-power-prefix[OF c0, rule-format]*)
apply *simp*
apply (*subgoal-tac n - aaa < ba*)
apply (*frule-tac k = ba* **in** *startsby-zero-power-prefix[OF d0, rule-format]*)
apply *simp*
apply *arith*
done

lemma *setsum-pair-less-iff*:
*setsum (%((k::nat),m). a k * b m * c (k + m)) {(k,m). k + m ≤ n} = setsum (%s. setsum (%i. a i * b (s - i) * c s) {0..s}) {0..n}* (**is** *?l = ?r*)
proof−
let *?KM= {(k,m). k + m ≤ n}*
let *?f = %s. UNION {(0::nat)..s} (%i. {(i,s - i)})*
have *th0*: *?KM = UNION {0..n} ?f*
apply (*simp add*: *expand-set-eq*)
apply *arith*
done
show *?l = ?r*
unfolding *th0*
apply (*subst setsum-UN-disjoint*)
apply *auto*
apply (*subst setsum-UN-disjoint*)
apply *auto*
done
qed

lemma *fps-compose-mult-distrib-lemma*:
assumes *c0*: *c$0 = (0::'a::idom)*
shows *((a oo c) * (b oo c))$n = setsum (%s. setsum (%i. a$i * b$(s - i) *

$(c\hat{}s)\ \$\ n)\ \{0..s\})\ \{0..n\}$ (**is** *?l = ?r*)
unfolding *product-composition-lemma*[*OF c0 c0*] *power-add*[*symmetric*]
unfolding *setsum-pair-less-iff*[**where** $a = \%k.\ a\$k$ **and** $b=\%m.\ b\$m$ **and** $c=\%s.\ (c\ \hat{}\ s)\$n$ **and** $n = n$] **..**

lemma *fps-compose-mult-distrib*:
assumes *c0*: $c\$0 = (0::'a::idom)$
shows $(a * b)\ oo\ c = (a\ oo\ c) * (b\ oo\ c)$ (**is** *?l = ?r*)
apply (*simp add*: *fps-eq-iff fps-compose-mult-distrib-lemma*[*OF c0*])
by (*simp add*: *fps-compose-nth fps-mult-nth setsum-left-distrib*)
lemma *fps-compose-setprod-distrib*:
assumes *c0*: $c\$0 = (0::'a::idom)$
shows $(setprod\ a\ S)\ oo\ c = setprod\ (\%k.\ a\ k\ oo\ c)\ S$ (**is** *?l = ?r*)
apply (*cases finite S*)
apply *simp-all*
apply (*induct S rule*: *finite-induct*)
apply *simp*
apply (*simp add*: *fps-compose-mult-distrib*[*OF c0*])
done

lemma *fps-compose-power*: **assumes** *c0*: $c\$0 = (0::'a::idom)$
shows $(a\ oo\ c)\hat{}n = a\hat{}n\ oo\ c$ (**is** *?l = ?r*)
proof−
{assume *n=0* **then have** *?thesis* **by** *simp***}**
moreover
{fix *m* **assume** *m*: *n = Suc m*
have *th0*: $a\hat{}n = setprod\ (\%k.\ a)\ \{0..m\}\ (a\ oo\ c)\ \hat{}\ n = setprod\ (\%k.\ a\ oo\ c)\ \{0..m\}$
by (*simp-all add*: *setprod-constant m*)
then have *?thesis*
by (*simp add*: *fps-compose-setprod-distrib*[*OF c0*])**}**
ultimately show *?thesis* **by** (*cases n*, *auto*)
qed

lemma *fps-const-mult-apply-left*:
$fps\text{-}const\ c * (a\ oo\ b) = (fps\text{-}const\ c * a)\ oo\ b$
by (*simp add*: *fps-eq-iff fps-compose-nth setsum-right-distrib mult-assoc*)

lemma *fps-const-mult-apply-right*:
$(a\ oo\ b) * fps\text{-}const\ (c::'a::comm\text{-}semiring\text{-}1) = (fps\text{-}const\ c * a)\ oo\ b$
by (*auto simp add*: *fps-const-mult-apply-left mult-commute*)

lemma *fps-compose-assoc*:
assumes *c0*: $c\$0 = (0::'a::idom)$ **and** *b0*: $b\$0 = 0$
shows $a\ oo\ (b\ oo\ c) = a\ oo\ b\ oo\ c$ (**is** *?l = ?r*)
proof−
{fix *n*
have $?l\$n = (setsum\ (\lambda i.\ (fps\text{-}const\ (a\$i) * b\hat{}i)\ oo\ c)\ \{0..n\})\$n$

by (*simp add*: *fps-compose-nth fps-compose-power*[*OF c0*] *fps-const-mult-apply-left setsum-right-distrib mult-assoc fps-setsum-nth*)
also have ... = ((*setsum* (λ*i*. *fps-const* (*a*$*i*) ∗ *b* ^ *i*) {*0*..*n*}) *oo c*)$*n*
by (*simp add*: *fps-compose-setsum-distrib*)
also have ... = *?r*$*n*
apply (*simp add*: *fps-compose-nth fps-setsum-nth setsum-left-distrib mult-assoc*)
apply (*rule setsum-cong2*)
apply (*rule setsum-mono-zero-right*)
apply (*auto simp add*: *not-le*)
by (*erule startsby-zero-power-prefix*[*OF b0*, *rule-format*])
finally have *?l*$*n* = *?r*$*n* **.}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *fps-X-power-compose*:
assumes *a0*: *a*$*0*=*0* **shows** *X* ^ *k oo a* = (*a*::(*'a*::*idom fps*)) ^ *k* (**is** *?l* = *?r*)
proof−
{assume *k*=*0* **hence** *?thesis* **by** *simp***}**
moreover
{fix *h* **assume** *h*: *k* = *Suc h*
{fix *n*
{assume *kn*: *k*>*n* **hence** *?l* $ *n* = *?r* $*n* **using** *a0 startsby-zero-power-prefix*[*OF a0*] *h*
by (*simp add*: *fps-compose-nth del*: *power-Suc*)**}**
moreover
{assume *kn*: *k* ≤ *n*
hence *?l*$*n* = *?r*$*n*
by (*simp add*: *fps-compose-nth mult-delta-left setsum-delta*)**}**
moreover have *k* >*n* ∨ *k*≤ *n* **by** *arith*
ultimately have *?l*$*n* = *?r*$*n* **by** *blast***}**
then have *?thesis* **unfolding** *fps-eq-iff* **by** *blast***}**
ultimately show *?thesis* **by** (*cases k*, *auto*)
qed

lemma *fps-inv-right*: **assumes** *a0*: *a*$*0* = *0* **and** *a1*: *a*$*1* ≠ *0*
shows *a oo fps-inv a* = *X*
proof−
let *?ia* = *fps-inv a*
let *?iaa* = *a oo fps-inv a*
have *th0*: *?ia* $ *0* = *0* **by** (*simp add*: *fps-inv-def*)
have *th1*: *?iaa* $ *0* = *0* **using** *a0 a1*
by (*simp add*: *fps-inv-def fps-compose-nth*)
have *th2*: *X*$*0* = *0* **by** *simp*
from *fps-inv*[*OF a0 a1*] **have** *a oo* (*fps-inv a oo a*) = *a oo X* **by** *simp*
then have (*a oo fps-inv a*) *oo a* = *X oo a*
by (*simp add*: *fps-compose-assoc*[*OF a0 th0*] *X-fps-compose-startby0*[*OF a0*])
with *fps-compose-inj-right*[*OF a0 a1*]
show *?thesis* **by** *simp*

qed

lemma *fps-inv-deriv*:
assumes *a0*:*a*$*0* = (*0*::*'a*::{*recpower*,*field*}) **and** *a1*: *a*$*1* ≠ *0*
shows *fps-deriv* (*fps-inv a*) = *inverse* (*fps-deriv a oo fps-inv a*)
proof−
let *?ia* = *fps-inv a*
let *?d* = *fps-deriv a oo ?ia*
let *?dia* = *fps-deriv ?ia*
have *ia0*: *?ia*$*0* = *0* **by** (*simp add*: *fps-inv-def*)
have *th0*: *?d*$*0* ≠ *0* **using** *a1* **by** (*simp add*: *fps-compose-nth fps-deriv-nth*)
from *fps-inv-right*[*OF a0 a1*] **have** *?d* ∗ *?dia* = *1*
by (*simp add*: *fps-compose-deriv*[*OF ia0*, *of a*, *symmetric*])
hence *inverse ?d* ∗ *?d* ∗ *?dia* = *inverse ?d* ∗ *1* **by** *simp*
with *inverse-mult-eq-1*[*OF th0*]
show *?dia* = *inverse ?d* **by** *simp*
qed

51.17 Elementary series

51.17.1 Exponential series

definition *E x* = *Abs-fps* (λ*n*. *x*^*n* / *of-nat* (*fact n*))

lemma *E-deriv*[*simp*]: *fps-deriv* (*E a*) = *fps-const* (*a*::*'a*::{*field*, *recpower*, *ring-char-0*})
∗ *E a* (**is** *?l* = *?r*)
proof−
{fix *n*
have *?l*$*n* = *?r* $ *n*
apply (*auto simp add*: *E-def field-simps power-Suc*[*symmetric*]*simp del*: *fact-Suc*
of-nat-Suc power-Suc)
by (*simp add*: *of-nat-mult ring-simps*)**}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *E-unique-ODE*:
fps-deriv a = *fps-const c* ∗ *a* ⟷ *a* = *fps-const* (*a*$*0*) ∗ *E* (*c* :: *'a*::{*field*,
ring-char-0, *recpower*})
(**is** *?lhs* ⟷ *?rhs*)
proof−
{assume *d*: *?lhs*
from *d* **have** *th*: ⋀*n*. *a* $ *Suc n* = *c* ∗ *a*$*n* / *of-nat* (*Suc n*)
by (*simp add*: *fps-deriv-def fps-eq-iff field-simps del*: *of-nat-Suc*)
{fix *n* **have** *a*$*n* = *a*$*0* ∗ *c* ^ *n*/ (*of-nat* (*fact n*))
apply (*induct n*)
apply *simp*
unfolding *th*
using *fact-gt-zero*
apply (*simp add*: *field-simps del*: *of-nat-Suc fact.simps*)
apply (*drule sym*)

```
    by (simp add: ring-simps of-nat-mult power-Suc)}
  note th' = this
  have ?rhs
    by (auto simp add: fps-eq-iff fps-const-mult-left E-def intro : th')}
moreover
{assume h: ?rhs
  have ?lhs
    apply (subst h)
    apply simp
    apply (simp only: h[symmetric])
  by simp}
ultimately show ?thesis by blast
qed

lemma E-add-mult: E (a + b) = E (a::'a::{ring-char-0, field, recpower}) * E b
(is ?l = ?r)
proof-
  have fps-deriv (?r) = fps-const (a+b) * ?r
    by (simp add: fps-const-add[symmetric] ring-simps del: fps-const-add)
  then have ?r = ?l apply (simp only: E-unique-ODE)
    by (simp add: fps-mult-nth E-def)
  then show ?thesis ..
qed

lemma E-nth[simp]: E a $ n = a^n / of-nat (fact n)
  by (simp add: E-def)

lemma E0[simp]: E (0::'a::{field, recpower}) = 1
  by (simp add: fps-eq-iff power-0-left)

lemma E-neg: E (- a) = inverse (E (a::'a::{ring-char-0, field, recpower}))
proof-
  from E-add-mult[of a - a] have th0: E a * E (- a) = 1
    by (simp )
  have th1: E a $ 0 ≠ 0 by simp
  from fps-inverse-unique[OF th1 th0] show ?thesis by simp
qed

lemma E-nth-deriv[simp]: fps-nth-deriv n (E (a::'a::{field, recpower, ring-char-0}))
= (fps-const a)^n * (E a)
  by (induct n, auto simp add: power-Suc)

lemma fps-compose-uminus: - (a::'a::ring-1 fps) oo c = - (a oo c)
  by (simp add: fps-eq-iff fps-compose-nth ring-simps setsum-negf[symmetric])

lemma fps-compose-sub-distrib:
  shows (a - b) oo (c::'a::ring-1 fps) = (a oo c) - (b oo c)
  unfolding diff-minus fps-compose-uminus fps-compose-add-distrib ..
```

lemma *X-fps-compose*:*X oo a = Abs-fps* (*λn. if n = 0 then* (*0*::*'a*::*comm-ring-1*) *else a*$*n*)
by (*simp add*: *fps-eq-iff fps-compose-nth mult-delta-left setsum-delta power-Suc*)

lemma *X-compose-E*[*simp*]: *X oo E* (*a*::*'a*::{*field*, *recpower*}) = *E a − 1*
by (*simp add*: *fps-eq-iff X-fps-compose*)

lemma *LE-compose*:
assumes *a*: *a≠0*
shows *fps-inv* (*E a − 1*) *oo* (*E a − 1*) = *X*
and (*E a − 1*) *oo fps-inv* (*E a − 1*) = *X*
proof−
let *?b* = *E a − 1*
have *b0*: *?b* $ *0* = *0* **by** *simp*
have *b1*: *?b* $ *1* ≠ *0* **by** (*simp add*: *a*)
from *fps-inv*[*OF b0 b1*] **show** *fps-inv* (*E a − 1*) *oo* (*E a − 1*) = *X* **.**
from *fps-inv-right*[*OF b0 b1*] **show** (*E a − 1*) *oo fps-inv* (*E a − 1*) = *X* **.**
qed

lemma *fps-const-inverse*:
inverse (*fps-const* (*a*::*'a*::{*field*, *division-by-zero*})) = *fps-const* (*inverse a*)
apply (*auto simp add*: *fps-eq-iff fps-inverse-def*) **by** (*case-tac n*, *auto*)

lemma *inverse-one-plus-X*:
inverse (*1* + *X*) = *Abs-fps* (*λn.* (*−* 1 ::*'a*::{*field*, *recpower*}) ^*n*)
(**is** *inverse ?l* = *?r*)
proof−
have *th*: *?l* ∗ *?r* = *1*
apply (*auto simp add*: *ring-simps fps-eq-iff X-mult-nth minus-one-power-iff*)
apply *presburger*+
done
have *th'*: *?l* $ *0* ≠ *0* **by** (*simp add*:)
from *fps-inverse-unique*[*OF th' th*] **show** *?thesis* **.**
qed

lemma *E-power-mult*: (*E* (*c*::*'a*::{*field*,*recpower*,*ring-char-0*})) ^*n* = *E* (*of-nat n* ∗ *c*)
by (*induct n*, *auto simp add*: *ring-simps E-add-mult power-Suc*)

51.17.2 Logarithmic series

definition (*L*::*'a*::{*field*, *ring-char-0*,*recpower*} *fps*)
= *Abs-fps* (*λn.* (*−* 1) ^ *Suc n* / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv L* = *inverse* (*1* + *X*)
unfolding *inverse-one-plus-X*
by (*simp add*: *L-def fps-eq-iff power-Suc del*: *of-nat-Suc*)

lemma *L-nth*: *L $ n = (− 1) ^ Suc n / of-nat n*
by (*simp add*: *L-def*)

lemma *L-E-inv*:
assumes *a*: *a≠ (0::'a::{field,division-by-zero,ring-char-0,recpower})*
shows *L = fps-const a * fps-inv (E a − 1)* (**is** *?l = ?r*)
proof−
let *?b = E a − 1*
have *b0*: *?b $ 0 = 0* **by** *simp*
have *b1*: *?b $ 1 ≠ 0* **by** (*simp add*: *a*)
have *fps-deriv (E a − 1) oo fps-inv (E a − 1) = (fps-const a * (E a − 1) + fps-const a) oo fps-inv (E a − 1)*
by (*simp add*: *ring-simps*)
also have *... = fps-const a * (X + 1)* **apply** (*simp add*: *fps-compose-add-distrib fps-const-mult-apply-left*[*symmetric*] *fps-inv-right*[*OF b0 b1*])
by (*simp add*: *ring-simps*)
finally have *eq*: *fps-deriv (E a − 1) oo fps-inv (E a − 1) = fps-const a * (X + 1)* **.**
from *fps-inv-deriv*[*OF b0 b1*, *unfolded eq*]
have *fps-deriv (fps-inv ?b) = fps-const (inverse a) / (X + 1)*
by (*simp add*: *fps-const-inverse eq fps-divide-def fps-inverse-mult*)
hence *fps-deriv (fps-const a * fps-inv ?b) = inverse (X + 1)*
using *a* **by** (*simp add*: *fps-divide-def field-simps*)
hence *fps-deriv ?l = fps-deriv ?r*
by (*simp add*: *fps-deriv-L add-commute*)
then show *?thesis* **unfolding** *fps-deriv-eq-iff*
by (*simp add*: *L-nth fps-inv-def*)
qed

51.17.3 Formal trigonometric functions

definition *fps-sin* (*c*::'*a*::{*field*, *recpower*, *ring-char-0*}) =
Abs-fps (*λn. if even n then 0 else (− 1) ^((n − 1) div 2) * c^n /(of-nat (fact n))*)

definition *fps-cos* (*c*::'*a*::{*field*, *recpower*, *ring-char-0*}) = *Abs-fps* (*λn. if even n then (− 1) ^ (n div 2) * c^n / (of-nat (fact n)) else 0*)

lemma *fps-sin-deriv*:
*fps-deriv (fps-sin c) = fps-const c * fps-cos c*
(**is** *?lhs = ?rhs*)
proof−
{fix *n*::*nat*
{assume *en*: *even n*
have *?lhs$n = of-nat (n+1) * (fps-sin c $ (n+1))* **by** *simp*
also have *... = of-nat (n+1) * ((− 1)^(n div 2) * c^Suc n / of-nat (fact (Suc n)))*
using *en* **by** (*simp add*: *fps-sin-def*)

```
    also have ... = (− 1) ^(n div 2) * c^Suc n * (of-nat (n+1) / (of-nat (Suc
n) * of-nat (fact n)))
      unfolding fact-Suc of-nat-mult
      by (simp add: field-simps del: of-nat-add of-nat-Suc)
    also have ... = (− 1) ^(n div 2) *c^Suc n / of-nat (fact n)
      by (simp add: field-simps del: of-nat-add of-nat-Suc)
    finally have ?lhs $n = ?rhs$n using en
      by (simp add: fps-cos-def ring-simps power-Suc )}
  then have ?lhs $ n = ?rhs $ n
    by (cases even n, simp-all add: fps-deriv-def fps-sin-def fps-cos-def) }
 then show ?thesis by (auto simp add: fps-eq-iff)
qed

lemma fps-cos-deriv:
 fps-deriv (fps-cos c) = fps-const (− c)* (fps-sin c)
 (is ?lhs = ?rhs)
proof−
 have th0: ⋀n. − ((− 1::'a) ^ n) = (− 1) ^Suc n by (simp add: power-Suc)
 have th1: ⋀n. odd n⟹ Suc ((n − 1) div 2) = Suc n div 2 by presburger
 {fix n::nat
  {assume en: odd n
    from en have n0: n ≠0 by presburger
    have ?lhs$n = of-nat (n+1) * (fps-cos c $ (n+1)) by simp
    also have ... = of-nat (n+1) * ((− 1) ^((n + 1) div 2) * c^Suc n / of-nat
(fact (Suc n)))
      using en by (simp add: fps-cos-def)
    also have ... = (− 1) ^((n + 1) div 2)*c^Suc n * (of-nat (n+1) / (of-nat
(Suc n) * of-nat (fact n)))
      unfolding fact-Suc of-nat-mult
      by (simp add: field-simps del: of-nat-add of-nat-Suc)
    also have ... = (− 1) ^((n + 1) div 2) * c^Suc n / of-nat (fact n)
      by (simp add: field-simps del: of-nat-add of-nat-Suc)
    also have ... = (− ((− 1) ^((n − 1) div 2))) * c^Suc n / of-nat (fact n)
      unfolding th0 unfolding th1[OF en] by simp
    finally have ?lhs $n = ?rhs$n using en
      by (simp add: fps-sin-def ring-simps power-Suc)}
  then have ?lhs $ n = ?rhs $ n
    by (cases even n, simp-all add: fps-deriv-def fps-sin-def
      fps-cos-def) }
 then show ?thesis by (auto simp add: fps-eq-iff)
qed

lemma fps-sin-cos-sum-of-squares:
 fps-cos c ^ 2 + fps-sin c ^ 2 = 1 (is ?lhs = 1)
proof−
 have fps-deriv ?lhs = 0
   apply (simp add: fps-deriv-power fps-sin-deriv fps-cos-deriv power-Suc)
  by (simp add: fps-power-def ring-simps fps-const-neg[symmetric] del: fps-const-neg)
 then have ?lhs = fps-const (?lhs $ 0)
```

unfolding *fps-deriv-eq-0-iff* **.**
also have ... = *1*
by (*auto simp add*: *fps-eq-iff fps-power-def numeral-2-eq-2 fps-mult-nth fps-cos-def fps-sin-def*)
finally show *?thesis* **.**
qed

definition *fps-tan c* = *fps-sin c* / *fps-cos c*

lemma *fps-tan-deriv*: *fps-deriv*(*fps-tan c*) = *fps-const c*/ (*fps-cos c* ^ *2*)
proof−
have *th0*: *fps-cos c* $ *0* ≠ *0* **by** (*simp add*: *fps-cos-def*)
show *?thesis*
using *fps-sin-cos-sum-of-squares*[*of c*]
apply (*simp add*: *fps-tan-def fps-divide-deriv*[*OF th0*] *fps-sin-deriv fps-cos-deriv add*: *fps-const-neg*[*symmetric*] *ring-simps power2-eq-square del*: *fps-const-neg*)
unfolding *right-distrib*[*symmetric*]
by *simp*
qed

end

52 Some applications of formal power series and some properties over complex numbers

theory *Formal-Power-Series-Examples*
imports *Formal-Power-Series Binomial Complex*
begin

53 The generalized binomial theorem

lemma *gbinomial-theorem*:
((*a*::'*a*::{*ring-char-0*, *field*, *division-by-zero*, *recpower*})+*b*) ^ *n* = (∑ *k*=*0*..*n*. *of-nat* (*n choose k*) * *a*^*k* * *b*^(*n*−*k*))
proof−
from *E-add-mult*[*of a b*]
have (*E* (*a* + *b*)) $ *n* = (*E a* * *E b*)$*n* **by** *simp*
then have (*a* + *b*) ^ *n* = (∑ *i*::*nat* = *0*::*nat*..*n*. *a* ^ *i* * *b* ^ (*n* − *i*) * (*of-nat* (*fact n*) / *of-nat* (*fact i* * *fact* (*n* − *i*))))
by (*simp add*: *field-simps fps-mult-nth of-nat-mult*[*symmetric*] *setsum-right-distrib*)
then show *?thesis*
apply *simp*
apply (*rule setsum-cong2*)
apply *simp*
apply (*frule binomial-fact*[**where** *?'a* = '*a*, *symmetric*])
by (*simp add*: *field-simps of-nat-mult*)
qed

And the nat-form – also available from Binomial.thy

lemma *binomial-theorem*: $(a+b)\ \hat{}\ n = (\sum k{=}0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
using *gbinomial-theorem*[*of of-nat a of-nat b n*]
unfolding *of-nat-add*[*symmetric*] *of-nat-power*[*symmetric*] *of-nat-mult*[*symmetric*] *of-nat-setsum*[*symmetric*]
by *simp*

54 The binomial series and Vandermonde's identity

definition *fps-binomial a* = *Abs-fps* ($\lambda n.$ *a gchoose n*)

lemma *fps-binomial-nth*[*simp*]: *fps-binomial a* $ *n* = *a gchoose n*
by (*simp add*: *fps-binomial-def*)

lemma *fps-binomial-ODE-unique*:
fixes *c* :: *'a*::{*field*, *recpower*,*ring-char-0*}
shows *fps-deriv a* = (*fps-const c* * *a*) / (*1* + *X*) ⟷ *a* = *fps-const* (*a*$*0*) * *fps-binomial c*
(**is** *?lhs* ⟷ *?rhs*)
proof−
let *?da* = *fps-deriv a*
let *?x1* = (*1* + *X*):: *'a fps*
let *?l* = *?x1* * *?da*
let *?r* = *fps-const c* * *a*
have *x10*: *?x1* $ *0* ≠ *0* **by** *simp*
have *?l* = *?r* ⟷ *inverse ?x1* * *?l* = *inverse ?x1* * *?r* **by** *simp*
also have ... ⟷ *?da* = (*fps-const c* * *a*) / *?x1*
apply (*simp only*: *fps-divide-def mult-assoc*[*symmetric*] *inverse-mult-eq-1*[*OF x10*])
by (*simp add*: *ring-simps*)
finally have *eq*: *?l* = *?r* ⟷ *?lhs* **by** *simp*
moreover
{**assume** *h*: *?l* = *?r*
{**fix** *n*
from *h* **have** *lrn*: *?l* $ *n* = *?r*$*n* **by** *simp*

from *lrn*
have *a*$ *Suc n* = ((*c* − *of-nat n*) / *of-nat* (*Suc n*)) * *a* $*n*
apply (*simp add*: *ring-simps del*: *of-nat-Suc*)
by (*cases n*, *simp-all add*: *field-simps del*: *of-nat-Suc*)
}
note *th0* = *this*
{**fix** *n* **have** *a*$*n* = (*c gchoose n*) * *a*$*0*
proof(*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc m*)

```
      thus ?case unfolding th0
        apply (simp add: field-simps del: of-nat-Suc)
        unfolding mult-assoc[symmetric] gbinomial-mult-1
        by (simp add: ring-simps)
    qed}
  note th1 = this
  have ?rhs
    apply (simp add: fps-eq-iff)
    apply (subst th1)
    by (simp add: ring-simps)}
 moreover
 {assume h: ?rhs
 have th00:⋀x y. x * (a$0 * y) = a$0 * (x*y) by (simp add: mult-commute)
  have ?l = ?r
    apply (subst h)
    apply (subst (2) h)
    apply (clarsimp simp add: fps-eq-iff ring-simps)
    unfolding mult-assoc[symmetric] th00 gbinomial-mult-1
    by (simp add: ring-simps gbinomial-mult-1)}
 ultimately show ?thesis by blast
qed

lemma fps-binomial-deriv: fps-deriv (fps-binomial c) = fps-const c * fps-binomial
c / (1 + X)
proof−
 let ?a = fps-binomial c
 have th0: ?a = fps-const (?a$0) * ?a by (simp)
 from iffD2[OF fps-binomial-ODE-unique, OF th0] show ?thesis .
qed

lemma fps-binomial-add-mult: fps-binomial (c+d) = fps-binomial c * fps-binomial
d (is ?l = ?r)
proof−
 let ?P = ?r − ?l
 let ?b = fps-binomial
 let ?db = λx. fps-deriv (?b x)
 have fps-deriv ?P = ?db c * ?b d + ?b c * ?db d − ?db (c + d) by simp
 also have ... = inverse (1 + X) * (fps-const c * ?b c * ?b d + fps-const d * ?b
c * ?b d − fps-const (c+d) * ?b (c + d))
   unfolding fps-binomial-deriv
   by (simp add: fps-divide-def ring-simps)
 also have ... = (fps-const (c + d)/ (1 + X)) * ?P
  by (simp add: ring-simps fps-divide-def fps-const-add[symmetric] del: fps-const-add)
 finally have th0: fps-deriv ?P = fps-const (c+d) * ?P / (1 + X)
   by (simp add: fps-divide-def)
 have ?P = fps-const (?P$0) * ?b (c + d)
   unfolding fps-binomial-ODE-unique[symmetric]
   using th0 by simp
 hence ?P = 0 by (simp add: fps-mult-nth)
```

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

lemma *fps-minomial-minus-one*: *fps-binomial* (− *1*) = *inverse* (*1* + *X*)
(**is** *?l* = *inverse ?r*)
proof−
have *th*: *?r*$*0* ≠ *0* **by** *simp*
have *th'*: *fps-deriv* (*inverse ?r*) = *fps-const* (− *1*) ∗ *inverse ?r* / (*1* + *X*)
by (*simp add*: *fps-inverse-deriv*[*OF th*] *fps-divide-def power2-eq-square mult-commute fps-const-neg*[*symmetric*] *del*: *fps-const-neg*)
have *eq*: *inverse ?r* $ *0* = *1*
by (*simp add*: *fps-inverse-def*)
from *iffD1*[*OF fps-binomial-ODE-unique*[*of inverse* (*1* + *X*) − *1*] *th'*] *eq*
show *?thesis* **by** (*simp add*: *fps-inverse-def*)
qed

lemma *gbinomial-Vandermond*: *setsum* (λ*k*. (*a gchoose k*) ∗ (*b gchoose* (*n* − *k*))) {*0*..*n*} = (*a* + *b*) *gchoose n*
proof−
let *?ba* = *fps-binomial a*
let *?bb* = *fps-binomial b*
let *?bab* = *fps-binomial* (*a* + *b*)
from *fps-binomial-add-mult*[*of a b*] **have** *?bab* $ *n* = (*?ba* ∗ *?bb*)$*n* **by** *simp*
then show *?thesis* **by** (*simp add*: *fps-mult-nth*)
qed

lemma *binomial-Vandermond*: *setsum* (λ*k*. (*a choose k*) ∗ (*b choose* (*n* − *k*))) {*0*..*n*} = (*a* + *b*) *choose n*
using *gbinomial-Vandermond*[*of* (*of-nat a*) *of-nat b n*]

apply (*simp only*: *binomial-gbinomial*[*symmetric*] *of-nat-mult*[*symmetric*] *of-nat-setsum*[*symmetric*] *of-nat-add*[*symmetric*])
by *simp*

lemma *binomial-symmetric*: **assumes** *kn*: *k* ≤ *n*
shows *n choose k* = *n choose* (*n* − *k*)
proof−
from *kn* **have** *kn'*: *n* − *k* ≤ *n* **by** *arith*
from *binomial-fact-lemma*[*OF kn*] *binomial-fact-lemma*[*OF kn'*]
have *fact k* ∗ *fact* (*n* − *k*) ∗ (*n choose k*) = *fact* (*n* − *k*) ∗ *fact* (*n* − (*n* − *k*)) ∗ (*n choose* (*n* − *k*)) **by** *simp*
then show *?thesis* **using** *kn* **by** *simp*
qed

lemma *binomial-Vandermond-same*: *setsum* (λ*k*. (*n choose k*) ^*2*) {*0*..*n*} = (*2*∗*n*) *choose n*
using *binomial-Vandermond*[*of n n n*,*symmetric*]
unfolding *nat-mult-2* **apply** (*simp add*: *power2-eq-square*)
apply (*rule setsum-cong2*)

by (*auto intro*: *binomial-symmetric*)

55 Relation between formal sine/cosine and the exponential FPS

lemma *Eii-sin-cos*:
E (*ii* ∗ *c*) = *fps-cos c* + *fps-const ii* ∗ *fps-sin c*
(**is** *?l* = *?r*)
proof−
{fix *n*::*nat*
{assume *en*: *even n*
from *en* **obtain** *m* **where** *m*: *n* = *2*∗*m*
unfolding *even-mult-two-ex* **by** *blast*

have *?l* $*n* = *?r*$*n*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib*
power-mult power-minus)**}**
moreover
{assume *on*: *odd n*
from *on* **obtain** *m* **where** *m*: *n* = *2*∗*m* + *1*
unfolding *odd-nat-equiv-def2* **by** (*auto simp add*: *nat-mult-2*)
have *?l* $*n* = *?r*$*n*
by (*simp add*: *m fps-sin-def fps-cos-def power-mult-distrib*
power-mult power-minus)**}**
ultimately have *?l* $*n* = *?r*$*n* **by** *blast***}**
then show *?thesis* **by** (*simp add*: *fps-eq-iff*)
qed

lemma *fps-sin-neg*[*simp*]: *fps-sin* (− *c*) = − *fps-sin c*
by (*simp add*: *fps-eq-iff fps-sin-def*)

lemma *fps-cos-neg*[*simp*]: *fps-cos* (− *c*) = *fps-cos c*
by (*simp add*: *fps-eq-iff fps-cos-def*)
lemma *E-minus-ii-sin-cos*: *E* (− (*ii* ∗ *c*)) = *fps-cos c* − *fps-const ii* ∗ *fps-sin c*
unfolding *minus-mult-right Eii-sin-cos* **by** *simp*

lemma *fps-const-minus*: *fps-const* (*c*::*'a*::*group-add*) − *fps-const d* = *fps-const* (*c*
− *d*) **by** (*simp add*: *fps-eq-iff fps-const-def*)

lemma *fps-number-of-fps-const*: *number-of i* = *fps-const* (*number-of i* :: *'a*:: {*comm-ring-1*,
number-ring})
apply (*subst* (*2*) *number-of-eq*)
apply(*rule int-induct*[*of* - *0*])
apply (*simp-all add*: *number-of-fps-def*)
by (*simp-all add*: *fps-const-add*[*symmetric*] *fps-const-minus*[*symmetric*])

lemma *fps-cos-Eii*:
fps-cos c = (*E* (*ii* ∗ *c*) + *E* (− *ii* ∗ *c*)) / *fps-const 2*

proof−
have *th*: *fps-cos c* + *fps-cos c* = *fps-cos c* ∗ *fps-const 2*
by (*simp add*: *fps-eq-iff fps-number-of-fps-const complex-number-of-def* [*symmetric*])
show *?thesis*
unfolding *Eii-sin-cos minus-mult-commute*
by (*simp add*: *fps-number-of-fps-const fps-divide-def fps-const-inverse th complex-number-of-def* [*symmetric*])
qed

lemma *fps-sin-Eii*:
fps-sin c = (*E* (*ii* ∗ *c*) − *E* (− *ii* ∗ *c*)) / *fps-const* (*2*∗*ii*)
proof−
have *th*: *fps-const* i ∗ *fps-sin c* + *fps-const* i ∗ *fps-sin c* = *fps-sin c* ∗ *fps-const* (*2* ∗ *ii*)
by (*simp add*: *fps-eq-iff fps-number-of-fps-const complex-number-of-def* [*symmetric*])
show *?thesis*
unfolding *Eii-sin-cos minus-mult-commute*
by (*simp add*: *fps-divide-def fps-const-inverse th*)
qed

lemma *fps-const-mult-2*: *fps-const* (*2*::*'a*::*number-ring*) ∗ *a* = *a* +*a*
by (*simp add*: *fps-eq-iff fps-number-of-fps-const*)

lemma *fps-const-mult-2-right*: *a* ∗ *fps-const* (*2*::*'a*::*number-ring*) = *a* +*a*
by (*simp add*: *fps-eq-iff fps-number-of-fps-const*)

lemma *fps-tan-Eii*:
fps-tan c = (*E* (*ii* ∗ *c*) − *E* (− *ii* ∗ *c*)) / (*fps-const ii* ∗ (*E* (*ii* ∗ *c*) + *E* (− *ii* ∗ *c*)))
unfolding *fps-tan-def fps-sin-Eii fps-cos-Eii mult-minus-left E-neg*
apply (*simp add*: *fps-divide-def fps-inverse-mult fps-const-mult*[*symmetric*] *fps-const-inverse del*: *fps-const-mult*)
by *simp*

lemma *fps-demoivre*: (*fps-cos a* + *fps-const ii* ∗ *fps-sin a*) ^*n* = *fps-cos* (*of-nat n* ∗ *a*) + *fps-const ii* ∗ *fps-sin* (*of-nat n* ∗ *a*)
unfolding *Eii-sin-cos*[*symmetric*] *E-power-mult*
by (*simp add*: *mult-ac*)

Now some trigonometric identities

lemma *fps-sin-add*:
fps-sin (*a*+*b*) = *fps-sin* (*a*::*complex*) ∗ *fps-cos b* + *fps-cos a* ∗ *fps-sin b*
proof−
let *?ca* = *fps-cos a*
let *?cb* = *fps-cos b*
let *?sa* = *fps-sin a*
let *?sb* = *fps-sin b*
let *?i* = *fps-const ii*
have *i*: *?i*∗*?i* = *fps-const* −*1* **by** *simp*
have *fps-sin* (*a* + *b*) =

((?ca + ?i * ?sa) * (?cb + ?i*?sb) − (?ca − ?i*?sa) * (?cb − ?i*?sb)) *
fps-const (− (i / 2))
apply(*simp add*: *fps-sin-Eii*[*of a+b*] *fps-divide-def minus-mult-commute*)
unfolding *right-distrib*
apply (*simp add*: *Eii-sin-cos E-minus-ii-sin-cos fps-const-inverse E-add-mult*)
by (*simp add*: *ring-simps*)
also have ... = (*?ca * ?cb + ?i*?ca * ?sb + ?i * ?sa * ?cb + (?i*?i)*?sa*?sb*
− *?ca*?cb + ?i*?ca * ?sb + ?i*?sa*?cb − (?i*?i)*?sa * ?sb*) * *fps-const* (− *ii/2*)
by (*simp add*: *ring-simps*)
also have ... = (*fps-const 2 * ?i * (?ca * ?sb + ?sa * ?cb)*) * *fps-const* (−
ii/2)
apply *simp*
apply (*simp add*: *ring-simps*)
apply (*simp add*: *ring-simps add*: *fps-const-mult*[*symmetric*] *del*:*fps-const-mult*)
unfolding *fps-const-mult-2-right*
by (*simp add*: *ring-simps*)
also have ... = (*fps-const 2 * ?i * fps-const* (− *ii/2*)) * (*?ca * ?sb + ?sa *
?cb)
by (*simp only*: *mult-ac*)
also have ... = *?sa * ?cb + ?ca*?sb*
by *simp*
finally show *?thesis* **.**
qed

lemma *fps-cos-add*:
fps-cos (*a*+*b*) = *fps-cos* (*a*::*complex*) * *fps-cos b* − *fps-sin a * fps-sin b*
proof−
let *?ca* = *fps-cos a*
let *?cb* = *fps-cos b*
let *?sa* = *fps-sin a*
let *?sb* = *fps-sin b*
let *?i* = *fps-const ii*
have *i*: *?i*?i* = *fps-const −1* **by** *simp*
have *i'*: ⋀*x*. *?i * (?i * x)* = − *x*
apply (*simp add*: *mult-assoc*[*symmetric*] *i*)
by (*simp add*: *fps-eq-iff*)
have *m1*: ⋀*x*. *x * fps-const* (*−1* ::*complex*) = − *x* ⋀*x*. *fps-const* (*−1* :: *complex*)
* *x* = − *x*
by (*auto simp add*: *fps-eq-iff*)

have *fps-cos* (*a* + *b*) =
((*?ca* + *?i* * *?sa*) * (*?cb* + *?i*?sb*) + (*?ca* − *?i*?sa*) * (*?cb* − *?i*?sb*)) *
fps-const (*1*/ *2*)
apply(*simp add*: *fps-cos-Eii*[*of a+b*] *fps-divide-def minus-mult-commute*)
unfolding *right-distrib minus-add-distrib*
apply (*simp add*: *Eii-sin-cos E-minus-ii-sin-cos fps-const-inverse E-add-mult*)
by (*simp add*: *ring-simps*)
also have ... = (*?ca * ?cb + ?i*?ca * ?sb + ?i * ?sa * ?cb + (?i*?i)*?sa*?sb*
+ *?ca*?cb − ?i*?ca * ?sb − ?i*?sa*?cb + (?i*?i)*?sa * ?sb*) * *fps-const* (*1*/*2*)

apply *simp*
by (*simp add*: *ring-simps i' m1*)
also have ... = (*fps-const 2* ∗ (*?ca* ∗ *?cb* − *?sa* ∗ *?sb*)) ∗ *fps-const* (*1*/*2*)
apply *simp*
by (*simp add*: *ring-simps m1 fps-const-mult-2-right*)
also have ... = (*fps-const 2* ∗ *fps-const* (*1*/*2*)) ∗ (*?ca* ∗ *?cb* − *?sa* ∗ *?sb*)
by (*simp only*: *mult-ac*)
also have ... = *?ca* ∗ *?cb* − *?sa*∗*?sb*
by *simp*
finally show *?thesis* **.**
qed

end

56 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).

56.1 Proof text

theorem *tnd*: *A* ∨ ¬ *A*
proof −
let *?P* = λ*X*. *X* = *False* ∨ *X* = *True* ∧ *A*
let *?Q* = λ*X*. *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 (rule ext)
        fix x show ?P x = ?Q x
        proof
          assume ?P x
          thus ?Q x
          proof
            assume x = False
            from this and a have x = False ∧ A ..
            thus ?Q x ..
          next
            assume x = True ∧ A
            hence x = True ..
            thus ?Q x ..
          qed
        next
          assume ?Q x
          thus ?P x
          proof
            assume x = False ∧ A
            hence x = False ..
            thus ?P x ..
          next
            assume x = True
            from this and a have x = True ∧ A ..
            thus ?P x ..
          qed
        qed
      qed
      with neq show False by contradiction
    qed
    thus ?thesis ..
  qed
qed
```

qed

56.2 Proof term of text

```
disjE · - · - · - · - · -
 (thm.Hilbert-Choice.someI · (λX. X = False ∨ X = True ∧ ?A) · - ·
   (disjI1 · - · - · - · (thm.HOL.refl · -))) ·
 (λH: -.
   disjE · - · - · - · - · -
    (thm.Hilbert-Choice.someI · (λX. X = False ∧ ?A ∨ X = True) · - ·
      (disjI2 · - · - · - · (thm.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))) ·
            (thm.HOL.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)))
```

56.3 Proof script

theorem *tnd'*: $A \vee \neg A$
apply (*subgoal-tac*
(((*SOME x*. x = *False* $\vee$ x = *True* $\wedge$ A) = *False*) $\vee$
((*SOME x*. x = *False* $\vee$ x = *True* $\wedge$ A) = *True*) $\wedge$ A) $\wedge$
(((*SOME x*. x = *False* $\wedge$ A $\vee$ x = *True*) = *False*) $\wedge$ A $\vee$
((*SOME x*. x = *False* $\wedge$ A $\vee$ 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*) $\vee$ (x = *True*) $\wedge$ A) $\neq$
(λx. (x = *False*) $\wedge$ A $\vee$ (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

56.4 Proof term of script

conjE · - · - · - · - · - · -
(*conjI* · - · - · - · -
(*thm.Hilbert-Choice.someI* · (λ*x*. *x* = *False* ∨ *x* = *True* ∧ *?A*) · - · -
(*disjI1* · - · - · - · - (*thm.HOL.refl* · -))) ·
(*thm.Hilbert-Choice.someI* · (λ*x*. *x* = *False* ∧ *?A* ∨ *x* = *True*) · - · -
(*disjI2* · - · - · - · - (*thm.HOL.refl* · -)))) ·
(**λ**(*H*: -) *Ha*: -.
disjE · - · - · - · - · - · - · *H* ·
(**λ***H*: -.
disjE · - · - · - · - · - · - · *Ha* ·
(**λ***H*: -. *conjE* · - · - · - · - · - · - · *H* · (**λ***H*: -. *disjI1* · - · - · -)) ·
(**λ***Ha*: -.
disjI2 · - · - · - · - ·
(*notI* · - · -
(**λ***Hb*: -.
notE · - · - · - · -
(*notI* · - · -
(**λ***Hb*: -.
False-neq-True · - · -
(*HOL.trans* · - · - · - · - · - · - (*HOL.sym* · - · - · - · - · *H*) ·
(*HOL.trans* · - · - · - · - · - · -
(*arg-cong* · (λ*x*. *x* = *False* ∨ *x* = *True* ∧ *?A*) ·
(λ*x*. *x* = *False* ∧ *?A* ∨ *x* = *True*) ·
Eps ·
Hb) ·
Ha)))) ·
(*thm.HOL.ext* · - · - · - · -
(**λ***x*. *iffI* · - · - · - · -
(**λ***H*: -.
disjE · - · - · - · - · - · - · *H* ·
(**λ***H*: -. *disjI1* · - · - · - · - (*conjI* · - · - · - · - · *H* · *Hb*)) ·
(**λ***H*: -.
conjE · - · - · - · - · - · - · *H* ·
(**λ**(*H*: -) *Ha*: -. *disjI2* · - · - · - · - · *H*))) ·
(**λ***H*: -.
disjE · - · - · - · - · - · - · *H* ·
(**λ***H*: -.
conjE · - · - · - · - · - · - · *H* ·
(**λ**(*H*: -) *Ha*: -. *disjI1* · - · - · - · - · *H*)) ·
(**λ***H*: -.
disjI2 · - · - · - · - (*conjI* · - · - · - · - · *H* · *Hb*))))))))) ·
(**λ***H*: -. *conjE* · - · - · - · - · - · - · *H* · (**λ***H*: -. *disjI1* · - · - · -)))

end

57 Installing an oracle for SVC (Stanford Validity Checker)

theory *SVC-Oracle*
imports *Main*
uses *svc-funcs.ML*
begin

consts
  iff-keep :: [*bool*, *bool*] => *bool*
  iff-unfold :: [*bool*, *bool*] => *bool*

hide *const iff-keep iff-unfold*

oracle *svc-oracle* = *Svc.oracle*

ML ⟪
(*
Installing the oracle for SVC (Stanford Validity Checker)

The following code merely CALLS the oracle;
  the soundness−critical functions are at svc-funcs.ML

Based upon the work of Søren T. Heilmann
*)

(*Generalize an Isabelle formula, replacing by Vars
  all subterms not intelligible to SVC.*)
fun svc-abstract t =
  let
    (*The oracle's result is given to the subgoal using compose-tac because
      its premises are matched against the assumptions rather than used
      to make subgoals. Therefore , abstraction must copy the parameters
      precisely and make them available to all generated Vars.*)
    val params = Term.strip-all-vars t
    and body = Term.strip-all-body t
    val Us = map #2 params
    val nPar = length params
    val vname = 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 = if can HOLogic.dest-number t then t else replace t;
   (*abstraction of a real/rational expression*)
   fun rat ((c as Const(@{const-name HOL.plus}, -)) $ x $ y) = c $ (rat x) $ (rat
y)
     | rat ((c as Const(@{const-name HOL.minus}, -)) $ x $ y) = c $ (rat x) $
(rat y)
     | rat ((c as Const(@{const-name HOL.divide}, -)) $ x $ y) = c $ (rat x) $
(rat y)
     | rat ((c as Const(@{const-name HOL.times}, -)) $ x $ y) = c $ (rat x) $
(rat y)
     | rat ((c as Const(@{const-name HOL.uminus}, -)) $ x) = c $ (rat x)
     | rat t = lit t
   (*abstraction of an integer expression: no div, mod*)
   fun int ((c as Const(@{const-name HOL.plus}, -)) $ x $ y) = c $ (int x) $ (int
y)
     | int ((c as Const(@{const-name HOL.minus}, -)) $ x $ y) = c $ (int x) $
(int y)
     | int ((c as Const(@{const-name HOL.times}, -)) $ x $ y) = c $ (int x) $
(int y)
     | int ((c as Const(@{const-name HOL.uminus}, -)) $ x) = c $ (int x)
     | int t = lit t
   (*abstraction of a natural number expression: no minus*)
    fun nat ((c as Const(@{const-name HOL.plus}, -)) $ x $ y) = c $ (nat x) $
(nat y)
     | nat ((c as Const(@{const-name HOL.times}, -)) $ x $ y) = c $ (nat x) $
(nat y)
     | nat ((c as Const(@{const-name Suc}, -)) $ x) = c $ (nat x)
     | nat t = lit t
   (*abstraction of a relation: =, <, <=*)
   fun rel (T, c $ x $ y) =
         if T = HOLogic.realT then c $ (rat x) $ (rat y)
         else if T = HOLogic.intT then c $ (int x) $ (int y)
         else if T = HOLogic.natT then c $ (nat x) $ (nat y)
         else if T = HOLogic.boolT then c $ (fm x) $ (fm y)
         else replace (c $ x $ y)   (*non-numeric comparison*)
   (*abstraction of a formula*)
   and fm ((c as Const(op &, -)) $ p $ q) = c $ (fm p) $ (fm q)
     | fm ((c as Const(op |, -)) $ p $ q) = c $ (fm p) $ (fm q)
     | fm ((c as Const(op -->, -)) $ p $ q) = c $ (fm p) $ (fm q)
     | fm ((c as Const(Not, -)) $ p) = c $ (fm p)
     | fm ((c as Const(True, -))) = c
     | fm ((c as Const(False, -))) = c
```

```
      | fm (t as Const(op =,  Type (fun, [T,-])) $ - $ -) = rel (T, t)
      | fm (t as Const(@{const-name HOL.less},  Type (fun, [T,-])) $ - $ -) = rel
(T, t)
      | fm (t as Const(@{const-name HOL.less-eq}, Type (fun, [T,-])) $ - $ -) = rel
(T, t)
      | fm t = replace t
    (*entry point, and abstraction of a meta-formula*)
    fun mt ((c as Const(Trueprop, -)) $ p) = c $ (fm p)
      | mt ((c as Const(==>, -)) $ p $ q)  = c $ (mt p) $ (mt q)
      | mt t = fm t  (*it might be a formula*)
  in (list-all (params, mt body), !pairs) end;


(*Present the entire subgoal to the oracle, assumptions and all, but possibly
  abstracted.  Use via compose-tac, which performs no lifting but will
  instantiate variables.*)

val svc-tac = CSUBGOAL (fn (ct, i) =>
  let
    val thy = Thm.theory-of-cterm ct;
    val (abs-goal, -) = svc-abstract (Thm.term-of ct);
    val th = svc-oracle (Thm.cterm-of thy abs-goal);
   in compose-tac (false, th, 0) i end
   handle TERM - => no-tac);
⟫

end
```

References

[1] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[2] K. McMillan. Lecture notes on verification of digital and hybrid systems. NATO summer school, http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial/toc.html.

[3] K. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.