Miscellaneous HOL Examples

November 22, 2007

Contents

1 Foundations of HOL **6**
- 1.1 Pure Logic . . . 6
  - 1.1.1 Basic logical connectives . . . 6
  - 1.1.2 Extensional equality . . . 6
  - 1.1.3 Derived connectives . . . 7
- 1.2 Classical logic . . . 8

2 Abstract Natural Numbers primitive recursion **9**

3 Proof by guessing **11**

4 Simple and efficient binary numerals **11**
- 4.1 Binary representation of natural numbers . . . 11
- 4.2 Direct operations – plain normalization . . . 12
- 4.3 Indirect operations – ML will produce witnesses . . . 12
- 4.4 Concrete syntax . . . 13
- 4.5 Examples . . . 13

5 Examples of recdef definitions **15**

6 Examples of function definitions **18**
- 6.1 Very basic . . . 18
- 6.2 Currying . . . 18
- 6.3 Nested recursion . . . 18
- 6.4 More general patterns . . . 19
  - 6.4.1 Overlapping patterns . . . 19
  - 6.4.2 Guards . . . 19
- 6.5 Mutual Recursion . . . 20
- 6.6 Definitions in local contexts . . . 20
- 6.7 Regression tests . . . 21

7 Some of the results in Inductive Invariants for Nested Recursion **24**

8 Example use if an inductive invariant to solve termination conditions 25

9 Using locales in Isabelle/Isar – outdated version! 27
9.1 Overview 27
9.2 Local contexts as mathematical structures 29
9.3 Explicit structures referenced implicitly 31
9.4 Simple meta-theory of structures 33

10 Test of Locale Interpretation 34

11 Interpretation of Defined Concepts 34
11.1 Lattices 34
11.1.1 Definitions 34
11.1.2 Total order <= on *int* 38
11.1.3 Total order <= on *nat* 39
11.1.4 Lattice *dvd* on *nat* 40
11.2 Group example with defined operations *inv* and *unit* 40
11.2.1 Locale declarations and lemmas 40
11.2.2 Interpretation of Functions 43

12 Monoids and Groups as predicates over record schemes 43

13 Binary arithmetic examples 44
13.1 Regression Testing for Cancellation Simprocs 44
13.2 Arithmetic Method Tests 46
13.3 The Integers 47
13.4 The Natural Numbers 49

14 Examples for hexadecimal and binary numerals 52

15 Antiquotations 52

16 Multiple nested quotations and anti-quotations 53

17 Partial equivalence relations 54
17.1 Partial equivalence 54
17.2 Equivalence on function spaces 54
17.3 Total equivalence 55
17.4 Quotient types 55
17.5 Equality on quotients 56
17.6 Picking representing elements 56

18 Summing natural numbers 57

19 Three Divides Theorem **58**
19.1 Abstract 58
19.2 Formal proof 59
19.2.1 Miscellaneous summation lemmas 59
19.2.2 Generalised Three Divides 59
19.2.3 Three Divides Natural 60

20 Higher-Order Logic: Intuitionistic predicate calculus problems **61**

21 CTL formulae **67**
21.1 Basic fixed point properties 68
21.2 The tree induction principle 69
21.3 An application of tree induction 70

22 Arithmetic **70**
22.1 Splitting of Operators: *max*, *min*, *abs*, *op* $-$, *nat*, *op mod*, *op div* 71
22.2 Meta-Logic 73
22.3 Various Other Examples 73

23 Binary trees **75**

24 Sorting: Basic Theory **77**

25 Merge Sort **78**

26 A question from "Bundeswettbewerb Mathematik" **79**

27 A lemma for Lagrange's theorem **80**

28 Groebner Basis Examples **81**
28.1 Basic examples 81
28.2 Lemmas for Lagrange's theorem 82
28.3 Colinearity is invariant by rotation 82

29 Milner-Tofte: Co-induction in Relational Semantics **83**

30 Case study: Unification Algorithm **98**
30.1 Basic definitions 98
30.2 Basic lemmas 99
30.3 Specification: Most general unifiers 99
30.4 The unification algorithm 100
30.5 Partial correctness 100
30.6 Properties used in termination proof 101
30.7 Termination proof 102

31 Some examples demonstrating the comm-ring method **102**

32 Small examples for evaluation mechanisms **103**

33 A simple random engine **105**

34 Primitive Recursive Functions **107**

35 The Full Theorem of Tarski **111**
35.1 Partial Order 113
35.2 sublattice 116
35.3 lub 116
35.4 glb 117
35.5 fixed points 117
35.6 lemmas for Tarski, lub 117
35.7 Tarski fixpoint theorem 1, first part 118
35.8 interval 118
35.9 Top and Bottom 120
35.10fixed points form a partial order 120

36 Implementation of carry chain incrementor and adder **121**
36.1 Carry chain incrementor 122

37 Hilbert's choice and classical logic **123**
37.1 Proof text 123
37.2 Proof term of text 123
37.3 Proof script 124
37.4 Proof term of script 124

38 Classical Predicate Calculus Problems **125**
38.1 Traditional Classical Reasoner 125
38.1.1 Pelletier's examples 126
38.1.2 Classical Logic: examples with quantifiers 127
38.1.3 Problems requiring quantifier duplication 128
38.1.4 Hard examples with quantifiers 128
38.1.5 Problems (mainly) involving equality or functions . . 132
38.2 Model Elimination Prover 134
38.2.1 Pelletier's examples 134
38.2.2 Classical Logic: examples with quantifiers 136
38.2.3 Hard examples with quantifiers 136

39 Set Theory examples: Cantor's Theorem, Schröder-Bernstein Theorem, etc. **142**
39.1 Examples for the *blast* paper 142

39.2 Cantor's Theorem: There is no surjection from a set to its powerset . . . 143
39.3 The Schröder-Berstein Theorem . . . 143
39.4 A simple party theorem . . . 144

40 Meson test cases **145**
40.1 Interactive examples . . . 145

41 Examples for Ferrante and Rackoff's quantifier elimination procedure **219**

42 Some examples for Presburger Arithmetic **224**

43 Generic reflection and reification **248**

44 Implementation of finite sets by lists **248**
44.1 Definitional rewrites . . . 248
44.2 Operations on lists . . . 249
44.2.1 Basic definitions . . . 249
44.2.2 Derived definitions . . . 250
44.3 Isomorphism proofs . . . 251
44.4 code generator setup . . . 253
44.4.1 type serializations . . . 253
44.4.2 const serializations . . . 253
44.5 Horrible detour . . . 264

45 Installing an oracle for SVC (Stanford Validity Checker) **266**

46 Examples for the 'refute' command **266**
46.1 Examples and Test Cases . . . 267
46.1.1 Propositional logic . . . 267
46.1.2 Predicate logic . . . 267
46.1.3 Equality . . . 268
46.1.4 First-Order Logic . . . 268
46.1.5 Higher-Order Logic . . . 270
46.1.6 Meta-logic . . . 272
46.1.7 Schematic variables . . . 272
46.1.8 Abstractions . . . 273
46.1.9 Sets . . . 273
46.1.10 arbitrary . . . 274
46.1.11 The . . . 274
46.1.12 Eps . . . 275
46.1.13 Subtypes (typedef), typedecl . . . 275
46.1.14 Inductive datatypes . . . 275
46.1.15 Records . . . 290

46.1.16 Inductively defined sets 291
46.1.17 Examples involving special functions 291
46.1.18 Axiomatic type classes and overloading 292

47 Examples for the 'quickcheck' command 294
47.1 Lists 295
47.2 Trees 296

1 Foundations of HOL

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

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

1.1 Pure Logic

classes *type*
defaultsort *type*

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

1.1.1 Basic logical connectives

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

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

1.1.2 Extensional equality

axiomatization
  $equal :: 'a \Rightarrow 'a \Rightarrow o \quad (\textbf{infixl} = 50)$
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*⟩

lemma [*trans*]: $x = y \Longrightarrow P\ y \Longrightarrow P\ x$
⟨*proof*⟩

lemma [*trans*]: $P\ x \Longrightarrow x = y \Longrightarrow P\ y$
⟨*proof*⟩

theorem *trans* [*trans*]: $x = y \Longrightarrow y = z \Longrightarrow x = z$
⟨*proof*⟩

theorem *iff1* [*elim*]: $A = B \Longrightarrow A \Longrightarrow B$
⟨*proof*⟩

theorem *iff2* [*elim*]: $A = B \Longrightarrow B \Longrightarrow A$
⟨*proof*⟩

1.1.3 Derived connectives

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

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

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

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

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

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

abbreviation
  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*⟩

theorem *trueI* [*intro*]: $\top$
⟨*proof*⟩

theorem *notI* [*intro*]: $(A \Longrightarrow \bot) \Longrightarrow \neg A$
⟨*proof*⟩

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

lemma *notE'*: $A \Longrightarrow \neg A \Longrightarrow B$
  ⟨*proof*⟩

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

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

theorem *conjE* [*elim*]: $A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

theorem *disjI1* [*intro*]: $A \Longrightarrow A \vee B$
⟨*proof*⟩

theorem *disjI2* [*intro*]: $B \Longrightarrow A \vee B$
⟨*proof*⟩

theorem *disjE* [*elim*]: $A \vee B \Longrightarrow (A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

theorem *exI* [*intro*]: $P\ a \Longrightarrow \exists x.\ P\ x$
⟨*proof*⟩

theorem *exE* [*elim*]: $\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

1.2 Classical logic

locale *classical* =
  assumes *classical*: $(\neg A \Longrightarrow A) \Longrightarrow A$

theorem (**in** *classical*)

$Peirce's\text{-}Law$: $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$
⟨*proof*⟩

theorem (**in** *classical*)
double-negation: $\neg\ \neg\ A \Longrightarrow A$
⟨*proof*⟩

theorem (**in** *classical*)
tertium-non-datur: $A \vee \neg\ A$
⟨*proof*⟩

theorem (**in** *classical*)
classical-cases: $(A \Longrightarrow C) \Longrightarrow (\neg\ A \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

lemma (**in** *classical*) $(\neg\ A \Longrightarrow A) \Longrightarrow A$
⟨*proof*⟩

end

2 Abstract Natural Numbers primitive recursion

theory *Abstract-NAT*
imports *Main*
begin

Axiomatic Natural Numbers (Peano) – a monomorphic theory.

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

lemma *zero-neq-succ* [*simp*]: $zero \neq succ\ m$
⟨*proof*⟩

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*⟩

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 $rec\ e\ r\ x = y$
  ⟨*proof*⟩

lemma *rec-zero* [*simp*]: $rec\ e\ r\ zero = e$
⟨*proof*⟩

lemma *rec-succ* [*simp*]: $rec\ e\ r\ (succ\ m) = r\ m\ (rec\ e\ r\ m)$
⟨*proof*⟩

Example: addition (monomorphic)

definition
  $add :: {'n} \Rightarrow {'n} \Rightarrow {'n}$ **where**
  $add\ m\ n = rec\ n\ (\lambda\text{-}\ k.\ succ\ k)\ m$

lemma *add-zero* [*simp*]: $add\ zero\ n = n$
  and *add-succ* [*simp*]: $add\ (succ\ m)\ n = succ\ (add\ m\ n)$
  ⟨*proof*⟩

lemma *add-assoc*: $add\ (add\ k\ m)\ n = add\ k\ (add\ m\ n)$
  ⟨*proof*⟩

lemma *add-zero-right*: $add\ m\ zero = m$
  ⟨*proof*⟩

lemma *add-succ-right*: $add\ m\ (succ\ n) = succ\ (add\ m\ n)$
  ⟨*proof*⟩

lemma $add\ (succ\ (succ\ (succ\ zero)))\ (succ\ (succ\ zero)) =$
    $succ\ (succ\ (succ\ (succ\ (succ\ zero))))$
  ⟨*proof*⟩

Example: replication (polymorphic)

definition
  $repl :: {'n} \Rightarrow {'a} \Rightarrow {'a}\ list$ **where**
  $repl\ n\ x = rec\ []\ (\lambda\text{-}\ xs.\ x\ \#\ xs)\ n$

lemma *repl-zero* [*simp*]: *repl zero x* = []
  and *repl-succ* [*simp*]: *repl* (*succ n*) *x* = *x* # *repl n x*
  ⟨*proof*⟩

lemma *repl* (*succ* (*succ* (*succ zero*))) *True* = [*True*, *True*, *True*]
  ⟨*proof*⟩

end

Just see that our abstract specification makes sense . . .

interpretation *NAT* [*0 Suc*]
⟨*proof*⟩

end

3 Proof by guessing

theory *Guess*
imports *Main*
begin

lemma *True*
⟨*proof*⟩

end

4 Simple and efficient binary numerals

theory *Binary*
imports *Main*
begin

4.1 Binary representation of natural numbers

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

lemma *bit-simps*:
    bit n False = *2 * n*
    bit n True = *2 * n + 1*
  ⟨*proof*⟩

⟨*ML*⟩

4.2 Direct operations – plain normalization

lemma *binary-norm*:
$bit\ 0\ False = 0$
$bit\ 0\ True = 1$
⟨*proof*⟩

lemma *binary-add*:
$n + 0 = n$
$0 + n = n$
$1 + 1 = bit\ 1\ False$
$bit\ n\ False + 1 = bit\ n\ True$
$bit\ n\ True + 1 = bit\ (n + 1)\ False$
$1 + bit\ n\ False = bit\ n\ True$
$1 + bit\ n\ True = bit\ (n + 1)\ False$
$bit\ m\ False + bit\ n\ False = bit\ (m + n)\ False$
$bit\ m\ False + bit\ n\ True = bit\ (m + n)\ True$
$bit\ m\ True + bit\ n\ False = bit\ (m + n)\ True$
$bit\ m\ True + bit\ n\ True = bit\ ((m + n) + 1)\ False$
⟨*proof*⟩

lemma *binary-mult*:
$n * 0 = 0$
$0 * n = 0$
$n * 1 = n$
$1 * n = n$
$bit\ m\ True * n = bit\ (m * n)\ False + n$
$bit\ m\ False * n = bit\ (m * n)\ False$
$n * bit\ m\ True = bit\ (m * n)\ False + n$
$n * bit\ m\ False = bit\ (m * n)\ False$
⟨*proof*⟩

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

4.3 Indirect operations – ML will produce witnesses

lemma *binary-less-eq*:
fixes $n :: nat$
shows $n \equiv m + k \Longrightarrow (m \leq n) \equiv True$
and $m \equiv n + k + 1 \Longrightarrow (m \leq n) \equiv False$
⟨*proof*⟩

lemma *binary-less*:
fixes $n :: nat$
shows $m \equiv n + k \Longrightarrow (m < n) \equiv False$
and $n \equiv m + k + 1 \Longrightarrow (m < n) \equiv True$
⟨*proof*⟩

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

shows $m \equiv n + k \Longrightarrow m - n \equiv k$
and $n \equiv m + k \Longrightarrow m - n \equiv 0$
⟨*proof*⟩

lemma *binary-divmod*:
fixes $n :: nat$
assumes $m \equiv n * k + l$ **and** $0 < n$ **and** $l < n$
shows $m\ div\ n \equiv k$
and $m\ mod\ n \equiv l$
⟨*proof*⟩

⟨*ML*⟩

4.4 Concrete syntax

syntax
-Binary :: *num-const* ⇒ *'a* ($-)

⟨*ML*⟩

4.5 Examples

lemma $6 = 6
⟨*proof*⟩

lemma *bit* (*bit* (*bit* 0 *False*) *False*) *True* = 1
⟨*proof*⟩

lemma *bit* (*bit* (*bit* 0 *False*) *False*) *True* = *bit* 0 *True*
⟨*proof*⟩

lemma $5 + $3 = $8
⟨*proof*⟩

lemma $5 * $3 = $15
⟨*proof*⟩

lemma $5 − $3 = $2
⟨*proof*⟩

lemma $3 − $5 = 0
⟨*proof*⟩

lemma $123456789 − $123 = $123456666
⟨*proof*⟩

lemma $1111111111222222222233333333334444444444 − $998877665544332211
=
$1111111111222222222232334455668900112233
⟨*proof*⟩

lemma $(1111111111222222222233333333334444444444::nat) - 998877665544332211 =$
$1111111111222222222232334455668900112233$
⟨proof⟩

lemma $(1111111111222222222233333333334444444444::int) - 998877665544332211 =$
$1111111111222222222232334455668900112233$
⟨proof⟩

lemma $\$1111111111222222222233333333334444444444 * \$998877665544332211 =$
$\$1109864072938022197293802219729380221972383090160869185684$
⟨proof⟩

lemma $\$1111111111222222222233333333334444444444 * \$998877665544332211 -$
$\$5555555555666666666677777777778888888888 =$
$\$1109864072938022191738246664062713555294605312381980296796$
⟨proof⟩

lemma $\$42 < \$4 = False$
⟨proof⟩

lemma $\$4 < \$42 = True$
⟨proof⟩

lemma $\$42 <= \$4 = False$
⟨proof⟩

lemma $\$4 <= \$42 = True$
⟨proof⟩

lemma $\$1111111111222222222233333333334444444444 < \$998877665544332211 = False$
⟨proof⟩

lemma $\$998877665544332211 < \$1111111111222222222233333333334444444444 = True$
⟨proof⟩

lemma $\$1111111111222222222233333333334444444444 <= \$998877665544332211 = False$
⟨proof⟩

lemma $\$998877665544332211 <= \$1111111111222222222233333333334444444444 = True$
⟨proof⟩

lemma $*1234 div* $*23* = $*53*
⟨*proof*⟩

lemma $*1234 mod* $*23* = $*15*
⟨*proof*⟩

lemma $*11111111112222222222223333333333444444444 div* $*998877665544332211*
=
$*1112359550673033707875*
⟨*proof*⟩

lemma $*11111111112222222222223333333333444444444 mod* $*998877665544332211*
=
$*422451743175 82819*
⟨*proof*⟩

lemma (*11111111112222222222223333333333444444444::int*) *div 998877665544332211*
=
1112359550673033707875
⟨*proof*⟩

lemma (*11111111112222222222223333333333444444444::int*) *mod 998877665544332211*
=
42245174317582819
⟨*proof*⟩

end

5 Examples of recdef definitions

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

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

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

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

lemma *fib 7 = 13*
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *g-zero*: *g x* = *0*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

Removing the termination condition from the generated thms:

lemma *mapf* (*Suc n*) = *concat* (*map mapf* (*replicate n n*))
  ⟨*proof*⟩

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

end

6 Examples of function definitions

theory *Fundefs*
imports *Main*
begin

6.1 Very basic

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

partial simp and induction rules:

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

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

thm *fib.cases*

total simp and induction rules:

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

6.2 Currying

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

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

6.3 Nested recursion

function *nz*
where
  nz 0 = *0*
| *nz* (*Suc x*) = *nz* (*nz x*)
⟨*proof*⟩

lemma *nz-is-zero*: — A lemma we need to prove termination
  assumes *trm*: *nz-dom x*
  shows *nz x* = *0*
⟨*proof*⟩

termination *nz*

$\langle proof \rangle$

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*)))
$\langle proof \rangle$

lemma *f91-estimate*:
assumes *trm*: *f91-dom n*
shows *n* < *f91 n* + *11*
$\langle proof \rangle$

termination
$\langle proof \rangle$

6.4 More general patterns

6.4.1 Overlapping patterns

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

fun *gcd2* :: *nat* ⇒ *nat* ⇒ *nat*
where
gcd2 x 0 = *x*
| *gcd2 0 y* = *y*
| *gcd2* (*Suc x*) (*Suc y*) = (*if x* < *y then gcd2* (*Suc x*) (*y* − *x*)
else gcd2 (*x* − *y*) (*Suc y*))

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

6.4.2 Guards

We can reformulate the above example using guarded patterns

function *gcd3* :: *nat* ⇒ *nat* ⇒ *nat*
where
gcd3 x 0 = *x*
| *gcd3 0 y* = *y*
| *x* < *y* ⟹ *gcd3* (*Suc x*) (*Suc y*) = *gcd3* (*Suc x*) (*y* − *x*)
| ¬ *x* < *y* ⟹ *gcd3* (*Suc x*) (*Suc y*) = *gcd3* (*x* − *y*) (*Suc y*)
$\langle proof \rangle$
termination $\langle proof \rangle$

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*⟩
termination ⟨*proof*⟩

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

6.5 Mutual Recursion

fun *evn od* :: *nat* ⇒ *bool*
where
  evn 0 = *True*
| *od 0* = *False*
| *evn* (*Suc n*) = *od n*
| *od* (*Suc n*) = *evn n*

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

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

6.6 Definitions in local contexts

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

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

fun *foldL* :: *'a list* ⇒ *'a*
where
  foldL [] = *un*

| *foldL* [*x*] = *x*
| *foldL* (*x*#*y*#*ys*) = *foldL* (*opr x y* # *ys*)

thm *foldL.simps*

lemma *foldR-foldL*: *foldR xs* = *foldL xs*
⟨*proof*⟩

thm *foldR-foldL*

end

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

6.7 Regression tests

The following examples mainly serve as tests for the function package

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

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

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

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

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

function $f2 :: (nat \times nat) \Rightarrow (nat \times nat)$
where
$f2\ p = (let\ (x,y) = p\ in\ f2\ (y,x))$
$\langle proof \rangle$

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

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

lemma $lem{:}x \in set\ l \Longrightarrow size\ x < Suc\ (tree\text{-}list\text{-}size\ l)$
$\langle proof \rangle$

function $treemap :: ('a \Rightarrow 'a) \Rightarrow 'a\ tree \Rightarrow 'a\ tree$
where
$treemap\ fn\ (Leaf\ n) = (Leaf\ \ (fn\ n))$
$|\ treemap\ fn\ (Branch\ l) = (Branch\ (map\ (treemap\ fn)\ l))$
$\langle proof \rangle$
termination $\langle proof \rangle$

declare $lem[simp]$

fun $tinc :: nat\ tree \Rightarrow nat\ tree$

where
  tinc (*Leaf n*) = *Leaf* (*Suc n*)
| *tinc* (*Branch l*) = *Branch* (*map tinc l*)

record *point* =
  Xcoord :: *int*
  Ycoord :: *int*

function *swp* :: *point* ⇒ *point*
where
  swp ⦇ *Xcoord* = *x*, *Ycoord* = *y* ⦈ = ⦇ *Xcoord* = *y*, *Ycoord* = *x* ⦈
⟨*proof*⟩
termination ⟨*proof*⟩

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

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

fun *big* :: *DT* ⇒ *nat*
where
  big A = *0*
| *big B* = *0*
| *big C* = *0*
| *big D* = *0*
| *big E* = *0*
| *big F* = *0*
| *big G* = *0*
| *big H* = *0*
| *big I* = *0*
| *big J* = *0*
| *big K* = *0*
| *big L* = *0*
| *big M* = *0*
| *big N* = *0*
| *big P* = *0*

| *big Q = 0*
| *big R = 0*
| *big S = 0*
| *big T = 0*
| *big U = 0*
| *big V = 0*

fun
f4 :: *nat* ⇒ *nat* ⇒ *bool*
where
f4 0 0 = True
| *f4 - - = False*

end

7 Some of the results in Inductive Invariants for Nested Recursion

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

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

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

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

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

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

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

theorem *indinv-wfrec*:

assumes *wf*: *wf r* **and**
inv: *indinv r S F*
shows $S\ x\ (wfrec\ r\ F\ x)$
⟨*proof*⟩

theorem *indinv-on-wfrec*:
assumes *WF*: *wf r* **and**
INV: *indinv-on r D S F* **and**
D: $x \in D$
shows $S\ x\ (wfrec\ r\ F\ x)$
⟨*proof*⟩

theorem *ind-fixpoint-on-lemma*:
assumes *WF*: *wf r* **and**
INV: $\forall f.\ \forall x\in D.\ (\forall y\in D.\ (y,x) \in r \dashrightarrow S\ y\ (wfrec\ r\ F\ y)\ \&\ f\ y = wfrec\ r\ F\ y)$
$\dashrightarrow S\ x\ (wfrec\ r\ F\ x)\ \&\ F\ f\ x = wfrec\ r\ F\ x$ **and**
D: $x \in D$
shows $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x\ \&\ S\ x\ (wfrec\ r\ F\ x)$
⟨*proof*⟩

theorem *ind-fixpoint-lemma*:
assumes *WF*: *wf r* **and**
INV: $\forall f\ x.\ (\forall y.\ (y,x) \in r \dashrightarrow S\ y\ (wfrec\ r\ F\ y)\ \&\ f\ y = wfrec\ r\ F\ y)$
$\dashrightarrow S\ x\ (wfrec\ r\ F\ x)\ \&\ F\ f\ x = wfrec\ r\ F\ x$
shows $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x\ \&\ S\ x\ (wfrec\ r\ F\ x)$
⟨*proof*⟩

theorem *tfl-indinv-wfrec*:
$[|\ f == wfrec\ r\ F;\ wf\ r;\ indinv\ r\ S\ F\ |]$
$==> S\ x\ (f\ x)$
⟨*proof*⟩

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

end

8 Example use if an inductive invariant to solve termination conditions

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

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

consts g :: nat => nat

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

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

recdef-tc *g-tc*[*simp*]: g
⟨*proof*⟩

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*]
⟨*proof*⟩

recdef-tc *g'-tc*[*simp*]: g'
⟨*proof*⟩

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*: $\exists k.\ n = Suc\ (Suc\ (2{*}k))$ ==> $g\text{-}even\ n = 3$
⟨*proof*⟩

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

theorem *g-even-n*: $\exists k.\ n = 2{*}k + 4$ ==> $g\text{-}even\ n = g\text{-}even\ (g\text{-}even\ (n - 2) - 1)$
⟨*proof*⟩

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*
⟨*proof*⟩

Proving the termination condition using the strengthened inductive invariant.

recdef-tc *ninety-one-tc*[*rule-format*]: *ninety-one*
⟨*proof*⟩

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*)))
⟨*proof*⟩

end

9 Using locales in Isabelle/Isar – outdated version!

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

9.1 Overview

Locales provide a mechanism for encapsulating local contexts. The original version due to Florian Kammüller [2] refers directly to Isabelle's meta-logic

[7], which is minimal higher-order logic with connectives $\bigwedge$ (universal quantification), $\Longrightarrow$ (implication), and $\equiv$ (equality).

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

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

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

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

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

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

- Separate language elements for *instantiation* of locales.

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

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

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

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

9.2 Local contexts as mathematical structures

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

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

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

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

theorem (**in** *group-context*)
  right-inv: $x \cdot x^{-1} = \mathbf{1}$
⟨*proof*⟩

theorem (**in** *group-context*)
  right-one: $x \cdot \mathbf{1} = x$
⟨*proof*⟩

Facts like *right-one* are available *group-context* as stated above. The exported version looses the additional infrastructure of Isar proof contexts (syntax etc.) retaining only the pure logical content: *group-context.right-one* becomes *group-context ?prod ?inv ?one* $\Longrightarrow$ *?prod ?x ?one* = *?x* (in Isabelle outermost $\bigwedge$ quantification is replaced by schematic variables).

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

theorem (**in** *group-context*)
  assumes *eq*: $e \cdot x = x$
  shows *one-equality*: $\mathbf{1} = e$
⟨*proof*⟩

theorem (**in** *group-context*)
  assumes *eq*: $x' \cdot x = \mathbf{1}$
  shows *inv-equality*: $x^{-1} = x'$
⟨*proof*⟩

theorem (**in** *group-context*)
  inv-prod: $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$
⟨*proof*⟩

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

theorem (**in** *abelian-group-context*)
  inv-prod': $(x \cdot y)^{-1} = x^{-1} \cdot y^{-1}$
⟨*proof*⟩

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

Some more properties of inversion in general group theory follow.

theorem (**in** *group-context*)
  inv-inv: $(x^{-1})^{-1} = x$
⟨*proof*⟩

theorem (**in** *group-context*)
  assumes *eq*: $x^{-1} = y^{-1}$
  shows *inv-inject*: $x = y$
⟨*proof*⟩

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

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

The corresponding introduction rule is as follows:

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

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

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

9.3 Explicit structures referenced implicitly

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

record $'a$ *semigroup* =
  prod :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\cdot\imath$ *70*)

record $'a$ *group* = $'a$ *semigroup* +
  inv :: $'a \Rightarrow 'a$ $((\text{-}^{-1}\imath)\ [\mathit{1000}]\ \mathit{999})$
  one :: $'a$ $(\mathbf{1}\imath)$

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

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

locale *semigroup* =
  fixes S (**structure**)

assumes *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

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

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

Note that we prefer to call the *group* record structure G rather than S inherited from *semigroup*. This does not affect our concrete syntax, which is only dependent on the *positional* arrangements of currently active structures (actually only one above), rather than names. In fact, these parameter names rarely occur in the term language at all (due to the "indexed syntax" facility of Isabelle). On the other hand, names of locale facts will get qualified accordingly, e.g. *S.assoc* versus *G.assoc*.

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

theorem (**in** *group*)
right-inv: $x \cdot x^{-1} = \mathbf{1}$
⟨*proof*⟩

theorem (**in** *group*)
right-one: $x \cdot \mathbf{1} = x$
⟨*proof*⟩

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

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

lemma
includes *group G*
includes *group H*
shows $x \cdot y \cdot \mathbf{1} = prod\ G\ (prod\ G\ x\ y)\ (one\ G)$
and $x \cdot_2 y \cdot_2 \mathbf{1}_2 = prod\ H\ (prod\ H\ x\ y)\ (one\ H)$
and $x \cdot \mathbf{1}_2 = prod\ G\ x\ (one\ H)$
⟨*proof*⟩

Note that the trivial statements above need to be given as a simultaneous goal in order to have type-inference make the implicit typing of structures G and H agree.

9.4 Simple meta-theory of structures

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

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

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

lemma *semigroup-product* [*intro*]:
  assumes S: *semigroup* S
    and T: *semigroup* T
  shows *semigroup* (*semigroup-product* S T)
⟨*proof*⟩

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

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

lemma
  includes *semigroup* S + *semigroup* T
  fixes U (**structure**)
  defines U $\equiv$ *semigroup-product* S T
  shows *semigroup* U
⟨*proof*⟩

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

constdefs
  group-product :: $'a$ *group* $\Rightarrow$ $'b$ *group* $\Rightarrow$ $('a \times 'b)$ *group*
  group-product G H $\equiv$
    semigroup.extend
      (*semigroup-product* (*semigroup.truncate* G) (*semigroup.truncate* H))
      (*group.fields* ($\lambda p.$ (*inv* G (*fst* p), *inv* H (*snd* p))) (*one* G, *one* H))

lemma *group-product-aux*:
  includes *group* G + *group* H
  fixes I (**structure**)
  defines I $\equiv$ *group-product* G H

shows *group I*
⟨*proof*⟩

theorem *group-product*: *group G* $\Longrightarrow$ *group H* $\Longrightarrow$ *group* (*group-product G H*)
⟨*proof*⟩

end

10 Test of Locale Interpretation

theory *LocaleTest2*
imports *GCD*
begin

11 Interpretation of Defined Concepts

Naming convention for global objects: prefixes D and d

11.1 Lattices

Much of the lattice proofs are from HOL/Lattice.

11.1.1 Definitions

locale *dpo* =
fixes *le* :: ['*a*, '*a*] => *bool* (**infixl** $\sqsubseteq$ *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*:
[| $x \sqsubseteq y$; $y \sqsubseteq z$; $z \sqsubseteq x$ |] ==> $x = y$ & $y = z$
⟨*proof*⟩

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

theorem *abs-test*:
op $\sqsubset$ = (%*x y*. $x \sqsubset y$)
⟨*proof*⟩

definition
is-inf :: ['*a*, '*a*, '*a*] => *bool*

where $\mathit{is\text{-}inf}\ x\ y\ i = (i \sqsubseteq x \wedge i \sqsubseteq y \wedge (\forall z.\ z \sqsubseteq x \wedge z \sqsubseteq y \longrightarrow z \sqsubseteq i))$

definition
$\mathit{is\text{-}sup} :: [\prime a,\ \prime a,\ \prime a] => \mathit{bool}$
where $\mathit{is\text{-}sup}\ x\ y\ s = (x \sqsubseteq s \wedge y \sqsubseteq s \wedge (\forall z.\ x \sqsubseteq z \wedge y \sqsubseteq z \longrightarrow s \sqsubseteq z))$

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
$\mathit{meet} :: [\prime a,\ \prime a] => \prime a$ (**infixl** $\sqcap$ *70*)
where $x \sqcap y = (\mathit{THE\ inf.\ is\text{-}inf}\ x\ y\ \mathit{inf})$

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

lemma *is-infI* [*intro?*]: $i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow$
$(\bigwedge z.\ z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i) \Longrightarrow \mathit{is\text{-}inf}\ x\ y\ i$
⟨*proof*⟩

lemma *is-inf-lower* [*elim?*]:
$\mathit{is\text{-}inf}\ x\ y\ i \Longrightarrow (i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

lemma *is-inf-greatest* [*elim?*]:
$\mathit{is\text{-}inf}\ x\ y\ i \Longrightarrow z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i$
⟨*proof*⟩

theorem *is-inf-uniq*: $\mathit{is\text{-}inf}\ x\ y\ i \Longrightarrow \mathit{is\text{-}inf}\ x\ y\ i' \Longrightarrow i = i'$
⟨*proof*⟩

theorem *is-inf-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow \mathit{is\text{-}inf}\ x\ y\ x$
⟨*proof*⟩

lemma *meet-equality* [*elim?*]: $\mathit{is\text{-}inf}\ x\ y\ i \Longrightarrow x \sqcap y = i$
⟨*proof*⟩

lemma *meetI* [*intro?*]:
$i \sqsubseteq x \Longrightarrow i \sqsubseteq y \Longrightarrow (\bigwedge z.\ z \sqsubseteq x \Longrightarrow z \sqsubseteq y \Longrightarrow z \sqsubseteq i) \Longrightarrow x \sqcap y = i$
⟨*proof*⟩

lemma *is-inf-meet* [*intro?*]: $\mathit{is\text{-}inf}\ x\ y\ (x \sqcap y)$
⟨*proof*⟩

lemma *meet-left* [*intro?*]:
$x \sqcap y \sqsubseteq x$
⟨*proof*⟩

lemma *meet-right* [*intro?*]:
$x \sqcap y \sqsubseteq y$
⟨*proof*⟩

lemma *meet-le* [*intro?*]:
$[| \ z \sqsubseteq x; \ z \sqsubseteq y \ |] ==> z \sqsubseteq x \sqcap y$
⟨*proof*⟩

lemma *is-supI* [*intro?*]: $x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow$
$(\bigwedge z. \ x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z) \Longrightarrow \textit{is-sup} \ x \ y \ s$
⟨*proof*⟩

lemma *is-sup-least* [*elim?*]:
$\textit{is-sup} \ x \ y \ s \Longrightarrow x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z$
⟨*proof*⟩

lemma *is-sup-upper* [*elim?*]:
$\textit{is-sup} \ x \ y \ s \Longrightarrow (x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow C) \Longrightarrow C$
⟨*proof*⟩

theorem *is-sup-uniq*: $\textit{is-sup} \ x \ y \ s \Longrightarrow \textit{is-sup} \ x \ y \ s' \Longrightarrow s = s'$
⟨*proof*⟩

theorem *is-sup-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow \textit{is-sup} \ x \ y \ y$
⟨*proof*⟩

lemma *join-equality* [*elim?*]: $\textit{is-sup} \ x \ y \ s \Longrightarrow x \sqcup y = s$
⟨*proof*⟩

lemma *joinI* [*intro?*]: $x \sqsubseteq s \Longrightarrow y \sqsubseteq s \Longrightarrow$
$(\bigwedge z. \ x \sqsubseteq z \Longrightarrow y \sqsubseteq z \Longrightarrow s \sqsubseteq z) \Longrightarrow x \sqcup y = s$
⟨*proof*⟩

lemma *is-sup-join* [*intro?*]: $\textit{is-sup} \ x \ y \ (x \sqcup y)$
⟨*proof*⟩

lemma *join-left* [*intro?*]:
$x \sqsubseteq x \sqcup y$
⟨*proof*⟩

lemma *join-right* [*intro?*]:
$y \sqsubseteq x \sqcup y$
⟨*proof*⟩

lemma *join-le* [*intro?*]:
[| $x \sqsubseteq z$; $y \sqsubseteq z$ |] ==> $x \sqcup y \sqsubseteq z$
⟨*proof*⟩

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

theorem *meet-commute*: $x \sqcap y = y \sqcap x$
⟨*proof*⟩

theorem *meet-join-absorb*: $x \sqcap (x \sqcup y) = x$
⟨*proof*⟩

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

theorem *join-commute*: $x \sqcup y = y \sqcup x$
⟨*proof*⟩

theorem *join-meet-absorb*: $x \sqcup (x \sqcap y) = x$
⟨*proof*⟩

theorem *meet-idem*: $x \sqcap x = x$
⟨*proof*⟩

theorem *meet-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
⟨*proof*⟩

theorem *meet-related2* [*elim?*]: $y \sqsubseteq x \Longrightarrow x \sqcap y = y$
⟨*proof*⟩

theorem *join-related* [*elim?*]: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
⟨*proof*⟩

theorem *join-related2* [*elim?*]: $y \sqsubseteq x \Longrightarrow x \sqcup y = x$
⟨*proof*⟩

Additional theorems

theorem *meet-connection*: $(x \sqsubseteq y) = (x \sqcap y = x)$
⟨*proof*⟩

theorem *meet-connection2*: $(x \sqsubseteq y) = (y \sqcap x = x)$
⟨*proof*⟩

theorem *join-connection*: $(x \sqsubseteq y) = (x \sqcup y = y)$
⟨*proof*⟩

theorem *join-connection2*: $(x \sqsubseteq y) = (x \sqcup y = y)$
⟨*proof*⟩

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*:
  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$⟨*proof*⟩

end

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

begin

lemma *less-total*: $x \sqsubset y \mid x = y \mid y \sqsubset x$
  ⟨*proof*⟩

end

interpretation *dlo* < *dlat*

⟨*proof*⟩

interpretation *dlo* < *ddlat*
⟨*proof*⟩

11.1.2 Total order <= on *int*

interpretation *int*: *dpo* [*op* <= :: [*int*, *int*] => *bool*]
  where (*dpo.less* (*op* <=) (*x*::*int*) *y*) = (*x* < *y*)⟨*proof*⟩

thm *int.circular*
lemma $[\![(x{::}int) \le y;\ y \le z;\ z \le x]\!] \Longrightarrow x = y \land y = z$
  ⟨*proof*⟩
thm *int.abs-test*
lemma (*op* < :: [*int*, *int*] => *bool*) = *op* <
  ⟨*proof*⟩

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*⟩

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

Interpreted theorems from the locales, involving defined terms.

thm *int.less-def*

from dpo

thm *int.meet-left*

from dlat

thm *int.meet-distr*

from ddlat

thm *int.less-total*

from dlo

11.1.3 Total order <= on *nat*

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

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*⟩

interpretation *nat*: *dlo* [*op* <= :: [*nat*, *nat*] => *bool*]
  ⟨*proof*⟩

Interpreted theorems from the locales, involving defined terms.

thm *nat.less-def*

from dpo

thm *nat.meet-left*

from dlat

thm *nat.meet-distr*

from ddlat

thm *nat.less-total*

from ldo

11.1.4 Lattice *dvd* on *nat*

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

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*⟩

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*)
  ⟨*proof*⟩
thm *nat-dvd.meet-left*

from dlat

lemma *gcd* (*x*, *y*) *dvd x*
  ⟨*proof*⟩

print-interps *dpo*
print-interps *dlat*

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

11.2.1 Locale declarations and lemmas

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

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

begin

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

lemma *inv-unique*:
  assumes *eq*: *y* ** *x* = *one x* ** *y'* = *one*
  shows *y* = *y'*
⟨*proof*⟩

lemma *unit-one* [*intro*, *simp*]:
unit one
⟨*proof*⟩

lemma *unit-l-inv-ex*:
unit x ==> ∃ *y*. *y* ** *x* = *one*
⟨*proof*⟩

lemma *unit-r-inv-ex*:
unit x ==> ∃ *y*. *x* ** *y* = *one*
⟨*proof*⟩

lemma *unit-l-inv*:
unit x ==> *inv x* ** *x* = *one*
⟨*proof*⟩

lemma *unit-r-inv*:
unit x ==> *x* ** *inv x* = *one*
⟨*proof*⟩

lemma *unit-inv-unit* [*intro*, *simp*]:
unit x ==> *unit* (*inv x*)
⟨*proof*⟩

lemma *unit-l-cancel* [*simp*]:
unit x ==> (*x* ** *y* = *x* ** *z*) = (*y* = *z*)
⟨*proof*⟩

lemma *unit-inv-inv* [*simp*]:
unit x ==> *inv* (*inv x*) = *x*
⟨*proof*⟩

lemma *inv-inj-on-unit*:
inj-on inv {*x*. *unit x*}
⟨*proof*⟩

lemma *unit-inv-comm*:
assumes *inv*: *x* ** *y* = *one*
and *G*: *unit x* *unit y*
shows *y* ** *x* = *one*
⟨*proof*⟩

end

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

begin

lemma *l-inv-ex* [*simp*]:
$\exists\, y.\ y ** x = one$
⟨*proof*⟩

lemma *r-inv-ex* [*simp*]:
$\exists\, y.\ x ** y = one$
⟨*proof*⟩

lemma *l-inv* [*simp*]:
inv $x ** x = one$
⟨*proof*⟩

lemma *l-cancel* [*simp*]:
$(x ** y = x ** z) = (y = z)$
⟨*proof*⟩

lemma *r-inv* [*simp*]:
$x **$ *inv* $x = one$
⟨*proof*⟩

lemma *r-cancel* [*simp*]:
$(y ** x = z ** x) = (y = z)$
⟨*proof*⟩

lemma *inv-one* [*simp*]:
inv one = *one*
⟨*proof*⟩

lemma *inv-inv* [*simp*]:
inv (*inv* x) = x
⟨*proof*⟩

lemma *inv-inj*:
inj-on inv UNIV
⟨*proof*⟩

lemma *inv-mult-group*:
inv $(x ** y)$ = *inv* $y **$ *inv* x
⟨*proof*⟩

lemma *inv-comm*:
$x ** y = one ==> y ** x = one$
⟨*proof*⟩

lemma *inv-equality*:
$y ** x = one ==>$ *inv* $x = y$
⟨*proof*⟩

end

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

begin

lemma *hom-one* [*simp*]:
  hom one = *zero*
⟨*proof*⟩

end

11.2.2 Interpretation of Functions

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

⟨*proof*⟩

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*
  ⟨*proof*⟩

interpretation *Dfun*: *Dgrp* [*op o id* :: *unit* => *unit*]
  where *Dmonoid.inv* (*op o*) *id f* = *inv* (*f* :: *unit* => *unit*)
⟨*proof*⟩

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

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

end

12 Monoids and Groups as predicates over record schemes

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

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

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

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

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

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

end

13 Binary arithmetic examples

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

13.1 Regression Testing for Cancellation Simprocs

lemma *l* + *2* + *2* + *2* + (*l* + *2*) + (*oo* + *2*) = (*uu*::*int*)
⟨*proof*⟩

lemma *2*∗*u* = (*u*::*int*)
⟨*proof*⟩

lemma (*i* + *j* + *12* + (*k*::*int*)) − *15* = *y*
⟨*proof*⟩

lemma (*i* + *j* + *12* + (*k*::*int*)) − *5* = *y*
⟨*proof*⟩

lemma *y* − *b* < (*b*::*int*)
⟨*proof*⟩

lemma *y* − (*3*∗*b* + *c*) < (*b*::*int*) − *2*∗*c*
⟨*proof*⟩

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

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

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

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

lemma $(i + j + 12 + (k::int)) = u + 15 + y$
⟨*proof*⟩

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

lemma $2*y + 3*z + 6*w + 2*y + 3*z + 2*u = 2*y' + 3*z' + 6*w' + 2*y' + 3*z' + u + (vv::int)$
⟨*proof*⟩

lemma $a + -(b+c) + b = (d::int)$
⟨*proof*⟩

lemma $a + -(b+c) - b = (d::int)$
⟨*proof*⟩

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

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

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

lemma $(i + j + -12 + (k::int)) - 15 = y$
⟨*proof*⟩

lemma $(i + j + 12 + (k::int)) - -15 = y$
⟨*proof*⟩

lemma $(i + j + -12 + (k::int)) - -15 = y$
⟨*proof*⟩

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

⟨*proof*⟩

13.2 Arithmetic Method Tests

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

lemma !!a::int. [| a < b; c < d |] ==> a−d+ 2 <= b+(−c)
⟨*proof*⟩

lemma !!a::int. [| a < b; c < d |] ==> a+c+ 1 < b+d
⟨*proof*⟩

lemma !!a::int. [| a <= b; b+b <= c |] ==> a+a <= c
⟨*proof*⟩

lemma !!a::int. [| a+b <= i+j; a<=b; i<=j |] ==> a+a <= j+j
⟨*proof*⟩

lemma !!a::int. [| a+b < i+j; a<b; i<j |] ==> a+a − − −1 < j+j − 3
⟨*proof*⟩

lemma !!a::int. a+b+c <= i+j+k & a<=b & b<=c & i<=j & j<=k −−> a+a+a <= k+k+k
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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

⟨*proof*⟩

13.3 The Integers

Addition

lemma $(13::int) + 19 = 32$
⟨*proof*⟩

lemma $(1234::int) + 5678 = 6912$
⟨*proof*⟩

lemma $(1359::int) + -2468 = -1109$
⟨*proof*⟩

lemma $(93746::int) + -46375 = 47371$
⟨*proof*⟩

Negation

lemma $-(65745::int) = -65745$
⟨*proof*⟩

lemma $-(-54321::int) = 54321$
⟨*proof*⟩

Multiplication

lemma $(13::int) * 19 = 247$
⟨*proof*⟩

lemma $(-84::int) * 51 = -4284$
⟨*proof*⟩

lemma $(255::int) * 255 = 65025$
⟨*proof*⟩

lemma $(1359::int) * -2468 = -3354012$
⟨*proof*⟩

lemma $(89::int) * 10 \neq 889$
⟨*proof*⟩

lemma $(13::int) < 18 - 4$
⟨*proof*⟩

lemma $(-345::int) < -242 + -100$
⟨*proof*⟩

lemma $(13557456::int) < 18678654$

⟨*proof*⟩

lemma $(999999::int) \le (1000001 + 1) - 2$
⟨*proof*⟩

lemma $(1234567::int) \le 1234567$
⟨*proof*⟩

No integer overflow!

lemma $1234567 * (1234567::int) < 1234567*1234567*1234567$
⟨*proof*⟩

Quotient and Remainder

lemma $(10::int)$ *div* $3 = 3$
⟨*proof*⟩

lemma $(10::int)$ *mod* $3 = 1$
⟨*proof*⟩

A negative divisor

lemma $(10::int)$ *div* $-3 = -4$
⟨*proof*⟩

lemma $(10::int)$ *mod* $-3 = -2$
⟨*proof*⟩

A negative dividend[1]

lemma $(-10::int)$ *div* $3 = -4$
⟨*proof*⟩

lemma $(-10::int)$ *mod* $3 = 2$
⟨*proof*⟩

A negative dividend *and* divisor

lemma $(-10::int)$ *div* $-3 = 3$
⟨*proof*⟩

lemma $(-10::int)$ *mod* $-3 = -1$
⟨*proof*⟩

A few bigger examples

lemma $(8452::int)$ *mod* $3 = 1$
⟨*proof*⟩

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

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

$\langle proof \rangle$

lemma $(1000006{::}int)\ mod\ 10 = 6$
$\langle proof \rangle$

Division by shifting

lemma $10000000\ div\ 2 = (5000000{::}int)$
$\langle proof \rangle$

lemma $10000001\ mod\ 2 = (1{::}int)$
$\langle proof \rangle$

lemma $10000055\ div\ 32 = (312501{::}int)$
$\langle proof \rangle$

lemma $10000055\ mod\ 32 = (23{::}int)$
$\langle proof \rangle$

lemma $100094\ div\ 144 = (695{::}int)$
$\langle proof \rangle$

lemma $100094\ mod\ 144 = (14{::}int)$
$\langle proof \rangle$

Powers

lemma $2\ \hat{}\ 10 = (1024{::}int)$
$\langle proof \rangle$

lemma $-3\ \hat{}\ 7 = (-2187{::}int)$
$\langle proof \rangle$

lemma $13\ \hat{}\ 7 = (62748517{::}int)$
$\langle proof \rangle$

lemma $3\ \hat{}\ 15 = (14348907{::}int)$
$\langle proof \rangle$

lemma $-5\ \hat{}\ 11 = (-48828125{::}int)$
$\langle proof \rangle$

13.4 The Natural Numbers

Successor

lemma $Suc\ 99999 = 100000$
$\langle proof \rangle$

Addition

lemma $(13::nat) + 19 = 32$
⟨*proof*⟩

lemma $(1234::nat) + 5678 = 6912$
⟨*proof*⟩

lemma $(973646::nat) + 6475 = 980121$
⟨*proof*⟩

Subtraction

lemma $(32::nat) - 14 = 18$
⟨*proof*⟩

lemma $(14::nat) - 15 = 0$
⟨*proof*⟩

lemma $(14::nat) - 1576644 = 0$
⟨*proof*⟩

lemma $(48273776::nat) - 3873737 = 44400039$
⟨*proof*⟩

Multiplication

lemma $(12::nat) * 11 = 132$
⟨*proof*⟩

lemma $(647::nat) * 3643 = 2357021$
⟨*proof*⟩

Quotient and Remainder

lemma $(10::nat)$ *div* $3 = 3$
⟨*proof*⟩

lemma $(10::nat)$ *mod* $3 = 1$
⟨*proof*⟩

lemma $(10000::nat)$ *div* $9 = 1111$
⟨*proof*⟩

lemma $(10000::nat)$ *mod* $9 = 1$
⟨*proof*⟩

lemma $(10000::nat)$ *div* $16 = 625$
⟨*proof*⟩

lemma $(10000::nat)$ *mod* $16 = 0$
⟨*proof*⟩

Powers

lemma $2 \hat{} 12 = (4096::nat)$
⟨*proof*⟩

lemma $3 \hat{} 10 = (59049::nat)$
⟨*proof*⟩

lemma $12 \hat{} 7 = (35831808::nat)$
⟨*proof*⟩

lemma $3 \hat{} 14 = (4782969::nat)$
⟨*proof*⟩

lemma $5 \hat{} 11 = (48828125::nat)$
⟨*proof*⟩

Testing the cancellation of complementary terms

lemma $y + (x + -x) = (0::int) + y$
⟨*proof*⟩

lemma $y + (-x + (- y + x)) = (0::int)$
⟨*proof*⟩

lemma $-x + (y + (- y + x)) = (0::int)$
⟨*proof*⟩

lemma $x + (x + (- x + (- x + (- y + - z)))) = (0::int) - y - z$
⟨*proof*⟩

lemma $x + x - x - x - y - z = (0::int) - y - z$
⟨*proof*⟩

lemma $x + y + z - (x + z) = y - (0::int)$
⟨*proof*⟩

lemma $x + (y + (y + (y + (-x + -x)))) = (0::int) + y - x + y + y$
⟨*proof*⟩

lemma $x + (y + (y + (y + (-y + -x)))) = y + (0::int) + y$
⟨*proof*⟩

lemma $x + y - x + z - x - y - z + x < (1::int)$
⟨*proof*⟩

end

14 Examples for hexadecimal and binary numerals

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

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

lemma $0xFF = 255$ ⟨*proof*⟩
lemma $0xF = 0b1111$ ⟨*proof*⟩

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

lemma $0x0A + 0x10 = (0x1A :: nat)$ ⟨*proof*⟩

The number of leading zeros is irrelevant

lemma $0b00010000 = 0x10$ ⟨*proof*⟩

Unary minus works as for decimal numerals

lemma $- 0x0A = - 10$ ⟨*proof*⟩

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 $0x01 = 1$ ⟨*proof*⟩
lemma $0x00 = 0$ ⟨*proof*⟩

lemma $0x01 = (1::nat)$ ⟨*proof*⟩
lemma $0b0000 = (0::int)$ ⟨*proof*⟩

end

15 Antiquotations

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

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

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

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

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

⟨*ML*⟩

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

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

end

16 Multiple nested quotations and anti-quotations

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

Multiple nested quotations and anti-quotations – basically a generalized version of de-Bruijn representation.

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

⟨*ML*⟩

basic examples

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

advanced examples

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

end

17 Partial equivalence relations

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

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

Oscar Slotosch: *Higher Order Quotients and their Implementation in Isabelle HOL.* Elsa L. Gunter and Amy Felty, editors, Theorem Proving in Higher Order Logics: TPHOLs '97, Springer LNCS 1275, 1997.

17.1 Partial equivalence

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

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

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

The domain of a partial equivalence relation is the set of reflexive elements. Due to symmetry and transitivity this characterizes exactly those elements that are connected with *any* other one.

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

lemma *domainI* [*intro*]: $x \sim x$ ==> $x \in$ *domain*
  ⟨*proof*⟩

lemma *domainD* [*dest*]: $x \in$ *domain* ==> $x \sim x$
  ⟨*proof*⟩

theorem *domainI'* [*elim?*]: $x \sim y$ ==> $x \in$ *domain*
⟨*proof*⟩

17.2 Equivalence on function spaces

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

defs (overloaded)

eqv-fun-def: $f \sim g == \forall x \in domain.\ \forall y \in domain.\ x \sim y \dashrightarrow f\,x \sim g\,y$

lemma *partial-equiv-funI* [*intro?*]:
$(!!x\ y.\ x \in domain ==> y \in domain ==> x \sim y ==> f\,x \sim g\,y) ==> f \sim g$
⟨*proof*⟩

lemma *partial-equiv-funD* [*dest?*]:
$f \sim g ==> x \in domain ==> y \in domain ==> x \sim y ==> f\,x \sim g\,y$
⟨*proof*⟩

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

instance *fun* :: (*partial-equiv*, *partial-equiv*) *partial-equiv*
⟨*proof*⟩

17.3 Total equivalence

The class of total equivalence relations on top of PERs. It coincides with the standard notion of equivalence, i.e. $\sim$ :: *'a => 'a => 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*]: $(x::'a::equiv) \in domain$
⟨*proof*⟩

theorem *equiv-cong* [*dest?*]: $f \sim g ==> x \sim y ==> f\,x \sim g\ (y::'a::equiv)$
⟨*proof*⟩

17.4 Quotient types

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

typedef *'a quot* = $\{\{x.\ a \sim x\}|\ a::'a.\ True\}$
⟨*proof*⟩

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
⟨*proof*⟩

lemma *quotE* [*elim*]: $R \in quot ==> (!!a.\ R = \{x.\ a \sim x\} ==> C) ==> C$
⟨*proof*⟩

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

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

theorem *quot-rep*: $\exists\, a.\ A = \lfloor a \rfloor$
⟨*proof*⟩

lemma *quot-cases* [*cases type*: *quot*]:
obtains (*rep*) *a* **where** $A = \lfloor a \rfloor$
⟨*proof*⟩

17.5 Equality on quotients

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

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

theorem *eqv-class-eqD'* [*dest?*]: $\lfloor a \rfloor = \lfloor b \rfloor ==> a \in domain ==> a \sim b$
⟨*proof*⟩

theorem *eqv-class-eqD* [*dest?*]: $\lfloor a \rfloor = \lfloor b \rfloor ==> a \sim (b::'a::equiv)$
⟨*proof*⟩

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

lemma *eqv-class-eq* [*simp*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim (b::'a::equiv))$
⟨*proof*⟩

17.6 Picking representing elements

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

theorem *pick-eqv'* [*intro?*, *simp*]: $a \in domain ==> pick\ \lfloor a \rfloor \sim a$
⟨*proof*⟩

theorem *pick-eqv* [*intro*, *simp*]: $pick\ \lfloor a \rfloor \sim (a::'a::equiv)$
⟨*proof*⟩

theorem *pick-inverse*: $\lfloor pick\ A \rfloor = (A::'a::equiv\ quot)$
⟨*proof*⟩

end

18 Summing natural numbers

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

Summing natural numbers, squares, cubes, etc.

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

lemmas [*simp*] =
ring-distribs
diff-mult-distrib diff-mult-distrib2 — for type nat

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

lemma *sum-of-odds*: $(\sum i=0..<n.\ Suc\ (i + i)) = n * n$
⟨*proof*⟩

The sum of the first n odd squares.

lemma *sum-of-odd-squares*:
$3 * (\sum i=0..<n.\ Suc(2*i) * Suc(2*i)) = n * (4 * n * n - 1)$
⟨*proof*⟩

The sum of the first n odd cubes

lemma *sum-of-odd-cubes*:
$(\sum i=0..<n.\ Suc\ (2*i) * Suc\ (2*i) * Suc\ (2*i)) =$
$n * n * (2 * n * n - 1)$
⟨*proof*⟩

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

lemma *sum-of-naturals*:
$2 * (\sum i=0..n.\ i) = n * Suc\ n$
⟨*proof*⟩

lemma *sum-of-squares*:
$6 * (\sum i=0..n.\ i * i) = n * Suc\ n * Suc\ (2 * n)$
⟨*proof*⟩

lemma *sum-of-cubes*:
$4 * (\sum i=0..n.\ i * i * i) = n * n * Suc\ n * Suc\ n$
⟨*proof*⟩

A cute identity:

lemma *sum-squared*: $(\sum i=0..n.\ i)\hat{}2 = (\sum i=0..n::nat.\ i\hat{}3)$
⟨*proof*⟩

Sum of fourth powers: three versions.

lemma *sum-of-fourth-powers*:
$30 * (\sum i{=}0..n.\ i * i * i * i) =$
$n * Suc\ n * Suc\ (2 * n) * (3 * n * n + 3 * n - 1)$
$\langle proof \rangle$

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

lemma *int-sum-of-fourth-powers*:
$30 * int\ (\sum i{=}0..{<}m.\ i * i * i * i) =$
$int\ m * (int\ m - 1) * (int(2 * m) - 1) *$
$(int(3 * m * m) - int(3 * m) - 1)$
$\langle proof \rangle$

lemma *of-nat-sum-of-fourth-powers*:
$30 * of\text{-}nat\ (\sum i{=}0..{<}m.\ i * i * i * i) =$
$of\text{-}nat\ m * (of\text{-}nat\ m - 1) * (of\text{-}nat\ (2 * m) - 1) *$
$(of\text{-}nat\ (3 * m * m) - of\text{-}nat\ (3 * m) - (1{::}int))$
$\langle proof \rangle$

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

lemma *sum-of-2-powers*: $(\sum i{=}0..{<}n.\ 2\hat{}i) = 2\hat{}n - (1{::}nat)$
$\langle proof \rangle$

lemma *sum-of-3-powers*: $2 * (\sum i{=}0..{<}n.\ 3\hat{}i) = 3\hat{}n - (1{::}nat)$
$\langle proof \rangle$

lemma *sum-of-powers*: $0 < k \Longrightarrow (k - 1) * (\sum i{=}0..{<}n.\ k\hat{}i) = k\hat{}n - (1{::}nat)$
$\langle proof \rangle$

end

19 Three Divides Theorem

theory *ThreeDivides*
imports *Main LaTeXsugar*
begin

19.1 Abstract

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

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

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

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

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

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

□

19.2 Formal proof

19.2.1 Miscellaneous summation lemmas

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

lemma *div-sum*:
  fixes $a::nat$ **and** $n::nat$
  shows $\forall x.\ a\ dvd\ A\ x \Longrightarrow a\ dvd\ (\sum x{<}n.\ A\ x * D\ x)$
$\langle proof \rangle$

19.2.2 Generalised Three Divides

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

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

lemma *digit-diff-split*:
  fixes $n::nat$ **and** $nd::nat$ **and** $x::nat$
  shows $n = (\sum x \in \{..{<}nd\}.\ (D\ x)*((10::nat)\hat{\ }x)) \Longrightarrow$
    $(n - (\sum x{<}nd.\ (D\ x))) = (\sum x{<}nd.\ (D\ x)*(10\hat{\ }x - 1))$
$\langle proof \rangle$

Now we prove that 3 always divides numbers of the form $10^x - 1$.

lemma *three-divs-0*:
  shows $(3::nat)\ dvd\ (10\hat{\ }x - 1)$
$\langle proof \rangle$

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

lemma *three-divs-1*:
  fixes $D :: nat \Rightarrow nat$
  shows $3\ dvd\ (\sum x{<}nd.\ D\ x * (10\hat{\ }x - 1))$
  $\langle proof \rangle$

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*⟩

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*⟩

19.2.3 Three Divides Natural

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

Definitions of length and digit sum.

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

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

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

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

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

Some properties of these functions follow.

lemma *nlen-zero*:
  $0 = nlen\ x \Longrightarrow x = 0$
  ⟨*proof*⟩

lemma *nlen-suc*:
  $Suc\ m = nlen\ n \Longrightarrow m = nlen\ (n\ div\ 10)$
  ⟨*proof*⟩

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

lemma *exp-exists*:
  $m = (\sum x{<}nlen\ m.\ (m\ div\ (10{::}nat)\ \hat{}x\ mod\ 10) * 10\hat{}x)$
⟨*proof*⟩

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*⟩

end

20 Higher-Order Logic: Intuitionistic predicate calculus problems

theory *Intuitionistic* **imports** *Main* **begin**

lemma $({\sim}{\sim}(P\&Q)) = (({\sim}{\sim}P)\ \&\ ({\sim}{\sim}Q))$
  ⟨*proof*⟩

lemma ${\sim}{\sim}\ (({\sim}P\ {-}{-}{>}\ Q)\ {-}{-}{>}\ ({\sim}P\ {-}{-}{>}\ {\sim}Q)\ {-}{-}{>}\ P)$
  ⟨*proof*⟩

lemma $({\sim}{\sim}(P{-}{-}{>}Q))\ = ({\sim}{\sim}P\ {-}{-}{>}\ {\sim}{\sim}Q)$
  ⟨*proof*⟩

lemma $({\sim}{\sim}{\sim}P) = ({\sim}P)$
  ⟨*proof*⟩

lemma ${\sim}{\sim}((P\ {-}{-}{>}\ Q\ |\ R)\ \ {-}{-}{>}\ \ (P{-}{-}{>}Q)\ |\ (P{-}{-}{>}R))$
  ⟨*proof*⟩

lemma $(P{=}Q) = (Q{=}P)$
  ⟨*proof*⟩

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

$\langle proof \rangle$

lemma $(((G{-}{-}{>}A)\ {-}{-}{>}\ J)\ {-}{-}{>}\ D\ {-}{-}{>}\ E)\ {-}{-}{>}\ (((H{-}{-}{>}B){-}{-}{>}I){-}{-}{>}C{-}{-}{>}J)$
$\ {-}{-}{>}\ (A{-}{-}{>}H)\ {-}{-}{>}\ F\ {-}{-}{>}\ G\ {-}{-}{>}\ (((C{-}{-}{>}B){-}{-}{>}I){-}{-}{>}D){-}{-}{>}(A{-}{-}{>}C)$
$\ {-}{-}{>}\ (((F{-}{-}{>}A){-}{-}{>}B)\ {-}{-}{>}\ I)\ {-}{-}{>}\ E$
$\langle proof \rangle$

lemma $P\ {-}{-}{>}\ {\sim}{\sim}P$
$\langle proof \rangle$

lemma ${\sim}{\sim}({\sim}{\sim}P\ {-}{-}{>}\ P)$
$\langle proof \rangle$

lemma ${\sim}{\sim}P\ \&\ {\sim}{\sim}(P\ {-}{-}{>}\ Q)\ {-}{-}{>}\ {\sim}{\sim}Q$
$\langle proof \rangle$

lemma $((P{=}Q)\ {-}{-}{>}\ P\&Q\&R)\ \&$
$((Q{=}R)\ {-}{-}{>}\ P\&Q\&R)\ \&$
$((R{=}P)\ {-}{-}{>}\ P\&Q\&R)\ {-}{-}{>}\ P\&Q\&R$
$\langle proof \rangle$

lemma $((P{=}Q)\ {-}{-}{>}\ P\&Q\&R\&S\&T)\ \&$
$((Q{=}R)\ {-}{-}{>}\ P\&Q\&R\&S\&T)\ \&$
$((R{=}S)\ {-}{-}{>}\ P\&Q\&R\&S\&T)\ \&$
$((S{=}T)\ {-}{-}{>}\ P\&Q\&R\&S\&T)\ \&$
$((T{=}P)\ {-}{-}{>}\ P\&Q\&R\&S\&T)\ {-}{-}{>}\ P\&Q\&R\&S\&T$
$\langle proof \rangle$

lemma $(ALL\ x.\ EX\ y.\ ALL\ z.\ p(x)\ \&\ q(y)\ \&\ r(z)) =$
$(ALL\ z.\ EX\ y.\ ALL\ x.\ p(x)\ \&\ q(y)\ \&\ r(z))$
$\langle proof \rangle$

lemma ${\sim}\ (EX\ x.\ ALL\ y.\ p\ y\ x = ({\sim}\ p\ x\ x))$
$\langle proof \rangle$

lemma $\sim\sim((P{-}{-}{>}Q) \ = \ (\sim Q \ {-}{-}{>} \sim P))$
⟨*proof*⟩

lemma $\sim\sim(\sim\sim P \ = \ P)$
⟨*proof*⟩

lemma $\sim(P{-}{-}{>}Q) \ {-}{-}{>} \ (Q{-}{-}{>}P)$
⟨*proof*⟩

lemma $\sim\sim((\sim P{-}{-}{>}Q) \ = \ (\sim Q \ {-}{-}{>} \ P))$
⟨*proof*⟩

lemma $\sim\sim((P|Q{-}{-}{>}P|R) \ {-}{-}{>} \ P|(Q{-}{-}{>}R))$
⟨*proof*⟩

lemma $\sim\sim(P \mid \sim P)$
⟨*proof*⟩

lemma $\sim\sim(P \mid \sim\sim\sim P)$
⟨*proof*⟩

lemma $\sim\sim(((P{-}{-}{>}Q) \ {-}{-}{>} \ P) \ \ {-}{-}{>} \ \ P)$
⟨*proof*⟩

lemma $((P|Q) \ \& \ (\sim P|Q) \ \& \ (P| \sim Q)) \ {-}{-}{>} \sim \ (\sim P \mid \sim Q)$
⟨*proof*⟩

lemma $(Q{-}{-}{>}R) \ {-}{-}{>} \ (R{-}{-}{>}P\&Q) \ {-}{-}{>} \ (P{-}{-}{>}(Q|R)) \ {-}{-}{>} \ (P{=}Q)$
⟨*proof*⟩

lemma $P{=}P$
⟨*proof*⟩

lemma $\sim\sim(((P \ = \ Q) \ = \ R) \ \ = \ \ (P \ = \ (Q \ = \ R)))$
⟨*proof*⟩

lemma $((P = Q) = R)$ $-->$ ~~$(P = (Q = R))$
⟨*proof*⟩

lemma $(P \mid (Q \ \& \ R))$ $= ((P \mid Q) \ \& \ (P \mid R))$
⟨*proof*⟩

lemma ~~$((P = Q) = ((Q \mid$ ~$P) \ \& \ ($~$Q|P)))$
⟨*proof*⟩

lemma ~~$((P$ $-->$ $Q) = ($~$P \mid Q))$
⟨*proof*⟩

lemma ~~$((P-->Q) \mid (Q-->P))$
⟨*proof*⟩

lemma ~~$(((P \ \& \ (Q-->R))-->S) = (($~$P \mid Q \mid S) \ \& \ ($~$P \mid$ ~$R \mid S)))$
⟨*proof*⟩

lemma $(P\&Q) = (P = (Q = (P|Q)))$
⟨*proof*⟩

lemma $(EX\ x.\ P(x)-->Q)$ $-->$ $(ALL\ x.\ P(x))$ $-->$ Q
⟨*proof*⟩

lemma $((ALL\ x.\ P(x))-->Q)$ $-->$ ~ $(ALL\ x.\ P(x)\ \&$ ~$Q)$
⟨*proof*⟩

lemma $((ALL\ x.$ ~$P(x))-->Q)$ $-->$ ~ $(ALL\ x.$ ~ $(P(x)|Q))$
⟨*proof*⟩

lemma $(ALL\ x.\ P(x)) \mid Q$ $-->$ $(ALL\ x.\ P(x) \mid Q)$
⟨*proof*⟩

lemma $(EX\ x.\ P$ $-->$ $Q(x))$ $-->$ $(P$ $-->$ $(EX\ x.\ Q(x)))$
⟨*proof*⟩

lemma ~~(EX x. ALL y z. (P(y)-->Q(z)) --> (P(x)-->Q(x)))
⟨proof⟩

lemma (ALL x y. EX z. ALL w. (P(x)&Q(y)-->R(z)&S(w)))
--> (EX x y. P(x) & Q(y)) --> (EX z. R(z))
⟨proof⟩

lemma (EX x. P-->Q(x)) & (EX x. Q(x)-->P) --> ~~(EX x. P=Q(x))
⟨proof⟩

lemma (ALL x. P = Q(x)) --> (P = (ALL x. Q(x)))
⟨proof⟩

lemma ~~ ((ALL x. P | Q(x)) = (P | (ALL x. Q(x))))
⟨proof⟩

lemma (EX x. P(x)) &
(ALL x. L(x) --> ~ (M(x) & R(x))) &
(ALL x. P(x) --> (M(x) & L(x))) &
((ALL x. P(x)-->Q(x)) | (EX x. P(x)&R(x)))
--> (EX x. Q(x)&P(x))
⟨proof⟩

lemma (EX x. P(x) & ~Q(x)) &
(ALL x. P(x) --> R(x)) &
(ALL x. M(x) & L(x) --> P(x)) &
((EX x. R(x) & ~ Q(x)) --> (ALL x. L(x) --> ~ R(x)))
--> (ALL x. M(x) --> ~L(x))
⟨proof⟩

lemma (ALL x. P(x) --> (ALL x. Q(x))) &
(~~(ALL x. Q(x)|R(x)) --> (EX x. Q(x)&S(x))) &
(~~(EX x. S(x)) --> (ALL x. L(x) --> M(x)))
--> (ALL x. P(x) & L(x) --> M(x))
⟨proof⟩

lemma $(((EX\ x.\ P(x))\ \&\ (EX\ y.\ Q(y)))\ {-}{-}{>}$
$(((ALL\ x.\ (P(x)\ {-}{-}{>}\ R(x)))\ \&\ (ALL\ y.\ (Q(y)\ {-}{-}{>}\ S(y))))\ =$
$(ALL\ x\ y.\ ((P(x)\ \&\ Q(y))\ {-}{-}{>}\ (R(x)\ \&\ S(y))))))$
⟨*proof*⟩

lemma $(ALL\ x.\ (P(x)\ |\ Q(x))\ {-}{-}{>}\ {\sim}\ R(x))\ \&$
$(ALL\ x.\ (Q(x)\ {-}{-}{>}\ {\sim}\ S(x))\ {-}{-}{>}\ P(x)\ \&\ R(x))$
${-}{-}{>}\ (ALL\ x.\ {\sim}{\sim}S(x))$
⟨*proof*⟩

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

lemma $(ALL\ x.\ P(x)\ \&\ (Q(x)|R(x)){-}{-}{>}S(x))\ \&$
$(ALL\ x.\ S(x)\ \&\ R(x)\ {-}{-}{>}\ L(x))\ \&$
$(ALL\ x.\ M(x)\ {-}{-}{>}\ R(x))$
${-}{-}{>}\ (ALL\ x.\ P(x)\ \&\ M(x)\ {-}{-}{>}\ L(x))$
⟨*proof*⟩

lemma $(ALL\ x.\ {\sim}{\sim}(P(a)\ \&\ (P(x){-}{-}{>}P(b)){-}{-}{>}P(c)))\ =$
$(ALL\ x.\ {\sim}{\sim}(({\sim}P(a)\ |\ P(x)\ |\ P(c))\ \&\ ({\sim}P(a)\ |\ {\sim}P(b)\ |\ P(c))))$
⟨*proof*⟩

lemma
$(ALL\ x.\ EX\ y.\ J\ x\ y)\ \&$
$(ALL\ x.\ EX\ y.\ G\ x\ y)\ \&$
$(ALL\ x\ y.\ J\ x\ y\ |\ G\ x\ y\ {-}{-}{>}\ (ALL\ z.\ J\ y\ z\ |\ G\ y\ z\ {-}{-}{>}\ H\ x\ z))$
${-}{-}{>}\ (ALL\ x.\ EX\ y.\ H\ x\ y)$
⟨*proof*⟩

lemma ${\sim}\ (EX\ x.\ ALL\ y.\ F\ y\ x = ({\sim}F\ y\ y))$
⟨*proof*⟩

lemma $(EX\ y.\ ALL\ x.\ F\ x\ y = F\ x\ x)\ {-}{-}{>}$
${\sim}(ALL\ x.\ EX\ y.\ ALL\ z.\ F\ z\ y = ({\sim}\ F\ z\ x))$
⟨*proof*⟩

lemma $(ALL\ x.\ f(x)\ {-}{-}{>}$

$(EX\ y.\ g(y)\ \&\ h\ x\ y\ \&\ (EX\ y.\ g(y)\ \&\ \sim\ h\ x\ y)))\ \ \&$
$(EX\ x.\ j(x)\ \&\ (ALL\ y.\ g(y)\ {-}{-}{>}\ h\ x\ y))$
${-}{-}{>}\ (EX\ x.\ j(x)\ \&\ {\sim}f(x))$
⟨*proof*⟩

lemma $(a{=}b\ |\ c{=}d)\ \&\ (a{=}c\ |\ b{=}d)\ {-}{-}{>}\ a{=}d\ |\ b{=}c$
⟨*proof*⟩

lemma $((EX\ z\ w.\ (ALL\ x\ y.\ (P\ x\ y = ((x = z)\ \&\ (y = w)))))\ {-}{-}{>}$
$(EX\ z.\ (ALL\ x.\ (EX\ w.\ ((ALL\ y.\ (P\ x\ y = (y = w))) = (x = z))))))$
⟨*proof*⟩

lemma $((EX\ z\ w.\ (ALL\ x\ y.\ (P\ x\ y = ((x = z)\ \&\ (y = w)))))\ {-}{-}{>}$
$(EX\ w.\ (ALL\ y.\ (EX\ z.\ ((ALL\ x.\ (P\ x\ y = (x = z))) = (y = w))))))$
⟨*proof*⟩

lemma $(ALL\ x.\ (EX\ y.\ P(y)\ \&\ x{=}f(y))\ {-}{-}{>}\ P(x)) = (ALL\ x.\ P(x)\ {-}{-}{>}$
$P(f(x)))$
⟨*proof*⟩

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

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

end

21 CTL formulae

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

We formalize basic concepts of Computational Tree Logic (CTL) [4, 3] within the simply-typed set theory of HOL.

By using the common technique of "shallow embedding", a CTL formula is identified with the corresponding set of states where it holds. Consequently, CTL operations such as negation, conjunction, disjunction simply become complement, intersection, union of sets. We only require a separate operation for implication, as point-wise inclusion is usually not encountered in plain set-theory.

lemmas [*intro*!] = *Int-greatest Un-upper2 Un-upper1 Int-lower1 Int-lower2*

types $'a\ ctl = 'a\ set$

definition
$imp :: 'a\ ctl \Rightarrow 'a\ ctl \Rightarrow 'a\ ctl$ (**infixr** $\rightarrow$ *75*) **where**
$p \rightarrow q = -\ p \cup q$

lemma [*intro*!]: $p \cap p \rightarrow q \subseteq q$ ⟨*proof*⟩
lemma [*intro*!]: $p \subseteq (q \rightarrow p)$ ⟨*proof*⟩

The CTL path operators are more interesting; they are based on an arbitrary, but fixed model $\mathcal{M}$, which is simply a transition relation over states $'a$.

axiomatization $\mathcal{M} :: ('a \times 'a)\ set$

The operators EX, EF, EG are taken as primitives, while AX, AF, AG are defined as derived ones. The formula EX p holds in a state s, iff there is a successor state s' (with respect to the model $\mathcal{M}$), such that p holds in s'. The formula EF p holds in a state s, iff there is a path in $\mathcal{M}$, starting from s, such that there exists a state s' on the path, such that p holds in s'. The formula EG p holds in a state s, iff there is a path, starting from s, such that for all states s' on the path, p holds in s'. It is easy to see that EF p and EG p may be expressed using least and greatest fixed points [4].

definition
EX (EX - [*80*] *90*) **where** EX $p = \{s.\ \exists s'.\ (s,\ s') \in \mathcal{M} \wedge s' \in p\}$
definition
EF (EF - [*80*] *90*) **where** EF $p = lfp\ (\lambda s.\ p \cup \text{EX}\ s)$
definition
EG (EG - [*80*] *90*) **where** EG $p = gfp\ (\lambda s.\ p \cap \text{EX}\ s)$

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

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

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

21.1 Basic fixed point properties

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

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

$\langle proof \rangle$

lemma *lfp-gfp'*: $- \mathit{lfp}\ f = \mathit{gfp}\ (\lambda s::'a\ set.\ - (f\ (- s)))$
$\langle proof \rangle$

lemma *gfp-lfp'*: $- \mathit{gfp}\ f = \mathit{lfp}\ (\lambda s::'a\ set.\ - (f\ (- s)))$
$\langle proof \rangle$

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

lemma *AF-lfp*: AF $p = \mathit{lfp}\ (\lambda s.\ p \cup \mathrm{AX}\ s)$ $\langle proof \rangle$
lemma *AG-gfp*: AG $p = \mathit{gfp}\ (\lambda s.\ p \cap \mathrm{AX}\ s)$ $\langle proof \rangle$

lemma *EF-fp*: EF $p = p \cup$ EX EF p
$\langle proof \rangle$

lemma *AF-fp*: AF $p = p \cup$ AX AF p
$\langle proof \rangle$

lemma *EG-fp*: EG $p = p \cap$ EX EG p
$\langle proof \rangle$

From the greatest fixed point definition of AG p, we derive as a consequence of the Knaster-Tarski theorem on the one hand that AG p is a fixed point of the monotonic function $\lambda s.\ p \cap \mathrm{AX}\ s$.

lemma *AG-fp*: AG $p = p \cap$ AX AG p
$\langle proof \rangle$

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 $p \subseteq p$
$\langle proof \rangle$

lemma *AG-fp-2*: AG $p \subseteq$ AX AG p
$\langle proof \rangle$

On the other hand, we have from the Knaster-Tarski fixed point theorem that any other post-fixed point of $\lambda s.\ p \cap \mathrm{AX}\ s$ is smaller than AG p. A post-fixed point is a set of states q such that $q \subseteq p \cap \mathrm{AX}\ q$. This leads to the following co-induction principle for AG p.

lemma *AG-I*: $q \subseteq p \cap \mathrm{AX}\ q \Longrightarrow q \subseteq \mathrm{AG}\ p$
$\langle proof \rangle$

21.2 The tree induction principle

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

lemma *AX-int*: AX $(p \cap q) = $ AX $p \cap$ AX q ⟨*proof*⟩
lemma *AX-mono*: $p \subseteq q \Longrightarrow$ AX $p \subseteq$ AX q ⟨*proof*⟩
lemma *AG-mono*: $p \subseteq q \Longrightarrow$ AG $p \subseteq$ AG q
⟨*proof*⟩

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

lemma *AG-AX*: AG $p \subseteq$ AX p
⟨*proof*⟩

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

lemma *AG-AG*: AG AG $p =$ AG p
⟨*proof*⟩

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

theorem *AG-induct*: $p \cap$ AG $(p \to$ AX $p) =$ AG p
⟨*proof*⟩

21.3 An application of tree induction

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

theorem *AG-AX-commute*: AG AX $p =$ AX AG p
⟨*proof*⟩

end

22 Arithmetic

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

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

`fast_arith_tac` is a very basic version of the tactic. It performs no meta-to-object-logic conversion, and only some splitting of operators. `simple_arith_tac` performs meta-to-object-logic conversion, full splitting of operators, and NNF normalization of the goal. The *arith* method combines them both, and tries other methods (e.g. *presburger*) as well. This is the one that you should use in your proofs!

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

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

lemma $(i::nat) <= max\ i\ j$
⟨*proof*⟩

lemma $(i::int) <= max\ i\ j$
⟨*proof*⟩

lemma $min\ i\ j <= (i::nat)$
⟨*proof*⟩

lemma $min\ i\ j <= (i::int)$
⟨*proof*⟩

lemma $min\ (i::nat)\ j <= max\ i\ j$
⟨*proof*⟩

lemma $min\ (i::int)\ j <= max\ i\ j$
⟨*proof*⟩

lemma $min\ (i::nat)\ j + max\ i\ j = i + j$
⟨*proof*⟩

lemma $min\ (i::int)\ j + max\ i\ j = i + j$
⟨*proof*⟩

lemma $(i::nat) < j ==> min\ i\ j < max\ i\ j$
⟨*proof*⟩

lemma $(i::int) < j ==> min\ i\ j < max\ i\ j$
⟨*proof*⟩

lemma $(0::int) <= abs\ i$
⟨*proof*⟩

lemma $(i::int) <= abs\ i$
⟨*proof*⟩

lemma $abs\ (abs\ (i::int)) = abs\ i$
⟨*proof*⟩

Also testing subgoals with bound variables.

lemma $!!x.\ (x::nat) <= y ==> x - y = 0$
  ⟨*proof*⟩

lemma $!!x.\ (x::nat) - y = 0 ==> x <= y$
  ⟨*proof*⟩

lemma $!!x.\ ((x::nat) <= y) = (x - y = 0)$
  ⟨*proof*⟩

lemma $[|\ (x::nat) < y;\ d < 1\ |] ==> x - y = d$
  ⟨*proof*⟩

lemma $[|\ (x::nat) < y;\ d < 1\ |] ==> x - y - x = d - x$
  ⟨*proof*⟩

lemma $(x::int) < y ==> x - y < 0$
  ⟨*proof*⟩

lemma $nat\ (i + j) <= nat\ i + nat\ j$
  ⟨*proof*⟩

lemma $i < j ==> nat\ (i - j) = 0$
  ⟨*proof*⟩

lemma $(i::nat)\ mod\ 0 = i$

  ⟨*proof*⟩

lemma $(i::nat)\ mod\ 1 = 0$

  ⟨*proof*⟩

lemma $(i::nat)\ mod\ 42 <= 41$
  ⟨*proof*⟩

lemma $(i::int)\ mod\ 0 = i$

  ⟨*proof*⟩

lemma $(i::int)\ mod\ 1 = 0$

  ⟨*proof*⟩

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

  ⟨*proof*⟩

lemma $-(i::int) * 1 = 0 ==> i = 0$

⟨*proof*⟩

lemma $[|\ (0::int) < abs\ i;\ abs\ i * 1 < abs\ i * j\ |] ==> 1 < abs\ i * j$
⟨*proof*⟩

22.2 Meta-Logic

lemma $x < Suc\ y == x <= y$
⟨*proof*⟩

lemma $((x::nat) == z ==> x \sim= y) ==> x \sim= y \mid z \sim= y$
⟨*proof*⟩

22.3 Various Other Examples

lemma $(x < Suc\ y) = (x <= y)$
⟨*proof*⟩

lemma $[|\ (x::nat) < y;\ y < z\ |] ==> x < z$
⟨*proof*⟩

lemma $(x::nat) < y\ \&\ y < z ==> x < z$
⟨*proof*⟩

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

lemma $(P::bool) = Q ==> Q = P$
⟨*proof*⟩

lemma $[|\ P = (x = 0);\ (\sim P) = (y = 0)\ |] ==> min\ (x::nat)\ y = 0$
⟨*proof*⟩

lemma $[|\ P = (x = 0);\ (\sim P) = (y = 0)\ |] ==> max\ (x::nat)\ y = x + y$
⟨*proof*⟩

lemma $[|\ (x::nat) \sim= y;\ a + 2 = b;\ a < y;\ y < b;\ a < x;\ x < b\ |] ==> False$
⟨*proof*⟩

lemma $[|\ (x::nat) > y;\ y > z;\ z > x\ |] ==> False$
⟨*proof*⟩

lemma $(x::nat) - 5 > y ==> y < x$
⟨*proof*⟩

lemma $(x::nat) \sim= 0 ==> 0 < x$
⟨*proof*⟩

lemma $[|\ (x::nat) \sim= y;\ x <= y\ |] ==> x < y$
⟨*proof*⟩

lemma [| $(x::nat) < y$; $P\ (x - y)$ |] ==> $P\ 0$
⟨proof⟩

lemma $(x - y) - (x::nat) = (x - x) - y$
⟨proof⟩

lemma [| $(a::nat) < b$; $c < d$ |] ==> $(a - b) = (c - d)$
⟨proof⟩

lemma $((a::nat) - (b - (c - (d - e)))) = (a - (b - (c - (d - e))))$
⟨proof⟩

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

⟨proof⟩

lemma $2 * (x::nat)$ ~= 1

⟨proof⟩

Constants.

lemma $(0::nat) < 1$
⟨proof⟩

lemma $(0::int) < 1$
⟨proof⟩

lemma $(47::nat) + 11 < 08 * 15$
⟨proof⟩

lemma $(47::int) + 11 < 08 * 15$
⟨proof⟩

Splitting of inequalities of different type.

lemma [| $(a::nat)$ ~= b; $(i::int)$ ~= j; $a < 2$; $b < 2$ |] ==>
$a + b$ <= $nat\ (max\ (abs\ i)\ (abs\ j))$

⟨*proof*⟩

Again, but different order.

lemma [| (*i*::*int*) ~= *j*; (*a*::*nat*) ~= *b*; *a* < *2*; *b* < *2* |] ==>
  a + *b* <= *nat* (*max* (*abs i*) (*abs j*))
  ⟨*proof*⟩

end

23 Binary trees

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

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

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

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

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

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

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

primrec
  bt-map f Lf = *Lf*

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

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

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

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

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

BT simplification

lemma *n-leaves-reflect*: *n-leaves (reflect t) = n-leaves t*
⟨*proof*⟩

lemma *n-nodes-reflect*: *n-nodes (reflect t) = n-nodes t*
⟨*proof*⟩

lemma *depth-reflect*: *depth (reflect t) = depth t*
⟨*proof*⟩

The famous relationship between the numbers of leaves and nodes.

lemma *n-leaves-nodes*: *n-leaves t = Suc (n-nodes t)*
⟨*proof*⟩

lemma *reflect-reflect-ident*: *reflect (reflect t) = t*
⟨*proof*⟩

lemma *bt-map-reflect*: *bt-map f (reflect t) = reflect (bt-map f t)*
⟨*proof*⟩

lemma *preorder-bt-map*: *preorder (bt-map f t) = map f (preorder t)*
⟨*proof*⟩

lemma *inorder-bt-map*: *inorder (bt-map f t) = map f (inorder t)*
⟨*proof*⟩

lemma *postorder-bt-map*: *postorder (bt-map f t) = map f (postorder t)*
⟨*proof*⟩

lemma *depth-bt-map* [*simp*]: *depth (bt-map f t) = depth t*
⟨*proof*⟩

lemma *n-leaves-bt-map* [*simp*]: *n-leaves* (*bt-map f t*) = *n-leaves t*
⟨*proof*⟩

lemma *preorder-reflect*: *preorder* (*reflect t*) = *rev* (*postorder t*)
⟨*proof*⟩

lemma *inorder-reflect*: *inorder* (*reflect t*) = *rev* (*inorder t*)
⟨*proof*⟩

lemma *postorder-reflect*: *postorder* (*reflect t*) = *rev* (*preorder t*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *append-Lf2* [*simp*]: *append t Lf* = *t*
⟨*proof*⟩

lemma *depth-append* [*simp*]: *depth* (*append t1 t2*) = *depth t1* + *depth t2*
⟨*proof*⟩

lemma *n-leaves-append* [*simp*]:
n-leaves (*append t1 t2*) = *n-leaves t1* * *n-leaves t2*
⟨*proof*⟩

lemma *bt-map-append*:
bt-map f (*append t1 t2*) = *append* (*bt-map f t1*) (*bt-map f t2*)
⟨*proof*⟩

end

24 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*
⟨*proof*⟩

lemma *sorted-append* [*simp*]:
  sorted le (*xs*@*ys*) =
    (*sorted le xs* & *sorted le ys* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *le x y*))
  ⟨*proof*⟩

end

25 Merge Sort

theory *MergeSort*
imports *Sorting*
begin

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

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

    merge(*xs*,[]) = *xs*

    merge([],*ys*) = *ys*

lemma *multiset-of-merge*[*simp*]:
     multiset-of (*merge*(*xs*,*ys*)) = *multiset-of xs* + *multiset-of ys*
⟨*proof*⟩

lemma *set-merge*[*simp*]: $set(merge(xs,ys)) = set\ xs \cup set\ ys$
⟨*proof*⟩

lemma *sorted-merge*[*simp*]:
$sorted\ (op \leq)\ (merge(xs,ys)) = (sorted\ (op \leq)\ xs\ \&\ sorted\ (op \leq)\ ys)$
⟨*proof*⟩

consts $msort :: ('a::linorder)\ list \Rightarrow 'a\ list$
recdef *msort measure size*
$msort\ [] = []$
$msort\ [x] = [x]$
$msort\ xs = merge(msort(take\ (size\ xs\ div\ 2)\ xs),$
$msort(drop\ (size\ xs\ div\ 2)\ xs))$

theorem *sorted-msort*: $sorted\ (op \leq)\ (msort\ xs)$
⟨*proof*⟩

theorem *multiset-of-msort*: $multiset\text{-}of\ (msort\ xs) = multiset\text{-}of\ xs$
⟨*proof*⟩

end

26 A question from "Bundeswettbewerb Mathematik"

theory *Puzzle* **imports** *Main* **begin**

consts $f :: nat => nat$

specification (f)
f-ax [*intro*!]: $f(f(n)) < f(Suc(n))$
⟨*proof*⟩

lemma *lemma0* [*rule-format*]: $\forall\, n.\ k{=}f(n) \dashrightarrow n <= f(n)$
⟨*proof*⟩

lemma *lemma1*: $n <= f(n)$
⟨*proof*⟩

lemma *f-mono* [*rule-format* (*no-asm*)]: $m <= n \dashrightarrow f(m) <= f(n)$
⟨*proof*⟩

lemma *f-id*: $f(n) = n$
⟨*proof*⟩

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*⟩

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)*
⟨*proof*⟩

A challenge by John Harrison. Takes about 17s on a 1.6GHz machine.

lemma fixes *p1* :: *'a::comm-ring* **shows**
*(sq p1 + sq q1 + sq r1 + sq s1 + sq t1 + sq u1 + sq v1 + sq w1) **
(sq p2 + sq q2 + sq r2 + sq s2 + sq t2 + sq u2 + sq v2 + sq w2)
*= sq (p1*p2 − q1*q2 − r1*r2 − s1*s2 − t1*t2 − u1*u2 − v1*v2 − w1*w2)*
+
*sq (p1*q2 + q1*p2 + r1*s2 − s1*r2 + t1*u2 − u1*t2 − v1*w2 + w1*v2)*
+
*sq (p1*r2 − q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 − v1*t2 − w1*u2)*
+
*sq (p1*s2 + q1*r2 − r1*q2 + s1*p2 + t1*w2 − u1*v2 + v1*u2 − w1*t2)*
+
*sq (p1*t2 − q1*u2 − r1*v2 − s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)*
+
*sq (p1*u2 + q1*t2 − r1*w2 + s1*v2 − t1*q2 + u1*p2 − v1*s2 + w1*r2)*
+
*sq (p1*v2 + q1*w2 + r1*t2 − s1*u2 − t1*r2 + u1*s2 + v1*p2 − w1*q2)*
+
*sq (p1*w2 − q1*v2 + r1*u2 + s1*t2 − t1*s2 − u1*r2 + v1*q2 + w1*p2)*
⟨*proof*⟩

end

28 Groebner Basis Examples

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

28.1 Basic examples

lemma $3\ \hat{}\ 3 == (?X::'a::\{number\text{-}ring,recpower\})$
⟨*proof*⟩

lemma $(x - (-2))\hat{}5 == ?X::int$
⟨*proof*⟩

lemma $(x - (-2))\hat{}5 * (y - 78)\ \hat{}\ 8 == ?X::int$
⟨*proof*⟩

lemma $((-3)\ \hat{}\ (Suc\ (Suc\ (Suc\ 0)))) == (X::'a::\{number\text{-}ring,recpower\})$
⟨*proof*⟩

lemma $((x::int) + y)\hat{}3 - 1 = (x - z)\hat{}2 - 10 \Longrightarrow x = z + 3 \Longrightarrow x = - y$
⟨*proof*⟩

lemma $(4::nat) + 4 = 3 + 5$
⟨*proof*⟩

lemma $(4::int) + 0 = 4$
⟨*proof*⟩

lemma
assumes $a * x\hat{}2 + b * x + c = (0::int)$ **and** $d * x\hat{}2 + e * x + f = 0$
shows $d\hat{}2*c\hat{}2 - 2*d*c*a*f + a\hat{}2*f\hat{}2 - e*d*b*c - e*b*a*f + a*e\hat{}2*c + f*d*b\hat{}2 = 0$
⟨*proof*⟩

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

theorem $x* (x^2 - x - 5) - 3 = (0::int) \longleftrightarrow (x = 3 \vee x = -1)$
⟨*proof*⟩

lemma
fixes $x::'a::\{idom,recpower,number\text{-}ring\}$
shows $x\hat{}2*y = x\hat{}2\ \&\ x*y\hat{}2 = y\hat{}2 \longleftrightarrow x=1\ \&\ y=1\ |\ x=0\ \&\ y=0$
⟨*proof*⟩

28.2 Lemmas for Lagrange's theorem

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

lemma
  fixes *x1* :: *'a*::{*idom*,*recpower*,*number-ring*}
  shows
  (*sq x1* + *sq x2* + *sq x3* + *sq x4*) ∗ (*sq y1* + *sq y2* + *sq y3* + *sq y4*) =
    sq (*x1*∗*y1* − *x2*∗*y2* − *x3*∗*y3* − *x4*∗*y4*) +
    sq (*x1*∗*y2* + *x2*∗*y1* + *x3*∗*y4* − *x4*∗*y3*) +
    sq (*x1*∗*y3* − *x2*∗*y4* + *x3*∗*y1* + *x4*∗*y2*) +
    sq (*x1*∗*y4* + *x2*∗*y3* − *x3*∗*y2* + *x4*∗*y1*)
  ⟨*proof*⟩

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

28.3 Colinearity is invariant by rotation

types *point* = *int* × *int*

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

lemma *collinear-inv-rotation*:
  assumes *collinear* (*Ax*, *Ay*) (*Bx*, *By*) (*Cx*, *Cy*) **and** $c^2 + s^2 = 1$
  shows *collinear* (*Ax* ∗ *c* − *Ay* ∗ *s*, *Ay* ∗ *c* + *Ax* ∗ *s*)
    (*Bx* ∗ *c* − *By* ∗ *s*, *By* ∗ *c* + *Bx* ∗ *s*) (*Cx* ∗ *c* − *Cy* ∗ *s*, *Cy* ∗ *c* + *Cx* ∗ *s*)

⟨*proof*⟩

lemma *EX* (*d::int*). $a*y - a*x = n*d \Longrightarrow$ *EX u v*. $a*u + n*v = 1 \Longrightarrow$ *EX e*. $y - x = n*e$
⟨*proof*⟩

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*⟩

lemma *infsys-p1*: *P a b* ==> *P* (*fst* (*a*,*b*)) (*snd* (*a*,*b*))
⟨*proof*⟩

lemma *infsys-p2*: *P* (*fst* (*a*,*b*)) (*snd* (*a*,*b*)) ==> *P a b*
⟨*proof*⟩

lemma *infsys-pp1*: *P a b c* ==> *P* (*fst*(*fst*((*a*,*b*),*c*))) (*snd*(*fst* ((*a*,*b*),*c*))) (*snd* ((*a*,*b*),*c*))
⟨*proof*⟩

lemma *infsys-pp2*: *P* (*fst*(*fst*((*a*,*b*),*c*))) (*snd*(*fst*((*a*,*b*),*c*))) (*snd*((*a*,*b*),*c*)) ==> *P a b c*
⟨*proof*⟩

lemma *lfp-intro2*: [| *mono*(*f*); *x*:*f*(*lfp*(*f*)) |] ==> *x*:*lfp*(*f*)
⟨*proof*⟩

lemma *lfp-elim2*:
assumes *lfp*: *x*:*lfp*(*f*)
and *mono*: *mono*(*f*)
and *r*: !!*y*. *y*:*f*(*lfp*(*f*)) ==> *P*(*y*)
shows *P*(*x*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *gfp-coind2*:
  assumes *cih*: $x{:}f(\{x\}\ Un\ gfp(f))$
    and *monoh*: $mono(f)$
  shows $x{:}gfp(f)$
⟨*proof*⟩

lemma *gfp-elim2*:
  assumes *gfph*: $x{:}gfp(f)$
    and *monoh*: $mono(f)$
    and *caseh*: $!!y.\ y{:}f(gfp(f)) ==> P(y)$
  shows $P(x)$
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *eval-const*: *ve* |– *e-const*(*c*) −−−> *v-const*(*c*)
⟨*proof*⟩

lemma *eval-var2*:
  ev:*ve-dom*(*ve*) ==> *ve* |– *e-var*(*ev*) −−−> *ve-app ve ev*
⟨*proof*⟩

lemma *eval-fn*:
  ve |– *fn ev* => *e* −−−> *v-clos*(<|*ev*,*e*,*ve*|>)
⟨*proof*⟩

lemma *eval-fix*:
   cl = <| *ev1*, *e*, *ve* + {*ev2* |−> *v-clos*(*cl*)} |> ==>
    ve |– *fix ev2*(*ev1*) = *e* −−−> *v-clos*(*cl*)
⟨*proof*⟩

lemma *eval-app1*:
   [| *ve* |– *e1* −−−> *v-const*(*c1*); *ve* |– *e2* −−−> *v-const*(*c2*) |] ==>
    ve |– *e1* @@ *e2* −−−> *v-const*(*c-app c1 c2*)
⟨*proof*⟩

lemma *eval-app2*:
   [| *ve* |– *e1* −−−> *v-clos*(<|*xm*,*em*,*vem*|>);
       ve |– *e2* −−−> *v2*;
       vem + {*xm* |−> *v2*} |– *em* −−−> *v*
    |] ==>
    ve |– *e1* @@ *e2* −−−> *v*
⟨*proof*⟩

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))
⟨proof⟩

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
⟨proof⟩

lemma *elab-fun-mono*: *mono(elab-fun)*
⟨proof⟩

lemma *elab-const*:
c isof ty ==> te |- e-const(c) ===> ty
⟨proof⟩

lemma *elab-var*:
x:te-dom(te) ==> te |- e-var(x) ===> te-app te x
⟨proof⟩

lemma *elab-fn*:

te + {*x* |=> *ty1*} |− *e* ===> *ty2* ==> *te* |− *fn x* => *e* ===> *ty1*−>*ty2*
⟨*proof*⟩

lemma *elab-fix*:
te + {*f* |=> *ty1*−>*ty2*} + {*x* |=> *ty1*} |− *e* ===> *ty2* ==>
te |− *fix f*(*x*) = *e* ===> *ty1*−>*ty2*
⟨*proof*⟩

lemma *elab-app*:
[| *te* |− *e1* ===> *ty1*−>*ty2*; *te* |− *e2* ===> *ty1* |] ==>
te |− *e1* @@ *e2* ===> *ty2*
⟨*proof*⟩

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*))
⟨*proof*⟩

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
⟨*proof*⟩

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*))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *elab-const-elim-lem*:
te |− *e* ===> *t* ==> (*e* = *e-const*(*c*) −−> *c isof t*)
⟨*proof*⟩

lemma *elab-const-elim*: *te* |− *e-const*(*c*) ===> *t* ==> *c isof t*
⟨*proof*⟩

lemma *elab-var-elim-lem*:
te |− e ===> t ==> (e = e-var(x) −−> t=te-app te x & x:te-dom(te))
⟨*proof*⟩

lemma *elab-var-elim*: te |− e-var(ev) ===> t ==> t=te-app te ev & ev : te-dom(te)
⟨*proof*⟩

lemma *elab-fn-elim-lem*:
te |− e ===> t ==>
(e = *fn* $x1$ => $e1$ −−>
(? $t1$ $t2$. t=t-fun $t1$ $t2$ & te + {$x1$ |=> $t1$} |− $e1$ ===> $t2$)
)
⟨*proof*⟩

lemma *elab-fn-elim*: te |− *fn* $x1$ => $e1$ ===> t ==>
(? $t1$ $t2$. t=$t1$−>$t2$ & te + {$x1$ |=> $t1$} |− $e1$ ===> $t2$)
⟨*proof*⟩

lemma *elab-fix-elim-lem*:
te |− e ===> t ==>
(e = *fix* f(x) = $e1$ −−>
(? $t1$ $t2$. t=$t1$−>$t2$ & te + {f |=> $t1$−>$t2$} + {x |=> $t1$} |− $e1$ ===> $t2$))
⟨*proof*⟩

lemma *elab-fix-elim*: te |− *fix* $ev1$($ev2$) = $e1$ ===> t ==>
(? $t1$ $t2$. t=$t1$−>$t2$ & te + {$ev1$ |=> $t1$−>$t2$} + {$ev2$ |=> $t1$} |− $e1$ ===> $t2$)
⟨*proof*⟩

lemma *elab-app-elim-lem*:
te |− e ===> $t2$ ==>
(e = $e1$ @@ $e2$ −−> (? $t1$. te |− $e1$ ===> $t1$−>$t2$ & te |− $e2$ ===> $t1$))
⟨*proof*⟩

lemma *elab-app-elim*: te |− $e1$ @@ $e2$ ===> $t2$ ==> (? $t1$. te |− $e1$ ===> $t1$−>$t2$ & te |− $e2$ ===> $t1$)
⟨*proof*⟩

lemma *mono-hasty-fun*: *mono*(*hasty-fun*)
⟨*proof*⟩

lemma *hasty-rel-const-coind*: *c isof t* ==> (*v-const*(*c*),*t*) : *hasty-rel*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *hasty-const*: *c isof t* ==> *v-const*(*c*) *hasty t*
⟨*proof*⟩

lemma *hasty-clos*:
te |− *fn ev* => *e* ===> *t* & *ve hastyenv te* ==> *v-clos*(<|*ev*,*e*,*ve*|>) *hasty t*

⟨*proof*⟩

lemma *hasty-elim-const-lem*:
v hasty t ==> (!c.(v = v-const(c) --> c isof t))
⟨*proof*⟩

lemma *hasty-elim-const*: *v-const(c) hasty t ==> c isof t*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *hasty-elim-clos*: *v-clos(<|ev,e,ve|>) hasty t ==>*
? te. te |- fn ev => e ===> t & ve hastyenv te
⟨*proof*⟩

lemma *hasty-env1*: *[| ve hastyenv te; v hasty t |] ==>*
ve + {ev |-> v} hastyenv te + {ev |=> t}
⟨*proof*⟩

lemma *consistency-const*: *[| ve hastyenv te ; te |- e-const(c) ===> t |] ==> v-const(c) hasty t*
⟨*proof*⟩

lemma *consistency-var*:
[| ev : ve-dom(ve); ve hastyenv te ; te |- e-var(ev) ===> t |] ==>
ve-app ve ev hasty t
⟨*proof*⟩

lemma *consistency-fn*: *[| ve hastyenv te ; te |- fn ev => e ===> t |] ==>*
v-clos(<| ev, e, ve |>) hasty t
⟨*proof*⟩

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
⟨proof⟩

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
⟨proof⟩

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
⟨proof⟩

lemma *consistency*: ve |- e ---> v ==>
(! t te. ve hastyenv te --> te |- e ===> t --> v hasty t)

⟨proof⟩

lemma *basic-consistency-lem*:
ve isofenv te ==> ve hastyenv te
⟨proof⟩

lemma *basic-consistency*:
[| ve isofenv te; ve |- e ---> v-const(c); te |- e ===> t |] ==> c isof t
⟨proof⟩

end

30 Case study: Unification Algorithm

theory *Unification*
imports *Main*
begin

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

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

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

30.1 Basic definitions

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

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

Applying a substitution to a variable:

fun $assoc :: 'a \Rightarrow 'b \Rightarrow ('a \times 'b)\ list \Rightarrow 'b$
where
  $assoc\ x\ d\ [] = d$
$|\ assoc\ x\ d\ ((p,q)\#t) = (if\ x = p\ then\ q\ else\ assoc\ x\ d\ t)$

Applying a substitution to a term:

fun $apply\text{-}subst :: 'a\ trm \Rightarrow 'a\ subst \Rightarrow 'a\ trm$ (**infixl** $\triangleleft$ *60*)
where
  $(Var\ v) \triangleleft s = assoc\ v\ (Var\ v)\ s$
$|\ (Const\ c) \triangleleft s = (Const\ c)$
$|\ (M \cdot N) \triangleleft s = (M \triangleleft s) \cdot (N \triangleleft s)$

Composition of substitutions:

fun
  $compose :: 'a\ subst \Rightarrow 'a\ subst \Rightarrow 'a\ subst$ (**infixl** $\bullet$ *80*)
where
  $[] \bullet bl = bl$

| $((a,b) \# al) \bullet bl = (a, b \lhd bl) \# (al \bullet bl)$

Equivalence of substitutions:

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

30.2 Basic lemmas

lemma *apply-empty*[*simp*]: $t \lhd [] = t$
⟨*proof*⟩

lemma *compose-empty*[*simp*]: $\sigma \bullet [] = \sigma$
⟨*proof*⟩

lemma *apply-compose*[*simp*]: $t \lhd (s1 \bullet s2) = t \lhd s1 \lhd s2$
⟨*proof*⟩

lemma *eqv-refl*[*intro*]: $s =_s s$
⟨*proof*⟩

lemma *eqv-trans*[*trans*]: $[\![s1 =_s s2; s2 =_s s3]\!] \Longrightarrow s1 =_s s3$
⟨*proof*⟩

lemma *eqv-sym*[*sym*]: $[\![s1 =_s s2]\!] \Longrightarrow s2 =_s s1$
⟨*proof*⟩

lemma *eqv-intro*[*intro*]: $(\bigwedge t.\ t \lhd \sigma = t \lhd \vartheta) \Longrightarrow \sigma =_s \vartheta$
⟨*proof*⟩

lemma *eqv-dest*[*dest*]: $s1 =_s s2 \Longrightarrow t \lhd s1 = t \lhd s2$
⟨*proof*⟩

lemma *compose-eqv*: $[\![\sigma =_s \sigma'; \vartheta =_s \vartheta']\!] \Longrightarrow (\sigma \bullet \vartheta) =_s (\sigma' \bullet \vartheta')$
⟨*proof*⟩

lemma *compose-assoc*: $(a \bullet b) \bullet c =_s a \bullet (b \bullet c)$
⟨*proof*⟩

30.3 Specification: Most general unifiers

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

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

lemma *MGUI*[*intro*]:

$[\![t \lhd \sigma = u \lhd \sigma; \bigwedge\vartheta.\ t \lhd \vartheta = u \lhd \vartheta \Longrightarrow \exists\gamma.\ \vartheta =_s \sigma \bullet \gamma]\!]$
$\Longrightarrow MGU\ \sigma\ t\ u$
$\langle proof \rangle$

lemma *MGU-sym*[*sym*]:
$MGU\ \sigma\ s\ t \Longrightarrow MGU\ \sigma\ t\ s$
$\langle proof \rangle$

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* ($M \cdot N$) = ($u = M \lor u = N \lor$ *occ u M* $\lor$ *occ u N*)

The unification algorithm:

function *unify* :: *'a trm* ⇒ *'a trm* ⇒ *'a subst option*
where
unify (*Const c*) ($M \cdot N$) = *None*
| *unify* ($M \cdot N$) (*Const c*) = *None*
| *unify* (*Const c*) (*Var v*) = *Some* [(*v*, *Const c*)]
| *unify* ($M \cdot N$) (*Var v*) = (*if* (*occ* (*Var v*) ($M \cdot N$))
then None
else Some [(*v*, $M \cdot N$)])
| *unify* (*Var v*) *M* = (*if* (*occ* (*Var v*) *M*)
then None
else Some [(*v*, *M*)])
| *unify* (*Const c*) (*Const d*) = (*if c=d then Some* [] *else None*)
| *unify* ($M \cdot N$) ($M' \cdot N'$) = (*case unify M M' of*
None ⇒ *None* |
Some ϑ ⇒ (*case unify* ($N \lhd \vartheta$) ($N' \lhd \vartheta$)
of None ⇒ *None* |
Some σ ⇒ *Some* ($\vartheta \bullet \sigma$)))
$\langle proof \rangle$

30.5 Partial correctness

Some lemmas about occ and MGU:

lemma *subst-no-occ*: $\neg$*occ* (*Var v*) $t \Longrightarrow$ *Var v* $\neq t$
$\Longrightarrow t \lhd [(v,s)] = t$
$\langle proof \rangle$

lemma *MGU-Var*[*intro*]:
assumes *no-occ*: $\neg$*occ* (*Var v*) *t*
shows *MGU* [(*v*,*t*)] (*Var v*) *t*
$\langle proof \rangle$

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

lemma $MGU\text{-}Const[simp]$: $MGU\ []\ (Const\ c)\ (Const\ d) = (c = d)$
⟨*proof*⟩

If unification terminates, then it computes most general unifiers:

lemma *unify-partial-correctness*:
assumes $unify\text{-}dom\ (M,\ N)$
assumes $unify\ M\ N = Some\ \sigma$
shows $MGU\ \sigma\ M\ N$
⟨*proof*⟩

30.6 Properties used in termination proof

The variables of a term:

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

lemma $vars\text{-}of\text{-}finite[intro]$: $finite\ (vars\text{-}of\ t)$
⟨*proof*⟩

Elimination of variables by a substitution:

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

lemma $elim\text{-}intro[intro]$: $(\bigwedge t.\ v \notin vars\text{-}of\ (t \lhd \sigma)) \Longrightarrow elim\ \sigma\ v$
⟨*proof*⟩

lemma $elim\text{-}dest[dest]$: $elim\ \sigma\ v \Longrightarrow v \notin vars\text{-}of\ (t \lhd \sigma)$
⟨*proof*⟩

lemma $elim\text{-}eqv$: $\sigma =_s \vartheta \Longrightarrow elim\ \sigma\ x = elim\ \vartheta\ x$
⟨*proof*⟩

Replacing a variable by itself yields an identity subtitution:

lemma $var\text{-}self[intro]$: $[(v,\ Var\ v)] =_s []$
⟨*proof*⟩

lemma $var\text{-}same$: $(t = Var\ v) = ([(v,\ t)] =_s [])$
⟨*proof*⟩

A lemma about occ and elim

lemma *remove-var*:
assumes $[simp]$: $v \notin vars\text{-}of\ s$

shows $v \notin$ *vars-of* $(t \lhd [(v, s)])$
⟨*proof*⟩

lemma *occ-elim*: $\neg$*occ* (*Var* v) t
$\Longrightarrow$ *elim* $[(v,t)]$ $v \vee [(v,t)] =_s []$
⟨*proof*⟩

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* $(t \lhd \sigma) \subseteq$ *vars-of* $M \cup$ *vars-of* $N \cup$ *vars-of* t
(**is** *?P* M N σ t)
⟨*proof*⟩

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 ($\exists$ $v \in$*vars-of* $M \cup$ *vars-of* N. *elim* σ v) $\vee$ $\sigma =_s []$
(**is** *?P* M N σ)
⟨*proof*⟩

30.7 Termination proof

termination *unify*
⟨*proof*⟩

end

31 Some examples demonstrating the comm-ring method

theory *Commutative-RingEx*
imports *Commutative-Ring*
begin

lemma $4*(x::int)\hat{}5*y\hat{}3*x\hat{}2*3 + x*z + 3\hat{}5 = 12*x\hat{}7*y\hat{}3 + z*x + 243$
⟨*proof*⟩

lemma $((x::int) + y)\hat{}2 = x\hat{}2 + y\hat{}2 + 2*x*y$
⟨*proof*⟩

lemma $((x::int) + y)\hat{}3 = x\hat{}3 + y\hat{}3 + 3*x\hat{}2*y + 3*y\hat{}2*x$
⟨*proof*⟩

lemma $((x{::}int) - y)\hat{\ }3 = x\hat{\ }3 + 3{*}x{*}y\hat{\ }2 + (-3){*}y{*}x\hat{\ }2 - y\hat{\ }3$
$\langle proof \rangle$

lemma $((x{::}int) - y)\hat{\ }2 = x\hat{\ }2 + y\hat{\ }2 - 2{*}x{*}y$
$\langle proof \rangle$

lemma $((a{::}int) + b + c)\hat{\ }2 = a\hat{\ }2 + b\hat{\ }2 + c\hat{\ }2 + 2{*}a{*}b + 2{*}b{*}c + 2{*}a{*}c$
$\langle proof \rangle$

lemma $((a{::}int) - b - c)\hat{\ }2 = a\hat{\ }2 + b\hat{\ }2 + c\hat{\ }2 - 2{*}a{*}b + 2{*}b{*}c - 2{*}a{*}c$
$\langle proof \rangle$

lemma $(a{::}int){*}b + a{*}c = a{*}(b{+}c)$
$\langle proof \rangle$

lemma $(a{::}int)\hat{\ }2 - b\hat{\ }2 = (a - b) * (a + b)$
$\langle proof \rangle$

lemma $(a{::}int)\hat{\ }3 - b\hat{\ }3 = (a - b) * (a\hat{\ }2 + a{*}b + b\hat{\ }2)$
$\langle proof \rangle$

lemma $(a{::}int)\hat{\ }3 + b\hat{\ }3 = (a + b) * (a\hat{\ }2 - a{*}b + b\hat{\ }2)$
$\langle proof \rangle$

lemma $(a{::}int)\hat{\ }4 - b\hat{\ }4 = (a - b) * (a + b){*}(a\hat{\ }2 + b\hat{\ }2)$
$\langle proof \rangle$

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

end

32 Small examples for evaluation mechanisms

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

evaluation oracle

lemma $True \vee False$ $\langle proof \rangle$
lemma $\neg (Suc\ 0 = Suc\ 1)$ $\langle proof \rangle$
lemma $[] = ([]{::}\ int\ list)$ $\langle proof \rangle$
lemma $[()] = [()]$ $\langle proof \rangle$
lemma $fst\ ([]{::}nat\ list,\ Suc\ 0) = []$ $\langle proof \rangle$

SML evaluation oracle

lemma $True \vee False$ $\langle proof \rangle$

lemma $\neg$ (*Suc 0* = *Suc 1*) ⟨*proof*⟩
lemma [] = ([]:: *int list*) ⟨*proof*⟩
lemma [()] = [()] ⟨*proof*⟩
lemma *fst* ([]::*nat list*, *Suc 0*) = [] ⟨*proof*⟩

normalization

lemma *True* $\vee$ *False* ⟨*proof*⟩
lemma $\neg$ (*Suc 0* = *Suc 1*) ⟨*proof*⟩
lemma [] = ([]:: *int list*) ⟨*proof*⟩
lemma [()] = [()] ⟨*proof*⟩
lemma *fst* ([]::*nat list*, *Suc 0*) = [] ⟨*proof*⟩

term evaluation

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

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

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

value (*10*::*int*) $\leq$ *12*
value (*code*) (*10*::*int*) $\leq$ *12*
value (*SML*) (*10*::*int*) $\leq$ *12*
value (*normal-form*) (*10*::*int*) $\leq$ *12*

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

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

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

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

value [(*nat 100*, ())]

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

a fancy datatype

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

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

end

33 A simple random engine

theory *Random*
imports *State-Monad Code-Integer*
begin

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

typedecl *randseed*

axiomatization
  random-shift :: *randseed* ⇒ *randseed*

axiomatization
  random-seed :: *randseed* ⇒ *nat*

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

lemma *random-bound*:
  assumes *0* < *n*
  shows *fst* (*random n s*) < *n*
⟨*proof*⟩

lemma *random-random-seed* [*simp*]:
snd (*random n s*) = *random-shift s* ⟨*proof*⟩

definition
select :: ′*a list* ⇒ *randseed* ⇒ ′*a* × *randseed* **where**
[*simp*]: *select xs* = (*do*
n ← *random* (*length xs*);
return (*nth xs n*)
done)

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

lemma
select (*x*#*xs*) *s* = *select-weight* (*map* (*Pair 1*) (*x*#*xs*)) *s*
⟨*proof*⟩

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

lemma *random-nat* [*code*]:
random n = (*do k* ← *random-int* (*int n*); *return* (*nat k*) *done*)
⟨*proof*⟩

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

⟨*ML*⟩

code-reserved *SML Random*

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

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

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

end

34 Primitive Recursive Functions

theory *Primrec* **imports** *Main* **begin**

Proof adopted from

Nora Szasz, A Machine Checked Proof that Ackermann's Function is not Primitive Recursive, In: Huet & Plotkin, eds., Logical Environments (CUP, 1993), 317-338.

See also E. Mendelson, Introduction to Mathematical Logic. (Van Nostrand, 1964), page 250, exercise 11.

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

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

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

The set of primitive recursive functions of type *nat list* ⇒ *nat*.

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

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

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

definition
  COMP :: (*nat list* => *nat*) => (*nat list* => *nat*) *list* => *nat list* => *nat* **where**
  COMP g fs l = *g* (*map* (λ*f*. *f l*) *fs*)

definition
  PREC :: (*nat list* => *nat*) => (*nat list* => *nat*) => *nat list* => *nat* **where**

$PREC\ f\ g\ l =$
$(case\ l\ of$
$[] => 0$
$|\ x \# l' => nat\text{-}rec\ (f\ l')\ (\lambda y\ r.\ g\ (r \# y \# l'))\ x)$
— Note that g is applied first to $PREC\ f\ g\ y$ and then to y!

inductive $PRIMREC :: (nat\ list => nat) => bool$
where
$SC: PRIMREC\ SC$
$|\ CONSTANT: PRIMREC\ (CONSTANT\ k)$
$|\ PROJ: PRIMREC\ (PROJ\ i)$
$|\ COMP: PRIMREC\ g ==> \forall f \in set\ fs.\ PRIMREC\ f ==> PRIMREC\ (COMP\ g\ fs)$
$|\ PREC: PRIMREC\ f ==> PRIMREC\ g ==> PRIMREC\ (PREC\ f\ g)$

Useful special cases of evaluation

lemma SC $[simp]$: $SC\ (x \# l) = Suc\ x$
$\langle proof \rangle$

lemma $CONSTANT$ $[simp]$: $CONSTANT\ k\ l = k$
$\langle proof \rangle$

lemma $PROJ\text{-}0$ $[simp]$: $PROJ\ 0\ (x \# l) = x$
$\langle proof \rangle$

lemma $COMP\text{-}1$ $[simp]$: $COMP\ g\ [f]\ l = g\ [f\ l]$
$\langle proof \rangle$

lemma $PREC\text{-}0$ $[simp]$: $PREC\ f\ g\ (0 \# l) = f\ l$
$\langle proof \rangle$

lemma $PREC\text{-}Suc$ $[simp]$: $PREC\ f\ g\ (Suc\ x \# l) = g\ (PREC\ f\ g\ (x \# l) \# x \# l)$
$\langle proof \rangle$

PROPERTY A 4

lemma $less\text{-}ack2$ $[iff]$: $j < ack\ (i, j)$
$\langle proof \rangle$

PROPERTY A 5-, the single-step lemma

lemma $ack\text{-}less\text{-}ack\text{-}Suc2$ $[iff]$: $ack(i, j) < ack\ (i, Suc\ j)$
$\langle proof \rangle$

PROPERTY A 5, monotonicity for $<$

lemma $ack\text{-}less\text{-}mono2$: $j < k ==> ack\ (i, j) < ack\ (i, k)$
$\langle proof \rangle$

PROPERTY A 5', monotonicity for $\leq$

lemma *ack-le-mono2*: $j \le k \Longrightarrow ack\ (i, j) \le ack\ (i, k)$
⟨*proof*⟩

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: $ack\ (i, Suc\ j) \le ack\ (Suc\ i, j)$
⟨*proof*⟩

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: $ack\ (i, j) < ack\ (Suc\ i, j)$
⟨*proof*⟩

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

lemma *less-ack1* [*iff*]: $i < ack\ (i, j)$
⟨*proof*⟩

PROPERTY A 8

lemma *ack-1* [*simp*]: $ack\ (Suc\ 0, j) = j + 2$
⟨*proof*⟩

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

lemma *ack-2* [*simp*]: $ack\ (Suc\ (Suc\ 0), j) = 2 * j + 3$
⟨*proof*⟩

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

lemma *ack-less-mono1-aux*: $ack\ (i, k) < ack\ (Suc\ (i + i'), k)$
⟨*proof*⟩

lemma *ack-less-mono1*: $i < j \Longrightarrow ack\ (i, k) < ack\ (j, k)$
⟨*proof*⟩

PROPERTY A 7', monotonicity for ≤

lemma *ack-le-mono1*: $i \le j \Longrightarrow ack\ (i, k) \le ack\ (j, k)$
⟨*proof*⟩

PROPERTY A 10

lemma *ack-nest-bound*: $ack(i1, ack\ (i2, j)) < ack\ (2 + (i1 + i2), j)$
⟨*proof*⟩

PROPERTY A 11

lemma *ack-add-bound*: $ack\ (i1, j) + ack\ (i2, j) < ack\ (4 + (i1 + i2), j)$
⟨*proof*⟩

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*: $i < ack\ (k, j) \Longrightarrow i + j < ack\ (4 + k, j)$
⟨*proof*⟩

Inductive definition of the *PR* functions

MAIN RESULT

lemma *SC-case*: $SC\ l < ack\ (1, list\text{-}add\ l)$
⟨*proof*⟩

lemma *CONSTANT-case*: $CONSTANT\ k\ l < ack\ (k, list\text{-}add\ l)$
⟨*proof*⟩

lemma *PROJ-case* [*rule-format*]: $\forall i.\ PROJ\ i\ l < ack\ (0, list\text{-}add\ l)$
⟨*proof*⟩

COMP case

lemma *COMP-map-aux*: $\forall f \in set\ fs.\ PRIMREC\ f \wedge (\exists kf.\ \forall l.\ f\ l < ack\ (kf, list\text{-}add\ l))$
$\Longrightarrow \exists k.\ \forall l.\ list\text{-}add\ (map\ (\lambda f.\ f\ l)\ fs) < ack\ (k, list\text{-}add\ l)$
⟨*proof*⟩

lemma *COMP-case*:
$\forall l.\ g\ l < ack\ (kg, list\text{-}add\ l) \Longrightarrow$
$\forall f \in set\ fs.\ PRIMREC\ f \wedge (\exists kf.\ \forall l.\ f\ l < ack(kf, list\text{-}add\ l))$
$\Longrightarrow \exists k.\ \forall l.\ COMP\ g\ fs\ \ l < ack(k, list\text{-}add\ l)$
⟨*proof*⟩

PREC case

lemma *PREC-case-aux*:
$\forall l.\ f\ l + list\text{-}add\ l < ack\ (kf, list\text{-}add\ l) \Longrightarrow$
$\forall l.\ g\ l + list\text{-}add\ l < ack\ (kg, list\text{-}add\ l) \Longrightarrow$
$PREC\ f\ g\ l + list\text{-}add\ l < ack\ (Suc\ (kf + kg), list\text{-}add\ l)$
⟨*proof*⟩

lemma *PREC-case*:
$\forall l.\ f\ l < ack\ (kf, list\text{-}add\ l) \Longrightarrow$
$\forall l.\ g\ l < ack\ (kg, list\text{-}add\ l) \Longrightarrow$
$\exists k.\ \forall l.\ PREC\ f\ g\ l < ack\ (k, list\text{-}add\ l)$
⟨*proof*⟩

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

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

end

35 The Full Theorem of Tarski

theory *Tarski* **imports** *Main FuncSet* **begin**

Minimal version of lattice theory plus the full theorem of Tarski: The fixed-points of a complete lattice themselves form a complete lattice.

Illustrates first-class theories, using the Sigma representation of structures. Tidied and converted to Isar by lcp.

```
record 'a potype =
  pset  :: 'a set
  order :: ('a * 'a) set

definition
  monotone :: ['a => 'a, 'a set, ('a *'a)set] => bool where
  monotone f A r = (∀ x∈A. ∀ y∈A. (x, y): r --> ((f x), (f y)) : r)

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

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

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

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

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

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

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

definition
```

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

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

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

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

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

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

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

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

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

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

locale (**open**) *PO* =

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

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

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

locale (**open**) *Tarski* = *CLF* +
fixes *Y* :: *'a set*
and *intY1* :: *'a set*
and *v* :: *'a*
assumes
Y-ss: *Y* ⊆ *P*
defines
intY1-def: *intY1* == *interval r (lub Y cl) (Top cl)*
and *v-def*: *v* == *glb {x. ((%x: intY1. f x) x, x): induced intY1 r &*
x: intY1}
(| pset=intY1, order=induced intY1 r|)

35.1 Partial Order

lemma (**in** *PO*) *PO-imp-refl*: *refl A r*
⟨*proof*⟩

lemma (**in** *PO*) *PO-imp-sym*: *antisym r*
⟨*proof*⟩

lemma (**in** *PO*) *PO-imp-trans*: *trans r*
⟨*proof*⟩

lemma (**in** *PO*) *reflE*: *x* ∈ *A* ==> *(x, x)* ∈ *r*
⟨*proof*⟩

lemma (**in** *PO*) *antisymE*: [| *(a, b)* ∈ *r*; *(b, a)* ∈ *r* |] ==> *a* = *b*
⟨*proof*⟩

lemma (**in** *PO*) *transE*: [| *(a, b)* ∈ *r*; *(b, c)* ∈ *r*|] ==> *(a,c)* ∈ *r*
⟨*proof*⟩

lemma (**in** *PO*) *monotoneE*:
  $[|$ *monotone f A r*; $x \in A$; $y \in A$; $(x,\ y) \in r\ |] ==> (f\,x,\ f\,y) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *po-subset-po*:
  $S \subseteq A ==> (|\ pset = S,\ order = induced\ S\ r\ |) \in PartialOrder$
⟨*proof*⟩

lemma (**in** *PO*) *indE*: $[|\ (x,\ y) \in induced\ S\ r;\ S \subseteq A\ |] ==> (x,\ y) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *indI*: $[|\ (x,\ y) \in r;\ x \in S;\ y \in S\ |] ==> (x,\ y) \in induced\ S\ r$
⟨*proof*⟩

lemma (**in** *CL*) *CL-imp-ex-isLub*: $S \subseteq A ==> \exists\,L.\ isLub\ S\ cl\ L$
⟨*proof*⟩

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

lemma *isLub-lub*: $(\exists\,L.\ isLub\ S\ cl\ L) = isLub\ S\ cl\ (lub\ S\ cl)$
⟨*proof*⟩

lemma *isGlb-glb*: $(\exists\,G.\ isGlb\ S\ cl\ G) = isGlb\ S\ cl\ (glb\ S\ cl)$
⟨*proof*⟩

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

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

lemma (**in** *PO*) *dualPO*: $dual\ cl \in PartialOrder$
⟨*proof*⟩

lemma *Rdual*:
  $\forall\,S.\ (S \subseteq A\ -->(\ \exists\,L.\ isLub\ S\ (|\ pset = A,\ order = r|)\ L))$
  $==> \forall\,S.\ (S \subseteq A\ --> (\exists\,G.\ isGlb\ S\ (|\ pset = A,\ order = r|)\ G))$
⟨*proof*⟩

lemma *lub-dual-glb*: $lub\ S\ cl = glb\ S\ (dual\ cl)$
⟨*proof*⟩

lemma *glb-dual-lub*: $glb\ S\ cl = lub\ S\ (dual\ cl)$
⟨*proof*⟩

lemma *CL-subset-PO*: $CompleteLattice \subseteq PartialOrder$
⟨*proof*⟩

lemmas *CL-imp-PO* = *CL-subset-PO* [*THEN subsetD*]

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

lemma (**in** *CL*) *CO-refl*: *refl A r*
⟨*proof*⟩

lemma (**in** *CL*) *CO-antisym*: *antisym r*
⟨*proof*⟩

lemma (**in** *CL*) *CO-trans*: *trans r*
⟨*proof*⟩

lemma *CompleteLatticeI*:
  [| *po* ∈ *PartialOrder*; (∀ *S*. *S* ⊆ *pset po* −−> (∃ *L*. *isLub S po L*));
    (∀ *S*. *S* ⊆ *pset po* −−> (∃ *G*. *isGlb S po G*))|]
  ==> *po* ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *CL*) *CL-dualCL*: *dual cl* ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *PO*) *dualA-iff*: *pset* (*dual cl*) = *pset cl*
⟨*proof*⟩

lemma (**in** *PO*) *dualr-iff*: ((*x*, *y*) ∈ (*order*(*dual cl*))) = ((*y*, *x*) ∈ *order cl*)
⟨*proof*⟩

lemma (**in** *PO*) *monotone-dual*:
  monotone f (*pset cl*) (*order cl*)
  ==> *monotone f* (*pset* (*dual cl*)) (*order*(*dual cl*))
⟨*proof*⟩

lemma (**in** *PO*) *interval-dual*:
  [| *x* ∈ *A*; *y* ∈ *A*|] ==> *interval r x y* = *interval* (*order*(*dual cl*)) *y x*
⟨*proof*⟩

lemma (**in** *PO*) *interval-not-empty*:
  [| *trans r*; *interval r a b* ≠ {} |] ==> (*a*, *b*) ∈ *r*
⟨*proof*⟩

lemma (**in** *PO*) *interval-imp-mem*: *x* ∈ *interval r a b* ==> (*a*, *x*) ∈ *r*
⟨*proof*⟩

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

lemma (**in** *PO*) *right-in-interval*:
$[| \ a \in A;\ b \in A;\ interval\ r\ a\ b \neq \{\}\ |] ==> b \in interval\ r\ a\ b$
$\langle proof \rangle$

35.2 sublattice

lemma (**in** *PO*) *sublattice-imp-CL*:
$S <<= cl \ ==> (|\ pset = S,\ order = induced\ S\ r\ |) \in CompleteLattice$
$\langle proof \rangle$

lemma (**in** *CL*) *sublatticeI*:
$[| \ S \subseteq A;\ (|\ pset = S,\ order = induced\ S\ r\ |) \in CompleteLattice\ |]$
$==> S <<= cl$
$\langle proof \rangle$

35.3 lub

lemma (**in** *CL*) *lub-unique*: $[| \ S \subseteq A;\ isLub\ S\ cl\ x;\ isLub\ S\ cl\ L|] ==> x = L$
$\langle proof \rangle$

lemma (**in** *CL*) *lub-upper*: $[|S \subseteq A;\ x \in S|] ==> (x,\ lub\ S\ cl) \in r$
$\langle proof \rangle$

lemma (**in** *CL*) *lub-least*:
$[| \ S \subseteq A;\ L \in A;\ \forall x \in S.\ (x,L) \in r\ |] ==> (lub\ S\ cl,\ L) \in r$
$\langle proof \rangle$

lemma (**in** *CL*) *lub-in-lattice*: $S \subseteq A ==> lub\ S\ cl \in A$
$\langle proof \rangle$

lemma (**in** *CL*) *lubI*:
$[| \ S \subseteq A;\ L \in A;\ \forall x \in S.\ (x,L) \in r;$
$\forall z \in A.\ (\forall y \in S.\ (y,z) \in r) --> (L,z) \in r\ |] ==> L = lub\ S\ cl$
$\langle proof \rangle$

lemma (**in** *CL*) *lubIa*: $[| \ S \subseteq A;\ isLub\ S\ cl\ L\ |] ==> L = lub\ S\ cl$
$\langle proof \rangle$

lemma (**in** *CL*) *isLub-in-lattice*: $isLub\ S\ cl\ L ==> L \in A$
$\langle proof \rangle$

lemma (**in** *CL*) *isLub-upper*: $[|isLub\ S\ cl\ L;\ y \in S|] ==> (y,\ L) \in r$
$\langle proof \rangle$

lemma (**in** *CL*) *isLub-least*:
$[| \ isLub\ S\ cl\ L;\ z \in A;\ \forall y \in S.\ (y,\ z) \in r|] ==> (L,\ z) \in r$
$\langle proof \rangle$

lemma (**in** *CL*) *isLubI*:
$[| \ L \in A;\ \forall y \in S.\ (y,\ L) \in r;$

$(\forall z \in A.\ (\forall y \in S.\ (y,\ z){:}r) \ {-}{-}{>}\ (L,\ z) \in r)|] \ ==>\ isLub\ S\ cl\ L$
⟨*proof*⟩

35.4 glb

lemma (**in** *CL*) *glb-in-lattice*: $S \subseteq A \ ==>\ glb\ S\ cl \in A$
⟨*proof*⟩

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

Reduce the sublattice property by using substructural properties; abandoned see *Tarski-4.ML*.

lemma (**in** *CLF*) [*simp*]:
f: $pset\ cl\ {-}{>}\ pset\ cl\ \&\ monotone\ f\ (pset\ cl)\ (order\ cl)$
⟨*proof*⟩

declare (**in** *CLF*) *f-cl* [*simp*]

lemma (**in** *CLF*) *f-in-funcset*: $f \in A\ {-}{>}\ A$
⟨*proof*⟩

lemma (**in** *CLF*) *monotone-f*: $monotone\ f\ A\ r$
⟨*proof*⟩

lemma (**in** *CLF*) *CLF-dual*: $(dual\ cl,\ f) \in CLF$
⟨*proof*⟩

35.5 fixed points

lemma *fix-subset*: $fix\ f\ A \subseteq A$
⟨*proof*⟩

lemma *fix-imp-eq*: $x \in fix\ f\ A \ ==>\ f\ x = x$
⟨*proof*⟩

lemma *fixf-subset*:
$[|\ A \subseteq B;\ x \in fix\ (\%y{:}\ A.\ f\ y)\ A\ |] \ ==>\ x \in fix\ f\ B$
⟨*proof*⟩

35.6 lemmas for Tarski, lub

lemma (**in** *CLF*) *lubH-le-flubH*:
$H = \{x.\ (x,\ f\ x) \in r\ \&\ x \in A\} \ ==>\ (lub\ H\ cl,\ f\ (lub\ H\ cl)) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *flubH-le-lubH*:
$[|\ \ H = \{x.\ (x,\ f\ x) \in r\ \&\ x \in A\}\ |] \ ==>\ (f\ (lub\ H\ cl),\ lub\ H\ cl) \in r$

⟨*proof*⟩

lemma (**in** *CLF*) *lubH-is-fixp*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} ==> lub\ H\ cl \in fix\ f\ A$
⟨*proof*⟩

lemma (**in** *CLF*) *fix-in-H*:
  $[|\ H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\};\ \ x \in P\ |] ==> x \in H$
⟨*proof*⟩

lemma (**in** *CLF*) *fixf-le-lubH*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} ==> \forall x \in fix\ f\ A.\ (x,\ lub\ H\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *lubH-least-fixf*:
  $H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}$
  $==> \forall L.\ (\forall y \in fix\ f\ A.\ (y,L) \in r) --> (lub\ H\ cl,\ L) \in r$
⟨*proof*⟩

35.7 Tarski fixpoint theorem 1, first part

lemma (**in** *CLF*) *T-thm-1-lub*: $lub\ P\ cl = lub\ \{x.\ (x, f\ x) \in r\ \&\ x \in A\}\ cl$
⟨*proof*⟩

lemma (**in** *CLF*) *glbH-is-fixp*: $H = \{x.\ (f\ x,\ x) \in r\ \&\ x \in A\} ==> glb\ H\ cl \in P$
— Tarski for glb
⟨*proof*⟩

lemma (**in** *CLF*) *T-thm-1-glb*: $glb\ P\ cl = glb\ \{x.\ (f\ x,\ x) \in r\ \&\ x \in A\}\ cl$
⟨*proof*⟩

35.8 interval

lemma (**in** *CLF*) *rel-imp-elem*: $(x,\ y) \in r ==> x \in A$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-subset*: $[|\ a \in A;\ b \in A\ |] ==> interval\ r\ a\ b \subseteq A$
⟨*proof*⟩

lemma (**in** *CLF*) *intervalI*:
  $[|\ (a,\ x) \in r;\ (x,\ b) \in r\ |] ==> x \in interval\ r\ a\ b$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-lemma1*:
  $[|\ S \subseteq interval\ r\ a\ b;\ x \in S\ |] ==> (a,\ x) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-lemma2*:
  $[|\ S \subseteq interval\ r\ a\ b;\ x \in S\ |] ==> (x,\ b) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *a-less-lub*:
$[| S \subseteq A; S \neq \{\};$
$\forall x \in S. (a,x) \in r; \forall y \in S. (y, L) \in r |] ==> (a,L) \in r$
$\langle proof \rangle$

lemma (**in** *CLF*) *glb-less-b*:
$[| S \subseteq A; S \neq \{\};$
$\forall x \in S. (x,b) \in r; \forall y \in S. (G, y) \in r |] ==> (G,b) \in r$
$\langle proof \rangle$

lemma (**in** *CLF*) *S-intv-cl*:
$[| a \in A; b \in A; S \subseteq interval\ r\ a\ b\ |]==> S \subseteq A$
$\langle proof \rangle$

lemma (**in** *CLF*) *L-in-interval*:
$[| a \in A; b \in A; S \subseteq interval\ r\ a\ b;$
$S \neq \{\}; isLub\ S\ cl\ L; interval\ r\ a\ b \neq \{\}\ |] ==> L \in interval\ r\ a\ b$
$\langle proof \rangle$

lemma (**in** *CLF*) *G-in-interval*:
$[| a \in A; b \in A; interval\ r\ a\ b \neq \{\}; S \subseteq interval\ r\ a\ b; isGlb\ S\ cl\ G;$
$S \neq \{\}\ |] ==> G \in interval\ r\ a\ b$
$\langle proof \rangle$

lemma (**in** *CLF*) *intervalPO*:
$[| a \in A; b \in A; interval\ r\ a\ b \neq \{\}\ |]$
$==> (| pset = interval\ r\ a\ b, order = induced\ (interval\ r\ a\ b)\ r\ |)$
$\in PartialOrder$
$\langle proof \rangle$

lemma (**in** *CLF*) *intv-CL-lub*:
$[| a \in A; b \in A; interval\ r\ a\ b \neq \{\}\ |]$
$==> \forall S. S \subseteq interval\ r\ a\ b\ -->$
$(\exists L. isLub\ S\ (| pset = interval\ r\ a\ b,$
$order = induced\ (interval\ r\ a\ b)\ r\ |)\ \ L)$
$\langle proof \rangle$

lemmas (**in** *CLF*) *intv-CL-glb* = *intv-CL-lub* [*THEN Rdual*]

lemma (**in** *CLF*) *interval-is-sublattice*:
$[| a \in A; b \in A; interval\ r\ a\ b \neq \{\}\ |]$
$==> interval\ r\ a\ b <<= cl$
$\langle proof \rangle$

lemmas (**in** *CLF*) *interv-is-compl-latt* =
interval-is-sublattice [*THEN sublattice-imp-CL*]

35.9 Top and Bottom

lemma (**in** *CLF*) *Top-dual-Bot*: *Top cl* = *Bot* (*dual cl*)
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-dual-Top*: *Bot cl* = *Top* (*dual cl*)
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-in-lattice*: *Bot cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *CLF*) *Top-in-lattice*: *Top cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *CLF*) *Top-prop*: *x* ∈ *A* ==> (*x*, *Top cl*) ∈ *r*
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-prop*: *x* ∈ *A* ==> (*Bot cl*, *x*) ∈ *r*
⟨*proof*⟩

lemma (**in** *CLF*) *Top-intv-not-empty*: *x* ∈ *A* ==> *interval r x* (*Top cl*) ≠ {}
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-intv-not-empty*: *x* ∈ *A* ==> *interval r* (*Bot cl*) *x* ≠ {}
⟨*proof*⟩

35.10 fixed points form a partial order

lemma (**in** *CLF*) *fixf-po*: (| *pset* = *P*, *order* = *induced P r*|) ∈ *PartialOrder*
⟨*proof*⟩

lemma (**in** *Tarski*) *Y-subset-A*: *Y* ⊆ *A*
⟨*proof*⟩

lemma (**in** *Tarski*) *lubY-in-A*: *lub Y cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *Tarski*) *lubY-le-flubY*: (*lub Y cl*, *f* (*lub Y cl*)) ∈ *r*
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-subset*: *intY1* ⊆ *A*
⟨*proof*⟩

lemmas (**in** *Tarski*) *intY1-elem* = *intY1-subset* [*THEN subsetD*]

lemma (**in** *Tarski*) *intY1-f-closed*: *x* ∈ *intY1* ⟹ *f x* ∈ *intY1*
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-func*: (%*x*: *intY1*. *f x*) ∈ *intY1* −> *intY1*
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-mono*:
monotone (%x: *intY1*. $f\ x$) *intY1* (*induced intY1 r*)
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-is-cl*:
(| *pset* = *intY1*, *order* = *induced intY1 r* |) ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *Tarski*) *v-in-P*: $v \in P$
⟨*proof*⟩

lemma (**in** *Tarski*) *z-in-interval*:
[| $z \in P$; $\forall\, y \in Y.\ (y,\ z) \in$ *induced* $P\ r$ |] ==> $z \in$ *intY1*
⟨*proof*⟩

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) ∈ *induced intY1 r*
⟨*proof*⟩

lemma (**in** *Tarski*) *tarski-full-lemma*:
$\exists\, L$. *isLub* Y (| *pset* = P, *order* = *induced* $P\ r$ |) L
⟨*proof*⟩

lemma *CompleteLatticeI-simp*:
[| (| *pset* = A, *order* = r |) ∈ *PartialOrder*;
$\forall\, S.\ S \subseteq A$ --> ($\exists\, L$. *isLub* S (| *pset* = A, *order* = r |) L) |]
==> (| *pset* = A, *order* = r |) ∈ *CompleteLattice*
⟨*proof*⟩

theorem (**in** *CLF*) *Tarski-full*:
(| *pset* = P, *order* = *induced* $P\ r$|) ∈ *CompleteLattice*
⟨*proof*⟩

end

36 Implementation of carry chain incrementor and adder

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

lemma [*simp*]: *bv-to-nat* [b] = *bitval* b
⟨*proof*⟩

lemma *bv-to-nat-helper'*:
$bv \neq$ [] ==> *bv-to-nat* bv = *bitval* (*hd* bv) ∗ *2* ˆ (*length* bv − *1*) + *bv-to-nat* (*tl* bv)

⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*simp*]: *length* (*half-adder a b*) = *2*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma [*simp*]: *length* (*full-adder a b c*) = *2*
⟨*proof*⟩

36.1 Carry chain incrementor

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

lemma *cci-nonnull*: *carry-chain-inc as c* ≠ []
⟨*proof*⟩

lemma *cci-length* [*simp*]: *length* (*carry-chain-inc as c*) = *length as* + *1*
⟨*proof*⟩

lemma *cci-correct*: *bv-to-nat* (*carry-chain-inc as c*) = *bv-to-nat as* + *bitval c*
⟨*proof*⟩

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* ≠ []
⟨*proof*⟩

lemma *cca-length*: *length as* = *length bs* ⟹
length (*carry-chain-adder as bs c*) = *Suc* (*length bs*)
⟨*proof*⟩

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*⟩

end

37 Hilbert's choice and classical logic

theory *Hilbert-Classical* **imports** *Main* **begin**

Derivation of the classical law of tertium-non-datur by means of Hilbert's choice operator (due to M. J. Beeson and J. Harrison).

37.1 Proof text

theorem *tnd*: *A* ∨ ¬ *A*
⟨*proof*⟩

37.2 Proof term of text

```
disjE · - · - · - •
 (someI · (λX. X = False ∨ X = True ∧ ?A) · - •
   (disjI1 · - · - • (HOL.refl · -))) •
 (λH: -.
   disjE · - · - · - •
    (someI · (λX. X = False ∧ ?A ∨ X = True) · - •
      (disjI2 · - · - • (HOL.refl · -))) •
    (λH: -. disjI1 · - · - • (conjE · - · - · - • H • (λ(H: -) H: -. H))) •
    (λHa: -.
      disjI2 · - · - •
       (notI · - •
         (λHb: -.
           notE · - · - •
            (notI · - •
              (λHb: -.
                False-neq-True · - •
                 (order-trans-rules-29 · - · - · - •
```

```
                (order-trans-rules-14 ·
                  (λa. a = (SOME X. X = False ∧ ?A ∨ X = True)) ·
                  - ·
                  - •
                  (arg-cong · (λX. X = False ∨ X = True ∧ ?A) ·
                    (λX. X = False ∧ ?A ∨ X = True) ·
                    Eps •
                    Hb) •
                  H) •
                Ha))) •
         (ext · - · - •
           (λX. iffI · - · - •
                 (λH: -.
                   disjE · - · - · - • H •
                    (λH: -. disjI1 · - · - • (conjI · - · - • H • Hb)) •
                    (λH: -.
                      disjI2 · - · - •
                       (conjE · - · - · - • H • (λ(H: -) Ha: -. H)))) •
                 (λH: -.
                   disjE · - · - · - • H •
                    (λH: -.
                      disjI1 · - · - •
                       (conjE · - · - · - • H • (λ(H: -) Ha: -. H))) •
                    (λH: -.
                      disjI2 · - · - • (conjI · - · - • H • Hb))))))))) •
(λH: -. disjI1 · - · - • (conjE · - · - · - • H • (λ(H: -) H: -. H)))
```

37.3 Proof script

theorem *tnd′*: $A \lor \neg\, A$
  ⟨*proof*⟩

37.4 Proof term of script

```
conjE · - · - · - •
 (conjI · - · - •
   (someI · (λx. x = False ∨ x = True ∧ ?A) · - •
     (disjI1 · - · - • (HOL.refl · -))) •
   (someI · (λx. x = False ∧ ?A ∨ x = True) · - •
     (disjI2 · - · - • (HOL.refl · -)))) •
 (λ(H: -) Ha: -.
   disjE · - · - · - • H •
    (λH: -.
      disjE · - · - · - • Ha •
       (λH: -. conjE · - · - · - • H • (λH: -. disjI1 · - · -)) •
       (λHa: -.
         disjI2 · - · - •
          (notI · - •
```

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

end

38 Classical Predicate Calculus Problems

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

38.1 Traditional Classical Reasoner

The machine "griffon" mentioned below is a 2.5GHz Power Mac G5.

Taken from *FOL/Classical.thy*. When porting examples from first-order logic, beware of the precedence of = versus ↔.

lemma $(P \;{-}{-}{>}\; Q \mid R) \;{-}{-}{>}\; (P{-}{-}{>}Q) \mid (P{-}{-}{>}R)$
⟨*proof*⟩

If and only if

lemma $(P{=}Q) = (Q = (P{::}bool))$
⟨*proof*⟩

lemma $\sim (P = (\sim P))$
$\langle proof \rangle$

Sample problems from F. J. Pelletier, Seventy-Five Problems for Testing Automatic Theorem Provers, J. Automated Reasoning 2 (1986), 191-216. Errata, JAR 4 (1988), 236-236.

The hardest problems – judging by experience with several theorem provers, including matrix ones – are 34 and 43.

38.1.1 Pelletier's examples

1

lemma $(P{-}{-}{>}Q) \;=\; (\sim Q \;{-}{-}{>}\; \sim P)$
$\langle proof \rangle$

2

lemma $(\sim \sim P) = \; P$
$\langle proof \rangle$

3

lemma $\sim(P{-}{-}{>}Q) \;{-}{-}{>}\; (Q{-}{-}{>}P)$
$\langle proof \rangle$

4

lemma $(\sim P{-}{-}{>}Q) \;=\; (\sim Q \;{-}{-}{>}\; P)$
$\langle proof \rangle$

5

lemma $((P|Q){-}{-}{>}(P|R)) \;{-}{-}{>}\; (P|(Q{-}{-}{>}R))$
$\langle proof \rangle$

6

lemma $P \mid \sim P$
$\langle proof \rangle$

7

lemma $P \mid \sim \sim \sim P$
$\langle proof \rangle$

8. Peirce's law

lemma $((P{-}{-}{>}Q) \;{-}{-}{>}\; P) \;\; {-}{-}{>} \;\; P$
$\langle proof \rangle$

9

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

$\langle proof \rangle$

10

lemma $(Q{-}{-}{>}R)$ & $(R{-}{-}{>}P\&Q)$ & $(P{-}{-}{>}Q|R)$ $-{-}{>}$ $(P{=}Q)$
$\langle proof \rangle$

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

lemma $P{=}(P{::}bool)$
$\langle proof \rangle$

12. "Dijkstra's law"

lemma $((P = Q) = R) = (P = (Q = R))$
$\langle proof \rangle$

13. Distributive law

lemma $(P \mid (Q \ \& \ R)) = ((P \mid Q) \ \& \ (P \mid R))$
$\langle proof \rangle$

14

lemma $(P = Q) = ((Q \mid {\sim}P) \ \& \ ({\sim}Q|P))$
$\langle proof \rangle$

15

lemma $(P \ {-}{-}{>} \ Q) = ({\sim}P \mid Q)$
$\langle proof \rangle$

16

lemma $(P{-}{-}{>}Q) \mid (Q{-}{-}{>}P)$
$\langle proof \rangle$

17

lemma $((P \ \& \ (Q{-}{-}{>}R)){-}{-}{>}S) \ = \ (({\sim}P \mid Q \mid S) \ \& \ ({\sim}P \mid {\sim}R \mid S))$
$\langle proof \rangle$

38.1.2 Classical Logic: examples with quantifiers

lemma $(\forall x. \ P(x) \ \& \ Q(x)) = ((\forall x. \ P(x)) \ \& \ (\forall x. \ Q(x)))$
$\langle proof \rangle$

lemma $(\exists x. \ P{-}{-}{>}Q(x)) \ = \ (P \ {-}{-}{>} \ (\exists x. \ Q(x)))$
$\langle proof \rangle$

lemma $(\exists x. \ P(x){-}{-}{>}Q) = ((\forall x. \ P(x)) \ {-}{-}{>} \ Q)$
$\langle proof \rangle$

lemma $((\forall x. \ P(x)) \mid Q) \ = \ (\forall x. \ P(x) \mid Q)$
$\langle proof \rangle$

From Wishnu Prasetya

lemma $(\forall s.\ q(s) \dashrightarrow r(s))$ & ~$r(s)$ & $(\forall s.$ ~$r(s)$ & ~$q(s) \dashrightarrow p(t) \mid q(t))$
$\dashrightarrow p(t) \mid r(t)$
⟨*proof*⟩

38.1.3 Problems requiring quantifier duplication

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

lemma $(\exists x.\ \forall y.\ P(x) = P(y))$ --> $((\exists x.\ P(x)) = (\forall y.\ P(y)))$
⟨*proof*⟩

Needs multiple instantiation of the quantifier.

lemma $(\forall x.\ P(x)$-->$P(f(x)))$ & $P(d)$-->$P(f(f(f(d))))$
⟨*proof*⟩

Needs double instantiation of the quantifier

lemma $\exists x.\ P(x)$ --> $P(a)$ & $P(b)$
⟨*proof*⟩

lemma $\exists z.\ P(z)$ --> $(\forall x.\ P(x))$
⟨*proof*⟩

lemma $\exists x.\ (\exists y.\ P(y))$ --> $P(x)$
⟨*proof*⟩

38.1.4 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P(y)$-->$P(x)$
⟨*proof*⟩

Problem 19

lemma $\exists x.\ \forall y\ z.\ (P(y)$-->$Q(z))$ --> $(P(x)$-->$Q(x))$
⟨*proof*⟩

Problem 20

lemma $(\forall x\ y.\ \exists z.\ \forall w.\ (P(x)$&$Q(y)$-->$R(z)$&$S(w)))$
--> $(\exists x\ y.\ P(x)$ & $Q(y))$ --> $(\exists z.\ R(z))$
⟨*proof*⟩

Problem 21

lemma $(\exists x.\ P$-->$Q(x))$ & $(\exists x.\ Q(x)$-->$P)$ --> $(\exists x.\ P=Q(x))$
⟨*proof*⟩

Problem 22

lemma $(\forall x.\ P = Q(x))$ $--\!>$ $(P = (\forall x.\ Q(x)))$
$\langle proof \rangle$

Problem 23

lemma $(\forall x.\ P \mid Q(x))$ $=$ $(P \mid (\forall x.\ Q(x)))$
$\langle proof \rangle$

Problem 24

lemma $\sim(\exists x.\ S(x)\&Q(x))$ $\&$ $(\forall x.\ P(x)$ $--\!>$ $Q(x)|R(x))$ $\&$
$(\sim(\exists x.\ P(x))$ $--\!>$ $(\exists x.\ Q(x)))$ $\&$ $(\forall x.\ Q(x)|R(x)$ $--\!>$ $S(x))$
$--\!>$ $(\exists x.\ P(x)\&R(x))$
$\langle proof \rangle$

Problem 25

lemma $(\exists x.\ P(x))$ $\&$
$(\forall x.\ L(x)$ $--\!>$ $\sim$ $(M(x)$ $\&$ $R(x)))$ $\&$
$(\forall x.\ P(x)$ $--\!>$ $(M(x)$ $\&$ $L(x)))$ $\&$
$((\forall x.\ P(x)--\!>Q(x))$ $|$ $(\exists x.\ P(x)\&R(x)))$
$--\!>$ $(\exists x.\ Q(x)\&P(x))$
$\langle proof \rangle$

Problem 26

lemma $((\exists x.\ p(x)) = (\exists x.\ q(x)))$ $\&$
$(\forall x.\ \forall y.\ p(x)$ $\&$ $q(y)$ $--\!>$ $(r(x) = s(y)))$
$--\!>$ $((\forall x.\ p(x)--\!>r(x)) = (\forall x.\ q(x)--\!>s(x)))$
$\langle proof \rangle$

Problem 27

lemma $(\exists x.\ P(x)$ $\&$ $\sim Q(x))$ $\&$
$(\forall x.\ P(x)$ $--\!>$ $R(x))$ $\&$
$(\forall x.\ M(x)$ $\&$ $L(x)$ $--\!>$ $P(x))$ $\&$
$((\exists x.\ R(x)$ $\&$ $\sim$ $Q(x))$ $--\!>$ $(\forall x.\ L(x)$ $--\!>$ $\sim$ $R(x)))$
$--\!>$ $(\forall x.\ M(x)$ $--\!>$ $\sim L(x))$
$\langle proof \rangle$

Problem 28. AMENDED

lemma $(\forall x.\ P(x)$ $--\!>$ $(\forall x.\ Q(x)))$ $\&$
$((\forall x.\ Q(x)|R(x))$ $--\!>$ $(\exists x.\ Q(x)\&S(x)))$ $\&$
$((\exists x.\ S(x))$ $--\!>$ $(\forall x.\ L(x)$ $--\!>$ $M(x)))$
$--\!>$ $(\forall x.\ P(x)$ $\&$ $L(x)$ $--\!>$ $M(x))$
$\langle proof \rangle$

Problem 29. Essentially the same as Principia Mathematica *11.71

lemma $(\exists x.\ F(x))$ $\&$ $(\exists y.\ G(y))$
$--\!>$ $(\ ((\forall x.\ F(x)--\!>H(x))$ $\&$ $(\forall y.\ G(y)--\!>J(y)))$ $=$
$(\forall x\ y.\ F(x)$ $\&$ $G(y)$ $--\!>$ $H(x)$ $\&$ $J(y)))$
$\langle proof \rangle$

Problem 30

lemma $(\forall x.\ P(x) \mid Q(x) \text{ --> } {}^{\sim}\ R(x))$ &
$(\forall x.\ (Q(x) \text{ --> } {}^{\sim}\ S(x)) \text{ --> } P(x)\ \&\ R(x))$
$\text{--> } (\forall x.\ S(x))$
$\langle proof \rangle$

Problem 31

lemma ${}^{\sim}(\exists x.\ P(x)\ \&\ (Q(x) \mid R(x)))$ &
$(\exists x.\ L(x)\ \&\ P(x))$ &
$(\forall x.\ {}^{\sim}\ R(x) \text{ --> } M(x))$
$\text{--> } (\exists x.\ L(x)\ \&\ M(x))$
$\langle proof \rangle$

Problem 32

lemma $(\forall x.\ P(x)\ \&\ (Q(x)|R(x))\text{-->}S(x))$ &
$(\forall x.\ S(x)\ \&\ R(x) \text{ --> } L(x))$ &
$(\forall x.\ M(x) \text{ --> } R(x))$
$\text{--> } (\forall x.\ P(x)\ \&\ M(x) \text{ --> } L(x))$
$\langle proof \rangle$

Problem 33

lemma $(\forall x.\ P(a)\ \&\ (P(x)\text{-->}P(b))\text{-->}P(c)) =$
$(\forall x.\ ({}^{\sim}P(a) \mid P(x) \mid P(c))\ \&\ ({}^{\sim}P(a) \mid {}^{\sim}P(b) \mid P(c)))$
$\langle proof \rangle$

Problem 34 AMENDED (TWICE!!)

Andrews's challenge

lemma $((\exists x.\ \forall y.\ p(x) = p(y)) =$
$((\exists x.\ q(x)) = (\forall y.\ p(y)))) =$
$((\exists x.\ \forall y.\ q(x) = q(y)) =$
$((\exists x.\ p(x)) = (\forall y.\ q(y))))$
$\langle proof \rangle$

Problem 35

lemma $\exists x\ y.\ P\ x\ y \text{ --> } (\forall u\ v.\ P\ u\ v)$
$\langle proof \rangle$

Problem 36

lemma $(\forall x.\ \exists y.\ J\ x\ y)$ &
$(\forall x.\ \exists y.\ G\ x\ y)$ &
$(\forall x\ y.\ J\ x\ y \mid G\ x\ y \text{ --> }$
$(\forall z.\ J\ y\ z \mid G\ y\ z \text{ --> } H\ x\ z))$
$\text{--> } (\forall x.\ \exists y.\ H\ x\ y)$
$\langle proof \rangle$

Problem 37

lemma $(\forall z.\ \exists w.\ \forall x.\ \exists y.$
$(P\ x\ z\ {-}{-}{>}P\ y\ w)\ \&\ P\ y\ z\ \&\ (P\ y\ w\ {-}{-}{>}\ (\exists u.\ Q\ u\ w)))\ \&$
$(\forall x\ z.\ {\sim}(P\ x\ z)\ {-}{-}{>}\ (\exists y.\ Q\ y\ z))\ \&$
$((\exists x\ y.\ Q\ x\ y)\ {-}{-}{>}\ (\forall x.\ R\ x\ x))$
${-}{-}{>}\ (\forall x.\ \exists y.\ R\ x\ y)$
$\langle proof \rangle$

Problem 38

lemma $(\forall x.\ p(a)\ \&\ (p(x)\ {-}{-}{>}\ (\exists y.\ p(y)\ \&\ r\ x\ y))\ {-}{-}{>}$
$(\exists z.\ \exists w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$(\forall x.\ ({\sim}p(a)\ |\ p(x)\ |\ (\exists z.\ \exists w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$({\sim}p(a)\ |\ {\sim}(\exists y.\ p(y)\ \&\ r\ x\ y)\ |$
$(\exists z.\ \exists w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
$\langle proof \rangle$

Problem 39

lemma ${\sim}\ (\exists x.\ \forall y.\ F\ y\ x = ({\sim}\ F\ y\ y))$
$\langle proof \rangle$

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
${-}{-}{>}\ \ {\sim}\ (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = ({\sim}\ F\ z\ x))$
$\langle proof \rangle$

Problem 41

lemma $(\forall z.\ \exists y.\ \forall x.\ f\ x\ y = (f\ x\ z\ \&\ {\sim}\ f\ x\ x))$
${-}{-}{>}\ {\sim}\ (\exists z.\ \forall x.\ f\ x\ z)$
$\langle proof \rangle$

Problem 42

lemma ${\sim}\ (\exists y.\ \forall x.\ p\ x\ y = ({\sim}\ (\exists z.\ p\ x\ z\ \&\ p\ z\ x)))$
$\langle proof \rangle$

Problem 43!!

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

Problem 44

lemma $(\forall x.\ f(x)\ {-}{-}{>}$
$(\exists y.\ g(y)\ \&\ h\ x\ y\ \&\ (\exists y.\ g(y)\ \&\ {\sim}\ h\ x\ y)))\ \ \&$
$(\exists x.\ j(x)\ \&\ (\forall y.\ g(y)\ {-}{-}{>}\ h\ x\ y))$
${-}{-}{>}\ (\exists x.\ j(x)\ \&\ {\sim}f(x))$
$\langle proof \rangle$

Problem 45

lemma $(\forall x.\ f(x)\ \&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ {-}{-}{>}\ j\ x\ y)$

$--> (\forall y.\ g(y)\ \&\ h\ x\ y\ --> k(y)))\ \&$
$\sim (\exists y.\ l(y)\ \&\ k(y))\ \&$
$(\exists x.\ f(x)\ \&\ (\forall y.\ h\ x\ y\ --> l(y))$
$\&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ --> j\ x\ y))$
$--> (\exists x.\ f(x)\ \&\ \sim (\exists y.\ g(y)\ \&\ h\ x\ y))$
⟨*proof*⟩

38.1.5 Problems (mainly) involving equality or functions

Problem 48

lemma $(a=b \mid c=d)\ \&\ (a=c \mid b=d)\ --> a=d \mid b=c$
⟨*proof*⟩

Problem 49 NOT PROVED AUTOMATICALLY. Hard because it involves substitution for Vars the type constraint ensures that x,y,z have the same type as a,b,u.

lemma $(\exists x\ y::'a.\ \forall z.\ z=x \mid z=y)\ \&\ P(a)\ \&\ P(b)\ \&\ (\sim a=b)$
$--> (\forall u::'a.\ P(u))$
⟨*proof*⟩

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

lemma $(\forall x.\ P\ a\ x \mid (\forall y.\ P\ x\ y))\ --> (\exists x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

Problem 51

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

Problem 52. Almost the same as 51.

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

Problem 55

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

lemma *lives*(*agatha*) & *lives*(*butler*) & *lives*(*charles*) &
(*killed agatha agatha* | *killed butler agatha* | *killed charles agatha*) &
(∀ *x y*. *killed x y* --> *hates x y* & ~*richer x y*) &
(∀ *x*. *hates agatha x* --> ~*hates charles x*) &
(*hates agatha agatha* & *hates agatha charles*) &
(∀ *x*. *lives*(*x*) & ~*richer x agatha* --> *hates butler x*) &
(∀ *x*. *hates agatha x* --> *hates butler x*) &
(∀ *x*. ~*hates x agatha* | ~*hates x butler* | ~*hates x charles*) -->
killed ?who agatha

⟨*proof*⟩

Problem 56

lemma $(\forall x.\ (\exists y.\ P(y)\ \&\ x{=}f(y))\ --> P(x)) = (\forall x.\ P(x)\ --> P(f(x)))$
⟨*proof*⟩

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$\quad (\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ --> P\ x\ z) \quad --> \quad P\ (f\ a\ b)\ (f\ a\ c)$
⟨*proof*⟩

Problem 58 NOT PROVED AUTOMATICALLY

lemma $(\forall x\ y.\ f(x){=}g(y))\ --> (\forall x\ y.\ f(f(x)){=}f(g(y)))$
⟨*proof*⟩

Problem 59

lemma $(\forall x.\ P(x) = ({\sim}P(f(x))))\ --> (\exists x.\ P(x)\ \&\ {\sim}P(f(x)))$
⟨*proof*⟩

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y\ --> P\ z\ (f\ x))\ \&\ P\ x\ y)$
⟨*proof*⟩

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ --> p(f\ x))\ --> p(f(f\ x))) \ =$
$\quad (\forall x.\ ({\sim}\ p\ a \mid p\ x \mid p(f(f\ x)))\ \&$
$\qquad ({\sim}\ p\ a \mid {\sim}\ p(f\ x) \mid p(f(f\ x))))$
⟨*proof*⟩

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

lemma $(\forall x.\ F(x)\ \&\ {\sim}G(x)\ --> (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$\quad (\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ --> K(y)))\ \&$
$\quad (\forall x.\ K(x)\ --> {\sim}G(x)) \ --> \ (\exists x.\ K(x)\ \&\ J(x))$
⟨*proof*⟩

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

lemma $(\forall x.\ F(x)\ \&\ {\sim}G(x)\ --> (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$\quad (\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ --> K(y)))\ \&$
$\quad (\forall x.\ K(x)\ --> {\sim}G(x)) \quad --> \quad (\exists x.\ K(x)\ --> {\sim}G(x))$
⟨*proof*⟩

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

lemma $(\forall x.\ \mathit{honest}(x)\ \&\ \mathit{industrious}(x)\ --> \mathit{healthy}(x))\ \&$

$\sim$ $(\exists\, x.\ grocer(x)\ \&\ healthy(x))$ &
$(\forall\, x.\ industrious(x)\ \&\ grocer(x)\ \text{-->}\ honest(x))$ &
$(\forall\, x.\ cyclist(x)\ \text{-->}\ industrious(x))$ &
$(\forall\, x.\ {\sim}healthy(x)\ \&\ cyclist(x)\ \text{-->}\ {\sim}honest(x))$
$\text{-->}\ (\forall\, x.\ grocer(x)\ \text{-->}\ {\sim}cyclist(x))$
⟨*proof*⟩

lemma $(\forall\, x\ y.\ R(x,y)\ |\ R(y,x))$ &
$(\forall\, x\ y.\ S(x,y)\ \&\ S(y,x)\ \text{-->}\ x{=}y)$ &
$(\forall\, x\ y.\ R(x,y)\ \text{-->}\ S(x,y))\quad \text{-->}\quad (\forall\, x\ y.\ S(x,y)\ \text{-->}\ R(x,y))$
⟨*proof*⟩

38.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y\ \&\ y < z\ \text{-->}\ {\sim}\ (z < (x{::}nat))$
⟨*proof*⟩

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 $(\forall\, x\ y\ z.\ R(x,y)\ \&\ R(y,z)\ \text{-->}\ R(x,z))$ &
$(\forall\, x.\ \exists\, y.\ R(x,y))\ \text{-->}$
$\sim\ (\forall\, x.\ P\ x = (\forall\, y.\ R(x,y)\ \text{-->}\ {\sim}\ P\ y))$
⟨*proof*⟩

38.2.1 Pelletier's examples

1

lemma $(P\ \text{-->}\ Q)\ =\ ({\sim}Q\ \text{-->}\ {\sim}P)$
⟨*proof*⟩

2

lemma $({\sim}\ {\sim}\ P) =\ P$
⟨*proof*⟩

3

lemma ${\sim}(P\text{-->}Q)\ \text{-->}\ (Q\text{-->}P)$
⟨*proof*⟩

4

lemma $({\sim}P\text{-->}Q)\ =\ ({\sim}Q\ \text{-->}\ P)$
⟨*proof*⟩

5

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

⟨*proof*⟩

6

lemma $P \mid \sim P$
⟨*proof*⟩

7

lemma $P \mid \sim\sim\sim P$
⟨*proof*⟩

8. Peirce's law

lemma $((P{-}{-}{>}Q)\ {-}{-}{>}\ P)\ \ {-}{-}{>}\ \ P$
⟨*proof*⟩

9

lemma $((P|Q)\ \&\ ({\sim}P|Q)\ \&\ (P|\ {\sim}Q))\ {-}{-}{>}\ {\sim}\ ({\sim}P\ |\ {\sim}Q)$
⟨*proof*⟩

10

lemma $(Q{-}{-}{>}R)\ \&\ (R{-}{-}{>}P\&Q)\ \&\ (P{-}{-}{>}Q|R)\ {-}{-}{>}\ (P{=}Q)$
⟨*proof*⟩

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

lemma $P{=}(P{::}bool)$
⟨*proof*⟩

12. "Dijkstra's law"

lemma $((P = Q) = R) = (P = (Q = R))$
⟨*proof*⟩

13. Distributive law

lemma $(P \mid (Q\ \&\ R)) = ((P \mid Q)\ \&\ (P \mid R))$
⟨*proof*⟩

14

lemma $(P = Q) = ((Q \mid {\sim}P)\ \&\ ({\sim}Q|P))$
⟨*proof*⟩

15

lemma $(P\ {-}{-}{>}\ Q) = ({\sim}P \mid Q)$
⟨*proof*⟩

16

lemma $(P{-}{-}{>}Q) \mid (Q{-}{-}{>}P)$
⟨*proof*⟩

17

lemma $((P\ \&\ (Q{-}{-}{>}R)){-}{-}{>}S)\ \ =\ \ (({\sim}P \mid Q \mid S)\ \&\ ({\sim}P \mid {\sim}R \mid S))$
⟨*proof*⟩

38.2.2 Classical Logic: examples with quantifiers

lemma $(\forall x.\ P\ x\ \&\ Q\ x) = ((\forall x.\ P\ x)\ \&\ (\forall x.\ Q\ x))$
⟨*proof*⟩

lemma $(\exists x.\ P\ \text{-->}\ Q\ x)\ =\ (P\ \text{-->}\ (\exists x.\ Q\ x))$
⟨*proof*⟩

lemma $(\exists x.\ P\ x\ \text{-->}\ Q) = ((\forall x.\ P\ x)\ \text{-->}\ Q)$
⟨*proof*⟩

lemma $((\forall x.\ P\ x)\ |\ Q)\ =\ (\forall x.\ P\ x\ |\ Q)$
⟨*proof*⟩

lemma $(\forall x.\ P\ x\ \text{-->}\ P(f\ x))\ \ \&\ \ P\ d\ \text{-->}\ P(f(f(f\ d)))$
⟨*proof*⟩

Needs double instantiation of EXISTS

lemma $\exists x.\ P\ x\ \text{-->}\ P\ a\ \&\ P\ b$
⟨*proof*⟩

lemma $\exists z.\ P\ z\ \text{-->}\ (\forall x.\ P\ x)$
⟨*proof*⟩

From a paper by Claire Quigley

lemma $\exists y.\ ((P\ c\ \&\ Q\ y)\ |\ (\exists z.\ {\sim}\ Q\ z))\ |\ (\exists x.\ {\sim}\ P\ x\ \&\ Q\ d)$
⟨*proof*⟩

38.2.3 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P\ y\ \text{-->}\ P\ x$
⟨*proof*⟩

Problem 19

lemma $\exists x.\ \forall y\ z.\ (P\ y\ \text{-->}\ Q\ z)\ \text{-->}\ (P\ x\ \text{-->}\ Q\ x)$
⟨*proof*⟩

Problem 20

lemma $(\forall x\ y.\ \exists z.\ \forall w.\ (P\ x\ \&\ Q\ y\ \text{-->}\ R\ z\ \&\ S\ w))$
$\quad \text{-->}\ (\exists x\ y.\ P\ x\ \&\ Q\ y)\ \text{-->}\ (\exists z.\ R\ z)$
⟨*proof*⟩

Problem 21

lemma $(\exists x.\ P\ \text{-->}\ Q\ x)\ \&\ (\exists x.\ Q\ x\ \text{-->}\ P)\ \text{-->}\ (\exists x.\ P{=}Q\ x)$
⟨*proof*⟩

Problem 22

lemma $(\forall x.\ P = Q\ x)$ $-->$ $(P = (\forall x.\ Q\ x))$
$\langle proof \rangle$

Problem 23

lemma $(\forall x.\ P \mid Q\ x)$ $=$ $(P \mid (\forall x.\ Q\ x))$
$\langle proof \rangle$

Problem 24

lemma $\sim(\exists x.\ S\ x\ \&\ Q\ x)\ \&\ (\forall x.\ P\ x\ -->\ Q\ x \mid R\ x)\ \&$
$(\sim(\exists x.\ P\ x)\ -->\ (\exists x.\ Q\ x))\ \&\ (\forall x.\ Q\ x \mid R\ x\ -->\ S\ x)$
$-->\ (\exists x.\ P\ x\ \&\ R\ x)$
$\langle proof \rangle$

Problem 25

lemma $(\exists x.\ P\ x)\ \&$
$(\forall x.\ L\ x\ -->\ \sim\ (M\ x\ \&\ R\ x))\ \&$
$(\forall x.\ P\ x\ -->\ (M\ x\ \&\ L\ x))\ \&$
$((\forall x.\ P\ x\ -->\ Q\ x) \mid (\exists x.\ P\ x\ \&\ R\ x))$
$-->\ (\exists x.\ Q\ x\ \&\ P\ x)$
$\langle proof \rangle$

Problem 26; has 24 Horn clauses

lemma $((\exists x.\ p\ x) = (\exists x.\ q\ x))\ \&$
$(\forall x.\ \forall y.\ p\ x\ \&\ q\ y\ -->\ (r\ x = s\ y))$
$-->\ ((\forall x.\ p\ x\ -->\ r\ x) = (\forall x.\ q\ x\ -->\ s\ x))$
$\langle proof \rangle$

Problem 27; has 13 Horn clauses

lemma $(\exists x.\ P\ x\ \&\ \sim Q\ x)\ \&$
$(\forall x.\ P\ x\ -->\ R\ x)\ \&$
$(\forall x.\ M\ x\ \&\ L\ x\ -->\ P\ x)\ \&$
$((\exists x.\ R\ x\ \&\ \sim\ Q\ x)\ -->\ (\forall x.\ L\ x\ -->\ \sim\ R\ x))$
$-->\ (\forall x.\ M\ x\ -->\ \sim L\ x)$
$\langle proof \rangle$

Problem 28. AMENDED; has 14 Horn clauses

lemma $(\forall x.\ P\ x\ -->\ (\forall x.\ Q\ x))\ \&$
$((\forall x.\ Q\ x \mid R\ x)\ -->\ (\exists x.\ Q\ x\ \&\ S\ x))\ \&$
$((\exists x.\ S\ x)\ -->\ (\forall x.\ L\ x\ -->\ M\ x))$
$-->\ (\forall x.\ P\ x\ \&\ L\ x\ -->\ M\ x)$
$\langle proof \rangle$

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

lemma $(\exists x.\ F\ x)\ \&\ (\exists y.\ G\ y)$
$-->\ (\ ((\forall x.\ F\ x\ -->\ H\ x)\ \&\ (\forall y.\ G\ y\ -->\ J\ y))\ =$
$(\forall x\ y.\ F\ x\ \&\ G\ y\ -->\ H\ x\ \&\ J\ y))$

$\langle proof \rangle$

Problem 30

lemma $(\forall x.\ P\ x \mid Q\ x \dashrightarrow \sim R\ x)$ & $(\forall x.\ (Q\ x \dashrightarrow \sim S\ x) \dashrightarrow P\ x$ & $R\ x)$
$\quad \dashrightarrow (\forall x.\ S\ x)$
$\langle proof \rangle$

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

lemma $\sim(\exists x.\ P\ x$ & $(Q\ x \mid R\ x))$ &
$(\exists x.\ L\ x$ & $P\ x)$ &
$(\forall x.\ \sim R\ x \dashrightarrow M\ x)$
$\dashrightarrow (\exists x.\ L\ x$ & $M\ x)$
$\langle proof \rangle$

Problem 32

lemma $(\forall x.\ P\ x$ & $(Q\ x \mid R\ x) \dashrightarrow S\ x)$ &
$(\forall x.\ S\ x$ & $R\ x \dashrightarrow L\ x)$ &
$(\forall x.\ M\ x \dashrightarrow R\ x)$
$\dashrightarrow (\forall x.\ P\ x$ & $M\ x \dashrightarrow L\ x)$
$\langle proof \rangle$

Problem 33; has 55 Horn clauses

lemma $(\forall x.\ P\ a$ & $(P\ x \dashrightarrow P\ b) \dashrightarrow P\ c) =$
$(\forall x.\ (\sim P\ a \mid P\ x \mid P\ c)$ & $(\sim P\ a \mid \sim P\ b \mid P\ c))$
$\langle proof \rangle$

Problem 34: Andrews's challenge has 924 Horn clauses

lemma $((\exists x.\ \forall y.\ p\ x = p\ y) = ((\exists x.\ q\ x) = (\forall y.\ p\ y))) =$
$((\exists x.\ \forall y.\ q\ x = q\ y) = ((\exists x.\ p\ x) = (\forall y.\ q\ y)))$
$\langle proof \rangle$

Problem 35

lemma $\exists x\ y.\ P\ x\ y \dashrightarrow (\forall u\ v.\ P\ u\ v)$
$\langle proof \rangle$

Problem 36; has 15 Horn clauses

lemma $(\forall x.\ \exists y.\ J\ x\ y)$ & $(\forall x.\ \exists y.\ G\ x\ y)$ &
$(\forall x\ y.\ J\ x\ y \mid G\ x\ y \dashrightarrow (\forall z.\ J\ y\ z \mid G\ y\ z \dashrightarrow H\ x\ z))$
$\dashrightarrow (\forall x.\ \exists y.\ H\ x\ y)$
$\langle proof \rangle$

Problem 37; has 10 Horn clauses

lemma $(\forall z.\ \exists w.\ \forall x.\ \exists y.$
$(P\ x\ z \dashrightarrow P\ y\ w)$ & $P\ y\ z$ & $(P\ y\ w \dashrightarrow (\exists u.\ Q\ u\ w)))$ &
$(\forall x\ z.\ \sim P\ x\ z \dashrightarrow (\exists y.\ Q\ y\ z))$ &
$((\exists x\ y.\ Q\ x\ y) \dashrightarrow (\forall x.\ R\ x\ x))$
$\dashrightarrow (\forall x.\ \exists y.\ R\ x\ y)$

$\langle proof \rangle$

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ --> (\exists y.\ p\ y\ \&\ r\ x\ y))\ -->$
$\quad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$\quad (\forall x.\ ({\sim}p\ a\ |\ p\ x\ |\ (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$\quad ({\sim}p\ a\ |\ {\sim}(\exists y.\ p\ y\ \&\ r\ x\ y)\ |$
$\quad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
$\langle proof \rangle$

Problem 39

lemma ${\sim}\ (\exists x.\ \forall y.\ F\ y\ x = ({\sim}F\ y\ y))$
$\langle proof \rangle$

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
$\quad -->\ {\sim}\ (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = ({\sim}F\ z\ x))$
$\langle proof \rangle$

Problem 41

lemma $(\forall z.\ (\exists y.\ (\forall x.\ f\ x\ y = (f\ x\ z\ \&\ {\sim}\ f\ x\ x))))$
$\quad -->\ {\sim}\ (\exists z.\ \forall x.\ f\ x\ z)$
$\langle proof \rangle$

Problem 42

lemma ${\sim}\ (\exists y.\ \forall x.\ p\ x\ y = ({\sim}\ (\exists z.\ p\ x\ z\ \&\ p\ z\ x)))$
$\langle proof \rangle$

Problem 43 NOW PROVED AUTOMATICALLY!!

lemma $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}bool)))$
$\quad -->\ (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}bool)))$
$\langle proof \rangle$

Problem 44: 13 Horn clauses; 7-step proof

lemma $(\forall x.\ f\ x\ -->\ (\exists y.\ g\ y\ \&\ h\ x\ y\ \&\ (\exists y.\ g\ y\ \&\ {\sim}\ h\ x\ y)))\ \&$
$\quad (\exists x.\ j\ x\ \&\ (\forall y.\ g\ y\ -->\ h\ x\ y))$
$\quad -->\ (\exists x.\ j\ x\ \&\ {\sim}f\ x)$
$\langle proof \rangle$

Problem 45; has 27 Horn clauses; 54-step proof

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

$\quad$ --> $(\exists x.\ f\ x\ \&\ \sim(\exists y.\ g\ y\ \&\ h\ x\ y))$
⟨*proof*⟩

Problem 46; has 26 Horn clauses; 21-step proof

lemma $(\forall x.\ f\ x\ \&\ (\forall y.\ f\ y\ \&\ h\ y\ x \dashrightarrow g\ y) \dashrightarrow g\ x)\ \&$
$\quad ((\exists x.\ f\ x\ \&\ {\sim}g\ x) \dashrightarrow$
$\quad (\exists x.\ f\ x\ \&\ {\sim}g\ x\ \&\ (\forall y.\ f\ y\ \&\ {\sim}g\ y \dashrightarrow j\ x\ y)))\ \&$
$\quad (\forall x\ y.\ f\ x\ \&\ f\ y\ \&\ h\ x\ y \dashrightarrow {\sim}j\ y\ x)$
$\quad \dashrightarrow (\forall x.\ f\ x \dashrightarrow g\ x)$
⟨*proof*⟩

Problem 47. Schubert's Steamroller. 26 clauses; 63 Horn clauses. 87094 inferences so far. Searching to depth 36

lemma $(\forall x.\ wolf\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ wolf\ x)\ \&$
$\quad (\forall x.\ fox\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ fox\ x)\ \&$
$\quad (\forall x.\ bird\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ bird\ x)\ \&$
$\quad (\forall x.\ caterpillar\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ caterpillar\ x)\ \&$
$\quad (\forall x.\ snail\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ snail\ x)\ \&$
$\quad (\forall x.\ grain\ x \longrightarrow plant\ x)\ \&\ (\exists x.\ grain\ x)\ \&$
$\quad (\forall x.\ animal\ x \longrightarrow$
$\quad\quad ((\forall y.\ plant\ y \longrightarrow eats\ x\ y)\ \vee$
$\quad\quad (\forall y.\ animal\ y\ \&\ smaller\text{-}than\ y\ x\ \&$
$\quad\quad\quad (\exists z.\ plant\ z\ \&\ eats\ y\ z) \longrightarrow eats\ x\ y)))\ \&$
$\quad (\forall x\ y.\ bird\ y\ \&\ (snail\ x \vee caterpillar\ x) \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$\quad (\forall x\ y.\ bird\ x\ \&\ fox\ y \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$\quad (\forall x\ y.\ fox\ x\ \&\ wolf\ y \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$\quad (\forall x\ y.\ wolf\ x\ \&\ (fox\ y \vee grain\ y) \longrightarrow {\sim}eats\ x\ y)\ \&$
$\quad (\forall x\ y.\ bird\ x\ \&\ caterpillar\ y \longrightarrow eats\ x\ y)\ \&$
$\quad (\forall x\ y.\ bird\ x\ \&\ snail\ y \longrightarrow {\sim}eats\ x\ y)\ \&$
$\quad (\forall x.\ (caterpillar\ x \vee snail\ x) \longrightarrow (\exists y.\ plant\ y\ \&\ eats\ x\ y))$
$\quad \longrightarrow (\exists x\ y.\ animal\ x\ \&\ animal\ y\ \&\ (\exists z.\ grain\ z\ \&\ eats\ y\ z\ \&\ eats\ x\ y))$
⟨*proof*⟩

The Los problem. Circulated by John Harrison

lemma $(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z \dashrightarrow P\ x\ z)\ \&$
$\quad (\forall x\ y\ z.\ Q\ x\ y\ \&\ Q\ y\ z \dashrightarrow Q\ x\ z)\ \&$
$\quad (\forall x\ y.\ P\ x\ y \dashrightarrow P\ y\ x)\ \&$
$\quad (\forall x\ y.\ P\ x\ y\ |\ Q\ x\ y)$
$\quad \dashrightarrow (\forall x\ y.\ P\ x\ y)\ |\ (\forall x\ y.\ Q\ x\ y)$
⟨*proof*⟩

A similar example, suggested by Johannes Schumann and credited to Pelletier

lemma $(\forall x\ y\ z.\ P\ x\ y \dashrightarrow P\ y\ z \dashrightarrow P\ x\ z) \dashrightarrow$
$\quad (\forall x\ y\ z.\ Q\ x\ y \dashrightarrow Q\ y\ z \dashrightarrow Q\ x\ z) \dashrightarrow$
$\quad (\forall x\ y.\ Q\ x\ y \dashrightarrow Q\ y\ x) \dashrightarrow (\forall x\ y.\ P\ x\ y\ |\ Q\ x\ y) \dashrightarrow$
$\quad (\forall x\ y.\ P\ x\ y)\ |\ (\forall x\ y.\ Q\ x\ y)$
⟨*proof*⟩

Problem 50. What has this to do with equality?

lemma $(\forall x.\ P\ a\ x \mid (\forall y.\ P\ x\ y))\ \texttt{-->}\ (\exists x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

Problem 54: NOT PROVED

lemma $(\forall y{::}'a.\ \exists z.\ \forall x.\ F\ x\ z = (x{=}y))\ \texttt{-->}$
$\sim (\exists w.\ \forall x.\ F\ x\ w = (\forall u.\ F\ x\ u\ \texttt{-->}\ (\exists y.\ F\ y\ u\ \&\ \sim (\exists z.\ F\ z\ u\ \&\ F\ z\ y))))$
⟨*proof*⟩

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988). *meson* cannot report who killed Agatha.

lemma *lives agatha* & *lives butler* & *lives charles* &
(*killed agatha agatha* | *killed butler agatha* | *killed charles agatha*) &
($\forall$*x y*. *killed x y* --> *hates x y* & ~*richer x y*) &
($\forall$*x*. *hates agatha x* --> ~*hates charles x*) &
(*hates agatha agatha* & *hates agatha charles*) &
($\forall$*x*. *lives x* & ~*richer x agatha* --> *hates butler x*) &
($\forall$*x*. *hates agatha x* --> *hates butler x*) &
($\forall$*x*. ~*hates x agatha* | ~*hates x butler* | ~*hates x charles*) -->
($\exists$*x*. *killed x agatha*)
⟨*proof*⟩

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ \texttt{-->}\ P\ x\ z)\quad \texttt{-->}\quad P\ (f\ a\ b)\ (f\ a\ c)$
⟨*proof*⟩

Problem 58: Challenge found on info-hol

lemma $\forall P\ Q\ R\ x.\ \exists v\ w.\ \forall y\ z.\ P\ x\ \&\ Q\ y\ \texttt{-->}\ (P\ v \mid R\ w)\ \&\ (R\ z\ \texttt{-->}\ Q\ v)$
⟨*proof*⟩

Problem 59

lemma $(\forall x.\ P\ x = ({\sim}P(f\ x)))\ \texttt{-->}\ (\exists x.\ P\ x\ \&\ {\sim}P(f\ x))$
⟨*proof*⟩

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y\ \texttt{-->}\ P\ z\ (f\ x))\ \&\ P\ x\ y)$
⟨*proof*⟩

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ \texttt{-->}\ p(f\ x))\ \texttt{-->}\ p(f(f\ x)))\ =$
$(\forall x.\ ({\sim}\ p\ a \mid p\ x \mid p(f(f\ x)))\ \&$
$({\sim}\ p\ a \mid {\sim}\ p(f\ x) \mid p(f(f\ x))))$
⟨*proof*⟩

* Charles Morgan's problems *

lemma
  assumes a: $\forall x\ y.\ \ T(i\ x(i\ y\ x))$
    and b: $\forall x\ y\ z.\ T(i\ (i\ x\ (i\ y\ z))\ (i\ (i\ x\ y)\ (i\ x\ z)))$
    and c: $\forall x\ y.\ \ T(i\ (i\ (n\ x)\ (n\ y))\ (i\ y\ x))$
    and c': $\forall x\ y.\ \ T(i\ (i\ y\ x)\ (i\ (n\ x)\ (n\ y)))$
    and d: $\forall x\ y.\ \ T(i\ x\ y)\ \&\ T\ x\ {-}{-}{>}\ T\ y$
  shows *True*
$\langle proof \rangle$

Problem 71, as found in TPTP (SYN007+1.005)

lemma $p1 = (p2 = (p3 = (p4 = (p5 = (p1 = (p2 = (p3 = (p4 = p5))))))))$
$\langle proof \rangle$

end

39 Set Theory examples: Cantor's Theorem, Schröder-Bernstein Theorem, etc.

theory *set* **imports** *Main* **begin**

These two are cited in Benzmueller and Kohlhase's system description of LEO, CADE-15, 1998 (pages 139-143) as theorems LEO could not prove.

lemma $(X = Y \cup Z) =$
  $(Y \subseteq X \wedge Z \subseteq X \wedge (\forall V.\ Y \subseteq V \wedge Z \subseteq V \longrightarrow X \subseteq V))$
  $\langle proof \rangle$

lemma $(X = Y \cap Z) =$
  $(X \subseteq Y \wedge X \subseteq Z \wedge (\forall V.\ V \subseteq Y \wedge V \subseteq Z \longrightarrow V \subseteq X))$
  $\langle proof \rangle$

Trivial example of term synthesis: apparently hard for some provers!

lemma $a \neq b \Longrightarrow a \in ?X \wedge b \notin ?X$
  $\langle proof \rangle$

39.1 Examples for the *blast* paper

lemma $(\bigcup x \in C.\ f\ x \cup g\ x) = \bigcup (f\ `\ C) \ \cup\ \bigcup (g\ `\ C)$
  — Union-image, called *Un-Union-image* in Main HOL
  $\langle proof \rangle$

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
  $\langle proof \rangle$

lemma *singleton-example-1*:
  $\bigwedge S{::}'a\ set\ set.\ \forall x \in S.\ \forall y \in S.\ x \subseteq y \Longrightarrow \exists z.\ S \subseteq \{z\}$
  $\langle proof \rangle$

lemma *singleton-example-2*:
$\forall x \in S. \bigcup S \subseteq x \Longrightarrow \exists z.\ S \subseteq \{z\}$
— Variant of the problem above.
⟨*proof*⟩

lemma $\exists!x.\ f\ (g\ x) = x \Longrightarrow \exists!y.\ g\ (f\ y) = y$
— A unique fixpoint theorem — *fast*/*best*/*meson* all fail.
⟨*proof*⟩

39.2 Cantor's Theorem: There is no surjection from a set to its powerset

lemma *cantor1*: $\neg\ (\exists f{::}\ 'a \Rightarrow 'a\ set.\ \forall S.\ \exists x.\ f\ x = S)$
— Requires best-first search because it is undirectional.
⟨*proof*⟩

lemma $\forall f{::}\ 'a \Rightarrow 'a\ set.\ \forall x.\ f\ x \neq ?S\ f$
— This form displays the diagonal term.
⟨*proof*⟩

lemma $?S \notin range\ (f :: 'a \Rightarrow 'a\ set)$
— This form exploits the set constructs.
⟨*proof*⟩

lemma $?S \notin range\ (f :: 'a \Rightarrow 'a\ set)$
— Or just this!
⟨*proof*⟩

39.3 The Schröder-Berstein Theorem

lemma *disj-lemma*: $-\ (f\ `\ X) = g\ `\ (-X) \Longrightarrow f\ a = g\ b \Longrightarrow a \in X \Longrightarrow b \in X$
⟨*proof*⟩

lemma *surj-if-then-else*:
$-(f\ `\ X) = g\ `\ (-X) \Longrightarrow surj\ (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g\ z)$
⟨*proof*⟩

lemma *bij-if-then-else*:
$inj\text{-}on\ f\ X \Longrightarrow inj\text{-}on\ g\ (-X) \Longrightarrow -(f\ `\ X) = g\ `\ (-X) \Longrightarrow$
$h = (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g\ z) \Longrightarrow inj\ h \wedge surj\ h$
⟨*proof*⟩

lemma *decomposition*: $\exists X.\ X = -\ (g\ `\ (-\ (f\ `\ X)))$
⟨*proof*⟩

theorem *Schroeder-Bernstein*:
$inj\ (f :: 'a \Rightarrow 'b) \Longrightarrow inj\ (g :: 'b \Rightarrow 'a)$
$\Longrightarrow \exists h{::}\ 'a \Rightarrow 'b.\ inj\ h \wedge surj\ h$

$\langle proof \rangle$

39.4 A simple party theorem

At any party there are two people who know the same number of people. Provided the party consists of at least two people and the knows relation is symmetric. Knowing yourself does not count — otherwise knows needs to be reflexive. (From Freek Wiedijk's talk at TPHOLs 2007.)

lemma *equal-number-of-acquaintances*:
assumes *Domain R <= A* **and** *sym R* **and** *card A* $\geq$ *2*
shows $\neg$ *inj-on* (*%a. card*(*R* `` {*a*} − {*a*})) *A*
$\langle proof \rangle$

From W. W. Bledsoe and Guohui Feng, SET-VAR. JAR 11 (3), 1993, pages 293-314.

Isabelle can prove the easy examples without any special mechanisms, but it can't prove the hard ones.

lemma $\exists A. (\forall x \in A. x \leq (0{::}int))$
— Example 1, page 295.
$\langle proof \rangle$

lemma $D \in F \Longrightarrow \exists G. \forall A \in G. \exists B \in F. A \subseteq B$
— Example 2.
$\langle proof \rangle$

lemma $P\ a \Longrightarrow \exists A. (\forall x \in A. P\ x) \wedge (\exists y. y \in A)$
— Example 3.
$\langle proof \rangle$

lemma $a < b \wedge b < (c{::}int) \Longrightarrow \exists A. a \notin A \wedge b \in A \wedge c \notin A$
— Example 4.
$\langle proof \rangle$

lemma $P\ (f\ b) \Longrightarrow \exists s\ A. (\forall x \in A. P\ x) \wedge f\ s \in A$
— Example 5, page 298.
$\langle proof \rangle$

lemma $P\ (f\ b) \Longrightarrow \exists s\ A. (\forall x \in A. P\ x) \wedge f\ s \in A$
— Example 6.
$\langle proof \rangle$

lemma $\exists A. a \notin A$
— Example 7.
$\langle proof \rangle$

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.

⟨*proof*⟩

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.
⟨*proof*⟩

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.
⟨*proof*⟩

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.
⟨*proof*⟩

end

40 Meson test cases

theory *Meson-Test*
imports *Main*
begin

WARNING: there are many potential conflicts between variables used below and constants declared in HOL!

hide *const subset member quotient between*

Test data for the MESON proof procedure (Excludes the equality problems 51, 52, 56, 58)

40.1 Interactive examples

⟨*ML*⟩

MORE and MUCH HARDER test data for the MESON proof procedure (courtesy John Harrison).

abbreviation *EQU001-0-ax equal* $\equiv (\forall X.\ equal(X{::}'a,X))$ &
$(\forall Y\ X.\ equal(X{::}'a,Y)$ --> $equal(Y{::}'a,X))$ &
$(\forall Y\ X\ Z.\ equal(X{::}'a,Y)$ & $equal(Y{::}'a,Z)$ --> $equal(X{::}'a,Z))$

abbreviation *BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum* $\equiv$
$(\forall X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall Y\ X\ Z.\ sum(X::'a,Y,Z) \ {-}{-}{>}\ sum(Y::'a,X,Z))$ &
$(\forall Y\ X\ Z.\ product(X::'a,Y,Z) \ {-}{-}{>}\ product(Y::'a,X,Z))$ &
$(\forall X.\ sum(additive\text{-}identity::'a,X,X))$ &
$(\forall X.\ sum(X::'a,additive\text{-}identity,X))$ &
$(\forall X.\ product(multiplicative\text{-}identity::'a,X,X))$ &
$(\forall X.\ product(X::'a,multiplicative\text{-}identity,X))$ &
$(\forall Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X::'a,Y,V1)$ & $product(X::'a,Z,V2)$ & $sum(Y::'a,Z,V3)$ & $product(X::'a,V3,V4) \ {-}{-}{>}\ sum(V1::'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X::'a,Y,V1)$ & $product(X::'a,Z,V2)$ & $sum(Y::'a,Z,V3)$ & $sum(V1::'a,V2,V4) \ {-}{-}{>}\ product(X::'a,V3,V4))$ &
$(\forall Y\ Z\ V3\ X\ V1\ V2\ V4.\ product(Y::'a,X,V1)$ & $product(Z::'a,X,V2)$ & $sum(Y::'a,Z,V3)$ & $product(V3::'a,X,V4) \ {-}{-}{>}\ sum(V1::'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ V3\ X\ V4.\ product(Y::'a,X,V1)$ & $product(Z::'a,X,V2)$ & $sum(Y::'a,Z,V3)$ & $sum(V1::'a,V2,V4) \ {-}{-}{>}\ product(V3::'a,X,V4))$ &
$(\forall Y\ Z\ X\ V3\ V1\ V2\ V4.\ sum(X::'a,Y,V1)$ & $sum(X::'a,Z,V2)$ & $product(Y::'a,Z,V3)$ & $sum(X::'a,V3,V4) \ {-}{-}{>}\ product(V1::'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ X\ V3\ V4.\ sum(X::'a,Y,V1)$ & $sum(X::'a,Z,V2)$ & $product(Y::'a,Z,V3)$ & $product(V1::'a,V2,V4) \ {-}{-}{>}\ sum(X::'a,V3,V4))$ &
$(\forall Y\ Z\ V3\ X\ V1\ V2\ V4.\ sum(Y::'a,X,V1)$ & $sum(Z::'a,X,V2)$ & $product(Y::'a,Z,V3)$ & $sum(V3::'a,X,V4) \ {-}{-}{>}\ product(V1::'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ V3\ X\ V4.\ sum(Y::'a,X,V1)$ & $sum(Z::'a,X,V2)$ & $product(Y::'a,Z,V3)$ & $product(V1::'a,V2,V4) \ {-}{-}{>}\ sum(V3::'a,X,V4))$ &
$(\forall X.\ sum(INVERSE(X),X,multiplicative\text{-}identity))$ &
$(\forall X.\ sum(X::'a,INVERSE(X),multiplicative\text{-}identity))$ &
$(\forall X.\ product(INVERSE(X),X,additive\text{-}identity))$ &
$(\forall X.\ product(X::'a,INVERSE(X),additive\text{-}identity))$ &
$(\forall X\ Y\ U\ V.\ sum(X::'a,Y,U)$ & $sum(X::'a,Y,V) \ {-}{-}{>}\ equal(U::'a,V))$ &
$(\forall X\ Y\ U\ V.\ product(X::'a,Y,U)$ & $product(X::'a,Y,V) \ {-}{-}{>}\ equal(U::'a,V))$

abbreviation *BOO002-0-eq INVERSE multiply add product sum equal* $\equiv$
$(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $sum(X::'a,W,Z) \ {-}{-}{>}\ sum(Y::'a,W,Z))$ &
$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $sum(W::'a,X,Z) \ {-}{-}{>}\ sum(W::'a,Y,Z))$ &
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $sum(W::'a,Z,X) \ {-}{-}{>}\ sum(W::'a,Z,Y))$ &
$(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z) \ {-}{-}{>}\ product(Y::'a,W,Z))$
&
$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z) \ {-}{-}{>}\ product(W::'a,Y,Z))$
&
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X) \ {-}{-}{>}\ product(W::'a,Z,Y))$
&
$(\forall X\ Y\ W.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(add(X::'a,W),add(Y::'a,W)))$ &
$(\forall X\ W\ Y.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(add(W::'a,X),add(W::'a,Y)))$ &
$(\forall X\ Y\ W.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall X\ W\ Y.\ equal(X::'a,Y) \ {-}{-}{>}\ equal(multiply(W::'a,X),multiply(W::'a,Y)))$

&
$(\forall X\ Y.\ equal(X{::}'a,Y) \ {-}{-}{>}\ equal(INVERSE(X),INVERSE(Y)))$

lemma *BOO003-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim product(x{::}'a,x,x))\ {-}{-}{>}\ False$
⟨*proof*⟩

lemma *BOO004-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim sum(x{::}'a,x,x))\ {-}{-}{>}\ False$
⟨*proof*⟩

lemma *BOO005-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim sum(x{::}'a,multiplicative\text{-}identity,multiplicative\text{-}identity))\ {-}{-}{>}\ False$
⟨*proof*⟩

lemma *BOO006-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim product(x{::}'a,additive\text{-}identity,additive\text{-}identity))\ {-}{-}{>}\ False$
⟨*proof*⟩

lemma *BOO011-1*:
EQU001-0-ax equal &
BOO002-0-ax equal INVERSE multiplicative-identity additive-identity multiply product add sum &
BOO002-0-eq INVERSE multiply add product sum equal &
$(\sim equal(INVERSE(additive\text{-}identity),multiplicative\text{-}identity))\ {-}{-}{>}\ False$
⟨*proof*⟩

abbreviation *CAT003-0-ax f1 compos codomain domain equal there-exists equiv-*

alent ≡
 $(\forall\, Y\, X.\ \mathit{equivalent}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{there\text{-}exists}(X))$ &
 $(\forall\, X\ Y.\ \mathit{equivalent}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equal}(X{::}'a,Y))$ &
 $(\forall\, X\ Y.\ \mathit{there\text{-}exists}(X)\ \&\ \mathit{equal}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equivalent}(X{::}'a,Y))$ &
 $(\forall\, X.\ \mathit{there\text{-}exists}(\mathit{domain}(X))\ {-}{-}{>}\ \mathit{there\text{-}exists}(X))$ &
 $(\forall\, X.\ \mathit{there\text{-}exists}(\mathit{codomain}(X))\ {-}{-}{>}\ \mathit{there\text{-}exists}(X))$ &
 $(\forall\, Y\, X.\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y))\ {-}{-}{>}\ \mathit{there\text{-}exists}(\mathit{domain}(X)))$ &
 $(\forall\, X\ Y.\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y))\ {-}{-}{>}\ \mathit{equal}(\mathit{domain}(X),\mathit{codomain}(Y)))$
&
 $(\forall\, X\, Y.\ \mathit{there\text{-}exists}(\mathit{domain}(X))\ \&\ \mathit{equal}(\mathit{domain}(X),\mathit{codomain}(Y))\ {-}{-}{>}\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y)))$
&
 $(\forall\, X\ Y\ Z.\ \mathit{equal}(\mathit{compos}(X{::}'a,\mathit{compos}(Y{::}'a,Z)),\mathit{compos}(\mathit{compos}(X{::}'a,Y),Z)))$
&
 $(\forall\, X.\ \mathit{equal}(\mathit{compos}(X{::}'a,\mathit{domain}(X)),X))$ &
 $(\forall\, X.\ \mathit{equal}(\mathit{compos}(\mathit{codomain}(X),X),X))$ &
 $(\forall\, X\ Y.\ \mathit{equivalent}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{there\text{-}exists}(Y))$ &
 $(\forall\, X\, Y.\ \mathit{there\text{-}exists}(X)\ \&\ \mathit{there\text{-}exists}(Y)\ \&\ \mathit{equal}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equivalent}(X{::}'a,Y))$
&
 $(\forall\, Y\, X.\ \mathit{there\text{-}exists}(\mathit{compos}(X{::}'a,Y))\ {-}{-}{>}\ \mathit{there\text{-}exists}(\mathit{codomain}(X)))$ &
 $(\forall\, X\ Y.\ \mathit{there\text{-}exists}(\mathit{f1}(X{::}'a,Y))\ |\ \mathit{equal}(X{::}'a,Y))$ &
 $(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,\mathit{f1}(X{::}'a,Y))\ |\ \mathit{equal}(Y{::}'a,\mathit{f1}(X{::}'a,Y))\ |\ \mathit{equal}(X{::}'a,Y))$
&
 $(\forall\, X\, Y.\ \mathit{equal}(X{::}'a,\mathit{f1}(X{::}'a,Y))\ \&\ \mathit{equal}(Y{::}'a,\mathit{f1}(X{::}'a,Y))\ {-}{-}{>}\ \mathit{equal}(X{::}'a,Y))$

abbreviation *CAT003-0-eq f1 compos codomain domain equivalent there-exists equal* ≡
 $(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,Y)\ \&\ \mathit{there\text{-}exists}(X)\ {-}{-}{>}\ \mathit{there\text{-}exists}(Y))$ &
 $(\forall\, X\ Y\ Z.\ \mathit{equal}(X{::}'a,Y)\ \&\ \mathit{equivalent}(X{::}'a,Z)\ {-}{-}{>}\ \mathit{equivalent}(Y{::}'a,Z))$ &
 $(\forall\, X\ Z\ Y.\ \mathit{equal}(X{::}'a,Y)\ \&\ \mathit{equivalent}(Z{::}'a,X)\ {-}{-}{>}\ \mathit{equivalent}(Z{::}'a,Y))$ &
 $(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{domain}(X),\mathit{domain}(Y)))$ &
 $(\forall\, X\ Y.\ \mathit{equal}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{codomain}(X),\mathit{codomain}(Y)))$ &
 $(\forall\, X\ Y\ Z.\ \mathit{equal}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{compos}(X{::}'a,Z),\mathit{compos}(Y{::}'a,Z)))$ &
 $(\forall\, X\ Z\ Y.\ \mathit{equal}(X{::}'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{compos}(Z{::}'a,X),\mathit{compos}(Z{::}'a,Y)))$ &
 $(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-}{-}{>}\ \mathit{equal}(\mathit{f1}(A{::}'a,C),\mathit{f1}(B{::}'a,C)))$ &
 $(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-}{-}{>}\ \mathit{equal}(\mathit{f1}(F'{::}'a,D),\mathit{f1}(F'{::}'a,E)))$

lemma *CAT001-3*:
 EQU001-0-ax equal &
 CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
 CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
 $(\mathit{there\text{-}exists}(\mathit{compos}(a{::}'a,b)))$ &
 $(\forall\, Y\, X\, Z.\ \mathit{equal}(\mathit{compos}(\mathit{compos}(a{::}'a,b),X),Y)\ \&\ \mathit{equal}(\mathit{compos}(\mathit{compos}(a{::}'a,b),Z),Y)$
$\ {-}{-}{>}\ \mathit{equal}(X{::}'a,Z))$ &
 $(\mathit{there\text{-}exists}(\mathit{compos}(b{::}'a,h)))$ &
 $(\mathit{equal}(\mathit{compos}(b{::}'a,h),\mathit{compos}(b{::}'a,g)))$ &
 $({\sim}\mathit{equal}(h{::}'a,g))\ {-}{-}{>}\ \mathit{False}$
 $\langle \mathit{proof} \rangle$

lemma *CAT003-3*:
EQU001-0-ax equal &
CAT003-0-ax f1 compos codomain domain equal there-exists equivalent &
CAT003-0-eq f1 compos codomain domain equivalent there-exists equal &
(*there-exists*(*compos*(a::'a,b))) &
($\forall$ Y X Z. *equal*(*compos*(X::'a,*compos*(a::'a,b)),Y) & *equal*(*compos*(Z::'a,*compos*(a::'a,b)),Y) −−> *equal*(X::'a,Z)) &
(*there-exists*(h)) &
(*equal*(*compos*(h::'a,a),*compos*(g::'a,a))) &
(~*equal*(g::'a,h)) −−> *False*
⟨*proof*⟩

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-map*(Y) −−> *defined*(X::'a,Z)) &
($\forall$ X. *identity-map*(*domain*(X))) &
($\forall$ X. *identity-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-map*(X) −−> *product*(X::'a,Y,Y)) &
($\forall$ Y X. *defined*(X::'a,Y) & *identity-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)) &

(∀ X Y. equal(X::'a,Y) --> equal(domain(X),domain(Y))) &
(∀ X Y. equal(X::'a,Y) --> equal(codomain(X),codomain(Y))) &
(∀ X Y. equal(X::'a,Y) & identity-map(X) --> identity-map(Y)) &
(∀ X Y Z. equal(X::'a,Y) & defined(X::'a,Z) --> defined(Y::'a,Z)) &
(∀ X Z Y. equal(X::'a,Y) & defined(Z::'a,X) --> defined(Z::'a,Y)) &
(∀ X Z Y. equal(X::'a,Y) --> equal(compos(Z::'a,X),compos(Z::'a,Y))) &
(∀ X Y Z. equal(X::'a,Y) --> equal(compos(X::'a,Z),compos(Y::'a,Z)))

lemma *CAT005-1*:
EQU001-0-ax equal &
CAT001-0-ax equal codomain domain identity-map compos product defined &
CAT001-0-eq compos defined identity-map codomain domain product equal &
(defined(a::'a,d)) &
(identity-map(d)) &
(~equal(domain(a),d)) --> False
⟨proof⟩

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
⟨proof⟩

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
⟨proof⟩

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(

&
$(\forall\, Y.\ {\sim}\mathit{equal}(Y{::}'a,\mathit{apply}(\mathit{combinator}{::}'a,Y)))\ {-}{-}{>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL023-1*:
EQU001-0-ax equal &
$(\forall\, X\ Y\ Z.\ \mathit{equal}(\mathit{apply}(\mathit{apply}(\mathit{apply}(b{::}'a,X),Y),Z),\mathit{apply}(X{::}'a,\mathit{apply}(Y{::}'a,Z))))$
&
$(\forall\, X\ Y\ Z.\ \mathit{equal}(\mathit{apply}(\mathit{apply}(\mathit{apply}(n{::}'a,X),Y),Z),\mathit{apply}(\mathit{apply}(\mathit{apply}(X{::}'a,Z),Y),Z)))$
&
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-}{-}{>}\ \mathit{equal}(\mathit{apply}(A{::}'a,C),\mathit{apply}(B{::}'a,C)))$ &
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-}{-}{>}\ \mathit{equal}(\mathit{apply}(F'{::}'a,D),\mathit{apply}(F'{::}'a,E)))$ &
$(\forall\, Y.\ {\sim}\mathit{equal}(Y{::}'a,\mathit{apply}(\mathit{combinator}{::}'a,Y)))\ {-}{-}{>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL032-1*:
EQU001-0-ax equal &
$(\forall\, X.\ \mathit{equal}(\mathit{apply}(m{::}'a,X),\mathit{apply}(X{::}'a,X)))$ &
$(\forall\, Y\ X\ Z.\ \mathit{equal}(\mathit{apply}(\mathit{apply}(\mathit{apply}(q{::}'a,X),Y),Z),\mathit{apply}(Y{::}'a,\mathit{apply}(X{::}'a,Z))))$
&
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-}{-}{>}\ \mathit{equal}(\mathit{apply}(A{::}'a,C),\mathit{apply}(B{::}'a,C)))$ &
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-}{-}{>}\ \mathit{equal}(\mathit{apply}(F'{::}'a,D),\mathit{apply}(F'{::}'a,E)))$ &
$(\forall\, G\ H.\ \mathit{equal}(G{::}'a,H)\ {-}{-}{>}\ \mathit{equal}(f(G),f(H)))$ &
$(\forall\, Y.\ {\sim}\mathit{equal}(\mathit{apply}(Y{::}'a,f(Y)),\mathit{apply}(f(Y),\mathit{apply}(Y{::}'a,f(Y)))))\ {-}{-}{>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

lemma *COL052-2*:
EQU001-0-ax equal &
$(\forall\, X\ Y\ W.\ \mathit{equal}(\mathit{response}(\mathit{compos}(X{::}'a,Y),W),\mathit{response}(X{::}'a,\mathit{response}(Y{::}'a,W))))$
&
$(\forall\, X\ Y.\ \mathit{agreeable}(X)\ {-}{-}{>}\ \mathit{equal}(\mathit{response}(X{::}'a,\mathit{common\text{-}bird}(Y)),\mathit{response}(Y{::}'a,\mathit{common\text{-}bird}(Y))))$
&
$(\forall\, Z\ X.\ \mathit{equal}(\mathit{response}(X{::}'a,Z),\mathit{response}(\mathit{compatible}(X),Z))\ {-}{-}{>}\ \mathit{agreeable}(X))$
&
$(\forall\, A\ B.\ \mathit{equal}(A{::}'a,B)\ {-}{-}{>}\ \mathit{equal}(\mathit{common\text{-}bird}(A),\mathit{common\text{-}bird}(B)))$ &
$(\forall\, C\ D.\ \mathit{equal}(C{::}'a,D)\ {-}{-}{>}\ \mathit{equal}(\mathit{compatible}(C),\mathit{compatible}(D)))$ &
$(\forall\, Q\ R.\ \mathit{equal}(Q{::}'a,R)\ \&\ \mathit{agreeable}(Q)\ {-}{-}{>}\ \mathit{agreeable}(R))$ &
$(\forall\, A\ B\ C.\ \mathit{equal}(A{::}'a,B)\ {-}{-}{>}\ \mathit{equal}(\mathit{compos}(A{::}'a,C),\mathit{compos}(B{::}'a,C)))$ &
$(\forall\, D\ F'\ E.\ \mathit{equal}(D{::}'a,E)\ {-}{-}{>}\ \mathit{equal}(\mathit{compos}(F'{::}'a,D),\mathit{compos}(F'{::}'a,E)))$ &
$(\forall\, G\ H\ I'.\ \mathit{equal}(G{::}'a,H)\ {-}{-}{>}\ \mathit{equal}(\mathit{response}(G{::}'a,I'),\mathit{response}(H{::}'a,I')))$ &
$(\forall\, J\ L\ K'.\ \mathit{equal}(J{::}'a,K')\ {-}{-}{>}\ \mathit{equal}(\mathit{response}(L{::}'a,J),\mathit{response}(L{::}'a,K')))$ &
$(\mathit{agreeable}(c))$ &
$({\sim}\mathit{agreeable}(a))$ &
$(\mathit{equal}(c{::}'a,\mathit{compos}(a{::}'a,b)))\ {-}{-}{>}\ \mathit{False}$
$\langle \mathit{proof} \rangle$

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨proof⟩

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
⟨proof⟩

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)) &

$(\forall X.\ \textit{algorithm}(X)\ \&\ \textit{program-decides}(X)\ {-}{-}{>}\ \textit{algorithm-program-decides}(X))$
$\&$
$(\forall Y\ X.\ \textit{program-halts2}(X{::}'a,Y)\ {-}{-}{>}\ \textit{program}(X))\ \&$
$(\forall X\ Y.\ \textit{program-halts2}(X{::}'a,Y)\ {-}{-}{>}\ \textit{halts2}(X{::}'a,Y))\ \&$
$(\forall X\ Y.\ \textit{program}(X)\ \&\ \textit{halts2}(X{::}'a,Y)\ {-}{-}{>}\ \textit{program-halts2}(X{::}'a,Y))\ \&$
$(\forall W\ X\ Y\ Z.\ \textit{halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}\ \textit{halts3}(X{::}'a,Y,Z))\ \&$
$(\forall Y\ Z\ X\ W.\ \textit{halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}\ \textit{outputs}(X{::}'a,W))\ \&$
$(\forall Y\ Z\ X\ W.\ \textit{halts3}(X{::}'a,Y,Z)\ \&\ \textit{outputs}(X{::}'a,W)\ {-}{-}{>}\ \textit{halts3-outputs}(X{::}'a,Y,Z,W))$
$\&$
$(\forall Y\ X.\ \textit{program-not-halts2}(X{::}'a,Y)\ {-}{-}{>}\ \textit{program}(X))\ \&$
$(\forall X\ Y.\ {\sim}(\textit{program-not-halts2}(X{::}'a,Y)\ \&\ \textit{halts2}(X{::}'a,Y)))\ \&$
$(\forall X\ Y.\ \textit{program}(X)\ {-}{-}{>}\ \textit{program-not-halts2}(X{::}'a,Y)\ |\ \textit{halts2}(X{::}'a,Y))\ \&$
$(\forall W\ X\ Y.\ \textit{halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{halts2}(X{::}'a,Y))\ \&$
$(\forall Y\ X\ W.\ \textit{halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{outputs}(X{::}'a,W))\ \&$
$(\forall Y\ X\ W.\ \textit{halts2}(X{::}'a,Y)\ \&\ \textit{outputs}(X{::}'a,W)\ {-}{-}{>}\ \textit{halts2-outputs}(X{::}'a,Y,W))$
$\&$
$(\forall X\ W\ Y\ Z.\ \textit{program-halts2-halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}\ \textit{program-halts2}(Y{::}'a,Z))$
$\&$
$(\forall X\ Y\ Z\ W.\ \textit{program-halts2-halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}\ \textit{halts3-outputs}(X{::}'a,Y,Z,W))$
$\&$
$(\forall X\ Y\ Z\ W.\ \textit{program-halts2}(Y{::}'a,Z)\ \&\ \textit{halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}$
$\textit{program-halts2-halts3-outputs}(X{::}'a,Y,Z,W))\ \&$
$(\forall X\ W\ Y\ Z.\ \textit{program-not-halts2-halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}\ \textit{program-not-halts2}(Y{::}'a,Z))$
$\&$
$(\forall X\ Y\ Z\ W.\ \textit{program-not-halts2-halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}\ \textit{halts3-outputs}(X{::}'a,Y,Z,W))$
$\&$
$(\forall X\ Y\ Z\ W.\ \textit{program-not-halts2}(Y{::}'a,Z)\ \&\ \textit{halts3-outputs}(X{::}'a,Y,Z,W)\ {-}{-}{>}$
$\textit{program-not-halts2-halts3-outputs}(X{::}'a,Y,Z,W))\ \&$
$(\forall X\ W\ Y.\ \textit{program-halts2-halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{program-halts2}(Y{::}'a,Y))$
$\&$
$(\forall X\ Y\ W.\ \textit{program-halts2-halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{halts2-outputs}(X{::}'a,Y,W))$
$\&$
$(\forall X\ Y\ W.\ \textit{program-halts2}(Y{::}'a,Y)\ \&\ \textit{halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{program-halts2-halts2-outputs}(X{::}'a$
$\&$
$(\forall X\ W\ Y.\ \textit{program-not-halts2-halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{program-not-halts2}(Y{::}'a,Y))$
$\&$
$(\forall X\ Y\ W.\ \textit{program-not-halts2-halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}\ \textit{halts2-outputs}(X{::}'a,Y,W))$
$\&$
$(\forall X\ Y\ W.\ \textit{program-not-halts2}(Y{::}'a,Y)\ \&\ \textit{halts2-outputs}(X{::}'a,Y,W)\ {-}{-}{>}$
$\textit{program-not-halts2-halts2-outputs}(X{::}'a,Y,W))\ \&$
$(\forall X.\ \textit{algorithm-program-decides}(X)\ {-}{-}{>}\ \textit{program-program-decides}(c1))\ \&$
$(\forall W\ Y\ Z.\ \textit{program-program-decides}(W)\ {-}{-}{>}\ \textit{program-halts2-halts3-outputs}(W{::}'a,Y,Z,\textit{good}))$
$\&$
$(\forall W\ Y\ Z.\ \textit{program-program-decides}(W)\ {-}{-}{>}\ \textit{program-not-halts2-halts3-outputs}(W{::}'a,Y,Z,\textit{bad}))$
$\&$
$(\forall W.\ \textit{program}(W)\ \&\ \textit{program-halts2-halts3-outputs}(W{::}'a,\textit{f3}(W),\textit{f3}(W),\textit{good})$
$\&\ \textit{program-not-halts2-halts3-outputs}(W{::}'a,\textit{f3}(W),\textit{f3}(W),\textit{bad})\ {-}{-}{>}\ \textit{program}(c2))$
$\&$
$(\forall W\ Y.\ \textit{program}(W)\ \&\ \textit{program-halts2-halts3-outputs}(W{::}'a,\textit{f3}(W),\textit{f3}(W),\textit{good})$

& *program-not-halts2-halts3-outputs*$(W::'a,f3(W),f3(W),bad)$ --> *program-halts2-halts2-outputs*$(c2::'a,Y,g$
&
(∀ W Y. *program*(W) & *program-halts2-halts3-outputs*$(W::'a,f3(W),f3(W),good)$
& *program-not-halts2-halts3-outputs*$(W::'a,f3(W),f3(W),bad)$ --> *program-not-halts2-halts2-outputs*$(c2::'a,$
&
(∀ V. *program*(V) & *program-halts2-halts2-outputs*$(V::'a,f4(V),good)$ & *program-not-halts2-halts2-outputs*$(V$
--> *program*(c3)) &
(∀ V Y. *program*(V) & *program-halts2-halts2-outputs*$(V::'a,f4(V),good)$ & *program-not-halts2-halts2-outputs*
& *program-halts2*$(Y::'a,Y)$ --> *halts2*$(c3::'a,Y)$) &
(∀ V Y. *program*(V) & *program-halts2-halts2-outputs*$(V::'a,f4(V),good)$ & *program-not-halts2-halts2-outputs*
--> *program-not-halts2-halts2-outputs*$(c3::'a,Y,bad)$) &
(*algorithm-program-decides*(c4)) --> *False*
⟨*proof*⟩

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*$(parent\text{-}of(X::'a,Y),or(C::'a,D))$)
&
(∀ X. *contradictory*$(negate(X),X)$) &
(∀ X. *contradictory*$(X::'a,negate(X))$) &
(∀ X. *siblings*$(left\text{-}child\text{-}of(X),right\text{-}child\text{-}of(X))$) &
(∀ D E. *equal*$(D::'a,E)$ --> *equal*$(left\text{-}child\text{-}of(D),left\text{-}child\text{-}of(E))$) &
(∀ 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*$(parent\text{-}of(N::'a,P),parent\text{-}of(O'::'a,P))$)
&
(∀ Q S' R. *equal*$(Q::'a,R)$ --> *equal*$(parent\text{-}of(S'::'a,Q),parent\text{-}of(S'::'a,R))$)
&
(∀ T' U. *equal*$(T'::'a,U)$ --> *equal*$(right\text{-}child\text{-}of(T'),right\text{-}child\text{-}of(U))$) &
(∀ 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*$(n\text{-}left::'a,or(EMPTY::'a,atom))$) &
(*failure-node*$(n\text{-}right::'a,or(EMPTY::'a,negate(atom)))$) &
(*equal*$(n\text{-}left::'a,left\text{-}child\text{-}of(n))$) &
(*equal*$(n\text{-}right::'a,right\text{-}child\text{-}of(n))$) &
(∀ Z. ~*failure-node*$(Z::'a,or(EMPTY::'a,EMPTY))$) --> *False*
⟨*proof*⟩

abbreviation *GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2 lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal between* ≡
(∀ X Y. between(X::'a,Y,X) --> equal(X::'a,Y)) &
(∀ V X Y Z. between(X::'a,Y,V) & between(Y::'a,Z,V) --> between(X::'a,Y,Z))
&
(∀ Y X V Z. between(X::'a,Y,Z) & between(X::'a,Y,V) --> equal(X::'a,Y) | between(X::'a,Z,V) | between(X::'a,V,Z)) &
(∀ Y X. equidistant(X::'a,Y,Y,X)) &
(∀ Z X Y. equidistant(X::'a,Y,Z,Z) --> equal(X::'a,Y)) &
(∀ X Y Z V V2 W. equidistant(X::'a,Y,Z,V) & equidistant(X::'a,Y,V2,W) --> equidistant(Z::'a,V,V2,W)) &
(∀ W X Z V Y. between(X::'a,W,V) & between(Y::'a,V,Z) --> between(X::'a,outer-pasch(W::'a,X,Y,Z,V
&
(∀ W X Y Z V. between(X::'a,W,V) & between(Y::'a,V,Z) --> between(Z::'a,W,outer-pasch(W::'a,X,Y,Z
&
(∀ W X Y Z V. between(X::'a,V,W) & between(Y::'a,V,Z) --> equal(X::'a,V) | between(X::'a,Z,euclid1(W::'a,X,Y,Z,V))) &
(∀ W X Y Z V. between(X::'a,V,W) & between(Y::'a,V,Z) --> equal(X::'a,V) | between(X::'a,Y,euclid2(W::'a,X,Y,Z,V))) &
(∀ W X Y Z V. between(X::'a,V,W) & between(Y::'a,V,Z) --> equal(X::'a,V) | between(euclid1(W::'a,X,Y,Z,V),W,euclid2(W::'a,X,Y,Z,V))) &
(∀ X1 Y1 X Y Z V Z1 V1. equidistant(X::'a,Y,X1,Y1) & equidistant(Y::'a,Z,Y1,Z1) & equidistant(X::'a,V,X1,V1) & equidistant(Y::'a,V,Y1,V1) & between(X::'a,Y,Z) & between(X1::'a,Y1,Z1) --> equal(X::'a,Y) | equidistant(Z::'a,V,Z1,V1)) &
(∀ X Y W V. between(X::'a,Y,extension(X::'a,Y,W,V))) &
(∀ X Y W V. equidistant(Y::'a,extension(X::'a,Y,W,V),W,V)) &
(~between(lower-dimension-point-1::'a,lower-dimension-point-2,lower-dimension-point-3))
&
(~between(lower-dimension-point-2::'a,lower-dimension-point-3,lower-dimension-point-1))
&
(~between(lower-dimension-point-3::'a,lower-dimension-point-1,lower-dimension-point-2))
&
(∀ Z X Y W V. equidistant(X::'a,W,X,V) & equidistant(Y::'a,W,Y,V) & equidistant(Z::'a,W,Z,V) --> between(X::'a,Y,Z) | between(Y::'a,Z,X) | between(Z::'a,X,Y) | equal(W::'a,V)) &
(∀ X Y Z X1 Z1 V. equidistant(V::'a,X,V,X1) & equidistant(V::'a,Z,V,Z1) & between(V::'a,X,Z) & between(X::'a,Y,Z) --> equidistant(V::'a,Y,Z,continuous(X::'a,Y,Z,X1,Z1,V)))
&
(∀ X Y Z X1 V Z1. equidistant(V::'a,X,V,X1) & equidistant(V::'a,Z,V,Z1) & between(V::'a,X,Z) & between(X::'a,Y,Z) --> between(X1::'a,continuous(X::'a,Y,Z,X1,Z1,V),Z1))

abbreviation *GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between equal* ≡
(∀ X Y W Z. equal(X::'a,Y) & between(X::'a,W,Z) --> between(Y::'a,W,Z))
&
(∀ X W Y Z. equal(X::'a,Y) & between(W::'a,X,Z) --> between(W::'a,Y,Z))
&

$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ between(W::'a,Z,X)\ {-}{-}{>}\ between(W::'a,Z,Y))$
&
$(\forall X\ Y\ V\ W\ Z.\ equal(X::'a,Y)\ \&\ equidistant(X::'a,V,W,Z)\ {-}{-}{>}\ equidistant(Y::'a,V,W,Z))$ &
$(\forall X\ V\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,X,W,Z)\ {-}{-}{>}\ equidistant(V::'a,Y,W,Z))$ &
$(\forall X\ V\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,X,Z)\ {-}{-}{>}\ equidistant(V::'a,W,Y,Z))$ &
$(\forall X\ V\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ equidistant(V::'a,W,Z,X)\ {-}{-}{>}\ equidistant(V::'a,W,Z,Y))$ &
$(\forall X\ Y\ V1\ V2\ V3\ V4.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(outer\text{-}pasch(X::'a,V1,V2,V3,V4),outer\text{-}pasch(Y::'a,V1,V$
&
$(\forall X\ V1\ Y\ V2\ V3\ V4.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(outer\text{-}pasch(V1::'a,X,V2,V3,V4),outer\text{-}pasch(V1::'a,Y,V$
&
$(\forall X\ V1\ V2\ Y\ V3\ V4.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(outer\text{-}pasch(V1::'a,V2,X,V3,V4),outer\text{-}pasch(V1::'a,V2,Y$
&
$(\forall X\ V1\ V2\ V3\ Y\ V4.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(outer\text{-}pasch(V1::'a,V2,V3,X,V4),outer\text{-}pasch(V1::'a,V2,V$
&
$(\forall X\ V1\ V2\ V3\ V4\ Y.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(outer\text{-}pasch(V1::'a,V2,V3,V4,X),outer\text{-}pasch(V1::'a,V2,V$
&
$(\forall A\ B\ C\ D\ E\ F'.\ equal(A::'a,B)\ {-}{-}{>}\ equal(euclid1(A::'a,C,D,E,F'),euclid1(B::'a,C,D,E,F')))$
&
$(\forall G\ I'\ H\ J\ K'\ L.\ equal(G::'a,H)\ {-}{-}{>}\ equal(euclid1(I'::'a,G,J,K',L),euclid1(I'::'a,H,J,K',L)))$
&
$(\forall M\ O'\ P\ N\ Q\ R.\ equal(M::'a,N)\ {-}{-}{>}\ equal(euclid1(O'::'a,P,M,Q,R),euclid1(O'::'a,P,N,Q,R)))$
&
$(\forall S'\ U\ V\ W\ T'\ X.\ equal(S'::'a,T')\ {-}{-}{>}\ equal(euclid1(U::'a,V,W,S',X),euclid1(U::'a,V,W,T',X)))$
&
$(\forall Y\ A1\ B1\ C1\ D1\ Z.\ equal(Y::'a,Z)\ {-}{-}{>}\ equal(euclid1(A1::'a,B1,C1,D1,Y),euclid1(A1::'a,B1,C1,D1,Z)$
&
$(\forall E1\ F1\ G1\ H1\ I1\ J1.\ equal(E1::'a,F1)\ {-}{-}{>}\ equal(euclid2(E1::'a,G1,H1,I1,J1),euclid2(F1::'a,G1,H1,I1,$
&
$(\forall K1\ M1\ L1\ N1\ O1\ P1.\ equal(K1::'a,L1)\ {-}{-}{>}\ equal(euclid2(M1::'a,K1,N1,O1,P1),euclid2(M1::'a,L1,N1,$
&
$(\forall Q1\ S1\ T1\ R1\ U1\ V1.\ equal(Q1::'a,R1)\ {-}{-}{>}\ equal(euclid2(S1::'a,T1,Q1,U1,V1),euclid2(S1::'a,T1,R1,U$
&
$(\forall W1\ Y1\ Z1\ A2\ X1\ B2.\ equal(W1::'a,X1)\ {-}{-}{>}\ equal(euclid2(Y1::'a,Z1,A2,W1,B2),euclid2(Y1::'a,Z1,A2$
&
$(\forall C2\ E2\ F2\ G2\ H2\ D2.\ equal(C2::'a,D2)\ {-}{-}{>}\ equal(euclid2(E2::'a,F2,G2,H2,C2),euclid2(E2::'a,F2,G2,H$
&
$(\forall X\ Y\ V1\ V2\ V3.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(extension(X::'a,V1,V2,V3),extension(Y::'a,V1,V2,V3)))$
&
$(\forall X\ V1\ Y\ V2\ V3.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(extension(V1::'a,X,V2,V3),extension(V1::'a,Y,V2,V3)))$
&
$(\forall X\ V1\ V2\ Y\ V3.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(extension(V1::'a,V2,X,V3),extension(V1::'a,V2,Y,V3)))$
&
$(\forall X\ V1\ V2\ V3\ Y.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(extension(V1::'a,V2,V3,X),extension(V1::'a,V2,V3,Y)))$
&
$(\forall X\ Y\ V1\ V2\ V3\ V4\ V5.\ equal(X::'a,Y)\ {-}{-}{>}\ equal(continuous(X::'a,V1,V2,V3,V4,V5),continuous(Y::'a,$

&
($\forall$ *X V1 Y V2 V3 V4 V5. equal(X::'a,Y) --> equal(continuous(V1::'a,X,V2,V3,V4,V5),continuous(V1::'a*
&
($\forall$ *X V1 V2 Y V3 V4 V5. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,X,V3,V4,V5),continuous(V1::'a*
&
($\forall$ *X V1 V2 V3 Y V4 V5. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,X,V4,V5),continuous(V1::'a*
&
($\forall$ *X V1 V2 V3 V4 Y V5. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,V4,X,V5),continuous(V1::'a*
&
($\forall$ *X V1 V2 V3 V4 V5 Y. equal(X::'a,Y) --> equal(continuous(V1::'a,V2,V3,V4,V5,X),continuous(V1::'a*

lemma *GEO003-1*:
EQU001-0-ax equal &
GEO001-0-ax continuous lower-dimension-point-3 lower-dimension-point-2 lower-dimension-point-1 extension euclid2 euclid1 outer-pasch equidistant equal between &
GEO001-0-eq continuous extension euclid2 euclid1 outer-pasch equidistant between equal &
(~*between(a::'a,b,b)) --> False*
⟨*proof*⟩

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* ≡
($\forall$ *Y X. equidistant(X::'a,Y,Y,X))* &
($\forall$ *X Y Z V V2 W. equidistant(X::'a,Y,Z,V)* & *equidistant(X::'a,Y,V2,W) --> equidistant(Z::'a,V,V2,W))* &
($\forall$ *Z X Y. equidistant(X::'a,Y,Z,Z) --> equal(X::'a,Y))* &
($\forall$ *X Y W V. between(X::'a,Y,extension(X::'a,Y,W,V)))* &
($\forall$ *X Y W V. equidistant(Y::'a,extension(X::'a,Y,W,V),W,V))* &
($\forall$ *X1 Y1 X Y Z V Z1 V1. equidistant(X::'a,Y,X1,Y1)* & *equidistant(Y::'a,Z,Y1,Z1)* & *equidistant(X::'a,V,X1,V1)* & *equidistant(Y::'a,V,Y1,V1)* & *between(X::'a,Y,Z)* & *between(X1::'a,Y1,Z1) --> equal(X::'a,Y) | equidistant(Z::'a,V,Z1,V1))* &
($\forall$ *X Y. between(X::'a,Y,X) --> equal(X::'a,Y))* &
($\forall$ *U V W X Y. between(U::'a,V,W)* & *between(Y::'a,X,W) --> between(V::'a,inner-pasch(U::'a,V,W,X*
&
($\forall$ *V W X Y U. between(U::'a,V,W)* & *between(Y::'a,X,W) --> between(X::'a,inner-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))*
&
($\forall$ *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))* &

$(\forall U\ V\ W\ X\ Y.\ \mathit{between}(U::'a,W,Y)\ \&\ \mathit{between}(V::'a,W,X)\ {-}{-}{>}\ \mathit{equal}(U::'a,W)$
$|\ \mathit{between}(U::'a,V,\mathit{euclid1}(U::'a,V,W,X,Y)))\ \&$
$(\forall U\ V\ W\ X\ Y.\ \mathit{between}(U::'a,W,Y)\ \&\ \mathit{between}(V::'a,W,X)\ {-}{-}{>}\ \mathit{equal}(U::'a,W)$
$|\ \mathit{between}(U::'a,X,\mathit{euclid2}(U::'a,V,W,X,Y)))\ \&$
$(\forall U\ V\ W\ X\ Y.\ \mathit{between}(U::'a,W,Y)\ \&\ \mathit{between}(V::'a,W,X)\ {-}{-}{>}\ \mathit{equal}(U::'a,W)$
$|\ \mathit{between}(\mathit{euclid1}(U::'a,V,W,X,Y),Y,\mathit{euclid2}(U::'a,V,W,X,Y)))\ \&$
$(\forall U\ V\ V1\ W\ X\ X1.\ \mathit{equidistant}(U::'a,V,U,V1)\ \&\ \mathit{equidistant}(U::'a,X,U,X1)\ \&$
$\mathit{between}(U::'a,V,X)\ \&\ \mathit{between}(V::'a,W,X)\ {-}{-}{>}\ \mathit{between}(V1::'a,\mathit{continuous}(U::'a,V,V1,W,X,X1),X1))$
$\&$
$(\forall U\ V\ V1\ W\ X\ X1.\ \mathit{equidistant}(U::'a,V,U,V1)\ \&\ \mathit{equidistant}(U::'a,X,U,X1)\ \&$
$\mathit{between}(U::'a,V,X)\ \&\ \mathit{between}(V::'a,W,X)\ {-}{-}{>}\ \mathit{equidistant}(U::'a,W,U,\mathit{continuous}(U::'a,V,V1,W,X,X1))$
$\&$
$(\forall X\ Y\ W\ Z.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{between}(X::'a,W,Z)\ {-}{-}{>}\ \mathit{between}(Y::'a,W,Z))$
$\&$
$(\forall X\ W\ Y\ Z.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{between}(W::'a,X,Z)\ {-}{-}{>}\ \mathit{between}(W::'a,Y,Z))$
$\&$
$(\forall X\ W\ Z\ Y.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{between}(W::'a,Z,X)\ {-}{-}{>}\ \mathit{between}(W::'a,Z,Y))$
$\&$
$(\forall X\ Y\ V\ W\ Z.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{equidistant}(X::'a,V,W,Z)\ {-}{-}{>}\ \mathit{equidis}$-
$\mathit{tant}(Y::'a,V,W,Z))\ \&$
$(\forall X\ V\ Y\ W\ Z.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{equidistant}(V::'a,X,W,Z)\ {-}{-}{>}\ \mathit{equidis}$-
$\mathit{tant}(V::'a,Y,W,Z))\ \&$
$(\forall X\ V\ W\ Y\ Z.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{equidistant}(V::'a,W,X,Z)\ {-}{-}{>}\ \mathit{equidis}$-
$\mathit{tant}(V::'a,W,Y,Z))\ \&$
$(\forall X\ V\ W\ Z\ Y.\ \mathit{equal}(X::'a,Y)\ \&\ \mathit{equidistant}(V::'a,W,Z,X)\ {-}{-}{>}\ \mathit{equidis}$-
$\mathit{tant}(V::'a,W,Z,Y))\ \&$
$(\forall X\ Y\ V1\ V2\ V3\ V4.\ \mathit{equal}(X::'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{inner\text{-}pasch}(X::'a,V1,V2,V3,V4),\mathit{inner\text{-}pasch}(Y::'a,V1,V$
$\&$
$(\forall X\ V1\ Y\ V2\ V3\ V4.\ \mathit{equal}(X::'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{inner\text{-}pasch}(V1::'a,X,V2,V3,V4),\mathit{inner\text{-}pasch}(V1::'a,Y,V$
$\&$
$(\forall X\ V1\ V2\ Y\ V3\ V4.\ \mathit{equal}(X::'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{inner\text{-}pasch}(V1::'a,V2,X,V3,V4),\mathit{inner\text{-}pasch}(V1::'a,V2,$
$\&$
$(\forall X\ V1\ V2\ V3\ Y\ V4.\ \mathit{equal}(X::'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{inner\text{-}pasch}(V1::'a,V2,V3,X,V4),\mathit{inner\text{-}pasch}(V1::'a,V2,$
$\&$
$(\forall X\ V1\ V2\ V3\ V4\ Y.\ \mathit{equal}(X::'a,Y)\ {-}{-}{>}\ \mathit{equal}(\mathit{inner\text{-}pasch}(V1::'a,V2,V3,V4,X),\mathit{inner\text{-}pasch}(V1::'a,V2,$
$\&$
$(\forall A\ B\ C\ D\ E\ F'.\ \mathit{equal}(A::'a,B)\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid1}(A::'a,C,D,E,F'),\mathit{euclid1}(B::'a,C,D,E,F')))$
$\&$
$(\forall G\ I'\ H\ J\ K'\ L.\ \mathit{equal}(G::'a,H)\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid1}(I'::'a,G,J,K',L),\mathit{euclid1}(I'::'a,H,J,K',L)))$
$\&$
$(\forall M\ O'\ P\ N\ Q\ R.\ \mathit{equal}(M::'a,N)\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid1}(O'::'a,P,M,Q,R),\mathit{euclid1}(O'::'a,P,N,Q,R)))$
$\&$
$(\forall S'\ U\ V\ W\ T'\ X.\ \mathit{equal}(S'::'a,T')\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid1}(U::'a,V,W,S',X),\mathit{euclid1}(U::'a,V,W,T',X)))$
$\&$
$(\forall Y\ A1\ B1\ C1\ D1\ Z.\ \mathit{equal}(Y::'a,Z)\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid1}(A1::'a,B1,C1,D1,Y),\mathit{euclid1}(A1::'a,B1,C1,D1,Z)$
$\&$
$(\forall E1\ F1\ G1\ H1\ I1\ J1.\ \mathit{equal}(E1::'a,F1)\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid2}(E1::'a,G1,H1,I1,J1),\mathit{euclid2}(F1::'a,G1,H1,I1,$
$\&$
$(\forall K1\ M1\ L1\ N1\ O1\ P1.\ \mathit{equal}(K1::'a,L1)\ {-}{-}{>}\ \mathit{equal}(\mathit{euclid2}(M1::'a,K1,N1,O1,P1),\mathit{euclid2}(M1::'a,L1,N1,$

&
 (∀ *Q1 S1 T1 R1 U1 V1*. *equal*(*Q1*::′*a*,*R1*) −−> *equal*(*euclid2*(*S1*::′*a*,*T1*,*Q1*,*U1*,*V1*),*euclid2*(*S1*::′*a*,*T1*,*R1*,*U*
&
 (∀ *W1 Y1 Z1 A2 X1 B2*. *equal*(*W1*::′*a*,*X1*) −−> *equal*(*euclid2*(*Y1*::′*a*,*Z1*,*A2*,*W1*,*B2*),*euclid2*(*Y1*::′*a*,*Z1*,*A2*
&
 (∀ *C2 E2 F2 G2 H2 D2*. *equal*(*C2*::′*a*,*D2*) −−> *equal*(*euclid2*(*E2*::′*a*,*F2*,*G2*,*H2*,*C2*),*euclid2*(*E2*::′*a*,*F2*,*G2*,*H*
&
 (∀ *X Y V1 V2 V3*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*extension*(*X*::′*a*,*V1*,*V2*,*V3*),*extension*(*Y*::′*a*,*V1*,*V2*,*V3*)))
&
 (∀ *X V1 Y V2 V3*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*extension*(*V1*::′*a*,*X*,*V2*,*V3*),*extension*(*V1*::′*a*,*Y*,*V2*,*V3*)))
&
 (∀ *X V1 V2 Y V3*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*extension*(*V1*::′*a*,*V2*,*X*,*V3*),*extension*(*V1*::′*a*,*V2*,*Y*,*V3*)))
&
 (∀ *X V1 V2 V3 Y*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*extension*(*V1*::′*a*,*V2*,*V3*,*X*),*extension*(*V1*::′*a*,*V2*,*V3*,*Y*)))
&
 (∀ *X Y V1 V2 V3 V4 V5*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*continuous*(*X*::′*a*,*V1*,*V2*,*V3*,*V4*,*V5*),*continuous*(*Y*::′*a*,
&
 (∀ *X V1 Y V2 V3 V4 V5*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*continuous*(*V1*::′*a*,*X*,*V2*,*V3*,*V4*,*V5*),*continuous*(*V1*::′*a*
&
 (∀ *X V1 V2 Y V3 V4 V5*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*continuous*(*V1*::′*a*,*V2*,*X*,*V3*,*V4*,*V5*),*continuous*(*V1*::′*a*
&
 (∀ *X V1 V2 V3 Y V4 V5*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*continuous*(*V1*::′*a*,*V2*,*V3*,*X*,*V4*,*V5*),*continuous*(*V1*::′*a*
&
 (∀ *X V1 V2 V3 V4 Y V5*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*continuous*(*V1*::′*a*,*V2*,*V3*,*V4*,*X*,*V5*),*continuous*(*V1*::′*a*
&
 (∀ *X V1 V2 V3 V4 V5 Y*. *equal*(*X*::′*a*,*Y*) −−> *equal*(*continuous*(*V1*::′*a*,*V2*,*V3*,*V4*,*V5*,*X*),*continuous*(*V1*::′*a*

lemma *GEO017-2*:
 EQU001-0-ax equal &
 GEO002-ax-eq continuous euclid2 euclid1 lower-dimension-point-3
 lower-dimension-point-2 lower-dimension-point-1 inner-pasch extension
 between equal equidistant &
 (*equidistant*(*u*::′*a*,*v*,*w*,*x*)) &
 (∼*equidistant*(*u*::′*a*,*v*,*x*,*w*)) −−> *False*
 ⟨*proof*⟩

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
⟨proof⟩

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
⟨proof⟩

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)) &

$(trapezoid(a::'a,b,c,d))$ &
$(\sim eq(a::'a,c,b,c,a,d))$ −−> *False*
$\langle proof \rangle$

abbreviation *GRP003-0-ax equal multiply INVERSE identity product* ≡
$(\forall X.\ product(identity::'a,X,X))$ &
$(\forall X.\ product(X::'a,identity,X))$ &
$(\forall X.\ product(INVERSE(X),X,identity))$ &
$(\forall X.\ product(X::'a,INVERSE(X),identity))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y\ Z\ W.\ product(X::'a,Y,Z)$ & $product(X::'a,Y,W)$ −−> $equal(Z::'a,W))$
&
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
−−> $product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
−−> $product(U::'a,Z,W))$

abbreviation *GRP003-0-eq product multiply INVERSE equal* ≡
$(\forall X\ Y.\ equal(X::'a,Y)$ −−> $equal(INVERSE(X),INVERSE(Y)))$ &
$(\forall X\ Y\ W.\ equal(X::'a,Y)$ −−> $equal(multiply(X::'a,W),multiply(Y::'a,W)))$
&
$(\forall X\ W\ Y.\ equal(X::'a,Y)$ −−> $equal(multiply(W::'a,X),multiply(W::'a,Y)))$
&
$(\forall X\ Y\ W\ Z.\ equal(X::'a,Y)$ & $product(X::'a,W,Z)$ −−> $product(Y::'a,W,Z))$
&
$(\forall X\ W\ Y\ Z.\ equal(X::'a,Y)$ & $product(W::'a,X,Z)$ −−> $product(W::'a,Y,Z))$
&
$(\forall X\ W\ Z\ Y.\ equal(X::'a,Y)$ & $product(W::'a,Z,X)$ −−> $product(W::'a,Z,Y))$

lemma *GRP001-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall X.\ product(X::'a,X,identity))$ &
$(product(a::'a,b,c))$ &
$(\sim product(b::'a,a,c))$ −−> *False*
$\langle proof \rangle$

lemma *GRP008-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
$(\forall A\ B.\ equal(A::'a,B)$ −−> $equal(h(A),h(B)))$ &
$(\forall C\ D.\ equal(C::'a,D)$ −−> $equal(j(C),j(D)))$ &
$(\forall A\ B.\ equal(A::'a,B)$ & $q(A)$ −−> $q(B))$ &
$(\forall B\ A\ C.\ q(A)$ & $product(A::'a,B,C)$ −−> $product(B::'a,A,C))$ &
$(\forall A.\ product(j(A),A,h(A)) \mid product(A::'a,j(A),h(A)) \mid q(A))$ &

(∀ A. product(j(A),A,h(A)) & product(A::'a,j(A),h(A)) --> q(A)) &
(~q(identity)) --> False
⟨proof⟩

lemma *GRP013-1*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
(∀ A. product(A::'a,A,identity)) &
(product(a::'a,b,c)) &
(product(INVERSE(a),INVERSE(b),d)) &
(∀ A C B. product(INVERSE(A),INVERSE(B),C) --> product(A::'a,C,B)) &
(~product(c::'a,d,identity)) --> False
⟨proof⟩

lemma *GRP037-3*:
EQU001-0-ax equal &
GRP003-0-ax equal multiply INVERSE identity product &
GRP003-0-eq product multiply INVERSE equal &
(∀ A B C. subgroup-member(A) & subgroup-member(B) & product(A::'a,INVERSE(B),C) --> subgroup-member(C)) &
(∀ A B. equal(A::'a,B) & subgroup-member(A) --> subgroup-member(B)) &
(∀ A. subgroup-member(A) --> product(Gidentity::'a,A,A)) &
(∀ A. subgroup-member(A) --> product(A::'a,Gidentity,A)) &
(∀ A. subgroup-member(A) --> product(A::'a,Ginverse(A),Gidentity)) &
(∀ A. subgroup-member(A) --> product(Ginverse(A),A,Gidentity)) &
(∀ A. subgroup-member(A) --> subgroup-member(Ginverse(A))) &
(∀ A B. equal(A::'a,B) --> equal(Ginverse(A),Ginverse(B))) &
(∀ A C D B. product(A::'a,B,C) & product(A::'a,D,C) --> equal(D::'a,B)) &
(∀ B C D A. product(A::'a,B,C) & product(D::'a,B,C) --> equal(D::'a,A)) &
(subgroup-member(a)) &
(subgroup-member(Gidentity)) &
(~equal(INVERSE(a),Ginverse(a))) --> False
⟨proof⟩

lemma *GRP031-2*:
(∀ X Y. product(X::'a,Y,multiply(X::'a,Y))) &
(∀ X Y Z W. product(X::'a,Y,Z) & product(X::'a,Y,W) --> equal(Z::'a,W)) &
(∀ Y U Z X V W. product(X::'a,Y,U) & product(Y::'a,Z,V) & product(U::'a,Z,W) --> product(X::'a,V,W)) &
(∀ Y X V U Z W. product(X::'a,Y,U) & product(Y::'a,Z,V) & product(X::'a,V,W) --> product(U::'a,Z,W)) &
(∀ A. product(A::'a,INVERSE(A),identity)) &
(∀ A. product(A::'a,identity,A)) &
(∀ A. ~product(A::'a,a,identity)) --> False

⟨*proof*⟩

lemma *GRP034-4*:
(∀ *X Y*. *product*(*X*::′*a*,*Y*,*multiply*(*X*::′*a*,*Y*))) &
(∀ *X*. *product*(*identity*::′*a*,*X*,*X*)) &
(∀ *X*. *product*(*X*::′*a*,*identity*,*X*)) &
(∀ *X*. *product*(*X*::′*a*,*INVERSE*(*X*),*identity*)) &
(∀ *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*)) &
(∀ *B A C*. *subgroup-member*(*A*) & *subgroup-member*(*B*) & *product*(*B*::′*a*,*INVERSE*(*A*),*C*)
−−> *subgroup-member*(*C*)) &
(*subgroup-member*(*a*)) &
(∼*subgroup-member*(*INVERSE*(*a*))) −−> *False*
⟨*proof*⟩

lemma *GRP047-2*:
(∀ *X*. *product*(*identity*::′*a*,*X*,*X*)) &
(∀ *X*. *product*(*INVERSE*(*X*),*X*,*identity*)) &
(∀ *X Y*. *product*(*X*::′*a*,*Y*,*multiply*(*X*::′*a*,*Y*))) &
(∀ *X Y Z W*. *product*(*X*::′*a*,*Y*,*Z*) & *product*(*X*::′*a*,*Y*,*W*) −−> *equal*(*Z*::′*a*,*W*))
&
(∀ *Y U Z X V W*. *product*(*X*::′*a*,*Y*,*U*) & *product*(*Y*::′*a*,*Z*,*V*) & *product*(*U*::′*a*,*Z*,*W*)
−−> *product*(*X*::′*a*,*V*,*W*)) &
(∀ *Y X V U Z W*. *product*(*X*::′*a*,*Y*,*U*) & *product*(*Y*::′*a*,*Z*,*V*) & *product*(*X*::′*a*,*V*,*W*)
−−> *product*(*U*::′*a*,*Z*,*W*)) &
(∀ *X W Z Y*. *equal*(*X*::′*a*,*Y*) & *product*(*W*::′*a*,*Z*,*X*) −−> *product*(*W*::′*a*,*Z*,*Y*))
&
(*equal*(*a*::′*a*,*b*)) &
(∼*equal*(*multiply*(*c*::′*a*,*a*),*multiply*(*c*::′*a*,*b*))) −−> *False*
⟨*proof*⟩

lemma *GRP130-1-002*:
(*group-element*(*e-1*)) &
(*group-element*(*e-2*)) &
(∼*equal*(*e-1*::′*a*,*e-2*)) &
(∼*equal*(*e-2*::′*a*,*e-1*)) &
(∀ *X Y*. *group-element*(*X*) & *group-element*(*Y*) −−> *product*(*X*::′*a*,*Y*,*e-1*) |
product(*X*::′*a*,*Y*,*e-2*)) &
(∀ *X Y W Z*. *product*(*X*::′*a*,*Y*,*W*) & *product*(*X*::′*a*,*Y*,*Z*) −−> *equal*(*W*::′*a*,*Z*))
&
(∀ *X Y W Z*. *product*(*X*::′*a*,*W*,*Y*) & *product*(*X*::′*a*,*Z*,*Y*) −−> *equal*(*W*::′*a*,*Z*))
&
(∀ *Y X W Z*. *product*(*W*::′*a*,*Y*,*X*) & *product*(*Z*::′*a*,*Y*,*X*) −−> *equal*(*W*::′*a*,*Z*))
&

$(\forall\, Z1\ Z2\ Y\ X.\ product(X::'a,Y,Z1)\ \&\ product(X::'a,Z1,Z2) \dashrightarrow product(Z2::'a,Y,X))$
$\dashrightarrow False$
$\langle proof \rangle$

abbreviation *GRP004-0-ax INVERSE identity multiply equal* $\equiv$
$(\forall\, X.\ equal(multiply(identity::'a,X),X))\ \&$
$(\forall\, X.\ equal(multiply(INVERSE(X),X),identity))\ \&$
$(\forall\, X\ Y\ Z.\ equal(multiply(multiply(X::'a,Y),Z),multiply(X::'a,multiply(Y::'a,Z))))$
$\&$
$(\forall\, A\ B.\ equal(A::'a,B) \dashrightarrow equal(INVERSE(A),INVERSE(B)))\ \&$
$(\forall\, C\ D\ E.\ equal(C::'a,D) \dashrightarrow equal(multiply(C::'a,E),multiply(D::'a,E)))\ \&$
$(\forall\, F'\ H\ G.\ equal(F'::'a,G) \dashrightarrow equal(multiply(H::'a,F'),multiply(H::'a,G)))$

abbreviation *GRP004-2-ax multiply least-upper-bound greatest-lower-bound equal*
$\equiv$
$(\forall\, Y\ X.\ equal(greatest\text{-}lower\text{-}bound(X::'a,Y),greatest\text{-}lower\text{-}bound(Y::'a,X)))\ \&$
$(\forall\, Y\ X.\ equal(least\text{-}upper\text{-}bound(X::'a,Y),least\text{-}upper\text{-}bound(Y::'a,X)))\ \&$
$(\forall\, X\ Y\ Z.\ equal(greatest\text{-}lower\text{-}bound(X::'a,greatest\text{-}lower\text{-}bound(Y::'a,Z)),greatest\text{-}lower\text{-}bound(greatest\text{-}lowe$
$\&$
$(\forall\, X\ Y\ Z.\ equal(least\text{-}upper\text{-}bound(X::'a,least\text{-}upper\text{-}bound(Y::'a,Z)),least\text{-}upper\text{-}bound(least\text{-}upper\text{-}bound(X:$
$\&$
$(\forall\, X.\ equal(least\text{-}upper\text{-}bound(X::'a,X),X))\ \&$
$(\forall\, X.\ equal(greatest\text{-}lower\text{-}bound(X::'a,X),X))\ \&$
$(\forall\, Y\ X.\ equal(least\text{-}upper\text{-}bound(X::'a,greatest\text{-}lower\text{-}bound(X::'a,Y)),X))\ \&$
$(\forall\, Y\ X.\ equal(greatest\text{-}lower\text{-}bound(X::'a,least\text{-}upper\text{-}bound(X::'a,Y)),X))\ \&$
$(\forall\, Y\ X\ Z.\ equal(multiply(X::'a,least\text{-}upper\text{-}bound(Y::'a,Z)),least\text{-}upper\text{-}bound(multiply(X::'a,Y),multiply(X:$
$\&$
$(\forall\, Y\ X\ Z.\ equal(multiply(X::'a,greatest\text{-}lower\text{-}bound(Y::'a,Z)),greatest\text{-}lower\text{-}bound(multiply(X::'a,Y),multip$
$\&$
$(\forall\, Y\ Z\ X.\ equal(multiply(least\text{-}upper\text{-}bound(Y::'a,Z),X),least\text{-}upper\text{-}bound(multiply(Y::'a,X),multiply(Z::'a,$
$\&$
$(\forall\, Y\ Z\ X.\ equal(multiply(greatest\text{-}lower\text{-}bound(Y::'a,Z),X),greatest\text{-}lower\text{-}bound(multiply(Y::'a,X),multiply($
$\&$
$(\forall\, A\ B\ C.\ equal(A::'a,B) \dashrightarrow equal(greatest\text{-}lower\text{-}bound(A::'a,C),greatest\text{-}lower\text{-}bound(B::'a,C)))$
$\&$
$(\forall\, A\ C\ B.\ equal(A::'a,B) \dashrightarrow equal(greatest\text{-}lower\text{-}bound(C::'a,A),greatest\text{-}lower\text{-}bound(C::'a,B)))$
$\&$
$(\forall\, A\ B\ C.\ equal(A::'a,B) \dashrightarrow equal(least\text{-}upper\text{-}bound(A::'a,C),least\text{-}upper\text{-}bound(B::'a,C)))$
$\&$
$(\forall\, A\ C\ B.\ equal(A::'a,B) \dashrightarrow equal(least\text{-}upper\text{-}bound(C::'a,A),least\text{-}upper\text{-}bound(C::'a,B)))$
$\&$
$(\forall\, A\ B\ C.\ equal(A::'a,B) \dashrightarrow equal(multiply(A::'a,C),multiply(B::'a,C)))\ \&$
$(\forall\, A\ C\ B.\ equal(A::'a,B) \dashrightarrow equal(multiply(C::'a,A),multiply(C::'a,B)))$

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 &

$(equal(least\text{-}upper\text{-}bound(a::'a,b),b))$ &
$(\sim equal(greatest\text{-}lower\text{-}bound(multiply(a::'a,c),multiply(b::'a,c)),multiply(a::'a,c)))$
$--> False$
$\langle proof \rangle$

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 &
$(equal(least\text{-}upper\text{-}bound(a::'a,b),b))$ &
$(\sim equal(least\text{-}upper\text{-}bound(multiply(INVERSE(c),multiply(a::'a,c)),multiply(INVERSE(c),multiply(b::'a,c))$
$--> False$
$\langle proof \rangle$

abbreviation *HEN002-0-ax identity Zero Divide equal mless-equal* $\equiv$
$(\forall X\ Y.\ mless\text{-}equal(X::'a,Y) --> equal(Divide(X::'a,Y),Zero))$ &
$(\forall X\ Y.\ equal(Divide(X::'a,Y),Zero) --> mless\text{-}equal(X::'a,Y))$ &
$(\forall Y\ X.\ mless\text{-}equal(Divide(X::'a,Y),X))$ &
$(\forall X\ Y\ Z.\ mless\text{-}equal(Divide(Divide(X::'a,Z),Divide(Y::'a,Z)),Divide(Divide(X::'a,Y),Z)))$
&
$(\forall X.\ mless\text{-}equal(Zero::'a,X))$ &
$(\forall X\ Y.\ mless\text{-}equal(X::'a,Y)$ & $mless\text{-}equal(Y::'a,X) --> equal(X::'a,Y))$ &
$(\forall X.\ mless\text{-}equal(X::'a,identity))$

abbreviation *HEN002-0-eq mless-equal Divide equal* $\equiv$
$(\forall A\ B\ C.\ equal(A::'a,B) --> equal(Divide(A::'a,C),Divide(B::'a,C)))$ &
$(\forall D\ F'\ E.\ equal(D::'a,E) --> equal(Divide(F'::'a,D),Divide(F'::'a,E)))$ &
$(\forall G\ H\ I'.\ equal(G::'a,H)$ & $mless\text{-}equal(G::'a,I') --> mless\text{-}equal(H::'a,I'))$ &
$(\forall J\ L\ K'.\ equal(J::'a,K')$ & $mless\text{-}equal(L::'a,J) --> mless\text{-}equal(L::'a,K'))$

lemma *HEN003-3*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
$(\sim equal(Divide(a::'a,a),Zero)) --> False$
$\langle proof \rangle$

lemma *HEN007-2*:
EQU001-0-ax equal &
$(\forall X\ Y.\ mless\text{-}equal(X::'a,Y) --> quotient(X::'a,Y,Zero))$ &
$(\forall X\ Y.\ quotient(X::'a,Y,Zero) --> mless\text{-}equal(X::'a,Y))$ &
$(\forall Y\ Z\ X.\ quotient(X::'a,Y,Z) --> mless\text{-}equal(Z::'a,X))$ &
$(\forall Y\ X\ V3\ V2\ V1\ Z\ V4\ V5.\ quotient(X::'a,Y,V1)$ & $quotient(Y::'a,Z,V2)$ & $quotient(X::'a,Z,V3)$ & $quotient(V3::'a,V2,V4)$ & $quotient(V1::'a,Z,V5) -->$ $mless\text{-}equal(V4::'a,V5))$ &
$(\forall X.\ mless\text{-}equal(Zero::'a,X))$ &

(∀ 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))
&
(∀ X W Y Z. equal(X::'a,Y) & quotient(W::'a,X,Z) --> quotient(W::'a,Y,Z))
&
(∀ X W Z Y. equal(X::'a,Y) & quotient(W::'a,Z,X) --> quotient(W::'a,Z,Y))
&
(∀ X Z Y. equal(X::'a,Y) & mless-equal(Z::'a,X) --> mless-equal(Z::'a,Y)) &
(∀ X Y Z. equal(X::'a,Y) & mless-equal(X::'a,Z) --> mless-equal(Y::'a,Z)) &
(∀ X Y W. equal(X::'a,Y) --> equal(Divide(X::'a,W),Divide(Y::'a,W))) &
(∀ X W Y. equal(X::'a,Y) --> equal(Divide(W::'a,X),Divide(W::'a,Y))) &
(∀ X. quotient(X::'a,identity,Zero)) &
(∀ X. quotient(Zero::'a,X,Zero)) &
(∀ X. quotient(X::'a,X,Zero)) &
(∀ X. quotient(X::'a,Zero,X)) &
(∀ Y X Z. mless-equal(X::'a,Y) & mless-equal(Y::'a,Z) --> mless-equal(X::'a,Z))
&
(∀ W1 X Z W2 Y. quotient(X::'a,Y,W1) & mless-equal(W1::'a,Z) & quotient(X::'a,Z,W2)
--> mless-equal(W2::'a,Y)) &
(mless-equal(x::'a,y)) &
(quotient(z::'a,y,zQy)) &
(quotient(z::'a,x,zQx)) &
(~mless-equal(zQy::'a,zQx)) --> False
⟨proof⟩

lemma *HEN008-4*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
(∀ X. equal(Divide(X::'a,identity),Zero)) &
(∀ X. equal(Divide(Zero::'a,X),Zero)) &
(∀ X. equal(Divide(X::'a,X),Zero)) &
(equal(Divide(a::'a,Zero),a)) &
(∀ Y X Z. mless-equal(X::'a,Y) & mless-equal(Y::'a,Z) --> mless-equal(X::'a,Z))
&
(∀ X Z Y. mless-equal(Divide(X::'a,Y),Z) --> mless-equal(Divide(X::'a,Z),Y))
&
(∀ Y Z X. mless-equal(X::'a,Y) --> mless-equal(Divide(Z::'a,Y),Divide(Z::'a,X)))
&
(mless-equal(a::'a,b)) &
(~mless-equal(Divide(a::'a,c),Divide(b::'a,c))) --> False
⟨proof⟩

lemma *HEN009-5*:
EQU001-0-ax equal &
$(\forall\, Y\ X.\ equal(Divide(Divide(X{::}'a,Y),X),Zero))$ &
$(\forall\, X\ Y\ Z.\ equal(Divide(Divide(Divide(X{::}'a,Z),Divide(Y{::}'a,Z)),Divide(Divide(X{::}'a,Y),Z)),Zero))$
&
$(\forall\, X.\ equal(Divide(Zero{::}'a,X),Zero))$ &
$(\forall\, X\ Y.\ equal(Divide(X{::}'a,Y),Zero)\ \&\ equal(Divide(Y{::}'a,X),Zero) \dashrightarrow equal(X{::}'a,Y))$
&
$(\forall\, X.\ equal(Divide(X{::}'a,identity),Zero))$ &
$(\forall\, A\ B\ C.\ equal(A{::}'a,B) \dashrightarrow equal(Divide(A{::}'a,C),Divide(B{::}'a,C)))$ &
$(\forall\, D\ F'\ E.\ equal(D{::}'a,E) \dashrightarrow equal(Divide(F'{::}'a,D),Divide(F'{::}'a,E)))$ &
$(\forall\, Y\ X\ Z.\ equal(Divide(X{::}'a,Y),Zero)\ \&\ equal(Divide(Y{::}'a,Z),Zero) \dashrightarrow$
$equal(Divide(X{::}'a,Z),Zero))$ &
$(\forall\, X\ Z\ Y.\ equal(Divide(Divide(X{::}'a,Y),Z),Zero) \dashrightarrow equal(Divide(Divide(X{::}'a,Z),Y),Zero))$
&
$(\forall\, Y\ Z\ X.\ equal(Divide(X{::}'a,Y),Zero) \dashrightarrow equal(Divide(Divide(Z{::}'a,Y),Divide(Z{::}'a,X)),Zero))$
&
$(\sim equal(Divide(identity{::}'a,a),Divide(identity{::}'a,Divide(identity{::}'a,Divide(identity{::}'a,a)))))$
&
$(equal(Divide(identity{::}'a,a),b))$ &
$(equal(Divide(identity{::}'a,b),c))$ &
$(equal(Divide(identity{::}'a,c),d))$ &
$(\sim equal(b{::}'a,d)) \dashrightarrow False$
$\langle proof \rangle$

lemma *HEN012-3*:
EQU001-0-ax equal &
HEN002-0-ax identity Zero Divide equal mless-equal &
HEN002-0-eq mless-equal Divide equal &
$(\sim mless\text{-}equal(a{::}'a,a)) \dashrightarrow False$
$\langle proof \rangle$

lemma *LCL010-1*:
$(\forall\, X\ Y.\ is\text{-}a\text{-}theorem(equivalent(X{::}'a,Y))\ \&\ is\text{-}a\text{-}theorem(X) \dashrightarrow is\text{-}a\text{-}theorem(Y))$
&
$(\forall\, X\ Z\ Y.\ is\text{-}a\text{-}theorem(equivalent(equivalent(X{::}'a,Y),equivalent(equivalent(X{::}'a,Z),equivalent(Z{::}'a,Y)))))$
&
$(\sim is\text{-}a\text{-}theorem(equivalent(equivalent(a{::}'a,b),equivalent(equivalent(c{::}'a,b),equivalent(a{::}'a,c)))))$
$\dashrightarrow False$
$\langle proof \rangle$

lemma *LCL077-2*:
$(\forall\, X\ Y.\ is\text{-}a\text{-}theorem(implies(X,Y))\ \&\ is\text{-}a\text{-}theorem(X) \dashrightarrow is\text{-}a\text{-}theorem(Y))$
&
$(\forall\, Y\ X.\ is\text{-}a\text{-}theorem(implies(X,implies(Y,X))))$ &

(∀ 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*
⟨*proof*⟩

lemma *LCL082-1*:
(∀ X Y. *is-a-theorem*(*implies*(X::'a,Y)) & *is-a-theorem*(X) −−> *is-a-theorem*(Y))
&
(∀ Y Z U X. *is-a-theorem*(*implies*(*implies*(*implies*(X::'a,Y),Z),*implies*(*implies*(Z::'a,X),*implies*(U::'a,X)))))
&
(~*is-a-theorem*(*implies*(a::'a,*implies*(b::'a,a)))) −−> *False*
⟨*proof*⟩

lemma *LCL111-1*:
(∀ X Y. *is-a-theorem*(*implies*(X,Y)) & *is-a-theorem*(X) −−> *is-a-theorem*(Y))
&
(∀ Y X. *is-a-theorem*(*implies*(X,*implies*(Y,X)))) &
(∀ Y X Z. *is-a-theorem*(*implies*(*implies*(X,Y),*implies*(*implies*(Y,Z),*implies*(X,Z)))))
&
(∀ Y X. *is-a-theorem*(*implies*(*implies*(*implies*(X,Y),Y),*implies*(*implies*(Y,X),X))))
&
(∀ Y X. *is-a-theorem*(*implies*(*implies*(*not*(X),*not*(Y)),*implies*(Y,X)))) &
(~*is-a-theorem*(*implies*(*implies*(a,b),*implies*(*implies*(c,a),*implies*(c,b))))) −−> *False*
⟨*proof*⟩

lemma *LCL143-1*:
(∀ X. *equal*(X,X)) &
(∀ Y X. *equal*(X,Y) −−> *equal*(Y,X)) &
(∀ Y X Z. *equal*(X,Y) & *equal*(Y,Z) −−> *equal*(X,Z)) &
(∀ X. *equal*(*implies*(*true*,X),X)) &
(∀ Y X Z. *equal*(*implies*(*implies*(X,Y),*implies*(*implies*(Y,Z),*implies*(X,Z))),*true*))
&
(∀ Y X. *equal*(*implies*(*implies*(X,Y),Y),*implies*(*implies*(Y,X),X))) &
(∀ Y X. *equal*(*implies*(*implies*(*not*(X),*not*(Y)),*implies*(Y,X)),*true*)) &
(∀ A B C. *equal*(A,B) −−> *equal*(*implies*(A,C),*implies*(B,C))) &
(∀ D F' E. *equal*(D,E) −−> *equal*(*implies*(F',D),*implies*(F',E))) &
(∀ G H. *equal*(G,H) −−> *equal*(*not*(G),*not*(H))) &
(∀ X Y. *equal*(*big-V*(X,Y),*implies*(*implies*(X,Y),Y))) &
(∀ X Y. *equal*(*big-hat*(X,Y),*not*(*big-V*(*not*(X),*not*(Y))))) &
(∀ X Y. *ordered*(X,Y) −−> *equal*(*implies*(X,Y),*true*)) &
(∀ X Y. *equal*(*implies*(X,Y),*true*) −−> *ordered*(X,Y)) &
(∀ A B C. *equal*(A,B) −−> *equal*(*big-V*(A,C),*big-V*(B,C))) &
(∀ D F' E. *equal*(D,E) −−> *equal*(*big-V*(F',D),*big-V*(F',E))) &

$(\forall\, G\ H\ I'.\ equal(G,H) \dashrightarrow equal(big\text{-}hat(G,I'),big\text{-}hat(H,I')))$ &
$(\forall\, J\ L\ K'.\ equal(J,K') \dashrightarrow equal(big\text{-}hat(L,J),big\text{-}hat(L,K')))$ &
$(\forall\, M\ N\ O'.\ equal(M,N)$ & $ordered(M,O') \dashrightarrow ordered(N,O'))$ &
$(\forall\, P\ R\ Q.\ equal(P,Q)$ & $ordered(R,P) \dashrightarrow ordered(R,Q))$ &
$(ordered(x,y))$ &
$(\sim ordered(implies(z,x),implies(z,y))) \dashrightarrow False$
$\langle proof \rangle$

lemma *LCL182-1*:
$(\forall\, A.\ axiom(or(not(or(A,A)),A)))$ &
$(\forall\, B\ A.\ axiom(or(not(A),or(B,A))))$ &
$(\forall\, B\ A.\ axiom(or(not(or(A,B)),or(B,A))))$ &
$(\forall\, B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C)))))$ &
$(\forall\, A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B)))))$ &
$(\forall\, X.\ axiom(X) \dashrightarrow theorem(X))$ &
$(\forall\, X\ Y.\ axiom(or(not(Y),X))$ & $theorem(Y) \dashrightarrow theorem(X))$ &
$(\forall\, X\ Y\ Z.\ axiom(or(not(X),Y))$ & $theorem(or(not(Y),Z)) \dashrightarrow theorem(or(not(X),Z)))$
&
$(\sim theorem(or(not(or(not(p),q)),or(not(not(q)),not(p))))) \dashrightarrow False$
$\langle proof \rangle$

lemma *LCL200-1*:
$(\forall\, A.\ axiom(or(not(or(A,A)),A)))$ &
$(\forall\, B\ A.\ axiom(or(not(A),or(B,A))))$ &
$(\forall\, B\ A.\ axiom(or(not(or(A,B)),or(B,A))))$ &
$(\forall\, B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C)))))$ &
$(\forall\, A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B)))))$ &
$(\forall\, X.\ axiom(X) \dashrightarrow theorem(X))$ &
$(\forall\, X\ Y.\ axiom(or(not(Y),X))$ & $theorem(Y) \dashrightarrow theorem(X))$ &
$(\forall\, X\ Y\ Z.\ axiom(or(not(X),Y))$ & $theorem(or(not(Y),Z)) \dashrightarrow theorem(or(not(X),Z)))$
&
$(\sim theorem(or(not(not(or(p,q))),not(q)))) \dashrightarrow False$
$\langle proof \rangle$

lemma *LCL215-1*:
$(\forall\, A.\ axiom(or(not(or(A,A)),A)))$ &
$(\forall\, B\ A.\ axiom(or(not(A),or(B,A))))$ &
$(\forall\, B\ A.\ axiom(or(not(or(A,B)),or(B,A))))$ &
$(\forall\, B\ A\ C.\ axiom(or(not(or(A,or(B,C))),or(B,or(A,C)))))$ &
$(\forall\, A\ C\ B.\ axiom(or(not(or(not(A),B)),or(not(or(C,A)),or(C,B)))))$ &
$(\forall\, X.\ axiom(X) \dashrightarrow theorem(X))$ &
$(\forall\, X\ Y.\ axiom(or(not(Y),X))$ & $theorem(Y) \dashrightarrow theorem(X))$ &
$(\forall\, X\ Y\ Z.\ axiom(or(not(X),Y))$ & $theorem(or(not(Y),Z)) \dashrightarrow theorem(or(not(X),Z)))$
&
$(\sim theorem(or(not(or(not(p),q)),or(not(or(p,q)),q)))) \dashrightarrow False$
$\langle proof \rangle$

lemma *LCL230-2*:
$(q \dashrightarrow p \mid r)$ &
$(\sim p)$ &
(q) &
$(\sim r)$ --> *False*
$\langle proof \rangle$

lemma *LDA003-1*:
EQU001-0-ax equal &
$(\forall\, Y\ X\ Z.\ equal(f(X::'a,f(Y::'a,Z)),f(f(X::'a,Y),f(X::'a,Z))))$ &
$(\forall\, X\ Y.\ left(X::'a,f(X::'a,Y)))$ &
$(\forall\, Y\ X\ Z.\ left(X::'a,Y)$ & $left(Y::'a,Z)$ --> $left(X::'a,Z))$ &
$(equal(num2::'a,f(num1::'a,num1)))$ &
$(equal(num3::'a,f(num2::'a,num1)))$ &
$(equal(u::'a,f(num2::'a,num2)))$ &
$(\forall\, A\ B\ C.\ equal(A::'a,B)$ --> $equal(f(A::'a,C),f(B::'a,C)))$ &
$(\forall\, D\ F'\ E.\ equal(D::'a,E)$ --> $equal(f(F'::'a,D),f(F'::'a,E)))$ &
$(\forall\, G\ H\ I'.\ equal(G::'a,H)$ & $left(G::'a,I')$ --> $left(H::'a,I'))$ &
$(\forall\, J\ L\ K'.\ equal(J::'a,K')$ & $left(L::'a,J)$ --> $left(L::'a,K'))$ &
$(\sim left(num3::'a,u))$ --> *False*
$\langle proof \rangle$

lemma *MSC002-1*:
$(at(something::'a,here,now))$ &
$(\forall\, Place\ Situation.\ hand\text{-}at(Place::'a,Situation)$ --> $hand\text{-}at(Place::'a,let\text{-}go(Situation)))$
&
$(\forall\, Place\ Another\text{-}place\ Situation.\ hand\text{-}at(Place::'a,Situation)$ --> $hand\text{-}at(Another\text{-}place::'a,go(Another\text{-}pl$
&
$(\forall\, Thing\ Situation.\ \sim held(Thing::'a,let\text{-}go(Situation)))$ &
$(\forall\, Situation\ Thing.\ at(Thing::'a,here,Situation)$ --> $red(Thing))$ &
$(\forall\, Thing\ Place\ Situation.\ at(Thing::'a,Place,Situation)$ --> $at(Thing::'a,Place,let\text{-}go(Situation)))$
&
$(\forall\, Thing\ Place\ Situation.\ at(Thing::'a,Place,Situation)$ --> $at(Thing::'a,Place,pick\text{-}up(Situation)))$
&
$(\forall\, Thing\ Place\ Situation.\ at(Thing::'a,Place,Situation)$ --> $grabbed(Thing::'a,pick\text{-}up(go(Place::'a,let\text{-}go(Si$
&
$(\forall\, Thing\ Situation.\ red(Thing)$ & $put(Thing::'a,there,Situation)$ --> $answer(Situation))$
&
$(\forall\, Place\ Thing\ Another\text{-}place\ Situation.\ at(Thing::'a,Place,Situation)$ & $grabbed(Thing::'a,Situation)$
--> $put(Thing::'a,Another\text{-}place,go(Another\text{-}place::'a,Situation)))$ &
$(\forall\, Thing\ Place\ Another\text{-}place\ Situation.\ at(Thing::'a,Place,Situation)$ --> $held(Thing::'a,Situation)$
$\mid at(Thing::'a,Place,go(Another\text{-}place::'a,Situation)))$ &
$(\forall\, One\text{-}place\ Thing\ Place\ Situation.\ hand\text{-}at(One\text{-}place::'a,Situation)$ & $held(Thing::'a,Situation)$
--> $at(Thing::'a,Place,go(Place::'a,Situation)))$ &

(∀ *Place Thing Situation. hand-at*(*Place*::′*a*,*Situation*) & *at*(*Thing*::′*a*,*Place*,*Situation*)
−−> *held*(*Thing*::′*a*,*pick-up*(*Situation*))) &
(∀ *Situation.* ~*answer*(*Situation*)) −−> *False*
⟨*proof*⟩

lemma *MSC003-1*:
(∀ *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-o*
−−> *has-parts*(*Big-part*::′*a*,*mtimes*(*Number-of-mid-parts*::′*a*,*Number-of-small-parts*),*Small-part*))
&
(*in*′(*john*::′*a*,*boy*)) &
(∀ *X*. *in*′(*X*::′*a*,*boy*) −−> *in*′(*X*::′*a*,*human*)) &
(∀ *X*. *in*′(*X*::′*a*,*hand*) −−> *has-parts*(*X*::′*a*,*num5*,*fingers*)) &
(∀ *X*. *in*′(*X*::′*a*,*human*) −−> *has-parts*(*X*::′*a*,*num2*,*arm*)) &
(∀ *X*. *in*′(*X*::′*a*,*arm*) −−> *has-parts*(*X*::′*a*,*num1*,*hand*)) &
(~*has-parts*(*john*::′*a*,*mtimes*(*num2*::′*a*,*num1*),*hand*)) −−> *False*
⟨*proof*⟩

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-o*
−−> *has-parts*(*Big-part*::′*a*,*mtimes*(*Number-of-mid-parts*::′*a*,*Number-of-small-parts*),*Small-part*))
&
(*in*′(*john*::′*a*,*boy*)) &
(∀ *X*. *in*′(*X*::′*a*,*boy*) −−> *in*′(*X*::′*a*,*human*)) &
(∀ *X*. *in*′(*X*::′*a*,*hand*) −−> *has-parts*(*X*::′*a*,*num5*,*fingers*)) &
(∀ *X*. *in*′(*X*::′*a*,*human*) −−> *has-parts*(*X*::′*a*,*num2*,*arm*)) &
(∀ *X*. *in*′(*X*::′*a*,*arm*) −−> *has-parts*(*X*::′*a*,*num1*,*hand*)) &
(~*has-parts*(*john*::′*a*,*mtimes*(*mtimes*(*num2*::′*a*,*num1*),*num5*),*fingers*)) −−> *False*
⟨*proof*⟩

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*))
&

($\forall X\ Y.\ value(X::'a,falsity)\ \&\ value(Y::'a,truth) \dashrightarrow value(xor(X::'a,Y),truth)$)
&
($\forall X\ Y.\ value(X::'a,falsity)\ \&\ value(Y::'a,falsity) \dashrightarrow value(xor(X::'a,Y),falsity)$)
&
($\forall\ Value.\ {\sim}value(xor(xor(xor(xor(truth::'a,falsity),falsity),truth),falsity),Value)$)
$\dashrightarrow$ *False*
$\langle proof \rangle$

lemma *MSC006-1*:
($\forall Y\ X\ Z.\ p(X::'a,Y)\ \&\ p(Y::'a,Z) \dashrightarrow p(X::'a,Z)$) &
($\forall Y\ X\ Z.\ q(X::'a,Y)\ \&\ q(Y::'a,Z) \dashrightarrow q(X::'a,Z)$) &
($\forall Y\ X.\ q(X::'a,Y) \dashrightarrow q(Y::'a,X)$) &
($\forall X\ Y.\ p(X::'a,Y)\ |\ q(X::'a,Y)$) &
(${\sim}p(a::'a,b)$) &
(${\sim}q(c::'a,d)$) $\dashrightarrow$ *False*
$\langle proof \rangle$

lemma *NUM001-1*:
($\forall A.\ equal(A::'a,A)$) &
($\forall B\ A\ C.\ equal(A::'a,B)\ \&\ equal(B::'a,C) \dashrightarrow equal(A::'a,C)$) &
($\forall B\ A.\ equal(add(A::'a,B),add(B::'a,A))$) &
($\forall A\ B\ C.\ equal(add(A::'a,add(B::'a,C)),add(add(A::'a,B),C))$) &
($\forall B\ A.\ equal(subtract(add(A::'a,B),B),A)$) &
($\forall A\ B.\ equal(A::'a,subtract(add(A::'a,B),B))$) &
($\forall A\ C\ B.\ equal(add(subtract(A::'a,B),C),subtract(add(A::'a,C),B))$) &
($\forall A\ C\ B.\ equal(subtract(add(A::'a,B),C),add(subtract(A::'a,C),B))$) &
($\forall A\ C\ B\ D.\ equal(A::'a,B)\ \&\ equal(C::'a,add(A::'a,D)) \dashrightarrow equal(C::'a,add(B::'a,D))$)
&
($\forall A\ C\ D\ B.\ equal(A::'a,B)\ \&\ equal(C::'a,add(D::'a,A)) \dashrightarrow equal(C::'a,add(D::'a,B))$)
&
($\forall A\ C\ B\ D.\ equal(A::'a,B)\ \&\ equal(C::'a,subtract(A::'a,D)) \dashrightarrow equal(C::'a,subtract(B::'a,D))$)
&
($\forall A\ C\ D\ B.\ equal(A::'a,B)\ \&\ equal(C::'a,subtract(D::'a,A)) \dashrightarrow equal(C::'a,subtract(D::'a,B))$)
&
(${\sim}equal(add(add(a::'a,b),c),add(a::'a,add(b::'a,c)))$) $\dashrightarrow$ *False*
$\langle proof \rangle$

abbreviation *NUM001-0-ax multiply successor num0 add equal* $\equiv$
($\forall A.\ equal(add(A::'a,num0),A)$) &
($\forall A\ B.\ equal(add(A::'a,successor(B)),successor(add(A::'a,B)))$) &
($\forall A.\ equal(multiply(A::'a,num0),num0)$) &
($\forall B\ A.\ equal(multiply(A::'a,successor(B)),add(multiply(A::'a,B),A))$) &
($\forall A\ B.\ equal(successor(A),successor(B)) \dashrightarrow equal(A::'a,B)$) &
($\forall A\ B.\ equal(A::'a,B) \dashrightarrow equal(successor(A),successor(B))$)

abbreviation *NUM001-1-ax predecessor-of-1st-minus-2nd successor add equal mless*
$\equiv$

($\forall$ *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-of-1st-minus-2nd*(*B*::′*a*,*A*)),*A*),*B*))

abbreviation *NUM001-2-ax equal mless divides* ≡
($\forall$ *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*)) &
(~*mless*(*b*::′*a*,*a*)) &
(*divides*(*c*::′*a*,*a*)) &
($\forall$ *A*. ~*equal*(*successor*(*A*),*num0*)) −−> *False*
⟨*proof*⟩

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*. ~*equal*(*successor*(*A*),*num0*)) −−> *False*
⟨*proof*⟩

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* ≡
($\forall$ *X U Y*. *subclass*(*X*::′*a*,*Y*) & *member*(*U*::′*a*,*X*) −−> *member*(*U*::′*a*,*Y*)) &
($\forall$ *X Y*. *member*(*not-subclass-element*(*X*::′*a*,*Y*),*X*) | *subclass*(*X*::′*a*,*Y*)) &
($\forall$ *X Y*. *member*(*not-subclass-element*(*X*::′*a*,*Y*),*Y*) −−> *subclass*(*X*::′*a*,*Y*)) &
($\forall$ *X*. *subclass*(*X*::′*a*,*universal-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-pair*(*X*::′*a*,*Y*)) −−> *equal*(*U*::′*a*,*X*) | *equal*(*U*::′*a*,*Y*))

&
(∀ X Y. member(X::'a,universal-class) --> member(X::'a,unordered-pair(X::'a,Y)))
&
(∀ X Y. member(Y::'a,universal-class) --> member(Y::'a,unordered-pair(X::'a,Y)))
&
(∀ X Y. member(unordered-pair(X::'a,Y),universal-class)) &
(∀ X. equal(unordered-pair(X::'a,X),singleton(X))) &
(∀ X Y. equal(unordered-pair(singleton(X),unordered-pair(X::'a,singleton(Y))),ordered-pair(X::'a,Y)))
&
(∀ V Y U X. member(ordered-pair(U::'a,V),cross-product(X::'a,Y)) --> member(U::'a,X)) &
(∀ U X V Y. member(ordered-pair(U::'a,V),cross-product(X::'a,Y)) --> member(V::'a,Y)) &
(∀ U V X Y. member(U::'a,X) & member(V::'a,Y) --> member(ordered-pair(U::'a,V),cross-product(X::'a
&
(∀ X Y Z. member(Z::'a,cross-product(X::'a,Y)) --> equal(ordered-pair(first(Z),second(Z)),Z))
&
(subclass(element-relation::'a,cross-product(universal-class::'a,universal-class)))
&
(∀ X Y. member(ordered-pair(X::'a,Y),element-relation) --> member(X::'a,Y))
&
(∀ X Y. member(ordered-pair(X::'a,Y),cross-product(universal-class::'a,universal-class)) & member(X::'a,Y) --> member(ordered-pair(X::'a,Y),element-relation)) &
(∀ Y Z X. member(Z::'a,intersection(X::'a,Y)) --> member(Z::'a,X)) &
(∀ X Z Y. member(Z::'a,intersection(X::'a,Y)) --> member(Z::'a,Y)) &
(∀ Z X Y. member(Z::'a,X) & member(Z::'a,Y) --> member(Z::'a,intersection(X::'a,Y)))
&
(∀ Z X. ~(member(Z::'a,complement(X)) & member(Z::'a,X))) &
(∀ Z X. member(Z::'a,universal-class) --> member(Z::'a,complement(X)) | member(Z::'a,X)) &
(∀ X Y. equal(complement(intersection(complement(X),complement(Y))),union(X::'a,Y)))
&
(∀ X Y. equal(intersection(complement(intersection(X::'a,Y)),complement(intersection(complement(X),com
&
(∀ Xr X Y. equal(intersection(Xr::'a,cross-product(X::'a,Y)),restrct(Xr::'a,X,Y)))
&
(∀ Xr X Y. equal(intersection(cross-product(X::'a,Y),Xr),restrct(Xr::'a,X,Y)))
&
(∀ Z X. ~(equal(restrct(X::'a,singleton(Z),universal-class),null-class) & member(Z::'a,domain-of(X)))) &
(∀ Z X. member(Z::'a,universal-class) --> equal(restrct(X::'a,singleton(Z),universal-class),null-class) | member(Z::'a,domain-of(X))) &
(∀ X. subclass(rot(X),cross-product(cross-product(universal-class::'a,universal-class),universal-class)))
&
(∀ V W U X. member(ordered-pair(ordered-pair(U::'a,V),W),rot(X)) --> member(ordered-pair(ordered-pair(V::'a,W),U),X)) &
(∀ U V W X. member(ordered-pair(ordered-pair(V::'a,W),U),X) & member(ordered-pair(ordered-pair(U::'a
--> member(ordered-pair(ordered-pair(U::'a,V),W),rot(X))) &
(∀ X. subclass(flip(X),cross-product(cross-product(universal-class::'a,universal-class),universal-class)))

&
(∀ V U W X. member(ordered-pair(ordered-pair(U::'a, V), W),flip(X)) --> member(ordered-pair(ordered-pair(V::'a, U), W),X)) &
(∀ U V W X. member(ordered-pair(ordered-pair(V::'a, U), W),X) & member(ordered-pair(ordered-pair(U::'a
--> member(ordered-pair(ordered-pair(U::'a, V), W),flip(X))) &
(∀ Y. equal(domain-of(flip(cross-product(Y::'a,universal-class))),INVERSE(Y)))
&
(∀ Z. equal(domain-of(INVERSE(Z)),range-of(Z))) &
(∀ Z X Y. equal(first(not-subclass-element(restrct(Z::'a,X,singleton(Y)),null-class)),domain(Z::'a,X,Y)))
&
(∀ Z X Y. equal(second(not-subclass-element(restrct(Z::'a,singleton(X),Y),null-class)),rng(Z::'a,X,Y)))
&
(∀ Xr X. equal(range-of(restrct(Xr::'a,X,universal-class)),image'(Xr::'a,X))) &
(∀ X. equal(union(X::'a,singleton(X)),successor(X))) &
(subclass(successor-relation::'a,cross-product(universal-class::'a,universal-class)))
&
(∀ X Y. member(ordered-pair(X::'a, Y),successor-relation) --> equal(successor(X), Y))
&
(∀ X Y. equal(successor(X), Y) & member(ordered-pair(X::'a, Y),cross-product(universal-class::'a,universal-
--> member(ordered-pair(X::'a, Y),successor-relation)) &
(∀ X. inductive(X) --> member(null-class::'a,X)) &
(∀ X. inductive(X) --> subclass(image'(successor-relation::'a,X),X)) &
(∀ X. member(null-class::'a,X) & subclass(image'(successor-relation::'a,X),X)
--> inductive(X)) &
(inductive(omega)) &
(∀ Y. inductive(Y) --> subclass(omega::'a, Y)) &
(member(omega::'a,universal-class)) &
(∀ X. equal(domain-of(restrct(element-relation::'a,universal-class,X)),sum-class(X)))
&
(∀ X. member(X::'a,universal-class) --> member(sum-class(X),universal-class))
&
(∀ X. equal(complement(image'(element-relation::'a,complement(X))),powerClass(X)))
&
(∀ U. member(U::'a,universal-class) --> member(powerClass(U),universal-class))
&
(∀ Yr Xr. subclass(compos(Yr::'a,Xr),cross-product(universal-class::'a,universal-class)))
&
(∀ Z Yr Xr Y. member(ordered-pair(Y::'a,Z),compos(Yr::'a,Xr)) --> member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y))))) &
(∀ Y Z Yr Xr. member(Z::'a,image'(Yr::'a,image'(Xr::'a,singleton(Y)))) & member(ordered-pair(Y::'a,Z),cross-product(universal-class::'a,universal-class)) --> member(ordered-pair(Y::'a,Z),compos(Yr::'a,Xr))) &
(∀ X. single-valued-class(X) --> subclass(compos(X::'a,INVERSE(X)),identity-relation))
&
(∀ X. subclass(compos(X::'a,INVERSE(X)),identity-relation) --> single-valued-class(X))
&
(∀ Xf. function(Xf) --> subclass(Xf::'a,cross-product(universal-class::'a,universal-class)))
&
(∀ Xf. function(Xf) --> subclass(compos(Xf::'a,INVERSE(Xf)),identity-relation))

&
(∀ Xf. subclass(Xf::'a,cross-product(universal-class::'a,universal-class)) & subclass(compos(Xf::'a,INVERSE(Xf)),identity-relation) --> function(Xf)) &
(∀ Xf X. function(Xf) & member(X::'a,universal-class) --> member(image'(Xf::'a,X),universal-class))
&
(∀ X. equal(X::'a,null-class) | member(regular(X),X)) &
(∀ X. equal(X::'a,null-class) | equal(intersection(X::'a,regular(X)),null-class)) &
(∀ Xf Y. equal(sum-class(image'(Xf::'a,singleton(Y))),apply(Xf::'a,Y))) &
(function(choice)) &
(∀ Y. member(Y::'a,universal-class) --> equal(Y::'a,null-class) | member(apply(choice::'a,Y),Y))
&
(∀ Xf. one-to-one(Xf) --> function(Xf)) &
(∀ Xf. one-to-one(Xf) --> function(INVERSE(Xf))) &
(∀ Xf. function(INVERSE(Xf)) & function(Xf) --> one-to-one(Xf)) &
(equal(intersection(cross-product(universal-class::'a,universal-class),intersection(cross-product(universal-clas
&
(equal(intersection(INVERSE(subset-relation),subset-relation),identity-relation))
&
(∀ Xr. equal(complement(domain-of(intersection(Xr::'a,identity-relation))),diagonalise(Xr)))
&
(∀ X. equal(intersection(domain-of(X),diagonalise(compos(INVERSE(element-relation),X))),cantor(X)))
&
(∀ Xf. operation(Xf) --> function(Xf)) &
(∀ Xf. operation(Xf) --> equal(cross-product(domain-of(domain-of(Xf)),domain-of(domain-of(Xf))),dom
&
(∀ Xf. operation(Xf) --> subclass(range-of(Xf),domain-of(domain-of(Xf))))
&
(∀ Xf. function(Xf) & equal(cross-product(domain-of(domain-of(Xf)),domain-of(domain-of(Xf))),domain-
& subclass(range-of(Xf),domain-of(domain-of(Xf))) --> operation(Xf)) &
(∀ Xf1 Xf2 Xh. compatible(Xh::'a,Xf1,Xf2) --> function(Xh)) &
(∀ Xf2 Xf1 Xh. compatible(Xh::'a,Xf1,Xf2) --> equal(domain-of(domain-of(Xf1)),domain-of(Xh)))
&
(∀ Xf1 Xh Xf2. compatible(Xh::'a,Xf1,Xf2) --> subclass(range-of(Xh),domain-of(domain-of(Xf2))))
&
(∀ Xh Xh1 Xf1 Xf2. function(Xh) & equal(domain-of(domain-of(Xf1)),domain-of(Xh))
& subclass(range-of(Xh),domain-of(domain-of(Xf2))) --> compatible(Xh1::'a,Xf1,Xf2))
&
(∀ Xh Xf2 Xf1. homomorphism(Xh::'a,Xf1,Xf2) --> operation(Xf1)) &
(∀ Xh Xf1 Xf2. homomorphism(Xh::'a,Xf1,Xf2) --> operation(Xf2)) &
(∀ Xh Xf1 Xf2. homomorphism(Xh::'a,Xf1,Xf2) --> compatible(Xh::'a,Xf1,Xf2))
&
(∀ Xf2 Xh Xf1 X Y. homomorphism(Xh::'a,Xf1,Xf2) & member(ordered-pair(X::'a,Y),domain-of(Xf1))
--> equal(apply(Xf2::'a,ordered-pair(apply(Xh::'a,X),apply(Xh::'a,Y))),apply(Xh::'a,apply(Xf1::'a,ordered-
&
(∀ Xh Xf1 Xf2. operation(Xf1) & operation(Xf2) & compatible(Xh::'a,Xf1,Xf2)
--> member(ordered-pair(not-homomorphism1(Xh::'a,Xf1,Xf2),not-homomorphism2(Xh::'a,Xf1,Xf2)),dom
| homomorphism(Xh::'a,Xf1,Xf2)) &
(∀ Xh Xf1 Xf2. operation(Xf1) & operation(Xf2) & compatible(Xh::'a,Xf1,Xf2)
& equal(apply(Xf2::'a,ordered-pair(apply(Xh::'a,not-homomorphism1(Xh::'a,Xf1,Xf2)),apply(Xh::'a,not-hom

$--> homomorphism(Xh::'a,Xf1,Xf2))$

abbreviation *SET004-0-eq subclass single-valued-class operation one-to-one member inductive homomorphism function compatible unordered-pair union sum-class successor singleton second rot restrct regular range-of rng powerClass ordered-pair not-subclass-element not-homomorphism2 not-homomorphism1 INVERSE intersection image′ flip first domain-of domain difference diagonalise cross-product compos complement cantor apply equal* ≡

$(\forall D\ E\ F'.\ equal(D::'a,E) --> equal(apply(D::'a,F'),apply(E::'a,F')))$ &
$(\forall G\ I'\ H.\ equal(G::'a,H) --> equal(apply(I'::'a,G),apply(I'::'a,H)))$ &
$(\forall J\ K'.\ equal(J::'a,K') --> equal(cantor(J),cantor(K')))$ &
$(\forall L\ M.\ equal(L::'a,M) --> equal(complement(L),complement(M)))$ &
$(\forall N\ O'\ P.\ equal(N::'a,O') --> equal(compos(N::'a,P),compos(O'::'a,P)))$ &
$(\forall Q\ S'\ R.\ equal(Q::'a,R) --> equal(compos(S'::'a,Q),compos(S'::'a,R)))$ &
$(\forall T'\ U\ V.\ equal(T'::'a,U) --> equal(cross\text{-}product(T'::'a,V),cross\text{-}product(U::'a,V)))$
&
$(\forall W\ Y\ X.\ equal(W::'a,X) --> equal(cross\text{-}product(Y::'a,W),cross\text{-}product(Y::'a,X)))$
&
$(\forall Z\ A1.\ equal(Z::'a,A1) --> equal(diagonalise(Z),diagonalise(A1)))$ &
$(\forall B1\ C1\ D1.\ equal(B1::'a,C1) --> equal(difference(B1::'a,D1),difference(C1::'a,D1)))$
&
$(\forall E1\ G1\ F1.\ equal(E1::'a,F1) --> equal(difference(G1::'a,E1),difference(G1::'a,F1)))$
&
$(\forall H1\ I1\ J1\ K1.\ equal(H1::'a,I1) --> equal(domain(H1::'a,J1,K1),domain(I1::'a,J1,K1)))$
&
$(\forall L1\ N1\ M1\ O1.\ equal(L1::'a,M1) --> equal(domain(N1::'a,L1,O1),domain(N1::'a,M1,O1)))$
&
$(\forall P1\ R1\ S1\ Q1.\ equal(P1::'a,Q1) --> equal(domain(R1::'a,S1,P1),domain(R1::'a,S1,Q1)))$
&
$(\forall T1\ U1.\ equal(T1::'a,U1) --> equal(domain\text{-}of(T1),domain\text{-}of(U1)))$ &
$(\forall V1\ W1.\ equal(V1::'a,W1) --> equal(first(V1),first(W1)))$ &
$(\forall X1\ Y1.\ equal(X1::'a,Y1) --> equal(flip(X1),flip(Y1)))$ &
$(\forall Z1\ A2\ B2.\ equal(Z1::'a,A2) --> equal(image'(Z1::'a,B2),image'(A2::'a,B2)))$
&
$(\forall C2\ E2\ D2.\ equal(C2::'a,D2) --> equal(image'(E2::'a,C2),image'(E2::'a,D2)))$
&
$(\forall F2\ G2\ H2.\ equal(F2::'a,G2) --> equal(intersection(F2::'a,H2),intersection(G2::'a,H2)))$
&
$(\forall I2\ K2\ J2.\ equal(I2::'a,J2) --> equal(intersection(K2::'a,I2),intersection(K2::'a,J2)))$
&
$(\forall L2\ M2.\ equal(L2::'a,M2) --> equal(INVERSE(L2),INVERSE(M2)))$ &
$(\forall N2\ O2\ P2\ Q2.\ equal(N2::'a,O2) --> equal(not\text{-}homomorphism1(N2::'a,P2,Q2),not\text{-}homomorphism1(O2$
&
$(\forall R2\ T2\ S2\ U2.\ equal(R2::'a,S2) --> equal(not\text{-}homomorphism1(T2::'a,R2,U2),not\text{-}homomorphism1(T2:$
&
$(\forall V2\ X2\ Y2\ W2.\ equal(V2::'a,W2) --> equal(not\text{-}homomorphism1(X2::'a,Y2,V2),not\text{-}homomorphism1(X$
&
$(\forall Z2\ A3\ B3\ C3.\ equal(Z2::'a,A3) --> equal(not\text{-}homomorphism2(Z2::'a,B3,C3),not\text{-}homomorphism2(A3:$

&
(∀ D3 F3 E3 G3. equal(D3::'a,E3) --> equal(not-homomorphism2(F3::'a,D3,G3),not-homomorphism2(F3
&
(∀ H3 J3 K3 I3. equal(H3::'a,I3) --> equal(not-homomorphism2(J3::'a,K3,H3),not-homomorphism2(J3::'
&
(∀ L3 M3 N3. equal(L3::'a,M3) --> equal(not-subclass-element(L3::'a,N3),not-subclass-element(M3::'a,N3
&
(∀ O3 Q3 P3. equal(O3::'a,P3) --> equal(not-subclass-element(Q3::'a,O3),not-subclass-element(Q3::'a,P3
&
(∀ R3 S3 T3. equal(R3::'a,S3) --> equal(ordered-pair(R3::'a,T3),ordered-pair(S3::'a,T3)))
&
(∀ U3 W3 V3. equal(U3::'a,V3) --> equal(ordered-pair(W3::'a,U3),ordered-pair(W3::'a,V3)))
&
(∀ X3 Y3. equal(X3::'a,Y3) --> equal(powerClass(X3),powerClass(Y3))) &
(∀ Z3 A4 B4 C4. equal(Z3::'a,A4) --> equal(rng(Z3::'a,B4,C4),rng(A4::'a,B4,C4)))
&
(∀ D4 F4 E4 G4. equal(D4::'a,E4) --> equal(rng(F4::'a,D4,G4),rng(F4::'a,E4,G4)))
&
(∀ H4 J4 K4 I4. equal(H4::'a,I4) --> equal(rng(J4::'a,K4,H4),rng(J4::'a,K4,I4)))
&
(∀ L4 M4. equal(L4::'a,M4) --> equal(range-of(L4),range-of(M4))) &
(∀ N4 O4. equal(N4::'a,O4) --> equal(regular(N4),regular(O4))) &
(∀ P4 Q4 R4 S4. equal(P4::'a,Q4) --> equal(restrct(P4::'a,R4,S4),restrct(Q4::'a,R4,S4)))
&
(∀ T4 V4 U4 W4. equal(T4::'a,U4) --> equal(restrct(V4::'a,T4,W4),restrct(V4::'a,U4,W4)))
&
(∀ X4 Z4 A5 Y4. equal(X4::'a,Y4) --> equal(restrct(Z4::'a,A5,X4),restrct(Z4::'a,A5,Y4)))
&
(∀ B5 C5. equal(B5::'a,C5) --> equal(rot(B5),rot(C5))) &
(∀ D5 E5. equal(D5::'a,E5) --> equal(second(D5),second(E5))) &
(∀ F5 G5. equal(F5::'a,G5) --> equal(singleton(F5),singleton(G5))) &
(∀ H5 I5. equal(H5::'a,I5) --> equal(successor(H5),successor(I5))) &
(∀ J5 K5. equal(J5::'a,K5) --> equal(sum-class(J5),sum-class(K5))) &
(∀ L5 M5 N5. equal(L5::'a,M5) --> equal(union(L5::'a,N5),union(M5::'a,N5)))
&
(∀ O5 Q5 P5. equal(O5::'a,P5) --> equal(union(Q5::'a,O5),union(Q5::'a,P5)))
&
(∀ R5 S5 T5. equal(R5::'a,S5) --> equal(unordered-pair(R5::'a,T5),unordered-pair(S5::'a,T5)))
&
(∀ U5 W5 V5. equal(U5::'a,V5) --> equal(unordered-pair(W5::'a,U5),unordered-pair(W5::'a,V5)))
&
(∀ X5 Y5 Z5 A6. equal(X5::'a,Y5) & compatible(X5::'a,Z5,A6) --> compatible(Y5::'a,Z5,A6)) &
(∀ B6 D6 C6 E6. equal(B6::'a,C6) & compatible(D6::'a,B6,E6) --> compatible(D6::'a,C6,E6)) &
(∀ F6 H6 I6 G6. equal(F6::'a,G6) & compatible(H6::'a,I6,F6) --> compatible(H6::'a,I6,G6)) &
(∀ J6 K6. equal(J6::'a,K6) & function(J6) --> function(K6)) &
(∀ L6 M6 N6 O6. equal(L6::'a,M6) & homomorphism(L6::'a,N6,O6) --> ho-

momorphism$(M6::'a,N6,O6)$) &
(∀ *P6 R6 Q6 S6*. *equal*$(P6::'a,Q6)$ & *homomorphism*$(R6::'a,P6,S6)$ −−> *homomorphism*$(R6::'a,Q6,S6)$) &
(∀ *T6 V6 W6 U6*. *equal*$(T6::'a,U6)$ & *homomorphism*$(V6::'a,W6,T6)$ −−> *homomorphism*$(V6::'a,W6,U6)$) &
(∀ *X6 Y6*. *equal*$(X6::'a,Y6)$ & *inductive*$(X6)$ −−> *inductive*$(Y6)$) &
(∀ *Z6 A7 B7*. *equal*$(Z6::'a,A7)$ & *member*$(Z6::'a,B7)$ −−> *member*$(A7::'a,B7)$)
&
(∀ *C7 E7 D7*. *equal*$(C7::'a,D7)$ & *member*$(E7::'a,C7)$ −−> *member*$(E7::'a,D7)$)
&
(∀ *F7 G7*. *equal*$(F7::'a,G7)$ & *one-to-one*$(F7)$ −−> *one-to-one*$(G7)$) &
(∀ *H7 I7*. *equal*$(H7::'a,I7)$ & *operation*$(H7)$ −−> *operation*$(I7)$) &
(∀ *J7 K7*. *equal*$(J7::'a,K7)$ & *single-valued-class*$(J7)$ −−> *single-valued-class*$(K7)$)
&
(∀ *L7 M7 N7*. *equal*$(L7::'a,M7)$ & *subclass*$(L7::'a,N7)$ −−> *subclass*$(M7::'a,N7)$)
&
(∀ *O7 Q7 P7*. *equal*$(O7::'a,P7)$ & *subclass*$(Q7::'a,O7)$ −−> *subclass*$(Q7::'a,P7)$)

abbreviation *SET004-1-ax range-of function maps apply application-function singleton-relation element-relation complement intersection single-valued3 singleton image′ domain single-valued2 second single-valued1 identity-relation INVERSE not-subclass-element first domain-of domain-relation composition-function compos equal ordered-pair member universal-class cross-product compose-class subclass* ≡
(∀ *X*. *subclass*(*compose-class*(X),*cross-product*(*universal-class*::′*a*,*universal-class*)))
&
(∀ *X Y Z*. *member*(*ordered-pair*$(Y::'a,Z)$,*compose-class*(X)) −−> *equal*(*compos*$(X::'a,Y)$,Z))
&
(∀ *Y Z X*. *member*(*ordered-pair*$(Y::'a,Z)$,*cross-product*(*universal-class*::′*a*,*universal-class*))
& *equal*(*compos*$(X::'a,Y)$,Z) −−> *member*(*ordered-pair*$(Y::'a,Z)$,*compose-class*(X)))
&
(*subclass*(*composition-function*::′*a*,*cross-product*(*universal-class*::′*a*,*cross-product*(*universal-class*::′*a*,*universa*
&
(∀ *X Y Z*. *member*(*ordered-pair*$(X::'a$,*ordered-pair*$(Y::'a,Z))$,*composition-function*)
−−> *equal*(*compos*$(X::'a,Y)$,Z)) &
(∀ *X Y*. *member*(*ordered-pair*$(X::'a,Y)$,*cross-product*(*universal-class*::′*a*,*universal-class*))
−−> *member*(*ordered-pair*$(X::'a$,*ordered-pair*$(Y::'a$,*compos*$(X::'a,Y)$)),*composition-function*))
&
(*subclass*(*domain-relation*::′*a*,*cross-product*(*universal-class*::′*a*,*universal-class*))) &
(∀ *X Y*. *member*(*ordered-pair*$(X::'a,Y)$,*domain-relation*) −−> *equal*(*domain-of*(X),Y))
&
(∀ *X*. *member*$(X::'a$,*universal-class*) −−> *member*(*ordered-pair*$(X::'a$,*domain-of*(X)),*domain-relation*))
&
(∀ *X*. *equal*(*first*(*not-subclass-element*(*compos*$(X::'a$,*INVERSE*(X)),*identity-relation*)),*single-valued1*(X)))
&
(∀ *X*. *equal*(*second*(*not-subclass-element*(*compos*$(X::'a$,*INVERSE*(X)),*identity-relation*)),*single-valued2*(X))
&
(∀ *X*. *equal*(*domain*$(X::'a$,*image′*(*INVERSE*(X),*singleton*(*single-valued1*(X))),*single-valued2*(X)),*single-valu*

&
$(equal(intersection(complement(compos(element\text{-}relation::'a,complement(identity\text{-}relation))),element\text{-}relatio$
&
$(subclass(application\text{-}function::'a,cross\text{-}product(universal\text{-}class::'a,cross\text{-}product(universal\text{-}class::'a,universal\text{-}$
&
$(\forall Z\ Y\ X.\ member(ordered\text{-}pair(X::'a,ordered\text{-}pair(Y::'a,Z)),application\text{-}function)$
$--> member(Y::'a,domain\text{-}of(X)))$ &
$(\forall X\ Y\ Z.\ member(ordered\text{-}pair(X::'a,ordered\text{-}pair(Y::'a,Z)),application\text{-}function)$
$--> equal(apply(X::'a,Y),Z))$ &
$(\forall Z\ X\ Y.\ member(ordered\text{-}pair(X::'a,ordered\text{-}pair(Y::'a,Z)),cross\text{-}product(universal\text{-}class::'a,cross\text{-}product($
& $member(Y::'a,domain\text{-}of(X)) --> member(ordered\text{-}pair(X::'a,ordered\text{-}pair(Y::'a,apply(X::'a,Y))),applic$
&
$(\forall X\ Y\ Xf.\ maps(Xf::'a,X,Y) --> function(Xf))$ &
$(\forall Y\ Xf\ X.\ maps(Xf::'a,X,Y) --> equal(domain\text{-}of(Xf),X))$ &
$(\forall X\ Xf\ Y.\ maps(Xf::'a,X,Y) --> subclass(range\text{-}of(Xf),Y))$ &
$(\forall Xf\ Y.\ function(Xf)$ & $subclass(range\text{-}of(Xf),Y) --> maps(Xf::'a,domain\text{-}of(Xf),Y))$

abbreviation *SET004-1-eq maps single-valued3 single-valued2 single-valued1 compose-class equal* $\equiv$
$(\forall L\ M.\ equal(L::'a,M) --> equal(compose\text{-}class(L),compose\text{-}class(M)))$ &
$(\forall N2\ O2.\ equal(N2::'a,O2) --> equal(single\text{-}valued1(N2),single\text{-}valued1(O2)))$
&
$(\forall P2\ Q2.\ equal(P2::'a,Q2) --> equal(single\text{-}valued2(P2),single\text{-}valued2(Q2)))$
&
$(\forall R2\ S2.\ equal(R2::'a,S2) --> equal(single\text{-}valued3(R2),single\text{-}valued3(S2)))$
&
$(\forall X2\ Y2\ Z2\ A3.\ equal(X2::'a,Y2)$ & $maps(X2::'a,Z2,A3) --> maps(Y2::'a,Z2,A3))$
&
$(\forall B3\ D3\ C3\ E3.\ equal(B3::'a,C3)$ & $maps(D3::'a,B3,E3) --> maps(D3::'a,C3,E3))$
&
$(\forall F3\ H3\ I3\ G3.\ equal(F3::'a,G3)$ & $maps(H3::'a,I3,F3) --> maps(H3::'a,I3,G3))$

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* $\equiv$
$(\forall X.\ equal(union(X::'a,INVERSE(X)),symmetrization\text{-}of(X)))$ &
$(\forall X\ Y.\ irreflexive(X::'a,Y) --> subclass(restrct(X::'a,Y,Y),complement(identity\text{-}relation)))$
&
$(\forall X\ Y.\ subclass(restrct(X::'a,Y,Y),complement(identity\text{-}relation)) --> irreflexive(X::'a,Y))$ &
$(\forall Y\ X.\ connected(X::'a,Y) --> subclass(cross\text{-}product(Y::'a,Y),union(identity\text{-}relation::'a,symmetrizatio$
&
$(\forall X\ Y.\ subclass(cross\text{-}product(Y::'a,Y),union(identity\text{-}relation::'a,symmetrization\text{-}of(X)))$

−−> connected(X::'a, Y)) &
(∀ Xr Y. transitive(Xr::'a, Y) −−> subclass(compos(restrct(Xr::'a, Y, Y),restrct(Xr::'a, Y, Y)),restrct(Xr::'a
&
(∀ Xr Y. subclass(compos(restrct(Xr::'a, Y, Y),restrct(Xr::'a, Y, Y)),restrct(Xr::'a, Y, Y))
−−> transitive(Xr::'a, Y)) &
(∀ Xr Y. asymmetric(Xr::'a, Y) −−> equal(restrct(intersection(Xr::'a,INVERSE(Xr)), Y, Y),null-class))
&
(∀ Xr Y. equal(restrct(intersection(Xr::'a,INVERSE(Xr)), Y, Y),null-class) −−>
asymmetric(Xr::'a, Y)) &
(∀ Xr Y Z. equal(segment(Xr::'a, Y,Z),domain-of(restrct(Xr::'a, Y,singleton(Z)))))
&
(∀ X Y. well-ordering(X::'a, Y) −−> connected(X::'a, Y)) &
(∀ Y Xr U. well-ordering(Xr::'a, Y) & subclass(U::'a, Y) −−> equal(U::'a,null-class)
| member(least(Xr::'a, U), U)) &
(∀ Y V Xr U. well-ordering(Xr::'a, Y) & subclass(U::'a, Y) & member(V::'a, U)
−−> member(least(Xr::'a, U), U)) &
(∀ Y Xr U. well-ordering(Xr::'a, Y) & subclass(U::'a, Y) −−> equal(segment(Xr::'a, U,least(Xr::'a, U)),null-
&
(∀ Y V U Xr. ~(well-ordering(Xr::'a, Y) & subclass(U::'a, Y) & member(V::'a, U)
& member(ordered-pair(V::'a,least(Xr::'a, U)),Xr))) &
(∀ Xr Y. connected(Xr::'a, Y) & equal(not-well-ordering(Xr::'a, Y),null-class)
−−> well-ordering(Xr::'a, Y)) &
(∀ Xr Y. connected(Xr::'a, Y) −−> subclass(not-well-ordering(Xr::'a, Y), Y) |
well-ordering(Xr::'a, Y)) &
(∀ V Xr Y. member(V::'a,not-well-ordering(Xr::'a, Y)) & equal(segment(Xr::'a,not-well-ordering(Xr::'a, Y),
& connected(Xr::'a, Y) −−> well-ordering(Xr::'a, Y)) &
(∀ Xr Y Z. section(Xr::'a, Y,Z) −−> subclass(Y::'a,Z)) &
(∀ Xr Z Y. section(Xr::'a, Y,Z) −−> subclass(domain-of(restrct(Xr::'a,Z, Y)), Y))
&
(∀ Xr Y Z. subclass(Y::'a,Z) & subclass(domain-of(restrct(Xr::'a,Z, Y)), Y) −−>
section(Xr::'a, Y,Z)) &
(∀ X. member(X::'a,ordinal-numbers) −−> well-ordering(element-relation::'a,X))
&
(∀ X. member(X::'a,ordinal-numbers) −−> subclass(sum-class(X),X)) &
(∀ X. well-ordering(element-relation::'a,X) & subclass(sum-class(X),X) & member(X::'a,universal-class) −−> member(X::'a,ordinal-numbers)) &
(∀ X. well-ordering(element-relation::'a,X) & subclass(sum-class(X),X) −−>
member(X::'a,ordinal-numbers) | equal(X::'a,ordinal-numbers)) &
(equal(union(singleton(null-class),image'(successor-relation::'a,ordinal-numbers)),kind-1-ordinals))
&
(equal(intersection(complement(kind-1-ordinals),ordinal-numbers),limit-ordinals))
&
(∀ X. subclass(rest-of(X),cross-product(universal-class::'a,universal-class))) &
(∀ V U X. member(ordered-pair(U::'a, V),rest-of(X)) −−> member(U::'a,domain-of(X)))
&
(∀ X U V. member(ordered-pair(U::'a, V),rest-of(X)) −−> equal(restrct(X::'a, U,universal-class), V))
&
(∀ U V X. member(U::'a,domain-of(X)) & equal(restrct(X::'a, U,universal-class), V)
−−> member(ordered-pair(U::'a, V),rest-of(X))) &

(*subclass*(*rest-relation*::'*a*,*cross-product*(*universal-class*::'*a*,*universal-class*))) &
(∀ X Y. *member*(*ordered-pair*(X::'*a*,Y),*rest-relation*) --> *equal*(*rest-of*(X),Y))
&
(∀ X. *member*(X::'*a*,*universal-class*) --> *member*(*ordered-pair*(X::'*a*,*rest-of*(X)),*rest-relation*))
&
(∀ X Z. *member*(X::'*a*,*recursion-equation-functions*(Z)) --> *function*(Z)) &
(∀ Z X. *member*(X::'*a*,*recursion-equation-functions*(Z)) --> *function*(X)) &
(∀ Z X. *member*(X::'*a*,*recursion-equation-functions*(Z)) --> *member*(*domain-of*(X),*ordinal-numbers*))
&
(∀ Z X. *member*(X::'*a*,*recursion-equation-functions*(Z)) --> *equal*(*compos*(Z::'*a*,*rest-of*(X)),X))
&
(∀ X Z. *function*(Z) & *function*(X) & *member*(*domain-of*(X),*ordinal-numbers*) &
equal(*compos*(Z::'*a*,*rest-of*(X)),X) --> *member*(X::'*a*,*recursion-equation-functions*(Z)))
&
(*subclass*(*union-of-range-map*::'*a*,*cross-product*(*universal-class*::'*a*,*universal-class*)))
&
(∀ X Y. *member*(*ordered-pair*(X::'*a*,Y),*union-of-range-map*) --> *equal*(*sum-class*(*range-of*(X)),Y))
&
(∀ X Y. *member*(*ordered-pair*(X::'*a*,Y),*cross-product*(*universal-class*::'*a*,*universal-class*))
& *equal*(*sum-class*(*range-of*(X)),Y) --> *member*(*ordered-pair*(X::'*a*,Y),*union-of-range-map*))
&
(∀ X Y. *equal*(*apply*(*recursion*(X::'*a*,*successor-relation*,*union-of-range-map*),Y),*ordinal-add*(X::'*a*,Y)))
&
(∀ X Y. *equal*(*recursion*(*null-class*::'*a*,*apply*(*add-relation*::'*a*,X),*union-of-range-map*),*ordinal-multiply*(X::'*a*,
&
(∀ X. *member*(X::'*a*,*omega*) --> *equal*(*integer-of*(X),X)) &
(∀ X. *member*(X::'*a*,*omega*) | *equal*(*integer-of*(X),*null-class*))

abbreviation *NUM004-0-eq well-ordering transitive section irreflexive connected asymmetric symmetrization-of segment rest-of recursion-equation-functions recursion ordinal-multiply ordinal-add not-well-ordering least integer-of equal* ≡
(∀ D E. *equal*(D::'*a*,E) --> *equal*(*integer-of*(D),*integer-of*(E))) &
(∀ F' G H. *equal*(F'::'*a*,G) --> *equal*(*least*(F'::'*a*,H),*least*(G::'*a*,H))) &
(∀ I' K' J. *equal*(I'::'*a*,J) --> *equal*(*least*(K'::'*a*,I'),*least*(K'::'*a*,J))) &
(∀ L M N. *equal*(L::'*a*,M) --> *equal*(*not-well-ordering*(L::'*a*,N),*not-well-ordering*(M::'*a*,N)))
&
(∀ O' Q P. *equal*(O'::'*a*,P) --> *equal*(*not-well-ordering*(Q::'*a*,O'),*not-well-ordering*(Q::'*a*,P)))
&
(∀ R S' T'. *equal*(R::'*a*,S') --> *equal*(*ordinal-add*(R::'*a*,T'),*ordinal-add*(S'::'*a*,T')))
&
(∀ U W V. *equal*(U::'*a*,V) --> *equal*(*ordinal-add*(W::'*a*,U),*ordinal-add*(W::'*a*,V)))
&
(∀ X Y Z. *equal*(X::'*a*,Y) --> *equal*(*ordinal-multiply*(X::'*a*,Z),*ordinal-multiply*(Y::'*a*,Z)))
&
(∀ A1 C1 B1. *equal*(A1::'*a*,B1) --> *equal*(*ordinal-multiply*(C1::'*a*,A1),*ordinal-multiply*(C1::'*a*,B1)))
&
(∀ F1 G1 H1 I1. *equal*(F1::'*a*,G1) --> *equal*(*recursion*(F1::'*a*,H1,I1),*recursion*(G1::'*a*,H1,I1)))
&

$(\forall\ J1\ L1\ K1\ M1.\ equal(J1::'a,K1) \ {-}{-}{>}\ equal(recursion(L1::'a,J1,M1),recursion(L1::'a,K1,M1)))$
&
$(\forall\ N1\ P1\ Q1\ O1.\ equal(N1::'a,O1) \ {-}{-}{>}\ equal(recursion(P1::'a,Q1,N1),recursion(P1::'a,Q1,O1)))$
&
$(\forall\ R1\ S1.\ equal(R1::'a,S1) \ {-}{-}{>}\ equal(recursion\text{-}equation\text{-}functions(R1),recursion\text{-}equation\text{-}functions(S1)))$
&
$(\forall\ T1\ U1.\ equal(T1::'a,U1) \ {-}{-}{>}\ equal(rest\text{-}of(T1),rest\text{-}of(U1)))$ &
$(\forall\ V1\ W1\ X1\ Y1.\ equal(V1::'a,W1) \ {-}{-}{>}\ equal(segment(V1::'a,X1,Y1),segment(W1::'a,X1,Y1)))$
&
$(\forall\ Z1\ B2\ A2\ C2.\ equal(Z1::'a,A2) \ {-}{-}{>}\ equal(segment(B2::'a,Z1,C2),segment(B2::'a,A2,C2)))$
&
$(\forall\ D2\ F2\ G2\ E2.\ equal(D2::'a,E2) \ {-}{-}{>}\ equal(segment(F2::'a,G2,D2),segment(F2::'a,G2,E2)))$
&
$(\forall\ H2\ I2.\ equal(H2::'a,I2) \ {-}{-}{>}\ equal(symmetrization\text{-}of(H2),symmetrization\text{-}of(I2)))$
&
$(\forall\ J2\ K2\ L2.\ equal(J2::'a,K2)\ \&\ asymmetric(J2::'a,L2) \ {-}{-}{>}\ asymmetric(K2::'a,L2))$
&
$(\forall\ M2\ O2\ N2.\ equal(M2::'a,N2)\ \&\ asymmetric(O2::'a,M2) \ {-}{-}{>}\ asymmetric(O2::'a,N2))$
&
$(\forall\ P2\ Q2\ R2.\ equal(P2::'a,Q2)\ \&\ connected(P2::'a,R2) \ {-}{-}{>}\ connected(Q2::'a,R2))$
&
$(\forall\ S2\ U2\ T2.\ equal(S2::'a,T2)\ \&\ connected(U2::'a,S2) \ {-}{-}{>}\ connected(U2::'a,T2))$
&
$(\forall\ V2\ W2\ X2.\ equal(V2::'a,W2)\ \&\ irreflexive(V2::'a,X2) \ {-}{-}{>}\ irreflexive(W2::'a,X2))$
&
$(\forall\ Y2\ A3\ Z2.\ equal(Y2::'a,Z2)\ \&\ irreflexive(A3::'a,Y2) \ {-}{-}{>}\ irreflexive(A3::'a,Z2))$
&
$(\forall\ B3\ C3\ D3\ E3.\ equal(B3::'a,C3)\ \&\ section(B3::'a,D3,E3) \ {-}{-}{>}\ section(C3::'a,D3,E3))$
&
$(\forall\ F3\ H3\ G3\ I3.\ equal(F3::'a,G3)\ \&\ section(H3::'a,F3,I3) \ {-}{-}{>}\ section(H3::'a,G3,I3))$
&
$(\forall\ J3\ L3\ M3\ K3.\ equal(J3::'a,K3)\ \&\ section(L3::'a,M3,J3) \ {-}{-}{>}\ section(L3::'a,M3,K3))$
&
$(\forall\ N3\ O3\ P3.\ equal(N3::'a,O3)\ \&\ transitive(N3::'a,P3) \ {-}{-}{>}\ transitive(O3::'a,P3))$
&
$(\forall\ Q3\ S3\ R3.\ equal(Q3::'a,R3)\ \&\ transitive(S3::'a,Q3) \ {-}{-}{>}\ transitive(S3::'a,R3))$
&
$(\forall\ T3\ U3\ V3.\ equal(T3::'a,U3)\ \&\ well\text{-}ordering(T3::'a,V3) \ {-}{-}{>}\ well\text{-}ordering(U3::'a,V3))$
&
$(\forall\ W3\ Y3\ X3.\ equal(W3::'a,X3)\ \&\ well\text{-}ordering(Y3::'a,W3) \ {-}{-}{>}\ well\text{-}ordering(Y3::'a,X3))$

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)))
&

$(\forall\, Situation.\ at(c::'a,Situation)\ {-}{-}{>}\ at(d::'a,go(d::'a,Situation)))\ \&$
$(\forall\, Situation.\ at(d::'a,Situation)\ {-}{-}{>}\ at(c::'a,go(c::'a,Situation)))\ \&$
$(\forall\, Situation.\ at(c::'a,Situation)\ {-}{-}{>}\ at(e::'a,go(e::'a,Situation)))\ \&$
$(\forall\, Situation.\ at(e::'a,Situation)\ {-}{-}{>}\ at(c::'a,go(c::'a,Situation)))\ \&$
$(\forall\, Situation.\ at(d::'a,Situation)\ {-}{-}{>}\ at(f::'a,go(f::'a,Situation)))\ \&$
$(\forall\, Situation.\ at(f::'a,Situation)\ {-}{-}{>}\ at(d::'a,go(d::'a,Situation)))\ \&$
$(at(f::'a,s0))\ \&$
$(\forall\, S'.\ {\sim}at(a::'a,S'))\ {-}{-}{>}\ False$
$\langle proof \rangle$

abbreviation *PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds* $\equiv$
$(\forall\, X\ Y\ State.\ holds(X::'a,State)\ \&\ holds(Y::'a,State)\ {-}{-}{>}\ holds(and'(X::'a,Y),State))$
$\&$
$(\forall\, State\ X.\ holds(EMPTY::'a,State)\ \&\ holds(clear(X),State)\ \&\ differ(X::'a,table)$
${-}{-}{>}\ holds(holding(X),do(pickup(X),State)))\ \&$
$(\forall\, Y\ X\ State.\ holds(on(X::'a,Y),State)\ \&\ holds(clear(X),State)\ \&\ holds(EMPTY::'a,State)$
${-}{-}{>}\ holds(clear(Y),do(pickup(X),State)))\ \&$
$(\forall\, Y\ State\ X\ Z.\ holds(on(X::'a,Y),State)\ \&\ differ(X::'a,Z)\ {-}{-}{>}\ holds(on(X::'a,Y),do(pickup(Z),State)))$
$\&$
$(\forall\, State\ X\ Z.\ holds(clear(X),State)\ \&\ differ(X::'a,Z)\ {-}{-}{>}\ holds(clear(X),do(pickup(Z),State)))$
$\&$
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State)\ {-}{-}{>}\ holds(EMPTY::'a,do(putdown(X::'a,Y$
$\&$
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State)\ {-}{-}{>}\ holds(on(X::'a,Y),do(putdown(X::'a,Y$
$\&$
$(\forall\, X\ Y\ State.\ holds(holding(X),State)\ \&\ holds(clear(Y),State)\ {-}{-}{>}\ holds(clear(X),do(putdown(X::'a,Y),St$
$\&$
$(\forall\, Z\ W\ X\ Y\ State.\ holds(on(X::'a,Y),State)\ {-}{-}{>}\ holds(on(X::'a,Y),do(putdown(Z::'a,W),State)))$
$\&$
$(\forall\, X\ State\ Z\ Y.\ holds(clear(Z),State)\ \&\ differ(Z::'a,Y)\ {-}{-}{>}\ holds(clear(Z),do(putdown(X::'a,Y),State)))$

abbreviation *PLA001-1-ax EMPTY clear s0 on holds table d c b a differ* $\equiv$
$(\forall\, Y\ X.\ differ(Y::'a,X)\ {-}{-}{>}\ differ(X::'a,Y))\ \&$
$(differ(a::'a,b))\ \&$
$(differ(a::'a,c))\ \&$
$(differ(a::'a,d))\ \&$
$(differ(a::'a,table))\ \&$
$(differ(b::'a,c))\ \&$
$(differ(b::'a,d))\ \&$
$(differ(b::'a,table))\ \&$
$(differ(c::'a,d))\ \&$
$(differ(c::'a,table))\ \&$
$(differ(d::'a,table))\ \&$
$(holds(on(a::'a,table),s0))\ \&$
$(holds(on(b::'a,table),s0))\ \&$
$(holds(on(c::'a,d),s0))\ \&$
$(holds(on(d::'a,table),s0))\ \&$
$(holds(clear(a),s0))\ \&$

$(holds(clear(b),s0))$ &
$(holds(clear(c),s0))$ &
$(holds(EMPTY::'a,s0))$ &
$(\forall State.\ holds(clear(table),State))$

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 &
$(\forall State.\ \sim holds(on(c::'a,table),State))$ $-->$ *False*
⟨*proof*⟩

lemma *PLA017-1*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
$(\forall State.\ \sim holds(on(a::'a,c),State))$ $-->$ *False*
⟨*proof*⟩

lemma *PLA022-1*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
$(\forall State.\ \sim holds(and'(on(c::'a,d),on(a::'a,c)),State))$ $-->$ *False*
⟨*proof*⟩

lemma *PLA022-2*:
PLA001-0-ax putdown on pickup do holding table differ clear EMPTY and' holds
&
PLA001-1-ax EMPTY clear s0 on holds table d c b a differ &
$(\forall State.\ \sim holds(and'(on(a::'a,c),on(c::'a,d)),State))$ $-->$ *False*
⟨*proof*⟩

lemma *PRV001-1*:
$(\forall X\ Y\ Z.\ q1(X::'a,Y,Z)$ & $mless\text{-}or\text{-}equal(X::'a,Y)$ $-->$ $q2(X::'a,Y,Z))$ &
$(\forall X\ Y\ Z.\ q1(X::'a,Y,Z)$ $-->$ $mless\text{-}or\text{-}equal(X::'a,Y) \mid q3(X::'a,Y,Z))$ &
$(\forall Z\ X\ Y.\ q2(X::'a,Y,Z)$ $-->$ $q4(X::'a,Y,Y))$ &
$(\forall Z\ Y\ X.\ q3(X::'a,Y,Z)$ $-->$ $q4(X::'a,Y,X))$ &
$(\forall X.\ mless\text{-}or\text{-}equal(X::'a,X))$ &
$(\forall X\ Y.\ mless\text{-}or\text{-}equal(X::'a,Y)$ & $mless\text{-}or\text{-}equal(Y::'a,X)$ $-->$ $equal(X::'a,Y))$
&
$(\forall Y\ X\ Z.\ mless\text{-}or\text{-}equal(X::'a,Y)$ & $mless\text{-}or\text{-}equal(Y::'a,Z)$ $-->$ $mless\text{-}or\text{-}equal(X::'a,Z))$
&
$(\forall Y\ X.\ mless\text{-}or\text{-}equal(X::'a,Y) \mid mless\text{-}or\text{-}equal(Y::'a,X))$ &

$(\forall X\ Y.\ equal(X::'a,Y) \dashrightarrow mless\text{-}or\text{-}equal(X::'a,Y))$ &
$(\forall X\ Y\ Z.\ equal(X::'a,Y)$ & $mless\text{-}or\text{-}equal(X::'a,Z) \dashrightarrow mless\text{-}or\text{-}equal(Y::'a,Z))$
&
$(\forall X\ Z\ Y.\ equal(X::'a,Y)$ & $mless\text{-}or\text{-}equal(Z::'a,X) \dashrightarrow mless\text{-}or\text{-}equal(Z::'a,Y))$
&
$(q1(a::'a,b,c))$ &
$(\forall W.\ {\sim}(q4(a::'a,b,W)$ & $mless\text{-}or\text{-}equal(a::'a,W)$ & $mless\text{-}or\text{-}equal(b::'a,W)$ &
$mless\text{-}or\text{-}equal(W::'a,a)))$ &
$(\forall W.\ {\sim}(q4(a::'a,b,W)$ & $mless\text{-}or\text{-}equal(a::'a,W)$ & $mless\text{-}or\text{-}equal(b::'a,W)$ &
$mless\text{-}or\text{-}equal(W::'a,b))) \dashrightarrow False$
$\langle proof \rangle$

abbreviation *SWV001-1-ax mless-THAN successor predecessor equal* $\equiv$
$(\forall X.\ equal(predecessor(successor(X)),X))$ &
$(\forall X.\ equal(successor(predecessor(X)),X))$ &
$(\forall X\ Y.\ equal(predecessor(X),predecessor(Y)) \dashrightarrow equal(X::'a,Y))$ &
$(\forall X\ Y.\ equal(successor(X),successor(Y)) \dashrightarrow equal(X::'a,Y))$ &
$(\forall X.\ mless\text{-}THAN(predecessor(X),X))$ &
$(\forall X.\ mless\text{-}THAN(X::'a,successor(X)))$ &
$(\forall X\ Y\ Z.\ mless\text{-}THAN(X::'a,Y)$ & $mless\text{-}THAN(Y::'a,Z) \dashrightarrow mless\text{-}THAN(X::'a,Z))$
&
$(\forall X\ Y.\ mless\text{-}THAN(X::'a,Y) \mid mless\text{-}THAN(Y::'a,X) \mid equal(X::'a,Y))$ &
$(\forall X.\ {\sim}mless\text{-}THAN(X::'a,X))$ &
$(\forall Y\ X.\ {\sim}(mless\text{-}THAN(X::'a,Y)$ & $mless\text{-}THAN(Y::'a,X)))$ &
$(\forall Y\ X\ Z.\ equal(X::'a,Y)$ & $mless\text{-}THAN(X::'a,Z) \dashrightarrow mless\text{-}THAN(Y::'a,Z))$
&
$(\forall Y\ Z\ X.\ equal(X::'a,Y)$ & $mless\text{-}THAN(Z::'a,X) \dashrightarrow mless\text{-}THAN(Z::'a,Y))$

abbreviation *SWV001-0-eq a successor predecessor equal* $\equiv$
$(\forall X\ Y.\ equal(X::'a,Y) \dashrightarrow equal(predecessor(X),predecessor(Y)))$ &
$(\forall X\ Y.\ equal(X::'a,Y) \dashrightarrow equal(successor(X),successor(Y)))$ &
$(\forall X\ Y.\ equal(X::'a,Y) \dashrightarrow equal(a(X),a(Y)))$

lemma *PRV003-1*:
EQU001-0-ax equal &
SWV001-1-ax mless-THAN successor predecessor equal &
SWV001-0-eq a successor predecessor equal &
$({\sim}mless\text{-}THAN(n::'a,j))$ &
$(mless\text{-}THAN(k::'a,j))$ &
$({\sim}mless\text{-}THAN(k::'a,i))$ &
$(mless\text{-}THAN(i::'a,n))$ &
$(mless\text{-}THAN(a(j),a(k)))$ &
$(\forall X.\ mless\text{-}THAN(X::'a,j)$ & $mless\text{-}THAN(a(X),a(k)) \dashrightarrow mless\text{-}THAN(X::'a,i))$
&
$(\forall X.\ mless\text{-}THAN(One::'a,i)$ & $mless\text{-}THAN(a(X),a(predecessor(i))) \dashrightarrow mless\text{-}THAN(X::'a,i)$
$\mid mless\text{-}THAN(n::'a,X))$ &
$(\forall X.\ {\sim}(mless\text{-}THAN(One::'a,X)$ & $mless\text{-}THAN(X::'a,i)$ & $mless\text{-}THAN(a(X),a(predecessor(X)))))$

&
  $(mless\text{-}THAN(j::'a,i))$ $-->$ *False*
  ⟨*proof*⟩

lemma *PRV005-1*:
  EQU001-0-ax equal &
  SWV001-1-ax mless-THAN successor predecessor equal &
  SWV001-0-eq a successor predecessor equal &
  $(\sim mless\text{-}THAN(n::'a,k))$ &
  $(\sim mless\text{-}THAN(k::'a,l))$ &
  $(\sim mless\text{-}THAN(k::'a,i))$ &
  $(mless\text{-}THAN(l::'a,n))$ &
  $(mless\text{-}THAN(One::'a,l))$ &
  $(mless\text{-}THAN(a(k),a(predecessor(l))))$ &
  $(\forall X.\ mless\text{-}THAN(X::'a,successor(n))$ & $mless\text{-}THAN(a(X),a(k))$ $-->$ $mless\text{-}THAN(X::'a,l))$
&
  $(\forall X.\ mless\text{-}THAN(One::'a,l)$ & $mless\text{-}THAN(a(X),a(predecessor(l)))$ $-->$ $mless\text{-}THAN(X::'a,l)$
| $mless\text{-}THAN(n::'a,X))$ &
  $(\forall X.\ \sim(mless\text{-}THAN(One::'a,X)$ & $mless\text{-}THAN(X::'a,l)$ & $mless\text{-}THAN(a(X),a(predecessor(X)))))$
$-->$ *False*
  ⟨*proof*⟩

lemma *PRV006-1*:
  EQU001-0-ax equal &
  SWV001-1-ax mless-THAN successor predecessor equal &
  SWV001-0-eq a successor predecessor equal &
  $(\sim mless\text{-}THAN(n::'a,m))$ &
  $(mless\text{-}THAN(i::'a,m))$ &
  $(mless\text{-}THAN(i::'a,n))$ &
  $(\sim mless\text{-}THAN(i::'a,One))$ &
  $(mless\text{-}THAN(a(i),a(m)))$ &
  $(\forall X.\ mless\text{-}THAN(X::'a,successor(n))$ & $mless\text{-}THAN(a(X),a(m))$ $-->$ $mless\text{-}THAN(X::'a,i))$
&
  $(\forall X.\ mless\text{-}THAN(One::'a,i)$ & $mless\text{-}THAN(a(X),a(predecessor(i)))$ $-->$ $mless\text{-}THAN(X::'a,i)$
| $mless\text{-}THAN(n::'a,X))$ &
  $(\forall X.\ \sim(mless\text{-}THAN(One::'a,X)$ & $mless\text{-}THAN(X::'a,i)$ & $mless\text{-}THAN(a(X),a(predecessor(X)))))$
$-->$ *False*
  ⟨*proof*⟩

lemma *PRV009-1*:
  $(\forall Y\ X.\ mless\text{-}or\text{-}equal(X::'a,Y) \mid mless(Y::'a,X))$ &
  $(mless(j::'a,i))$ &
  $(mless\text{-}or\text{-}equal(m::'a,p))$ &
  $(mless\text{-}or\text{-}equal(p::'a,q))$ &
  $(mless\text{-}or\text{-}equal(q::'a,n))$ &
  $(\forall X\ Y.\ mless\text{-}or\text{-}equal(m::'a,X)$ & $mless(X::'a,i)$ & $mless(j::'a,Y)$ & $mless\text{-}or\text{-}equal(Y::'a,n)$

$--> mless\text{-}or\text{-}equal(a(X),a(Y)))$ &
$(\forall X\ Y.\ mless\text{-}or\text{-}equal(m::'a,X)$ & $mless\text{-}or\text{-}equal(X::'a,Y)$ & $mless\text{-}or\text{-}equal(Y::'a,j)$
$--> mless\text{-}or\text{-}equal(a(X),a(Y)))$ &
$(\forall X\ Y.\ mless\text{-}or\text{-}equal(i::'a,X)$ & $mless\text{-}or\text{-}equal(X::'a,Y)$ & $mless\text{-}or\text{-}equal(Y::'a,n)$
$--> mless\text{-}or\text{-}equal(a(X),a(Y)))$ &
$(\sim mless\text{-}or\text{-}equal(a(p),a(q))) --> False$
$\langle proof \rangle$

lemma *PUZ012-1*:
$(\forall X.\ equal\text{-}fruits(X::'a,X))$ &
$(\forall X.\ equal\text{-}boxes(X::'a,X))$ &
$(\forall X\ Y.\ \sim(label(X::'a,Y)$ & $contains(X::'a,Y)))$ &
$(\forall X.\ contains(boxa::'a,X) \mid contains(boxb::'a,X) \mid contains(boxc::'a,X))$ &
$(\forall X.\ contains(X::'a,apples) \mid contains(X::'a,bananas) \mid contains(X::'a,oranges))$
&
$(\forall X\ Y\ Z.\ contains(X::'a,Y)$ & $contains(X::'a,Z) --> equal\text{-}fruits(Y::'a,Z))$ &
$(\forall Y\ X\ Z.\ contains(X::'a,Y)$ & $contains(Z::'a,Y) --> equal\text{-}boxes(X::'a,Z))$ &
$(\sim equal\text{-}boxes(boxa::'a,boxb))$ &
$(\sim equal\text{-}boxes(boxb::'a,boxc))$ &
$(\sim equal\text{-}boxes(boxa::'a,boxc))$ &
$(\sim equal\text{-}fruits(apples::'a,bananas))$ &
$(\sim equal\text{-}fruits(bananas::'a,oranges))$ &
$(\sim equal\text{-}fruits(apples::'a,oranges))$ &
$(label(boxa::'a,apples))$ &
$(label(boxb::'a,oranges))$ &
$(label(boxc::'a,bananas))$ &
$(contains(boxb::'a,apples))$ &
$(\sim(contains(boxa::'a,bananas)$ & $contains(boxc::'a,oranges))) --> False$
$\langle proof \rangle$

lemma *PUZ020-1*:
EQU001-0-ax equal &
$(\forall A\ B.\ equal(A::'a,B) --> equal(statement\text{-}by(A),statement\text{-}by(B)))$ &
$(\forall X.\ person(X) --> knight(X) \mid knave(X))$ &
$(\forall X.\ \sim(person(X)$ & $knight(X)$ & $knave(X)))$ &
$(\forall X\ Y.\ says(X::'a,Y)$ & $a\text{-}truth(Y) --> a\text{-}truth(Y))$ &
$(\forall X\ Y.\ \sim(says(X::'a,Y)$ & $equal(X::'a,Y)))$ &
$(\forall Y\ X.\ says(X::'a,Y) --> equal(Y::'a,statement\text{-}by(X)))$ &
$(\forall X\ Y.\ \sim(person(X)$ & $equal(X::'a,statement\text{-}by(Y))))$ &
$(\forall X.\ person(X)$ & $a\text{-}truth(statement\text{-}by(X)) --> knight(X))$ &
$(\forall X.\ person(X) --> a\text{-}truth(statement\text{-}by(X)) \mid knave(X))$ &
$(\forall X\ Y.\ equal(X::'a,Y)$ & $knight(X) --> knight(Y))$ &
$(\forall X\ Y.\ equal(X::'a,Y)$ & $knave(X) --> knave(Y))$ &
$(\forall X\ Y.\ equal(X::'a,Y)$ & $person(X) --> person(Y))$ &
$(\forall X\ Y\ Z.\ equal(X::'a,Y)$ & $says(X::'a,Z) --> says(Y::'a,Z))$ &
$(\forall X\ Z\ Y.\ equal(X::'a,Y)$ & $says(Z::'a,X) --> says(Z::'a,Y))$ &
$(\forall X\ Y.\ equal(X::'a,Y)$ & $a\text{-}truth(X) --> a\text{-}truth(Y))$ &

(∀ 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
⟨proof⟩

lemma *PUZ025-1*:
(∀ X. a-truth(truthteller(X)) | a-truth(liar(X))) &
(∀ X. ~(a-truth(truthteller(X)) & a-truth(liar(X)))) &
(∀ Truthteller Statement. a-truth(truthteller(Truthteller)) & a-truth(says(Truthteller::'a,Statement)) −−> a-truth(Statement)) &
(∀ Liar Statement. ~(a-truth(liar(Liar)) & a-truth(says(Liar::'a,Statement)) & a-truth(Statement))) &
(∀ Statement Truthteller. a-truth(Statement) & a-truth(says(Truthteller::'a,Statement)) −−> a-truth(truthteller(Truthteller))) &
(∀ Statement Liar. a-truth(says(Liar::'a,Statement)) −−> a-truth(Statement) | a-truth(liar(Liar))) &
(∀ Z X Y. people(X::'a,Y,Z) & a-truth(liar(X)) & a-truth(liar(Y)) −−> a-truth(equal-type(X::'a,Y))) &
(∀ Z X Y. people(X::'a,Y,Z) & a-truth(truthteller(X)) & a-truth(truthteller(Y)) −−> a-truth(equal-type(X::'a,Y))) &
(∀ X Y. a-truth(equal-type(X::'a,Y)) & a-truth(truthteller(X)) −−> a-truth(truthteller(Y))) &
(∀ X Y. a-truth(equal-type(X::'a,Y)) & a-truth(liar(X)) −−> a-truth(liar(Y))) &
(∀ X Y. a-truth(truthteller(X)) −−> a-truth(equal-type(X::'a,Y)) | a-truth(liar(Y))) &
(∀ X Y. a-truth(liar(X)) −−> a-truth(equal-type(X::'a,Y)) | a-truth(truthteller(Y))) &
(∀ Y X. a-truth(equal-type(X::'a,Y)) −−> a-truth(equal-type(Y::'a,X))) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(truthteller(X)) & a-truth(Y) −−> answer(yes)) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(truthteller(X)) −−> a-truth(Y) | answer(no)) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(liar(X)) & a-truth(Y) −−> answer(no)) &
(∀ X Y. ask-1-if-2(X::'a,Y) & a-truth(liar(X)) −−> a-truth(Y) | answer(yes)) &
(people(b::'a,c,a)) &
(people(a::'a,b,a)) &
(people(a::'a,c,b)) &

$(people(c::'a,b,a))$ &
$(a\text{-}truth(says(a::'a,equal\text{-}type(b::'a,c))))$ &
$(ask\text{-}1\text{-}if\text{-}2(c::'a,equal\text{-}type(a::'a,b)))$ &
$(\forall\, Answer.\ \sim answer(Answer))$ --> *False*
⟨*proof*⟩

lemma *PUZ029-1*:
$(\forall X.\ dances\text{-}on\text{-}tightropes(X) \mid eats\text{-}pennybuns(X) \mid old(X))$ &
$(\forall X.\ pig(X)$ & $liable\text{-}to\text{-}giddiness(X)$ --> $treated\text{-}with\text{-}respect(X))$ &
$(\forall X.\ wise(X)$ & $balloonist(X)$ --> $has\text{-}umbrella(X))$ &
$(\forall X.\ \sim(looks\text{-}ridiculous(X)$ & $eats\text{-}pennybuns(X)$ & $eats\text{-}lunch\text{-}in\text{-}public(X)))$ &
$(\forall X.\ balloonist(X)$ & $young(X)$ --> $liable\text{-}to\text{-}giddiness(X))$ &
$(\forall X.\ fat(X)$ & $looks\text{-}ridiculous(X)$ --> $dances\text{-}on\text{-}tightropes(X) \mid eats\text{-}lunch\text{-}in\text{-}public(X))$
&
$(\forall X.\ \sim(liable\text{-}to\text{-}giddiness(X)$ & $wise(X)$ & $dances\text{-}on\text{-}tightropes(X)))$ &
$(\forall X.\ pig(X)$ & $has\text{-}umbrella(X)$ --> $looks\text{-}ridiculous(X))$ &
$(\forall X.\ treated\text{-}with\text{-}respect(X)$ --> $dances\text{-}on\text{-}tightropes(X) \mid fat(X))$ &
$(\forall X.\ young(X) \mid old(X))$ &
$(\forall X.\ \sim(young(X)$ & $old(X)))$ &
$(wise(piggy))$ &
$(young(piggy))$ &
$(pig(piggy))$ &
$(balloonist(piggy))$ --> *False*
⟨*proof*⟩

abbreviation *RNG001-0-ax equal additive-inverse add multiply product additive-identity sum* $\equiv$
$(\forall X.\ sum(additive\text{-}identity::'a,X,X))$ &
$(\forall X.\ sum(X::'a,additive\text{-}identity,X))$ &
$(\forall X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))$ &
$(\forall X.\ sum(additive\text{-}inverse(X),X,additive\text{-}identity))$ &
$(\forall X.\ sum(X::'a,additive\text{-}inverse(X),additive\text{-}identity))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ sum(X::'a,Y,U)$ & $sum(Y::'a,Z,V)$ & $sum(U::'a,Z,W)$ -->
$sum(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ sum(X::'a,Y,U)$ & $sum(Y::'a,Z,V)$ & $sum(X::'a,V,W)$ -->
$sum(U::'a,Z,W))$ &
$(\forall Y\ X\ Z.\ sum(X::'a,Y,Z)$ --> $sum(Y::'a,X,Z))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(U::'a,Z,W)$
--> $product(X::'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X::'a,Y,U)$ & $product(Y::'a,Z,V)$ & $product(X::'a,V,W)$
--> $product(U::'a,Z,W))$ &
$(\forall Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X::'a,Y,V1)$ & $product(X::'a,Z,V2)$ & $sum(Y::'a,Z,V3)$
& $product(X::'a,V3,V4)$ --> $sum(V1::'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X::'a,Y,V1)$ & $product(X::'a,Z,V2)$ & $sum(Y::'a,Z,V3)$
& $sum(V1::'a,V2,V4)$ --> $product(X::'a,V3,V4))$ &
$(\forall Y\ Z\ V3\ X\ V1\ V2\ V4.\ product(Y::'a,X,V1)$ & $product(Z::'a,X,V2)$ & $sum(Y::'a,Z,V3)$

& product(V3::'a,X,V4) --> sum(V1::'a,V2,V4)) &
(∀ 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
⟨proof⟩

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)) &

$(\forall\, Y\, X\, V\, U\, Z\, W.\ product(X::'a,Y,U)\ \&\ product(Y::'a,Z,V)\ \&\ product(X::'a,V,W)$
$-->\ product(U::'a,Z,W))\ \&$
$(\forall\, Y\, Z\, X\, V3\, V1\, V2\, V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ product(X::'a,V3,V4)\ -->\ sum(V1::'a,V2,V4))\ \&$
$(\forall\, Y\, Z\, V1\, V2\, X\, V3\, V4.\ product(X::'a,Y,V1)\ \&\ product(X::'a,Z,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ sum(V1::'a,V2,V4)\ -->\ product(X::'a,V3,V4))\ \&$
$(\forall\, Y\, Z\, V3\, X\, V1\, V2\, V4.\ product(Y::'a,X,V1)\ \&\ product(Z::'a,X,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ product(V3::'a,X,V4)\ -->\ sum(V1::'a,V2,V4))\ \&$
$(\forall\, Y\, Z\, V1\, V2\, V3\, X\, V4.\ product(Y::'a,X,V1)\ \&\ product(Z::'a,X,V2)\ \&\ sum(Y::'a,Z,V3)$
$\&\ sum(V1::'a,V2,V4)\ -->\ product(V3::'a,X,V4))\ \&$
$(\forall\, X\ Y\ U\ V.\ sum(X::'a,Y,U)\ \&\ sum(X::'a,Y,V)\ -->\ equal(U::'a,V))\ \&$
$(\forall\, X\ Y\ U\ V.\ product(X::'a,Y,U)\ \&\ product(X::'a,Y,V)\ -->\ equal(U::'a,V))$
$\&$
$(\forall\, X\ Y.\ equal(X::'a,Y)\ -->\ equal(additive\text{-}inverse(X),additive\text{-}inverse(Y)))\ \&$
$(\forall\, X\ Y\ W.\ equal(X::'a,Y)\ -->\ equal(add(X::'a,W),add(Y::'a,W)))\ \&$
$(\forall\, X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ sum(X::'a,W,Z)\ -->\ sum(Y::'a,W,Z))\ \&$
$(\forall\, X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ sum(W::'a,X,Z)\ -->\ sum(W::'a,Y,Z))\ \&$
$(\forall\, X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ sum(W::'a,Z,X)\ -->\ sum(W::'a,Z,Y))\ \&$
$(\forall\, X\ Y\ W.\ equal(X::'a,Y)\ -->\ equal(multiply(X::'a,W),multiply(Y::'a,W)))$
$\&$
$(\forall\, X\ Y\ W\ Z.\ equal(X::'a,Y)\ \&\ product(X::'a,W,Z)\ -->\ product(Y::'a,W,Z))$
$\&$
$(\forall\, X\ W\ Y\ Z.\ equal(X::'a,Y)\ \&\ product(W::'a,X,Z)\ -->\ product(W::'a,Y,Z))$
$\&$
$(\forall\, X\ W\ Z\ Y.\ equal(X::'a,Y)\ \&\ product(W::'a,Z,X)\ -->\ product(W::'a,Z,Y))$

lemma *RNG001-5*:
EQU001-0-ax equal &
$(\forall\, X.\ sum(additive\text{-}identity::'a,X,X))\ \&$
$(\forall\, X.\ sum(X::'a,additive\text{-}identity,X))\ \&$
$(\forall\, X\ Y.\ product(X::'a,Y,multiply(X::'a,Y)))\ \&$
$(\forall\, X\ Y.\ sum(X::'a,Y,add(X::'a,Y)))\ \&$
$(\forall\, X.\ sum(additive\text{-}inverse(X),X,additive\text{-}identity))\ \&$
RNG-other-ax multiply add equal product additive-identity additive-inverse sum
&
$(\sim product(a::'a,additive\text{-}identity,additive\text{-}identity))\ -->\ False$
$\langle proof\rangle$

lemma *RNG011-5*:
EQU001-0-ax equal &
$(\forall\, A\ B\ C.\ equal(A::'a,B)\ -->\ equal(add(A::'a,C),add(B::'a,C)))\ \&$
$(\forall\, D\ F'\ E.\ equal(D::'a,E)\ -->\ equal(add(F'::'a,D),add(F'::'a,E)))\ \&$
$(\forall\, G\ H.\ equal(G::'a,H)\ -->\ equal(additive\text{-}inverse(G),additive\text{-}inverse(H)))\ \&$
$(\forall\, I'\ J\ K'.\ equal(I'::'a,J)\ -->\ equal(multiply(I'::'a,K'),multiply(J::'a,K')))\ \&$
$(\forall\, L\ N\ M.\ equal(L::'a,M)\ -->\ equal(multiply(N::'a,L),multiply(N::'a,M)))\ \&$
$(\forall\, A\ B\ C\ D.\ equal(A::'a,B)\ -->\ equal(associator(A::'a,C,D),associator(B::'a,C,D)))$

&
$(\forall\, E\ G\ F'\ H.\ equal(E{::}'a,F') \ {-}{-}{>}\ equal(associator(G{::}'a,E,H),associator(G{::}'a,F',H)))$
&
$(\forall\, I'\ K'\ L\ J.\ equal(I'{::}'a,J) \ {-}{-}{>}\ equal(associator(K'{::}'a,L,I'),associator(K'{::}'a,L,J)))$
&
$(\forall\, M\ N\ O'.\ equal(M{::}'a,N) \ {-}{-}{>}\ equal(commutator(M{::}'a,O'),commutator(N{::}'a,O')))$
&
$(\forall\, P\ R\ Q.\ equal(P{::}'a,Q) \ {-}{-}{>}\ equal(commutator(R{::}'a,P),commutator(R{::}'a,Q)))$
&
$(\forall\, Y\ X.\ equal(add(X{::}'a,Y),add(Y{::}'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(add(X{::}'a,Y),Z),add(X{::}'a,add(Y{::}'a,Z))))$ &
$(\forall\, X.\ equal(add(X{::}'a,additive\text{-}identity),X))$ &
$(\forall\, X.\ equal(add(additive\text{-}identity{::}'a,X),X))$ &
$(\forall\, X.\ equal(add(X{::}'a,additive\text{-}inverse(X)),additive\text{-}identity))$ &
$(\forall\, X.\ equal(add(additive\text{-}inverse(X),X),additive\text{-}identity))$ &
$(equal(additive\text{-}inverse(additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X\ Y.\ equal(add(X{::}'a,add(additive\text{-}inverse(X),Y)),Y))$ &
$(\forall\, X\ Y.\ equal(additive\text{-}inverse(add(X{::}'a,Y)),add(additive\text{-}inverse(X),additive\text{-}inverse(Y))))$
&
$(\forall\, X.\ equal(additive\text{-}inverse(additive\text{-}inverse(X)),X))$ &
$(\forall\, X.\ equal(multiply(X{::}'a,additive\text{-}identity),additive\text{-}identity))$ &
$(\forall\, X.\ equal(multiply(additive\text{-}identity{::}'a,X),additive\text{-}identity))$ &
$(\forall\, X\ Y.\ equal(multiply(additive\text{-}inverse(X),additive\text{-}inverse(Y)),multiply(X{::}'a,Y)))$
&
$(\forall\, X\ Y.\ equal(multiply(X{::}'a,additive\text{-}inverse(Y)),additive\text{-}inverse(multiply(X{::}'a,Y))))$
&
$(\forall\, X\ Y.\ equal(multiply(additive\text{-}inverse(X),Y),additive\text{-}inverse(multiply(X{::}'a,Y))))$
&
$(\forall\, Y\ X\ Z.\ equal(multiply(X{::}'a,add(Y{::}'a,Z)),add(multiply(X{::}'a,Y),multiply(X{::}'a,Z))))$
&
$(\forall\, X\ Y\ Z.\ equal(multiply(add(X{::}'a,Y),Z),add(multiply(X{::}'a,Z),multiply(Y{::}'a,Z))))$
&
$(\forall\, X\ Y.\ equal(multiply(multiply(X{::}'a,Y),Y),multiply(X{::}'a,multiply(Y{::}'a,Y))))$
&
$(\forall\, X\ Y\ Z.\ equal(associator(X{::}'a,Y,Z),add(multiply(multiply(X{::}'a,Y),Z),additive\text{-}inverse(multiply(X{::}'a,m$
&
$(\forall\, X\ Y.\ equal(commutator(X{::}'a,Y),add(multiply(Y{::}'a,X),additive\text{-}inverse(multiply(X{::}'a,Y)))))$
&
$(\forall\, X\ Y.\ equal(multiply(multiply(associator(X{::}'a,X,Y),X),associator(X{::}'a,X,Y)),additive\text{-}identity))$
&
$({\sim}\,equal(multiply(multiply(associator(a{::}'a,a,b),a),associator(a{::}'a,a,b)),additive\text{-}identity))$
$-{-}{>}$ *False*
$\langle proof\rangle$

lemma *RNG023-6*:
EQU001-0-ax equal &
$(\forall\, Y\ X.\ equal(add(X{::}'a,Y),add(Y{::}'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(X{::}'a,add(Y{::}'a,Z)),add(add(X{::}'a,Y),Z)))$ &

(∀ X. equal(add(additive-identity::'a,X),X)) &
(∀ X. equal(add(X::'a,additive-identity),X)) &
(∀ X. equal(multiply(additive-identity::'a,X),additive-identity)) &
(∀ X. equal(multiply(X::'a,additive-identity),additive-identity)) &
(∀ X. equal(add(additive-inverse(X),X),additive-identity)) &
(∀ X. equal(add(X::'a,additive-inverse(X)),additive-identity)) &
(∀ Y X Z. equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))
&
(∀ X Y Z. equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))
&
(∀ X. equal(additive-inverse(additive-inverse(X)),X)) &
(∀ X Y. equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))
&
(∀ X Y. equal(multiply(multiply(X::'a,X),Y),multiply(X::'a,multiply(X::'a,Y))))
&
(∀ X Y Z. equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive-inverse(multiply(X::'a,m
&
(∀ X Y. equal(commutator(X::'a,Y),add(multiply(Y::'a,X),additive-inverse(multiply(X::'a,Y)))))
&
(∀ D E F'. equal(D::'a,E) --> equal(add(D::'a,F'),add(E::'a,F'))) &
(∀ G I' H. equal(G::'a,H) --> equal(add(I'::'a,G),add(I'::'a,H))) &
(∀ J K'. equal(J::'a,K') --> equal(additive-inverse(J),additive-inverse(K'))) &
(∀ L M N O'. equal(L::'a,M) --> equal(associator(L::'a,N,O'),associator(M::'a,N,O')))
&
(∀ P R Q S'. equal(P::'a,Q) --> equal(associator(R::'a,P,S'),associator(R::'a,Q,S')))
&
(∀ T' V W U. equal(T'::'a,U) --> equal(associator(V::'a,W,T'),associator(V::'a,W,U)))
&
(∀ X Y Z. equal(X::'a,Y) --> equal(commutator(X::'a,Z),commutator(Y::'a,Z)))
&
(∀ A1 C1 B1. equal(A1::'a,B1) --> equal(commutator(C1::'a,A1),commutator(C1::'a,B1)))
&
(∀ D1 E1 F1. equal(D1::'a,E1) --> equal(multiply(D1::'a,F1),multiply(E1::'a,F1)))
&
(∀ G1 I1 H1. equal(G1::'a,H1) --> equal(multiply(I1::'a,G1),multiply(I1::'a,H1)))
&
(~equal(associator(x::'a,x,y),additive-identity)) --> False
⟨proof⟩

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)) &

$(\forall Y X Z.\ equal(multiply(X::'a,add(Y::'a,Z)),add(multiply(X::'a,Y),multiply(X::'a,Z))))$
&
$(\forall X Y Z.\ equal(multiply(add(X::'a,Y),Z),add(multiply(X::'a,Z),multiply(Y::'a,Z))))$
&
$(\forall X Y.\ equal(multiply(multiply(X::'a,Y),Y),multiply(X::'a,multiply(Y::'a,Y))))$
&
$(\forall X Y.\ equal(multiply(multiply(X::'a,X),Y),multiply(X::'a,multiply(X::'a,Y))))$
&
$(\forall X Y.\ equal(multiply(additive\text{-}inverse(X),Y),additive\text{-}inverse(multiply(X::'a,Y))))$
&
$(\forall X Y.\ equal(multiply(X::'a,additive\text{-}inverse(Y)),additive\text{-}inverse(multiply(X::'a,Y))))$
&
$(equal(additive\text{-}inverse(additive\text{-}identity),additive\text{-}identity))$ &
$(\forall Y X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall X Y Z.\ equal(add(X::'a,add(Y::'a,Z)),add(add(X::'a,Y),Z)))$ &
$(\forall Z X Y.\ equal(add(X::'a,Z),add(Y::'a,Z)) \dashrightarrow equal(X::'a,Y))$ &
$(\forall Z X Y.\ equal(add(Z::'a,X),add(Z::'a,Y)) \dashrightarrow equal(X::'a,Y))$ &
$(\forall D E F'.\ equal(D::'a,E) \dashrightarrow equal(add(D::'a,F'),add(E::'a,F')))$ &
$(\forall G I' H.\ equal(G::'a,H) \dashrightarrow equal(add(I'::'a,G),add(I'::'a,H)))$ &
$(\forall J K'.\ equal(J::'a,K') \dashrightarrow equal(additive\text{-}inverse(J),additive\text{-}inverse(K')))$ &
$(\forall D1 E1 F1.\ equal(D1::'a,E1) \dashrightarrow equal(multiply(D1::'a,F1),multiply(E1::'a,F1)))$
&
$(\forall G1 I1 H1.\ equal(G1::'a,H1) \dashrightarrow equal(multiply(I1::'a,G1),multiply(I1::'a,H1)))$
&
$(\forall X Y Z.\ equal(associator(X::'a,Y,Z),add(multiply(multiply(X::'a,Y),Z),additive\text{-}inverse(multiply(X::'a,m$
&
$(\forall L M N O'.\ equal(L::'a,M) \dashrightarrow equal(associator(L::'a,N,O'),associator(M::'a,N,O')))$
&
$(\forall P R Q S'.\ equal(P::'a,Q) \dashrightarrow equal(associator(R::'a,P,S'),associator(R::'a,Q,S')))$
&
$(\forall T' V W U.\ equal(T'::'a,U) \dashrightarrow equal(associator(V::'a,W,T'),associator(V::'a,W,U)))$
&
$(\forall X Y.\ \sim equal(multiply(multiply(Y::'a,X),Y),multiply(Y::'a,multiply(X::'a,Y))))$
&
$(\forall X Y Z.\ \sim equal(associator(Y::'a,X,Z),additive\text{-}inverse(associator(X::'a,Y,Z))))$
&
$(\forall X Y Z.\ \sim equal(associator(Z::'a,Y,X),additive\text{-}inverse(associator(X::'a,Y,Z))))$
&
$(\sim equal(multiply(multiply(cx::'a,multiply(cy::'a,cx)),cz),multiply(cx::'a,multiply(cy::'a,multiply(cx::'a,cz))))$
$\dashrightarrow False$
$\langle proof \rangle$

lemma *RNG038-2*:
$(\forall X.\ sum(X::'a,additive\text{-}identity,X))$ &
$(\forall X Y.\ product(X::'a,Y,multiply(X::'a,Y)))$ &
$(\forall X Y.\ sum(X::'a,Y,add(X::'a,Y)))$ &
RNG-other-ax multiply add equal product additive-identity additive-inverse sum
&

$(\forall X.\ product(additive\text{-}identity{::}'a,X,additive\text{-}identity))$ &
$(\forall X.\ product(X{::}'a,additive\text{-}identity,additive\text{-}identity))$ &
$(\forall X\ Y.\ equal(X{::}'a,additive\text{-}identity)$ --> $product(X{::}'a,h(X{::}'a,Y),Y))$ &
$(product(a{::}'a,b,additive\text{-}identity))$ &
$({\sim}\,equal(a{::}'a,additive\text{-}identity))$ &
$({\sim}\,equal(b{::}'a,additive\text{-}identity))$ --> *False*
⟨*proof*⟩

lemma *RNG040-2*:
EQU001-0-ax equal &
RNG001-0-eq product multiply sum add additive-inverse equal &
$(\forall X.\ sum(additive\text{-}identity{::}'a,X,X))$ &
$(\forall X.\ sum(X{::}'a,additive\text{-}identity,X))$ &
$(\forall X\ Y.\ product(X{::}'a,Y,multiply(X{::}'a,Y)))$ &
$(\forall X\ Y.\ sum(X{::}'a,Y,add(X{::}'a,Y)))$ &
$(\forall X.\ sum(additive\text{-}inverse(X),X,additive\text{-}identity))$ &
$(\forall X.\ sum(X{::}'a,additive\text{-}inverse(X),additive\text{-}identity))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ sum(X{::}'a,Y,U)$ & $sum(Y{::}'a,Z,V)$ & $sum(U{::}'a,Z,W)$ --> $sum(X{::}'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ sum(X{::}'a,Y,U)$ & $sum(Y{::}'a,Z,V)$ & $sum(X{::}'a,V,W)$ --> $sum(U{::}'a,Z,W))$ &
$(\forall Y\ X\ Z.\ sum(X{::}'a,Y,Z)$ --> $sum(Y{::}'a,X,Z))$ &
$(\forall Y\ U\ Z\ X\ V\ W.\ product(X{::}'a,Y,U)$ & $product(Y{::}'a,Z,V)$ & $product(U{::}'a,Z,W)$ --> $product(X{::}'a,V,W))$ &
$(\forall Y\ X\ V\ U\ Z\ W.\ product(X{::}'a,Y,U)$ & $product(Y{::}'a,Z,V)$ & $product(X{::}'a,V,W)$ --> $product(U{::}'a,Z,W))$ &
$(\forall Y\ Z\ X\ V3\ V1\ V2\ V4.\ product(X{::}'a,Y,V1)$ & $product(X{::}'a,Z,V2)$ & $sum(Y{::}'a,Z,V3)$ & $product(X{::}'a,V3,V4)$ --> $sum(V1{::}'a,V2,V4))$ &
$(\forall Y\ Z\ V1\ V2\ X\ V3\ V4.\ product(X{::}'a,Y,V1)$ & $product(X{::}'a,Z,V2)$ & $sum(Y{::}'a,Z,V3)$ & $sum(V1{::}'a,V2,V4)$ --> $product(X{::}'a,V3,V4))$ &
$(\forall X\ Y\ U\ V.\ sum(X{::}'a,Y,U)$ & $sum(X{::}'a,Y,V)$ --> $equal(U{::}'a,V))$ &
$(\forall X\ Y\ U\ V.\ product(X{::}'a,Y,U)$ & $product(X{::}'a,Y,V)$ --> $equal(U{::}'a,V))$ &
$(\forall A.\ product(A{::}'a,multiplicative\text{-}identity,A))$ &
$(\forall A.\ product(multiplicative\text{-}identity{::}'a,A,A))$ &
$(\forall A.\ product(A{::}'a,h(A),multiplicative\text{-}identity) \mid equal(A{::}'a,additive\text{-}identity))$ &
$(\forall A.\ product(h(A),A,multiplicative\text{-}identity) \mid equal(A{::}'a,additive\text{-}identity))$ &
$(\forall B\ A\ C.\ product(A{::}'a,B,C)$ --> $product(B{::}'a,A,C))$ &
$(\forall A\ B.\ equal(A{::}'a,B)$ --> $equal(h(A),h(B)))$ &
$(sum(b{::}'a,c,d))$ &
$(product(d{::}'a,a,additive\text{-}identity))$ &
$(product(b{::}'a,a,l))$ &
$(product(c{::}'a,a,n))$ &
$({\sim}\,sum(l{::}'a,n,additive\text{-}identity))$ --> *False*
⟨*proof*⟩

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 &
$(\forall\, A\ B.\ equal(A::'a,B) \ -{-}{>}\ equal(h(A),h(B)))$ &
$(\forall\, A.\ product(additive\text{-}identity::'a,A,additive\text{-}identity))$ &
$(\forall\, A.\ product(A::'a,additive\text{-}identity,additive\text{-}identity))$ &
$(\forall\, A.\ product(A::'a,multiplicative\text{-}identity,A))$ &
$(\forall\, A.\ product(multiplicative\text{-}identity::'a,A,A))$ &
$(\forall\, A.\ product(A::'a,h(A),multiplicative\text{-}identity) \mid equal(A::'a,additive\text{-}identity))$
&
$(\forall\, A.\ product(h(A),A,multiplicative\text{-}identity) \mid equal(A::'a,additive\text{-}identity))$ &
$(product(a::'a,b,additive\text{-}identity))$ &
$(\sim equal(a::'a,additive\text{-}identity))$ &
$(\sim equal(b::'a,additive\text{-}identity)) \ -{-}{>}\ False$
⟨*proof*⟩

lemma *ROB010-1*:
EQU001-0-ax equal &
$(\forall\, Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z))))$ &
$(\forall\, Y\ X.\ equal(negate(add(negate(add(X::'a,Y)),negate(add(X::'a,negate(Y))))),X))$
&
$(\forall\, A\ B\ C.\ equal(A::'a,B) \ -{-}{>}\ equal(add(A::'a,C),add(B::'a,C)))$ &
$(\forall\, D\ F'\ E.\ equal(D::'a,E) \ -{-}{>}\ equal(add(F'::'a,D),add(F'::'a,E)))$ &
$(\forall\, G\ H.\ equal(G::'a,H) \ -{-}{>}\ equal(negate(G),negate(H)))$ &
$(equal(negate(add(a::'a,negate(b))),c))$ &
$(\sim equal(negate(add(c::'a,negate(add(b::'a,a)))),a)) \ -{-}{>}\ False$
⟨*proof*⟩

lemma *ROB013-1*:
EQU001-0-ax equal &
$(\forall\, Y\ X.\ equal(add(X::'a,Y),add(Y::'a,X)))$ &
$(\forall\, X\ Y\ Z.\ equal(add(add(X::'a,Y),Z),add(X::'a,add(Y::'a,Z))))$ &
$(\forall\, Y\ X.\ equal(negate(add(negate(add(X::'a,Y)),negate(add(X::'a,negate(Y))))),X))$
&
$(\forall\, A\ B\ C.\ equal(A::'a,B) \ -{-}{>}\ equal(add(A::'a,C),add(B::'a,C)))$ &
$(\forall\, D\ F'\ E.\ equal(D::'a,E) \ -{-}{>}\ equal(add(F'::'a,D),add(F'::'a,E)))$ &
$(\forall\, G\ H.\ equal(G::'a,H) \ -{-}{>}\ equal(negate(G),negate(H)))$ &
$(equal(negate(add(a::'a,b)),c))$ &
$(\sim equal(negate(add(c::'a,negate(add(negate(b),a)))),a)) \ -{-}{>}\ False$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)) &

$(\forall\, Subset\ Superset.\ equal\text{-}sets(Subset{::}'a,Superset) \ {-}{-}{>}\ subset(Subset{::}'a,Superset))$
&
$(\forall\, Subset\ Superset.\ equal\text{-}sets(Superset{::}'a,Subset) \ {-}{-}{>}\ subset(Subset{::}'a,Superset))$
&
$(\forall\, Set2\ Set1.\ subset(Set1{::}'a,Set2)\ \&\ subset(Set2{::}'a,Set1) \ {-}{-}{>}\ equal\text{-}sets(Set2{::}'a,Set1))$
&
$(\forall\, Set2\ Intersection\ Element\ Set1.\ intersection(Set1{::}'a,Set2,Intersection)\ \&\ member(Element{::}'a,Intersection) \ {-}{-}{>}\ member(Element{::}'a,Set1))$ &
$(\forall\, Set1\ Intersection\ Element\ Set2.\ intersection(Set1{::}'a,Set2,Intersection)\ \&\ member(Element{::}'a,Intersection) \ {-}{-}{>}\ member(Element{::}'a,Set2))$ &
$(\forall\, Set2\ Set1\ Element\ Intersection.\ intersection(Set1{::}'a,Set2,Intersection)\ \&\ member(Element{::}'a,Set2)\ \&\ member(Element{::}'a,Set1) \ {-}{-}{>}\ member(Element{::}'a,Intersection))$
&
$(\forall\, Set2\ Intersection\ Set1.\ member(h(Set1{::}'a,Set2,Intersection),Intersection)\ |\ intersection(Set1{::}'a,Set2,Intersection)\ |\ member(h(Set1{::}'a,Set2,Intersection),Set1))$
&
$(\forall\, Set1\ Intersection\ Set2.\ member(h(Set1{::}'a,Set2,Intersection),Intersection)\ |\ intersection(Set1{::}'a,Set2,Intersection)\ |\ member(h(Set1{::}'a,Set2,Intersection),Set2))$
&
$(\forall\, Set1\ Set2\ Intersection.\ member(h(Set1{::}'a,Set2,Intersection),Intersection)\ \&\ member(h(Set1{::}'a,Set2,Intersection),Set2)\ \&\ member(h(Set1{::}'a,Set2,Intersection),Set1) \ {-}{-}{>}\ intersection(Set1{::}'a,Set2,Intersection))$ &
$(intersection(a{::}'a,b,aIb))$ &
$(intersection(b{::}'a,c,bIc))$ &
$(intersection(a{::}'a,bIc,aIbIc))$ &
$(\sim intersection(aIb{::}'a,c,aIbIc)) \ {-}{-}{>}\ False$
$\langle proof \rangle$

lemma *SET009-1*:
$(\forall\, Subset\ Element\ Superset.\ member(Element{::}'a,Subset)\ \&\ ssubset(Subset{::}'a,Superset) \ {-}{-}{>}\ member(Element{::}'a,Superset))$ &
$(\forall\, Superset\ Subset.\ ssubset(Subset{::}'a,Superset)\ |\ member(member\text{-}of\text{-}1\text{-}not\text{-}of\text{-}2(Subset{::}'a,Superset),Subset))$
&
$(\forall\, Subset\ Superset.\ member(member\text{-}of\text{-}1\text{-}not\text{-}of\text{-}2(Subset{::}'a,Superset),Superset) \ {-}{-}{>}\ ssubset(Subset{::}'a,Superset))$ &
$(\forall\, Subset\ Superset.\ equal\text{-}sets(Subset{::}'a,Superset) \ {-}{-}{>}\ ssubset(Subset{::}'a,Superset))$
&
$(\forall\, Subset\ Superset.\ equal\text{-}sets(Superset{::}'a,Subset) \ {-}{-}{>}\ ssubset(Subset{::}'a,Superset))$
&
$(\forall\, Set2\ Set1.\ ssubset(Set1{::}'a,Set2)\ \&\ ssubset(Set2{::}'a,Set1) \ {-}{-}{>}\ equal\text{-}sets(Set2{::}'a,Set1))$
&
$(\forall\, Set2\ Difference\ Element\ Set1.\ difference(Set1{::}'a,Set2,Difference)\ \&\ member(Element{::}'a,Difference) \ {-}{-}{>}\ member(Element{::}'a,Set1))$ &
$(\forall\, Element\ A\text{-}set\ Set1\ Set2.\ \sim(member(Element{::}'a,Set1)\ \&\ member(Element{::}'a,Set2)\ \&\ difference(A\text{-}set{::}'a,Set1,Set2)))$ &
$(\forall\, Set1\ Difference\ Element\ Set2.\ member(Element{::}'a,Set1)\ \&\ difference(Set1{::}'a,Set2,Difference) \ {-}{-}{>}\ member(Element{::}'a,Difference)\ |\ member(Element{::}'a,Set2))$ &

($\forall$ *Set1 Set2 Difference. difference*(*Set1*::'*a*,*Set2*,*Difference*) | *member*(*k*(*Set1*::'*a*,*Set2*,*Difference*),*Set1*) | *member*(*k*(*Set1*::'*a*,*Set2*,*Difference*),*Difference*)) &
($\forall$ *Set1 Set2 Difference. member*(*k*(*Set1*::'*a*,*Set2*,*Difference*),*Set2*) --> *member*(*k*(*Set1*::'*a*,*Set2*,*Difference*),*Difference*) | *difference*(*Set1*::'*a*,*Set2*,*Difference*)) &
($\forall$ *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*
⟨*proof*⟩

lemma *SET025-4*:
EQU001-0-ax equal &
($\forall$ *Y X. member*(*X*::'*a*,*Y*) --> *little-set*(*X*)) &
($\forall$ *X Y. little-set*(*f1*(*X*::'*a*,*Y*)) | *equal*(*X*::'*a*,*Y*)) &
($\forall$ *X Y. member*(*f1*(*X*::'*a*,*Y*),*X*) | *member*(*f1*(*X*::'*a*,*Y*),*Y*) | *equal*(*X*::'*a*,*Y*)) &
($\forall$ *X Y. member*(*f1*(*X*::'*a*,*Y*),*X*) & *member*(*f1*(*X*::'*a*,*Y*),*Y*) --> *equal*(*X*::'*a*,*Y*)) &
($\forall$ *X U Y. member*(*U*::'*a*,*non-ordered-pair*(*X*::'*a*,*Y*)) --> *equal*(*U*::'*a*,*X*) | *equal*(*U*::'*a*,*Y*)) &
($\forall$ *Y U X. little-set*(*U*) & *equal*(*U*::'*a*,*X*) --> *member*(*U*::'*a*,*non-ordered-pair*(*X*::'*a*,*Y*))) &
($\forall$ *X U Y. little-set*(*U*) & *equal*(*U*::'*a*,*Y*) --> *member*(*U*::'*a*,*non-ordered-pair*(*X*::'*a*,*Y*))) &
($\forall$ *X Y. little-set*(*non-ordered-pair*(*X*::'*a*,*Y*))) &
($\forall$ *X. equal*(*singleton-set*(*X*),*non-ordered-pair*(*X*::'*a*,*X*))) &
($\forall$ *X Y. equal*(*ordered-pair*(*X*::'*a*,*Y*),*non-ordered-pair*(*singleton-set*(*X*),*non-ordered-pair*(*X*::'*a*,*Y*)))) &
($\forall$ *X. ordered-pair-predicate*(*X*) --> *little-set*(*f2*(*X*))) &
($\forall$ *X. ordered-pair-predicate*(*X*) --> *little-set*(*f3*(*X*))) &
($\forall$ *X. ordered-pair-predicate*(*X*) --> *equal*(*X*::'*a*,*ordered-pair*(*f2*(*X*),*f3*(*X*)))) &
($\forall$ *X Y Z. little-set*(*Y*) & *little-set*(*Z*) & *equal*(*X*::'*a*,*ordered-pair*(*Y*::'*a*,*Z*)) --> *ordered-pair-predicate*(*X*)) &
($\forall$ *Z X. member*(*Z*::'*a*,*first*(*X*)) --> *little-set*(*f4*(*Z*::'*a*,*X*))) &
($\forall$ *Z X. member*(*Z*::'*a*,*first*(*X*)) --> *little-set*(*f5*(*Z*::'*a*,*X*))) &
($\forall$ *Z X. member*(*Z*::'*a*,*first*(*X*)) --> *equal*(*X*::'*a*,*ordered-pair*(*f4*(*Z*::'*a*,*X*),*f5*(*Z*::'*a*,*X*)))) &
($\forall$ *Z X. member*(*Z*::'*a*,*first*(*X*)) --> *member*(*Z*::'*a*,*f4*(*Z*::'*a*,*X*))) &
($\forall$ *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*))) &
($\forall$ *Z X. member*(*Z*::'*a*,*second*(*X*)) --> *little-set*(*f6*(*Z*::'*a*,*X*))) &
($\forall$ *Z X. member*(*Z*::'*a*,*second*(*X*)) --> *little-set*(*f7*(*Z*::'*a*,*X*))) &
($\forall$ *Z X. member*(*Z*::'*a*,*second*(*X*)) --> *equal*(*X*::'*a*,*ordered-pair*(*f6*(*Z*::'*a*,*X*),*f7*(*Z*::'*a*,*X*)))) &
($\forall$ *Z X. member*(*Z*::'*a*,*second*(*X*)) --> *member*(*Z*::'*a*,*f7*(*Z*::'*a*,*X*))) &
($\forall$ *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)) &
(∀ X Z Y. *member*(Z::'a,X) & *member*(Z::'a, Y) --> *member*(Z::'a,*intersection*(X::'a, Y)))
&
(∀ Z X. ~(*member*(Z::'a,*complement*(X)) & *member*(Z::'a,X))) &
(∀ Z X. *little-set*(Z) --> *member*(Z::'a,*complement*(X)) | *member*(Z::'a,X)) &
(∀ X Y. *equal*(*union*(X::'a, Y),*complement*(*intersection*(*complement*(X),*complement*(Y)))))
&
(∀ Z X. *member*(Z::'a,*domain-of*(X)) --> *ordered-pair-predicate*(*f8*(Z::'a,X)))
&
(∀ Z X. *member*(Z::'a,*domain-of*(X)) --> *member*(*f8*(Z::'a,X),X)) &
(∀ Z X. *member*(Z::'a,*domain-of*(X)) --> *equal*(Z::'a,*first*(*f8*(Z::'a,X)))) &
(∀ X Z Xp. *little-set*(Z) & *ordered-pair-predicate*(Xp) & *member*(Xp::'a,X) &
equal(Z::'a,*first*(Xp)) --> *member*(Z::'a,*domain-of*(X))) &
(∀ X Y Z. *member*(Z::'a,*cross-product*(X::'a, Y)) --> *ordered-pair-predicate*(Z))
&
(∀ Y Z X. *member*(Z::'a,*cross-product*(X::'a, Y)) --> *member*(*first*(Z),X)) &
(∀ X Z Y. *member*(Z::'a,*cross-product*(X::'a, Y)) --> *member*(*second*(Z), Y))
&
(∀ X Z Y. *little-set*(Z) & *ordered-pair-predicate*(Z) & *member*(*first*(Z),X) &
member(*second*(Z), Y) --> *member*(Z::'a,*cross-product*(X::'a, Y))) &
(∀ X Z. *member*(Z::'a,*inv1* X) --> *ordered-pair-predicate*(Z)) &
(∀ Z X. *member*(Z::'a,*inv1* X) --> *member*(*ordered-pair*(*second*(Z),*first*(Z)),X))
&
(∀ Z X. *little-set*(Z) & *ordered-pair-predicate*(Z) & *member*(*ordered-pair*(*second*(Z),*first*(Z)),X)
--> *member*(Z::'a,*inv1* X)) &
(∀ Z X. *member*(Z::'a,*rot-right*(X)) --> *little-set*(*f9*(Z::'a,X))) &
(∀ Z X. *member*(Z::'a,*rot-right*(X)) --> *little-set*(*f10*(Z::'a,X))) &
(∀ Z X. *member*(Z::'a,*rot-right*(X)) --> *little-set*(*f11*(Z::'a,X))) &
(∀ Z X. *member*(Z::'a,*rot-right*(X)) --> *equal*(Z::'a,*ordered-pair*(*f9*(Z::'a,X),*ordered-pair*(*f10*(Z::'a,X),*f11*
&
(∀ Z X. *member*(Z::'a,*rot-right*(X)) --> *member*(*ordered-pair*(*f10*(Z::'a,X),*ordered-pair*(*f11*(Z::'a,X),*f9*(Z
&
(∀ Z V W U X. *little-set*(Z) & *little-set*(U) & *little-set*(V) & *little-set*(W) &
equal(Z::'a,*ordered-pair*(U::'a,*ordered-pair*(V::'a, W))) & *member*(*ordered-pair*(V::'a,*ordered-pair*(W::'a, U))
--> *member*(Z::'a,*rot-right*(X))) &
(∀ Z X. *member*(Z::'a,*flip-range-of*(X)) --> *little-set*(*f12*(Z::'a,X))) &
(∀ Z X. *member*(Z::'a,*flip-range-of*(X)) --> *little-set*(*f13*(Z::'a,X))) &
(∀ Z X. *member*(Z::'a,*flip-range-of*(X)) --> *little-set*(*f14*(Z::'a,X))) &
(∀ Z X. *member*(Z::'a,*flip-range-of*(X)) --> *equal*(Z::'a,*ordered-pair*(*f12*(Z::'a,X),*ordered-pair*(*f13*(Z::'a,X
&
(∀ Z X. *member*(Z::'a,*flip-range-of*(X)) --> *member*(*ordered-pair*(*f12*(Z::'a,X),*ordered-pair*(*f14*(Z::'a,X),
&
(∀ Z U W V X. *little-set*(Z) & *little-set*(U) & *little-set*(V) & *little-set*(W) &

equal(Z::'a,ordered-pair(U::'a,ordered-pair(V::'a,W))) & member(ordered-pair(U::'a,ordered-pair(W::'a,V)),X)
--> member(Z::'a,flip-range-of(X))) &
(∀ X. equal(successor(X),union(X::'a,singleton-set(X)))) &
(∀ Z. ~member(Z::'a,empty-set)) &
(∀ Z. little-set(Z) --> member(Z::'a,universal-set)) &
(little-set(infinity)) &
(member(empty-set::'a,infinity)) &
(∀ X. member(X::'a,infinity) --> member(successor(X),infinity)) &
(∀ Z X. member(Z::'a,sigma(X)) --> member(f16(Z::'a,X),X)) &
(∀ Z X. member(Z::'a,sigma(X)) --> member(Z::'a,f16(Z::'a,X))) &
(∀ X Z Y. member(Y::'a,X) & member(Z::'a,Y) --> member(Z::'a,sigma(X)))
&
(∀ U. little-set(U) --> little-set(sigma(U))) &
(∀ X U Y. ssubset(X::'a,Y) & member(U::'a,X) --> member(U::'a,Y)) &
(∀ Y X. ssubset(X::'a,Y) | member(f17(X::'a,Y),X)) &
(∀ X Y. member(f17(X::'a,Y),Y) --> ssubset(X::'a,Y)) &
(∀ X Y. proper-subset(X::'a,Y) --> ssubset(X::'a,Y)) &
(∀ X Y. ~(proper-subset(X::'a,Y) & equal(X::'a,Y))) &
(∀ X Y. ssubset(X::'a,Y) --> proper-subset(X::'a,Y) | equal(X::'a,Y)) &
(∀ Z X. member(Z::'a,powerset(X)) --> ssubset(Z::'a,X)) &
(∀ Z X. little-set(Z) & ssubset(Z::'a,X) --> member(Z::'a,powerset(X))) &
(∀ U. little-set(U) --> little-set(powerset(U))) &
(∀ Z X. relation(Z) & member(X::'a,Z) --> ordered-pair-predicate(X)) &
(∀ Z. relation(Z) | member(f18(Z),Z)) &
(∀ Z. ordered-pair-predicate(f18(Z)) --> relation(Z)) &
(∀ U X V W. single-valued-set(X) & little-set(U) & little-set(V) & little-set(W)
& member(ordered-pair(U::'a,V),X) & member(ordered-pair(U::'a,W),X) -->
equal(V::'a,W)) &
(∀ X. single-valued-set(X) | little-set(f19(X))) &
(∀ X. single-valued-set(X) | little-set(f20(X))) &
(∀ X. single-valued-set(X) | little-set(f21(X))) &
(∀ X. single-valued-set(X) | member(ordered-pair(f19(X),f20(X)),X)) &
(∀ X. single-valued-set(X) | member(ordered-pair(f19(X),f21(X)),X)) &
(∀ X. equal(f20(X),f21(X)) --> single-valued-set(X)) &
(∀ Xf. function(Xf) --> relation(Xf)) &
(∀ Xf. function(Xf) --> single-valued-set(Xf)) &
(∀ Xf. relation(Xf) & single-valued-set(Xf) --> function(Xf)) &
(∀ Z X Xf. member(Z::'a,image'(X::'a,Xf)) --> ordered-pair-predicate(f22(Z::'a,X,Xf)))
&
(∀ Z X Xf. member(Z::'a,image'(X::'a,Xf)) --> member(f22(Z::'a,X,Xf),Xf))
&
(∀ Z Xf X. member(Z::'a,image'(X::'a,Xf)) --> member(first(f22(Z::'a,X,Xf)),X))
&
(∀ X Xf Z. member(Z::'a,image'(X::'a,Xf)) --> equal(second(f22(Z::'a,X,Xf)),Z))
&
(∀ Xf X Y Z. little-set(Z) & ordered-pair-predicate(Y) & member(Y::'a,Xf) &
member(first(Y),X) & equal(second(Y),Z) --> member(Z::'a,image'(X::'a,Xf)))
&
(∀ X Xf. little-set(X) & function(Xf) --> little-set(image'(X::'a,Xf))) &

$(\forall X\ U\ Y.\ \sim(disjoint(X::'a,Y)\ \&\ member(U::'a,X)\ \&\ member(U::'a,Y)))\ \&$
$(\forall Y\ X.\ disjoint(X::'a,Y)\ |\ member(f23(X::'a,Y),X))\ \&$
$(\forall X\ Y.\ disjoint(X::'a,Y)\ |\ member(f23(X::'a,Y),Y))\ \&$
$(\forall X.\ equal(X::'a,empty\text{-}set)\ |\ member(f24(X),X))\ \&$
$(\forall X.\ equal(X::'a,empty\text{-}set)\ |\ disjoint(f24(X),X))\ \&$
$(function(f25))\ \&$
$(\forall X.\ little\text{-}set(X)\ -->\ equal(X::'a,empty\text{-}set)\ |\ member(f26(X),X))\ \&$
$(\forall X.\ little\text{-}set(X)\ -->\ equal(X::'a,empty\text{-}set)\ |\ member(ordered\text{-}pair(X::'a,f26(X)),f25))$
$\&$
$(\forall Z\ X.\ member(Z::'a,range\text{-}of(X))\ -->\ ordered\text{-}pair\text{-}predicate(f27(Z::'a,X)))$
$\&$
$(\forall Z\ X.\ member(Z::'a,range\text{-}of(X))\ -->\ member(f27(Z::'a,X),X))\ \&$
$(\forall Z\ X.\ member(Z::'a,range\text{-}of(X))\ -->\ equal(Z::'a,second(f27(Z::'a,X))))\ \&$
$(\forall X\ Z\ Xp.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Xp)\ \&\ member(Xp::'a,X)\ \&$
$equal(Z::'a,second(Xp))\ -->\ member(Z::'a,range\text{-}of(X)))\ \&$
$(\forall Z.\ member(Z::'a,identity\text{-}relation)\ -->\ ordered\text{-}pair\text{-}predicate(Z))\ \&$
$(\forall Z.\ member(Z::'a,identity\text{-}relation)\ -->\ equal(first(Z),second(Z)))\ \&$
$(\forall Z.\ little\text{-}set(Z)\ \&\ ordered\text{-}pair\text{-}predicate(Z)\ \&\ equal(first(Z),second(Z))\ -->$
$member(Z::'a,identity\text{-}relation))\ \&$
$(\forall X\ Y.\ equal(restrct(X::'a,Y),intersection(X::'a,cross\text{-}product(Y::'a,universal\text{-}set))))$
$\&$
$(\forall Xf.\ one\text{-}to\text{-}one\text{-}function(Xf)\ -->\ function(Xf))\ \&$
$(\forall Xf.\ one\text{-}to\text{-}one\text{-}function(Xf)\ -->\ function(inv1\ Xf))\ \&$
$(\forall Xf.\ function(Xf)\ \&\ function(inv1\ Xf)\ -->\ one\text{-}to\text{-}one\text{-}function(Xf))\ \&$
$(\forall Z\ Xf\ Y.\ member(Z::'a,apply(Xf::'a,Y))\ -->\ ordered\text{-}pair\text{-}predicate(f28(Z::'a,Xf,Y)))$
$\&$
$(\forall Z\ Y\ Xf.\ member(Z::'a,apply(Xf::'a,Y))\ -->\ member(f28(Z::'a,Xf,Y),Xf))$
$\&$
$(\forall Z\ Xf\ Y.\ member(Z::'a,apply(Xf::'a,Y))\ -->\ equal(first(f28(Z::'a,Xf,Y)),Y))$
$\&$
$(\forall Z\ Xf\ Y.\ member(Z::'a,apply(Xf::'a,Y))\ -->\ member(Z::'a,second(f28(Z::'a,Xf,Y))))$
$\&$
$(\forall Xf\ Y\ Z\ W.\ ordered\text{-}pair\text{-}predicate(W)\ \&\ member(W::'a,Xf)\ \&\ equal(first(W),Y)$
$\&\ member(Z::'a,second(W))\ -->\ member(Z::'a,apply(Xf::'a,Y)))\ \&$
$(\forall Xf\ X\ Y.\ equal(apply\text{-}to\text{-}two\text{-}arguments(Xf::'a,X,Y),apply(Xf::'a,ordered\text{-}pair(X::'a,Y))))$
$\&$
$(\forall X\ Y\ Xf.\ maps(Xf::'a,X,Y)\ -->\ function(Xf))\ \&$
$(\forall Y\ Xf\ X.\ maps(Xf::'a,X,Y)\ -->\ equal(domain\text{-}of(Xf),X))\ \&$
$(\forall X\ Xf\ Y.\ maps(Xf::'a,X,Y)\ -->\ ssubset(range\text{-}of(Xf),Y))\ \&$
$(\forall X\ Xf\ Y.\ function(Xf)\ \&\ equal(domain\text{-}of(Xf),X)\ \&\ ssubset(range\text{-}of(Xf),Y)$
$-->\ maps(Xf::'a,X,Y))\ \&$
$(\forall Xf\ Xs.\ closed(Xs::'a,Xf)\ -->\ little\text{-}set(Xs))\ \&$
$(\forall Xs\ Xf.\ closed(Xs::'a,Xf)\ -->\ little\text{-}set(Xf))\ \&$
$(\forall Xf\ Xs.\ closed(Xs::'a,Xf)\ -->\ maps(Xf::'a,cross\text{-}product(Xs::'a,Xs),Xs))\ \&$
$(\forall Xf\ Xs.\ little\text{-}set(Xs)\ \&\ little\text{-}set(Xf)\ \&\ maps(Xf::'a,cross\text{-}product(Xs::'a,Xs),Xs)$
$-->\ closed(Xs::'a,Xf))\ \&$
$(\forall Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg))\ -->\ little\text{-}set(f29(Z::'a,Xf,Xg)))$
$\&$
$(\forall Z\ Xf\ Xg.\ member(Z::'a,composition(Xf::'a,Xg))\ -->\ little\text{-}set(f30(Z::'a,Xf,Xg)))$

&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> little-set(f31(Z::'a,Xf,Xg)))
&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> equal(Z::'a,ordered-pair(f29(Z::'a,Xf,Xg),f30(Z::'a
&
(∀ Z Xg Xf. member(Z::'a,composition(Xf::'a,Xg)) --> member(ordered-pair(f29(Z::'a,Xf,Xg),f31(Z::'a,X
&
(∀ Z Xf Xg. member(Z::'a,composition(Xf::'a,Xg)) --> member(ordered-pair(f31(Z::'a,Xf,Xg),f30(Z::'a,X
&
(∀ Z X Xf W Y Xg. little-set(Z) & little-set(X) & little-set(Y) & little-set(W) & equal(Z::'a,ordered-pair(X::'a,Y)) & member(ordered-pair(X::'a,W),Xf) & member(ordered-pair(W::'a,Y),Xg) --> member(Z::'a,composition(Xf::'a,Xg))) &
(∀ Xh Xs2 Xf2 Xs1 Xf1. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) --> closed(Xs1::'a,Xf1))
&
(∀ Xh Xs1 Xf1 Xs2 Xf2. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) --> closed(Xs2::'a,Xf2))
&
(∀ Xf1 Xf2 Xh Xs1 Xs2. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) --> maps(Xh::'a,Xs1,Xs2))
&
(∀ Xs2 Xs1 Xf1 Xf2 X Xh Y. homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) & member(X::'a,Xs1) & member(Y::'a,Xs1) --> equal(apply(Xh::'a,apply-to-two-arguments(Xf1::'a,X,Y)),apply-t
&
(∀ Xh Xf1 Xs2 Xf2 Xs1. closed(Xs1::'a,Xf1) & closed(Xs2::'a,Xf2) & maps(Xh::'a,Xs1,Xs2) --> homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) | member(f32(Xh::'a,Xs1,Xf1,Xs2,Xf2),Xs1))
&
(∀ Xh Xf1 Xs2 Xf2 Xs1. closed(Xs1::'a,Xf1) & closed(Xs2::'a,Xf2) & maps(Xh::'a,Xs1,Xs2) --> homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2) | member(f33(Xh::'a,Xs1,Xf1,Xs2,Xf2),Xs1))
&
(∀ Xh Xs1 Xf1 Xs2 Xf2. closed(Xs1::'a,Xf1) & closed(Xs2::'a,Xf2) & maps(Xh::'a,Xs1,Xs2) & equal(apply(Xh::'a,apply-to-two-arguments(Xf1::'a,f32(Xh::'a,Xs1,Xf1,Xs2,Xf2),f33(Xh::'a,Xs1,Xf1,Xs2,
--> homomorphism(Xh::'a,Xs1,Xf1,Xs2,Xf2)) &
(∀ A B C. equal(A::'a,B) --> equal(f1(A::'a,C),f1(B::'a,C))) &
(∀ D F' E. equal(D::'a,E) --> equal(f1(F'::'a,D),f1(F'::'a,E))) &
(∀ A2 B2. equal(A2::'a,B2) --> equal(f2(A2),f2(B2))) &
(∀ G4 H4. equal(G4::'a,H4) --> equal(f3(G4),f3(H4))) &
(∀ O7 P7 Q7. equal(O7::'a,P7) --> equal(f4(O7::'a,Q7),f4(P7::'a,Q7))) &
(∀ R7 T7 S7. equal(R7::'a,S7) --> equal(f4(T7::'a,R7),f4(T7::'a,S7))) &
(∀ U7 V7 W7. equal(U7::'a,V7) --> equal(f5(U7::'a,W7),f5(V7::'a,W7))) &
(∀ X7 Z7 Y7. equal(X7::'a,Y7) --> equal(f5(Z7::'a,X7),f5(Z7::'a,Y7))) &
(∀ A8 B8 C8. equal(A8::'a,B8) --> equal(f6(A8::'a,C8),f6(B8::'a,C8))) &
(∀ D8 F8 E8. equal(D8::'a,E8) --> equal(f6(F8::'a,D8),f6(F8::'a,E8))) &
(∀ G8 H8 I8. equal(G8::'a,H8) --> equal(f7(G8::'a,I8),f7(H8::'a,I8))) &
(∀ J8 L8 K8. equal(J8::'a,K8) --> equal(f7(L8::'a,J8),f7(L8::'a,K8))) &
(∀ M8 N8 O8. equal(M8::'a,N8) --> equal(f8(M8::'a,O8),f8(N8::'a,O8))) &
(∀ P8 R8 Q8. equal(P8::'a,Q8) --> equal(f8(R8::'a,P8),f8(R8::'a,Q8))) &
(∀ S8 T8 U8. equal(S8::'a,T8) --> equal(f9(S8::'a,U8),f9(T8::'a,U8))) &
(∀ V8 X8 W8. equal(V8::'a,W8) --> equal(f9(X8::'a,V8),f9(X8::'a,W8))) &
(∀ G H I'. equal(G::'a,H) --> equal(f10(G::'a,I'),f10(H::'a,I'))) &
(∀ 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'))) &

$(\forall P\ R\ Q.\ equal(P{::}'a,Q) \ {-}{-}{>}\ equal(f11(R{::}'a,P),f11(R{::}'a,Q)))\ \&$
$(\forall S'\ T'\ U.\ equal(S'{::}'a,T') \ {-}{-}{>}\ equal(f12(S'{::}'a,U),f12(T'{::}'a,U)))\ \&$
$(\forall V\ X\ W.\ equal(V{::}'a,W) \ {-}{-}{>}\ equal(f12(X{::}'a,V),f12(X{::}'a,W)))\ \&$
$(\forall Y\ Z\ A1.\ equal(Y{::}'a,Z) \ {-}{-}{>}\ equal(f13(Y{::}'a,A1),f13(Z{::}'a,A1)))\ \&$
$(\forall B1\ D1\ C1.\ equal(B1{::}'a,C1) \ {-}{-}{>}\ equal(f13(D1{::}'a,B1),f13(D1{::}'a,C1)))\ \&$
$(\forall E1\ F1\ G1.\ equal(E1{::}'a,F1) \ {-}{-}{>}\ equal(f14(E1{::}'a,G1),f14(F1{::}'a,G1)))\ \&$
$(\forall H1\ J1\ I1.\ equal(H1{::}'a,I1) \ {-}{-}{>}\ equal(f14(J1{::}'a,H1),f14(J1{::}'a,I1)))\ \&$
$(\forall K1\ L1\ M1.\ equal(K1{::}'a,L1) \ {-}{-}{>}\ equal(f16(K1{::}'a,M1),f16(L1{::}'a,M1)))\ \&$
$(\forall N1\ P1\ O1.\ equal(N1{::}'a,O1) \ {-}{-}{>}\ equal(f16(P1{::}'a,N1),f16(P1{::}'a,O1)))\ \&$
$(\forall Q1\ R1\ S1.\ equal(Q1{::}'a,R1) \ {-}{-}{>}\ equal(f17(Q1{::}'a,S1),f17(R1{::}'a,S1)))\ \&$
$(\forall T1\ V1\ U1.\ equal(T1{::}'a,U1) \ {-}{-}{>}\ equal(f17(V1{::}'a,T1),f17(V1{::}'a,U1)))\ \&$
$(\forall W1\ X1.\ equal(W1{::}'a,X1) \ {-}{-}{>}\ equal(f18(W1),f18(X1)))\ \&$
$(\forall Y1\ Z1.\ equal(Y1{::}'a,Z1) \ {-}{-}{>}\ equal(f19(Y1),f19(Z1)))\ \&$
$(\forall C2\ D2.\ equal(C2{::}'a,D2) \ {-}{-}{>}\ equal(f20(C2),f20(D2)))\ \&$
$(\forall E2\ F2.\ equal(E2{::}'a,F2) \ {-}{-}{>}\ equal(f21(E2),f21(F2)))\ \&$
$(\forall G2\ H2\ I2\ J2.\ equal(G2{::}'a,H2) \ {-}{-}{>}\ equal(f22(G2{::}'a,I2,J2),f22(H2{::}'a,I2,J2)))$
$\&$
$(\forall K2\ M2\ L2\ N2.\ equal(K2{::}'a,L2) \ {-}{-}{>}\ equal(f22(M2{::}'a,K2,N2),f22(M2{::}'a,L2,N2)))$
$\&$
$(\forall O2\ Q2\ R2\ P2.\ equal(O2{::}'a,P2) \ {-}{-}{>}\ equal(f22(Q2{::}'a,R2,O2),f22(Q2{::}'a,R2,P2)))$
$\&$
$(\forall S2\ T2\ U2.\ equal(S2{::}'a,T2) \ {-}{-}{>}\ equal(f23(S2{::}'a,U2),f23(T2{::}'a,U2)))\ \&$
$(\forall V2\ X2\ W2.\ equal(V2{::}'a,W2) \ {-}{-}{>}\ equal(f23(X2{::}'a,V2),f23(X2{::}'a,W2)))$
$\&$
$(\forall Y2\ Z2.\ equal(Y2{::}'a,Z2) \ {-}{-}{>}\ equal(f24(Y2),f24(Z2)))\ \&$
$(\forall A3\ B3.\ equal(A3{::}'a,B3) \ {-}{-}{>}\ equal(f26(A3),f26(B3)))\ \&$
$(\forall C3\ D3\ E3.\ equal(C3{::}'a,D3) \ {-}{-}{>}\ equal(f27(C3{::}'a,E3),f27(D3{::}'a,E3)))\ \&$
$(\forall F3\ H3\ G3.\ equal(F3{::}'a,G3) \ {-}{-}{>}\ equal(f27(H3{::}'a,F3),f27(H3{::}'a,G3)))\ \&$
$(\forall I3\ J3\ K3\ L3.\ equal(I3{::}'a,J3) \ {-}{-}{>}\ equal(f28(I3{::}'a,K3,L3),f28(J3{::}'a,K3,L3)))$
$\&$
$(\forall M3\ O3\ N3\ P3.\ equal(M3{::}'a,N3) \ {-}{-}{>}\ equal(f28(O3{::}'a,M3,P3),f28(O3{::}'a,N3,P3)))$
$\&$
$(\forall Q3\ S3\ T3\ R3.\ equal(Q3{::}'a,R3) \ {-}{-}{>}\ equal(f28(S3{::}'a,T3,Q3),f28(S3{::}'a,T3,R3)))$
$\&$
$(\forall U3\ V3\ W3\ X3.\ equal(U3{::}'a,V3) \ {-}{-}{>}\ equal(f29(U3{::}'a,W3,X3),f29(V3{::}'a,W3,X3)))$
$\&$
$(\forall Y3\ A4\ Z3\ B4.\ equal(Y3{::}'a,Z3) \ {-}{-}{>}\ equal(f29(A4{::}'a,Y3,B4),f29(A4{::}'a,Z3,B4)))$
$\&$
$(\forall C4\ E4\ F4\ D4.\ equal(C4{::}'a,D4) \ {-}{-}{>}\ equal(f29(E4{::}'a,F4,C4),f29(E4{::}'a,F4,D4)))$
$\&$
$(\forall I4\ J4\ K4\ L4.\ equal(I4{::}'a,J4) \ {-}{-}{>}\ equal(f30(I4{::}'a,K4,L4),f30(J4{::}'a,K4,L4)))$
$\&$
$(\forall M4\ O4\ N4\ P4.\ equal(M4{::}'a,N4) \ {-}{-}{>}\ equal(f30(O4{::}'a,M4,P4),f30(O4{::}'a,N4,P4)))$
$\&$
$(\forall Q4\ S4\ T4\ R4.\ equal(Q4{::}'a,R4) \ {-}{-}{>}\ equal(f30(S4{::}'a,T4,Q4),f30(S4{::}'a,T4,R4)))$
$\&$
$(\forall U4\ V4\ W4\ X4.\ equal(U4{::}'a,V4) \ {-}{-}{>}\ equal(f31(U4{::}'a,W4,X4),f31(V4{::}'a,W4,X4)))$
$\&$
$(\forall Y4\ A5\ Z4\ B5.\ equal(Y4{::}'a,Z4) \ {-}{-}{>}\ equal(f31(A5{::}'a,Y4,B5),f31(A5{::}'a,Z4,B5)))$

&
(∀ C5 E5 F5 D5. equal(C5::'a,D5) --> equal(f31(E5::'a,F5,C5),f31(E5::'a,F5,D5)))
&
(∀ G5 H5 I5 J5 K5 L5. equal(G5::'a,H5) --> equal(f32(G5::'a,I5,J5,K5,L5),f32(H5::'a,I5,J5,K5,L5)))
&
(∀ M5 O5 N5 P5 Q5 R5. equal(M5::'a,N5) --> equal(f32(O5::'a,M5,P5,Q5,R5),f32(O5::'a,N5,P5,Q5,R5
&
(∀ S5 U5 V5 T5 W5 X5. equal(S5::'a,T5) --> equal(f32(U5::'a,V5,S5,W5,X5),f32(U5::'a,V5,T5,W5,X5)
&
(∀ Y5 A6 B6 C6 Z5 D6. equal(Y5::'a,Z5) --> equal(f32(A6::'a,B6,C6,Y5,D6),f32(A6::'a,B6,C6,Z5,D6)))
&
(∀ E6 G6 H6 I6 J6 F6. equal(E6::'a,F6) --> equal(f32(G6::'a,H6,I6,J6,E6),f32(G6::'a,H6,I6,J6,F6)))
&
(∀ K6 L6 M6 N6 O6 P6. equal(K6::'a,L6) --> equal(f33(K6::'a,M6,N6,O6,P6),f33(L6::'a,M6,N6,O6,P6)
&
(∀ Q6 S6 R6 T6 U6 V6. equal(Q6::'a,R6) --> equal(f33(S6::'a,Q6,T6,U6,V6),f33(S6::'a,R6,T6,U6,V6)))
&
(∀ W6 Y6 Z6 X6 A7 B7. equal(W6::'a,X6) --> equal(f33(Y6::'a,Z6,W6,A7,B7),f33(Y6::'a,Z6,X6,A7,B7)
&
(∀ C7 E7 F7 G7 D7 H7. equal(C7::'a,D7) --> equal(f33(E7::'a,F7,G7,C7,H7),f33(E7::'a,F7,G7,D7,H7))
&
(∀ I7 K7 L7 M7 N7 J7. equal(I7::'a,J7) --> equal(f33(K7::'a,L7,M7,N7,I7),f33(K7::'a,L7,M7,N7,J7)))
&
(∀ 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 I' J. equal(G::'a,H) --> equal(apply-to-two-arguments(G::'a,I',J),apply-to-two-arguments(H::'a,I',
&
(∀ K' M L N. equal(K'::'a,L) --> equal(apply-to-two-arguments(M::'a,K',N),apply-to-two-arguments(M::'
&
(∀ O' Q R P. equal(O'::'a,P) --> equal(apply-to-two-arguments(Q::'a,R,O'),apply-to-two-arguments(Q::'a,
&
(∀ S' T'. equal(S'::'a,T') --> equal(complement(S'),complement(T'))) &
(∀ U V W. equal(U::'a,V) --> equal(composition(U::'a,W),composition(V::'a,W)))
&
(∀ X Z Y. equal(X::'a,Y) --> equal(composition(Z::'a,X),composition(Z::'a,Y)))
&
(∀ A1 B1. equal(A1::'a,B1) --> equal(inv1 A1,inv1 B1)) &
(∀ C1 D1 E1. equal(C1::'a,D1) --> equal(cross-product(C1::'a,E1),cross-product(D1::'a,E1)))
&
(∀ F1 H1 G1. equal(F1::'a,G1) --> equal(cross-product(H1::'a,F1),cross-product(H1::'a,G1)))
&
(∀ I1 J1. equal(I1::'a,J1) --> equal(domain-of(I1),domain-of(J1))) &
(∀ I10 J10. equal(I10::'a,J10) --> equal(first(I10),first(J10))) &
(∀ Q10 R10. equal(Q10::'a,R10) --> equal(flip-range-of(Q10),flip-range-of(R10)))
&
(∀ S10 T10 U10. equal(S10::'a,T10) --> equal(image'(S10::'a,U10),image'(T10::'a,U10)))
&
(∀ V10 X10 W10. equal(V10::'a,W10) --> equal(image'(X10::'a,V10),image'(X10::'a,W10)))
&

(∀ Y10 Z10 A11. equal(Y10::'a,Z10) --> equal(intersection(Y10::'a,A11),intersection(Z10::'a,A11)))
&
(∀ B11 D11 C11. equal(B11::'a,C11) --> equal(intersection(D11::'a,B11),intersection(D11::'a,C11)))
&
(∀ E11 F11 G11. equal(E11::'a,F11) --> equal(non-ordered-pair(E11::'a,G11),non-ordered-pair(F11::'a,G
&
(∀ H11 J11 I11. equal(H11::'a,I11) --> equal(non-ordered-pair(J11::'a,H11),non-ordered-pair(J11::'a,I11)
&
(∀ K11 L11 M11. equal(K11::'a,L11) --> equal(ordered-pair(K11::'a,M11),ordered-pair(L11::'a,M11)))
&
(∀ N11 P11 O11. equal(N11::'a,O11) --> equal(ordered-pair(P11::'a,N11),ordered-pair(P11::'a,O11)))
&
(∀ Q11 R11. equal(Q11::'a,R11) --> equal(powerset(Q11),powerset(R11))) &
(∀ S11 T11. equal(S11::'a,T11) --> equal(range-of(S11),range-of(T11))) &
(∀ U11 V11 W11. equal(U11::'a,V11) --> equal(restrct(U11::'a,W11),restrct(V11::'a,W11)))
&
(∀ X11 Z11 Y11. equal(X11::'a,Y11) --> equal(restrct(Z11::'a,X11),restrct(Z11::'a,Y11)))
&
(∀ A12 B12. equal(A12::'a,B12) --> equal(rot-right(A12),rot-right(B12))) &
(∀ C12 D12. equal(C12::'a,D12) --> equal(second(C12),second(D12))) &
(∀ K12 L12. equal(K12::'a,L12) --> equal(sigma(K12),sigma(L12))) &
(∀ M12 N12. equal(M12::'a,N12) --> equal(singleton-set(M12),singleton-set(N12)))
&
(∀ O12 P12. equal(O12::'a,P12) --> equal(successor(O12),successor(P12))) &
(∀ Q12 R12 S12. equal(Q12::'a,R12) --> equal(union(Q12::'a,S12),union(R12::'a,S12)))
&
(∀ T12 V12 U12. equal(T12::'a,U12) --> equal(union(V12::'a,T12),union(V12::'a,U12)))
&
(∀ W12 X12 Y12. equal(W12::'a,X12) & closed(W12::'a,Y12) --> closed(X12::'a,Y12))
&
(∀ Z12 B13 A13. equal(Z12::'a,A13) & closed(B13::'a,Z12) --> closed(B13::'a,A13))
&
(∀ C13 D13 E13. equal(C13::'a,D13) & disjoint(C13::'a,E13) --> disjoint(D13::'a,E13))
&
(∀ F13 H13 G13. equal(F13::'a,G13) & disjoint(H13::'a,F13) --> disjoint(H13::'a,G13))
&
(∀ I13 J13. equal(I13::'a,J13) & function(I13) --> function(J13)) &
(∀ K13 L13 M13 N13 O13 P13. equal(K13::'a,L13) & homomorphism(K13::'a,M13,N13,O13,P13)
--> homomorphism(L13::'a,M13,N13,O13,P13)) &
(∀ Q13 S13 R13 T13 U13 V13. equal(Q13::'a,R13) & homomorphism(S13::'a,Q13,T13,U13,V13)
--> homomorphism(S13::'a,R13,T13,U13,V13)) &
(∀ W13 Y13 Z13 X13 A14 B14. equal(W13::'a,X13) & homomorphism(Y13::'a,Z13,W13,A14,B14)
--> homomorphism(Y13::'a,Z13,X13,A14,B14)) &
(∀ C14 E14 F14 G14 D14 H14. equal(C14::'a,D14) & homomorphism(E14::'a,F14,G14,C14,H14)
--> homomorphism(E14::'a,F14,G14,D14,H14)) &
(∀ I14 K14 L14 M14 N14 J14. equal(I14::'a,J14) & homomorphism(K14::'a,L14,M14,N14,I14)
--> homomorphism(K14::'a,L14,M14,N14,J14)) &
(∀ O14 P14. equal(O14::'a,P14) & little-set(O14) --> little-set(P14)) &
(∀ Q14 R14 S14 T14. equal(Q14::'a,R14) & maps(Q14::'a,S14,T14) --> maps(R14::'a,S14,T14))

&
(∀ *U14 W14 V14 X14*. *equal*(*U14*::′*a*,*V14*) & *maps*(*W14*::′*a*,*U14*,*X14*) −−>
maps(*W14*::′*a*,*V14*,*X14*)) &
(∀ *Y14 A15 B15 Z14*. *equal*(*Y14*::′*a*,*Z14*) & *maps*(*A15*::′*a*,*B15*,*Y14*) −−> *maps*(*A15*::′*a*,*B15*,*Z14*))
&
(∀ *C15 D15 E15*. *equal*(*C15*::′*a*,*D15*) & *member*(*C15*::′*a*,*E15*) −−> *member*(*D15*::′*a*,*E15*))
&
(∀ *F15 H15 G15*. *equal*(*F15*::′*a*,*G15*) & *member*(*H15*::′*a*,*F15*) −−> *member*(*H15*::′*a*,*G15*))
&
(∀ *I15 J15*. *equal*(*I15*::′*a*,*J15*) & *one-to-one-function*(*I15*) −−> *one-to-one-function*(*J15*))
&
(∀ *K15 L15*. *equal*(*K15*::′*a*,*L15*) & *ordered-pair-predicate*(*K15*) −−> *ordered-pair-predicate*(*L15*))
&
(∀ *M15 N15 O15*. *equal*(*M15*::′*a*,*N15*) & *proper-subset*(*M15*::′*a*,*O15*) −−> *proper-subset*(*N15*::′*a*,*O15*))
&
(∀ *P15 R15 Q15*. *equal*(*P15*::′*a*,*Q15*) & *proper-subset*(*R15*::′*a*,*P15*) −−> *proper-subset*(*R15*::′*a*,*Q15*))
&
(∀ *S15 T15*. *equal*(*S15*::′*a*,*T15*) & *relation*(*S15*) −−> *relation*(*T15*)) &
(∀ *U15 V15*. *equal*(*U15*::′*a*,*V15*) & *single-valued-set*(*U15*) −−> *single-valued-set*(*V15*))
&
(∀ *W15 X15 Y15*. *equal*(*W15*::′*a*,*X15*) & *ssubset*(*W15*::′*a*,*Y15*) −−> *ssubset*(*X15*::′*a*,*Y15*))
&
(∀ *Z15 B16 A16*. *equal*(*Z15*::′*a*,*A16*) & *ssubset*(*B16*::′*a*,*Z15*) −−> *ssubset*(*B16*::′*a*,*A16*))
&
(~*little-set*(*ordered-pair*(*a*::′*a*,*b*))) −−> *False*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *SYN034-1*:

$(\forall A.\ p(A::'a,a) \mid p(A::'a,f(A)))$ &
$(\forall A.\ p(A::'a,a) \mid p(f(A),A))$ &
$(\forall A\ B.\ \sim(p(A::'a,B)\ \&\ p(B::'a,A)\ \&\ p(B::'a,a)))$ −−> *False*
⟨*proof*⟩

lemma *SYN071-1*:
EQU001-0-ax equal &
$(equal(a::'a,b) \mid equal(c::'a,d))$ &
$(equal(a::'a,c) \mid equal(b::'a,d))$ &
$(\sim equal(a::'a,d))$ &
$(\sim equal(b::'a,c))$ −−> *False*
⟨*proof*⟩

lemma *SYN349-1*:
$(\forall X\ Y.\ f(w(X),g(X::'a,Y))$ −−> $f(X::'a,g(X::'a,Y)))$ &
$(\forall X\ Y.\ f(X::'a,g(X::'a,Y))$ −−> $f(w(X),g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(Y::'a,g(X::'a,Y))$ −−> $f(g(X::'a,Y),Y) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(g(X::'a,Y),Y)\ \&\ f(Y::'a,g(X::'a,Y))$ −−> $f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),Y) \mid f(Y::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),Y)$ −−> $f(Y::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),w(X)))$ &
$(\forall Y\ X.\ f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X))$ −−> $f(g(X::'a,Y),Y) \mid f(Y::'a,g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(g(X::'a,Y),Y)\ \&\ f(g(X::'a,Y),w(X))$ −−> $f(X::'a,g(X::'a,Y)) \mid f(Y::'a,g(X::'a,Y)))$ &
$(\forall Y\ X.\ f(Y::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X))$ −−> $f(X::'a,g(X::'a,Y)) \mid f(g(X::'a,Y),Y))$ &
$(\forall Y\ X.\ \sim(f(X::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),Y)\ \&\ f(Y::'a,g(X::'a,Y))\ \&\ f(g(X::'a,Y),w(X))))$ −−> *False*
⟨*proof*⟩

lemma *SYN352-1*:
$(f(a::'a,b))$ &
$(\forall X\ Y.\ f(X::'a,Y)$ −−> $f(b::'a,z(X::'a,Y)) \mid f(Y::'a,z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(X::'a,Y) \mid f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(b::'a,z(X::'a,Y)) \mid f(X::'a,z(X::'a,Y)) \mid f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ f(b::'a,z(X::'a,Y))\ \&\ f(X::'a,z(X::'a,Y))$ −−> $f(z(X::'a,Y),z(X::'a,Y)))$ &
$(\forall X\ Y.\ \sim(f(X::'a,Y)\ \&\ f(X::'a,z(X::'a,Y))\ \&\ f(Y::'a,z(X::'a,Y))))$ &
$(\forall X\ Y.\ f(X::'a,Y)$ −−> $f(X::'a,z(X::'a,Y)) \mid f(Y::'a,z(X::'a,Y)))$ −−> *False*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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) | subset-collections(f5(X::'a, Vt), Vt)) &
(∀ X Vt. equal-sets(union-of-members(Vt),X) & element-of-collection(empty-set::'a, Vt) & element-of-collection(X::'a, Vt) & element-of-collection(union-of-members(f5(X::'a, Vt)), Vt) --> topological-space(X::'a, Vt) | element-of-collection(f3(X::'a, Vt), Vt)) &
(∀ X Vt. equal-sets(union-of-members(Vt),X) & element-of-collection(empty-set::'a, Vt) & element-of-collection(X::'a, Vt) --> topological-space(X::'a, Vt) | element-of-collection(f4(X::'a, Vt), Vt) | subset-collections(f5(X::'a, Vt), Vt)) &
(∀ X Vt. equal-sets(union-of-members(Vt),X) & element-of-collection(empty-set::'a, Vt) & element-of-collection(X::'a, Vt) & element-of-collection(union-of-members(f5(X::'a, Vt)), Vt) --> topological-space(X::'a, Vt) | element-of-collection(f4(X::'a, Vt), Vt)) &
(∀ X Vt. equal-sets(union-of-members(Vt),X) & element-of-collection(empty-set::'a, Vt) & element-of-collection(X::'a, Vt) & element-of-collection(intersection-of-sets(f3(X::'a, Vt),f4(X::'a, Vt)), Vt) --> topological-space(X::'a, Vt) | subset-collections(f5(X::'a, Vt), Vt)) &
(∀ X Vt. equal-sets(union-of-members(Vt),X) & element-of-collection(empty-set::'a, Vt) & element-of-collection(X::'a, Vt) & element-of-collection(intersection-of-sets(f3(X::'a, Vt),f4(X::'a, Vt)), Vt) & element-of-collection(union-of-members(f5(X::'a, Vt)), Vt) --> topological-space(X::'a, Vt)) &
(∀ U X Vt. open(U::'a,X, Vt) --> topological-space(X::'a, Vt)) &
(∀ X U Vt. open(U::'a,X, Vt) --> element-of-collection(U::'a, Vt)) &
(∀ X U Vt. topological-space(X::'a, Vt) & element-of-collection(U::'a, Vt) --> open(U::'a,X, Vt)) &
(∀ U X Vt. closed(U::'a,X, Vt) --> topological-space(X::'a, Vt)) &
(∀ U X Vt. closed(U::'a,X, Vt) --> open(relative-complement-sets(U::'a,X),X, Vt)) &
(∀ U X Vt. topological-space(X::'a, Vt) & open(relative-complement-sets(U::'a,X),X, Vt) --> closed(U::'a,X, Vt)) &
(∀ Vs X Vt. finer(Vt::'a, Vs,X) --> topological-space(X::'a, Vt)) &
(∀ Vt X Vs. finer(Vt::'a, Vs,X) --> topological-space(X::'a, Vs)) &
(∀ X Vs Vt. finer(Vt::'a, Vs,X) --> subset-collections(Vs::'a, Vt)) &
(∀ X Vs Vt. topological-space(X::'a, Vt) & topological-space(X::'a, Vs) & subset-collections(Vs::'a, Vt) --> finer(Vt::'a, Vs,X)) &
(∀ 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 X. equal-sets(union-of-members(Vf),X) --> basis(X::'a, Vf) | element-of-set(f7(X::'a, Vf),X))

&
(∀ X Vf. equal-sets(union-of-members(Vf),X) --> basis(X::'a,Vf) | element-of-collection(f8(X::'a,Vf),Vf)
&
(∀ X Vf. equal-sets(union-of-members(Vf),X) --> basis(X::'a,Vf) | element-of-collection(f9(X::'a,Vf),Vf)
&
(∀ X Vf. equal-sets(union-of-members(Vf),X) --> basis(X::'a,Vf) | element-of-set(f7(X::'a,Vf),intersectio
&
(∀ Uu9 X Vf. equal-sets(union-of-members(Vf),X) & element-of-set(f7(X::'a,Vf),Uu9)
& element-of-collection(Uu9::'a,Vf) & subset-sets(Uu9::'a,intersection-of-sets(f8(X::'a,Vf),f9(X::'a,Vf)))
--> basis(X::'a,Vf)) &
(∀ Vf U X. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
--> element-of-set(X::'a,f10(Vf::'a,U,X))) &
(∀ U X Vf. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
--> element-of-collection(f10(Vf::'a,U,X),Vf)) &
(∀ Vf X U. element-of-collection(U::'a,top-of-basis(Vf)) & element-of-set(X::'a,U)
--> subset-sets(f10(Vf::'a,U,X),U)) &
(∀ Vf U. element-of-collection(U::'a,top-of-basis(Vf)) | element-of-set(f11(Vf::'a,U),U))
&
(∀ Vf Uu11 U. element-of-set(f11(Vf::'a,U),Uu11) & element-of-collection(Uu11::'a,Vf)
& subset-sets(Uu11::'a,U) --> element-of-collection(U::'a,top-of-basis(Vf))) &
(∀ U Y X Vt. element-of-collection(U::'a,subspace-topology(X::'a,Vt,Y)) -->
topological-space(X::'a,Vt)) &
(∀ U Vt Y X. element-of-collection(U::'a,subspace-topology(X::'a,Vt,Y)) -->
subset-sets(Y::'a,X)) &
(∀ X Y U Vt. element-of-collection(U::'a,subspace-topology(X::'a,Vt,Y)) -->
element-of-collection(f12(X::'a,Vt,Y,U),Vt)) &
(∀ X Vt Y U. element-of-collection(U::'a,subspace-topology(X::'a,Vt,Y)) -->
equal-sets(U::'a,intersection-of-sets(Y::'a,f12(X::'a,Vt,Y,U)))) &
(∀ X Vt U Y Uu12. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) & element-of-collection(Uu12::'a,Vt)
& equal-sets(U::'a,intersection-of-sets(Y::'a,Uu12)) --> element-of-collection(U::'a,subspace-topology(X::'a
&
(∀ U Y X Vt. element-of-set(U::'a,interior(Y::'a,X,Vt)) --> topological-space(X::'a,Vt))
&
(∀ U Vt Y X. element-of-set(U::'a,interior(Y::'a,X,Vt)) --> subset-sets(Y::'a,X))
&
(∀ Y X Vt U. element-of-set(U::'a,interior(Y::'a,X,Vt)) --> element-of-set(U::'a,f13(Y::'a,X,Vt,U)))
&
(∀ X Vt U Y. element-of-set(U::'a,interior(Y::'a,X,Vt)) --> subset-sets(f13(Y::'a,X,Vt,U),Y))
&
(∀ Y U X Vt. element-of-set(U::'a,interior(Y::'a,X,Vt)) --> open(f13(Y::'a,X,Vt,U),X,Vt))
&
(∀ U Y Uu13 X Vt. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) & element-of-set(U::'a,Uu13)
& subset-sets(Uu13::'a,Y) & open(Uu13::'a,X,Vt) --> element-of-set(U::'a,interior(Y::'a,X,Vt)))
&
(∀ U Y X Vt. element-of-set(U::'a,closure(Y::'a,X,Vt)) --> topological-space(X::'a,Vt))
&
(∀ U Vt Y X. element-of-set(U::'a,closure(Y::'a,X,Vt)) --> subset-sets(Y::'a,X))
&
(∀ Y X Vt U V. element-of-set(U::'a,closure(Y::'a,X,Vt)) & subset-sets(Y::'a,V)

& closed(V::'a,X,Vt) --> element-of-set(U::'a,V)) &
(∀ Y X Vt U. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) --> element-of-set(U::'a,closure(Y::'a,X,
| subset-sets(Y::'a,f14(Y::'a,X,Vt,U))) &
(∀ Y U X Vt. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) --> element-of-set(U::'a,closure(Y::'a,X,
| closed(f14(Y::'a,X,Vt,U),X,Vt)) &
(∀ Y X Vt U. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) & element-of-set(U::'a,f14(Y::'a,X,Vt,U))
--> element-of-set(U::'a,closure(Y::'a,X,Vt))) &
(∀ U Y X Vt. neighborhood(U::'a,Y,X,Vt) --> topological-space(X::'a,Vt)) &
(∀ Y U X Vt. neighborhood(U::'a,Y,X,Vt) --> open(U::'a,X,Vt)) &
(∀ X Vt Y U. neighborhood(U::'a,Y,X,Vt) --> element-of-set(Y::'a,U)) &
(∀ X Vt Y U. topological-space(X::'a,Vt) & open(U::'a,X,Vt) & element-of-set(Y::'a,U)
--> neighborhood(U::'a,Y,X,Vt)) &
(∀ Z Y X Vt. limit-point(Z::'a,Y,X,Vt) --> topological-space(X::'a,Vt)) &
(∀ Z Vt Y X. limit-point(Z::'a,Y,X,Vt) --> subset-sets(Y::'a,X)) &
(∀ Z X Vt U Y. limit-point(Z::'a,Y,X,Vt) & neighborhood(U::'a,Z,X,Vt) -->
element-of-set(f15(Z::'a,Y,X,Vt,U),intersection-of-sets(U::'a,Y))) &
(∀ Y X Vt U Z. ~(limit-point(Z::'a,Y,X,Vt) & neighborhood(U::'a,Z,X,Vt) &
eq-p(f15(Z::'a,Y,X,Vt,U),Z))) &
(∀ Y Z X Vt. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) --> limit-point(Z::'a,Y,X,Vt)
| neighborhood(f16(Z::'a,Y,X,Vt),Z,X,Vt)) &
(∀ X Vt Y Uu16 Z. topological-space(X::'a,Vt) & subset-sets(Y::'a,X) & element-of-set(Uu16::'a,intersection
--> limit-point(Z::'a,Y,X,Vt) | eq-p(Uu16::'a,Z)) &
(∀ U Y X Vt. element-of-set(U::'a,boundary(Y::'a,X,Vt)) --> topological-space(X::'a,Vt))
&
(∀ U Y X Vt. element-of-set(U::'a,boundary(Y::'a,X,Vt)) --> element-of-set(U::'a,closure(Y::'a,X,Vt)))
&
(∀ U Y X Vt. element-of-set(U::'a,boundary(Y::'a,X,Vt)) --> element-of-set(U::'a,closure(relative-comple
&
(∀ U Y X Vt. topological-space(X::'a,Vt) & element-of-set(U::'a,closure(Y::'a,X,Vt))
& element-of-set(U::'a,closure(relative-complement-sets(Y::'a,X),X,Vt)) --> element-of-set(U::'a,boundary
&
(∀ X Vt. hausdorff(X::'a,Vt) --> topological-space(X::'a,Vt)) &
(∀ X-2 X-1 X Vt. hausdorff(X::'a,Vt) & element-of-set(X-1::'a,X) & element-of-set(X-2::'a,X)
--> eq-p(X-1::'a,X-2) | neighborhood(f17(X::'a,Vt,X-1,X-2),X-1,X,Vt)) &
(∀ X-1 X-2 X Vt. hausdorff(X::'a,Vt) & element-of-set(X-1::'a,X) & element-of-set(X-2::'a,X)
--> eq-p(X-1::'a,X-2) | neighborhood(f18(X::'a,Vt,X-1,X-2),X-2,X,Vt)) &
(∀ X Vt X-1 X-2. hausdorff(X::'a,Vt) & element-of-set(X-1::'a,X) & element-of-set(X-2::'a,X)
--> eq-p(X-1::'a,X-2) | disjoint-s(f17(X::'a,Vt,X-1,X-2),f18(X::'a,Vt,X-1,X-2)))
&
(∀ Vt X. topological-space(X::'a,Vt) --> hausdorff(X::'a,Vt) | element-of-set(f19(X::'a,Vt),X))
&
(∀ Vt X. topological-space(X::'a,Vt) --> hausdorff(X::'a,Vt) | element-of-set(f20(X::'a,Vt),X))
&
(∀ X Vt. topological-space(X::'a,Vt) & eq-p(f19(X::'a,Vt),f20(X::'a,Vt)) -->
hausdorff(X::'a,Vt)) &
(∀ X Vt Uu19 Uu20. topological-space(X::'a,Vt) & neighborhood(Uu19::'a,f19(X::'a,Vt),X,Vt)
& neighborhood(Uu20::'a,f20(X::'a,Vt),X,Vt) & disjoint-s(Uu19::'a,Uu20) -->
hausdorff(X::'a,Vt)) &
(∀ Va1 Va2 X Vt. separation(Va1::'a,Va2,X,Vt) --> topological-space(X::'a,Vt))

&
(∀ Va2 X Vt Va1. ~(separation(Va1::'a,Va2,X,Vt) & equal-sets(Va1::'a,empty-set)))
&
(∀ Va1 X Vt Va2. ~(separation(Va1::'a,Va2,X,Vt) & equal-sets(Va2::'a,empty-set)))
&
(∀ Va2 X Va1 Vt. separation(Va1::'a,Va2,X,Vt) --> element-of-collection(Va1::'a,Vt))
&
(∀ Va1 X Va2 Vt. separation(Va1::'a,Va2,X,Vt) --> element-of-collection(Va2::'a,Vt))
&
(∀ Vt Va1 Va2 X. separation(Va1::'a,Va2,X,Vt) --> equal-sets(union-of-sets(Va1::'a,Va2),X))
&
(∀ X Vt Va1 Va2. separation(Va1::'a,Va2,X,Vt) --> disjoint-s(Va1::'a,Va2))
&
(∀ Vt X Va1 Va2. topological-space(X::'a,Vt) & element-of-collection(Va1::'a,Vt)
& element-of-collection(Va2::'a,Vt) & equal-sets(union-of-sets(Va1::'a,Va2),X) &
disjoint-s(Va1::'a,Va2) --> separation(Va1::'a,Va2,X,Vt) | equal-sets(Va1::'a,empty-set)
| equal-sets(Va2::'a,empty-set)) &
(∀ X Vt. connected-space(X::'a,Vt) --> topological-space(X::'a,Vt)) &
(∀ Va1 Va2 X Vt. ~(connected-space(X::'a,Vt) & separation(Va1::'a,Va2,X,Vt)))
&
(∀ X Vt. topological-space(X::'a,Vt) --> connected-space(X::'a,Vt) | separation(f21(X::'a,Vt),f22(X::'a,Vt),X,Vt)) &
(∀ Va X Vt. connected-set(Va::'a,X,Vt) --> topological-space(X::'a,Vt)) &
(∀ Vt Va X. connected-set(Va::'a,X,Vt) --> subset-sets(Va::'a,X)) &
(∀ X Vt Va. connected-set(Va::'a,X,Vt) --> connected-space(Va::'a,subspace-topology(X::'a,Vt,Va)))
&
(∀ X Vt Va. topological-space(X::'a,Vt) & subset-sets(Va::'a,X) & connected-space(Va::'a,subspace-topology(X
--> connected-set(Va::'a,X,Vt)) &
(∀ Vf X Vt. open-covering(Vf::'a,X,Vt) --> topological-space(X::'a,Vt)) &
(∀ X Vf Vt. open-covering(Vf::'a,X,Vt) --> subset-collections(Vf::'a,Vt)) &
(∀ Vt Vf X. open-covering(Vf::'a,X,Vt) --> equal-sets(union-of-members(Vf),X))
&
(∀ Vt Vf X. topological-space(X::'a,Vt) & subset-collections(Vf::'a,Vt) & equal-sets(union-of-members(Vf),X
--> open-covering(Vf::'a,X,Vt)) &
(∀ X Vt. compact-space(X::'a,Vt) --> topological-space(X::'a,Vt)) &
(∀ X Vt Vf1. compact-space(X::'a,Vt) & open-covering(Vf1::'a,X,Vt) --> finite'(f23(X::'a,Vt,Vf1))) &
(∀ X Vt Vf1. compact-space(X::'a,Vt) & open-covering(Vf1::'a,X,Vt) --> subset-collections(f23(X::'a,Vt,V
&
(∀ Vf1 X Vt. compact-space(X::'a,Vt) & open-covering(Vf1::'a,X,Vt) --> open-covering(f23(X::'a,Vt,Vf1)
&
(∀ X Vt. topological-space(X::'a,Vt) --> compact-space(X::'a,Vt) | open-covering(f24(X::'a,Vt),X,Vt))
&
(∀ Uu24 X Vt. topological-space(X::'a,Vt) & finite'(Uu24) & subset-collections(Uu24::'a,f24(X::'a,Vt))
& open-covering(Uu24::'a,X,Vt) --> compact-space(X::'a,Vt)) &
(∀ Va X Vt. compact-set(Va::'a,X,Vt) --> topological-space(X::'a,Vt)) &
(∀ Vt Va X. compact-set(Va::'a,X,Vt) --> subset-sets(Va::'a,X)) &
(∀ X Vt Va. compact-set(Va::'a,X,Vt) --> compact-space(Va::'a,subspace-topology(X::'a,Vt,Va)))
&

(∀ X Vt Va. *topological-space*(X::'a,Vt) & *subset-sets*(Va::'a,X) & *compact-space*(Va::'a,*subspace-topology*(X:
−−> *compact-set*(Va::'a,X,Vt)) &
(*basis*(cx::'a,f)) &
(∀ U. *element-of-collection*(U::'a,*top-of-basis*(f))) &
(∀ V. *element-of-collection*(V::'a,*top-of-basis*(f))) &
(∀ U V. ~*element-of-collection*(*intersection-of-sets*(U::'a,V),*top-of-basis*(f))) −−>
False
⟨*proof*⟩

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),*inter
&
(∀ 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))) &

$(\forall\, V.\ \mathit{element\text{-}of\text{-}collection}(V{::}'a,\mathit{top\text{-}of\text{-}basis}(f)))$ &
$(\forall\, U\ V.\ {\sim}\mathit{element\text{-}of\text{-}collection}(\mathit{intersection\text{-}of\text{-}sets}(U{::}'a,V),\mathit{top\text{-}of\text{-}basis}(f)))$ --->
False
⟨*proof*⟩

lemma *TOP005-2*:
$(\forall\, Vf\ U.\ \mathit{element\text{-}of\text{-}set}(U{::}'a,\mathit{union\text{-}of\text{-}members}(Vf))$ ---> $\mathit{element\text{-}of\text{-}set}(U{::}'a,\mathit{f1}(Vf{::}'a,U)))$
&
$(\forall\, U\ Vf.\ \mathit{element\text{-}of\text{-}set}(U{::}'a,\mathit{union\text{-}of\text{-}members}(Vf))$ ---> $\mathit{element\text{-}of\text{-}collection}(\mathit{f1}(Vf{::}'a,U),Vf))$
&
$(\forall\, Vf\ U\ X.\ \mathit{element\text{-}of\text{-}collection}(U{::}'a,\mathit{top\text{-}of\text{-}basis}(Vf))$ & $\mathit{element\text{-}of\text{-}set}(X{::}'a,U)$
---> $\mathit{element\text{-}of\text{-}set}(X{::}'a,\mathit{f10}(Vf{::}'a,U,X)))$ &
$(\forall\, U\ X\ Vf.\ \mathit{element\text{-}of\text{-}collection}(U{::}'a,\mathit{top\text{-}of\text{-}basis}(Vf))$ & $\mathit{element\text{-}of\text{-}set}(X{::}'a,U)$
---> $\mathit{element\text{-}of\text{-}collection}(\mathit{f10}(Vf{::}'a,U,X),Vf))$ &
$(\forall\, Vf\ X\ U.\ \mathit{element\text{-}of\text{-}collection}(U{::}'a,\mathit{top\text{-}of\text{-}basis}(Vf))$ & $\mathit{element\text{-}of\text{-}set}(X{::}'a,U)$
---> $\mathit{subset\text{-}sets}(\mathit{f10}(Vf{::}'a,U,X),U))$ &
$(\forall\, Vf\ U.\ \mathit{element\text{-}of\text{-}collection}(U{::}'a,\mathit{top\text{-}of\text{-}basis}(Vf))\ |\ \mathit{element\text{-}of\text{-}set}(\mathit{f11}(Vf{::}'a,U),U))$
&
$(\forall\, Vf\ Uu11\ U.\ \mathit{element\text{-}of\text{-}set}(\mathit{f11}(Vf{::}'a,U),Uu11)$ & $\mathit{element\text{-}of\text{-}collection}(Uu11{::}'a,Vf)$
& $\mathit{subset\text{-}sets}(Uu11{::}'a,U)$ ---> $\mathit{element\text{-}of\text{-}collection}(U{::}'a,\mathit{top\text{-}of\text{-}basis}(Vf)))$ &
$(\forall\, X\ U\ Y.\ \mathit{element\text{-}of\text{-}set}(U{::}'a,X)$ ---> $\mathit{subset\text{-}sets}(X{::}'a,Y)\ |\ \mathit{element\text{-}of\text{-}set}(U{::}'a,Y))$
&
$(\forall\, Y\ X\ Z.\ \mathit{subset\text{-}sets}(X{::}'a,Y)$ & $\mathit{element\text{-}of\text{-}collection}(Y{::}'a,Z)$ ---> $\mathit{subset\text{-}sets}(X{::}'a,\mathit{union\text{-}of\text{-}members}(Z$
&
$(\forall\, X\ U\ Y.\ \mathit{subset\text{-}collections}(X{::}'a,Y)$ & $\mathit{element\text{-}of\text{-}collection}(U{::}'a,X)$ --->
$\mathit{element\text{-}of\text{-}collection}(U{::}'a,Y))$ &
$(\mathit{subset\text{-}collections}(g{::}'a,\mathit{top\text{-}of\text{-}basis}(f)))$ &
$({\sim}\mathit{element\text{-}of\text{-}collection}(\mathit{union\text{-}of\text{-}members}(g),\mathit{top\text{-}of\text{-}basis}(f)))$ ---> *False*
⟨*proof*⟩

end

41 Examples for Ferrante and Rackoff's quantifier elimination procedure

theory *Dense-Linear-Order-Ex*
imports *Main*
begin

lemma
$\exists\,(y{::}'a{::}\{\mathit{ordered\text{-}field},\mathit{recpower},\mathit{number\text{-}ring},\ \mathit{division\text{-}by\text{-}zero}\})\ {<}2.\ x + 3*\ y <$
$0 \wedge x - y > 0$
⟨*proof*⟩

lemma $\sim\ (ALL\ x\ (y{::}'a{::}\{\mathit{ordered\text{-}field},\mathit{recpower},\mathit{number\text{-}ring},\ \mathit{division\text{-}by\text{-}zero}\}).$
$x < y$ ---> $10{*}x < 11{*}y)$

⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y*. *x* < *y* −−> (*10*∗(*x* + *5*∗*y* + −*1*) < *60*∗*y*)
⟨*proof*⟩

lemma *EX* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y*. *x* ~= *y* −−> *x* < *y*
⟨*proof*⟩

lemma *EX* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y*. (*x* ~= *y* & *10*∗*x* ~= *9*∗*y* & *10*∗*x* < *y*) −−> *x* < *y*
⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y*. (*x* ~= *y* & *5*∗*x* <= *y*) −−> *500*∗*x* <= *100*∗*y*
⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). (*EX* (*y*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). *4*∗*x* + *3*∗*y* <= *0* & *4*∗*x* + *3*∗*y* >= −*1*)
⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) < *0*. (*EX* (*y*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) > *0*. *7*∗*x* + *y* > *0* & *x* − *y* <= *9*)
⟨*proof*⟩

lemma *EX* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). (*0* < *x* & *x* < *1*) −−> (*ALL y* > *1*. *x* + *y* ~= *1*)
⟨*proof*⟩

lemma *EX x*. (*ALL* (*y*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). *y* < *2* −−> *2*∗(*y* − *x*) $\leq$ *0*)
⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). *x* < *10* | *x* > *20* | (*EX y*. *y*>= *0* & *y* <= *10* & *x*+*y* = *20*)
⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y z*. *x* + *y* < *z* −−> *y* >= *z* −−> *x* < *0*
⟨*proof*⟩

lemma *EX* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y z*. *x* + *7*∗*y* < *5*∗ *z* & *5*∗*y* >= *7*∗*z* & *x* < *0*
⟨*proof*⟩

lemma *ALL* (*x*::′*a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y z*.

$abs\ (x + y) <= z\ -->\ (abs\ z = z)$
  ⟨*proof*⟩

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y\ z.\ x + 7*y - 5*\ z < 0\ \&\ 5*y + 7*z + 3*x < 0$
  ⟨*proof*⟩

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y\ z.\ (abs\ (5*x+3*y+z) <= 5*x+3*y+z\ \&\ abs\ (5*x+3*y+z) >= -\ (5*x+3*y+z))\ |\ (abs\ (5*x+3*y+z) >= 5*x+3*y+z\ \&\ abs\ (5*x+3*y+z) <= -\ (5*x+3*y+z))$
  ⟨*proof*⟩

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ x < y\ -->\ (EX\ z>0.\ x+z = y)$
  ⟨*proof*⟩

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ x < y\ -->\ (EX\ z>0.\ x+z = y)$
  ⟨*proof*⟩

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ (EX\ z>0.\ abs\ (x - y) <= z\ )$
  ⟨*proof*⟩

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ (ALL\ z<0.\ (z < x\ -->\ z <= y)\ \&\ (z > y\ -->\ z >= x))$
  ⟨*proof*⟩

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ (ALL\ z>=0.\ abs\ (3*x+7*y) <= 2*z + 1)$
  ⟨*proof*⟩

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})\ y.\ (ALL\ z<0.\ (z < x\ -->\ z <= y)\ \&\ (z > y\ -->\ z >= x))$
  ⟨*proof*⟩

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\})>0.\ (ALL\ y.\ (EX\ z.\ 13*\ abs\ z \neq abs\ (12*y - x)\ \&\ 5*x - 3*(abs\ y) <= 7*z))$
  ⟨*proof*⟩

lemma $EX\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).\ abs\ (4*x + 17) < 4\ \&\ (ALL\ y\ .\ abs\ (x*34 - 34*y - 9) \neq 0 \longrightarrow (EX\ z.\ 5*x - 3*abs\ y <= 7*z))$
  ⟨*proof*⟩

lemma $ALL\ (x::'a::\{ordered\text{-}field,recpower,number\text{-}ring,\ division\text{-}by\text{-}zero\}).\ (EX\ y > abs\ (23*x - 9).\ (ALL\ z > abs\ (3*y - 19*\ abs\ x).\ x+z > 2*y))$
  ⟨*proof*⟩

lemma *ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). (*EX* *y*< *abs* (*3*∗*x* − *1*). (*ALL* *z* >= (*3*∗*abs* *x* − *1*). *abs* (*12*∗*x* − *13*∗*y* + *19*∗*z*) > *abs* (*23*∗*x*)))
⟨*proof*⟩

lemma *EX* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). *abs* *x* < *100* & (*ALL* *y* > *x*. (*EX* *z*<*2*∗*y* − *x*. *5*∗*x* − *3*∗*y* <= *7*∗*z*))
⟨*proof*⟩

lemma *ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y* *z* *w*. *7*∗*x*<*3*∗*y* −−> *5*∗*y* < *7*∗*z* −−> *z* < *2*∗*w* −−> *7*∗(*2*∗*w*−*x*) > *2*∗*y*
⟨*proof*⟩

lemma *EX* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y* *z* *w*. *5*∗*x* + *3*∗*z* − *17*∗*w* + *abs* (*y* − *8*∗*x* + *z*) <= *89*
⟨*proof*⟩

lemma *EX* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y* *z* *w*. *5*∗*x* + *3*∗*z* − *17*∗*w* + *7*∗ (*y* − *8*∗*x* + *z*) <= *max* *y* (*7*∗*z* − *x* + *w*)
⟨*proof*⟩

lemma *EX* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y* *z* *w*. *min* (*5*∗*x* + *3*∗*z*) (*17*∗*w*) + *5*∗ *abs* (*y* − *8*∗*x* + *z*) <= *max* *y* (*7*∗*z* − *x* + *w*)
⟨*proof*⟩

lemma *ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y* *z*. (*EX* *w* >= (*x*+*y*+*z*). *w* <= *abs* *x* + *abs* *y* + *abs* *z*)
⟨*proof*⟩

lemma ~(*ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). (*EX* *y* *z* *w*. *3*∗ *x* + *z*∗*4* = *3*∗*y* & *x* + *y* < *z* & *x*> *w* & *3*∗*x* < *w* + *y*))
⟨*proof*⟩

lemma *ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y*. (*EX* *z* *w*. *abs* (*x*−*y*) = (*z*−*w*) & *z*∗*1234* < *233*∗*x* & *w* ~= *y*)
⟨*proof*⟩

lemma *ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}). (*EX* *y* *z* *w*. *min* (*5*∗*x* + *3*∗*z*) (*17*∗*w*) + *5*∗ *abs* (*y* − *8*∗*x* + *z*) <= *max* *y* (*7*∗*z* − *x* + *w*))
⟨*proof*⟩

lemma *EX* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y* *z*. (*ALL* *w* >= *abs* (*x*+*y*+*z*). *w* >= *abs* *x* + *abs* *y* + *abs* *z*)
⟨*proof*⟩

lemma *EX* *z*. (*ALL* (*x*::*'a*::{*ordered-field*,*recpower*,*number-ring*, *division-by-zero*}) *y*. (*EX* *w* >= (*x*+*y*+*z*). *w* <= *abs* *x* + *abs* *y* + *abs* *z*))
⟨*proof*⟩

lemma *EX z. (ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) < abs z. (EX y w. x< y & x < z & x> w & 3*x < w + y))*
⟨*proof*⟩

lemma *ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y. (EX z. (ALL w. abs (x−y) = abs (z−w) −−> z < x & w ~= y))*
⟨*proof*⟩

lemma *EX y. (ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX z w. min (5*x + 3*z) (17*w) + 5* abs (y − 8*x + z) <= max y (7*z − x + w)))*
⟨*proof*⟩

lemma *EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) z. (ALL w >= 13*x − 4*z. (EX y. w >= abs x + abs y + z))*
⟨*proof*⟩

lemma *EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (ALL y < x. (EX z > (x+y).*
*(ALL w. 5*w + 10*x − z >= y −−> w + 7*x + 3*z >= 2*y)))*
⟨*proof*⟩

lemma *EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (ALL y. (EX z > y.*
*(ALL w . w < 13 −−> w + 10*x − z >= y −−> 5*w + 7*x + 13*z >= 2*y)))*
⟨*proof*⟩

lemma *EX (x::'a::{ordered-field,recpower,number-ring, division-by-zero}) y z w. min (5*x + 3*z) (17*w) + 5* abs (y − 8*x + z) <= max y (7*z − x + w)*
⟨*proof*⟩

lemma *ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX y. (ALL z>19. y <= x + z & (EX w. abs (y − x) < w)))*
⟨*proof*⟩

lemma *ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX y. (ALL z>19. y <= x + z & (EX w. abs (x + z) < w − y)))*
⟨*proof*⟩

lemma *ALL (x::'a::{ordered-field,recpower,number-ring, division-by-zero}). (EX y. abs y ~= abs x & (ALL z> max x y. (EX w. w ~= y & w ~= z & 3*w − z >= x + y)))*
⟨*proof*⟩

end

42 Some examples for Presburger Arithmetic

theory *PresburgerEx*
imports *Presburger*
begin

lemma $\bigwedge m\ n\ ja\ ia.\ [\![\neg\ m \le j;\ \neg\ n \le i;\ e \ne 0;\ Suc\ j \le ja]\!] \Longrightarrow \exists\, m.\ \forall\, ja\ ia.\ m \le ja \longrightarrow (if\ j = ja \wedge i = ia\ then\ e\ else\ 0) = 0$ ⟨*proof*⟩
lemma $(0{::}nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
⟨*proof*⟩
lemma $(0{::}nat) < emBits\ mod\ 8 \Longrightarrow 8 + emBits\ div\ 8 * 8 - emBits = 8 - emBits\ mod\ 8$
⟨*proof*⟩

theorem $(\forall\,(y{::}int).\ 3\ dvd\ y) ==> \forall\,(x{::}int).\ b < x --> a \le x$
⟨*proof*⟩

theorem !! $(y{::}int)\ (z{::}int)\ (n{::}int).\ 3\ dvd\ z ==> 2\ dvd\ (y{::}int) ==>$
$(\exists\,(x{::}int).\ 2{*}x = y)\ \&\ (\exists\,(k{::}int).\ 3{*}k = z)$
⟨*proof*⟩

theorem !! $(y{::}int)\ (z{::}int)\ n.\ Suc(n{::}nat) < 6 ==> 3\ dvd\ z ==>$
$2\ dvd\ (y{::}int) ==> (\exists\,(x{::}int).\ 2{*}x = y)\ \&\ (\exists\,(k{::}int).\ 3{*}k = z)$
⟨*proof*⟩

theorem $\forall\,(x{::}nat).\ \exists\,(y{::}nat).\ (0{::}nat) \le 5 --> y = 5 + x$
⟨*proof*⟩

Slow: about 7 seconds on a 1.6GHz machine.

theorem $\forall\,(x{::}nat).\ \exists\,(y{::}nat).\ y = 5 + x \mid x\ div\ 6 + 1 = 2$
⟨*proof*⟩

theorem $\exists\,(x{::}int).\ 0 < x$
⟨*proof*⟩

theorem $\forall\,(x{::}int)\ y.\ x < y --> 2 * x + 1 < 2 * y$
⟨*proof*⟩

theorem $\forall\,(x{::}int)\ y.\ 2 * x + 1 \ne 2 * y$
⟨*proof*⟩

theorem $\exists\,(x{::}int)\ y.\ 0 < x\ \&\ 0 \le y\ \&\ 3 * x - 5 * y = 1$
⟨*proof*⟩

theorem $\sim\ (\exists\,(x{::}int)\ (y{::}int)\ (z{::}int).\ 4{*}x + (-6{::}int){*}y = 1)$
⟨*proof*⟩

theorem $\forall\,(x{::}int).\ b < x --> a \le x$

⟨*proof*⟩

theorem $\sim (\exists (x{::}int).\ False)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (a{::}int) < 3 * x \dashrightarrow b < 3 * x$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (2\ dvd\ x) \dashrightarrow (\exists (y{::}int).\ x = 2{*}y)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (2\ dvd\ x) \dashrightarrow (\exists (y{::}int).\ x = 2{*}y)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ (2\ dvd\ x) = (\exists (y{::}int).\ x = 2{*}y)$
⟨*proof*⟩

theorem $\forall (x{::}int).\ ((2\ dvd\ x) = (\forall (y{::}int).\ x \neq 2{*}y + 1))$
⟨*proof*⟩

theorem $\sim (\forall (x{::}int).$
$((2\ dvd\ x) = (\forall (y{::}int).\ x \neq 2{*}y{+}1) \mid$
$(\exists (q{::}int)\ (u{::}int)\ i.\ 3{*}i + 2{*}q - u < 17)$
$\dashrightarrow 0 < x \mid ((\sim 3\ dvd\ x)\ \&(x + 8 = 0))))$
⟨*proof*⟩

theorem $\sim (\forall (i{::}int).\ 4 \le i \dashrightarrow (\exists x\ y.\ 0 \le x\ \&\ 0 \le y\ \&\ 3 * x + 5 * y = i))$
⟨*proof*⟩

theorem $\forall (i{::}int).\ 8 \le i \dashrightarrow (\exists x\ y.\ 0 \le x\ \&\ 0 \le y\ \&\ 3 * x + 5 * y = i)$
⟨*proof*⟩

theorem $\exists (j{::}int).\ \forall i.\ j \le i \dashrightarrow (\exists x\ y.\ 0 \le x\ \&\ 0 \le y\ \&\ 3 * x + 5 * y = i)$
⟨*proof*⟩

theorem $\sim (\forall j\ (i{::}int).\ j \le i \dashrightarrow (\exists x\ y.\ 0 \le x\ \&\ 0 \le y\ \&\ 3 * x + 5 * y = i))$
⟨*proof*⟩

Slow: about 5 seconds on a 1.6GHz machine.

theorem $(\exists m{::}nat.\ n = 2 * m) \dashrightarrow (n + 1)\ div\ 2 = n\ div\ 2$
⟨*proof*⟩

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 $\llbracket$ *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* $\rrbracket$
$\Longrightarrow$ *x1 = x10* & *x2 = (x11::int)*
⟨*proof*⟩

end

theory *Reflected-Presburger*
imports *GCD Efficient-Nat*
uses (*coopereif.ML*) (*coopertac.ML*)
begin

function
iupt :: *int* ⇒ *int* ⇒ *int list*
where
iupt i j = (*if j* < *i then* [] *else i* # *iupt* (*i*+*1*) *j*)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *iupt-set*: *set* (*iupt i j*) = {*i..j*}
⟨*proof*⟩

datatype *num* = *C int* | *Bound nat* | *CN nat int num* | *Neg num* | *Add num num*|
Sub num num
| *Mul int num*

consts *num-size* :: *num* ⇒ *nat*
primrec
num-size (*C c*) = *1*
num-size (*Bound n*) = *1*
num-size (*Neg a*) = *1* + *num-size a*
num-size (*Add a b*) = *1* + *num-size a* + *num-size b*
num-size (*Sub a b*) = *3* + *num-size a* + *num-size b*
num-size (*CN n c a*) = *4* + *num-size a*
num-size (*Mul c a*) = *1* + *num-size a*

consts *Inum* :: *int list* ⇒ *num* ⇒ *int*
primrec
Inum bs (*C c*) = *c*
Inum bs (*Bound n*) = *bs*!*n*
Inum bs (*CN n c a*) = *c* ∗ (*bs*!*n*) + (*Inum bs a*)

Inum bs (*Neg a*) = −(*Inum bs a*)
Inum bs (*Add a b*) = *Inum bs a* + *Inum bs b*
Inum bs (*Sub a b*) = *Inum bs a* − *Inum bs b*
Inum bs (*Mul c a*) = *c*∗ *Inum bs a*

datatype *fm* =
T| *F*| *Lt num*| *Le num*| *Gt num*| *Ge num*| *Eq num*| *NEq num*| *Dvd int num*| *NDvd int num*|
NOT fm| *And fm fm*| *Or fm fm*| *Imp fm fm*| *Iff fm fm*| *E fm*| *A fm*
| *Closed nat* | *NClosed nat*

consts *fmsize* :: *fm* ⇒ *nat*
recdef *fmsize measure size*
fmsize (*NOT p*) = *1* + *fmsize p*
fmsize (*And p q*) = *1* + *fmsize p* + *fmsize q*
fmsize (*Or p q*) = *1* + *fmsize p* + *fmsize q*
fmsize (*Imp p q*) = *3* + *fmsize p* + *fmsize q*
fmsize (*Iff p q*) = *3* + *2*∗(*fmsize p* + *fmsize q*)
fmsize (*E p*) = *1* + *fmsize p*
fmsize (*A p*) = *4*+ *fmsize p*
fmsize (*Dvd i t*) = *2*
fmsize (*NDvd i t*) = *2*
fmsize p = *1*

lemma *fmsize-pos*: *fmsize p* > *0*
⟨*proof*⟩

consts *Ifm* ::*bool list* ⇒ *int list* ⇒ *fm* ⇒ *bool*
primrec
Ifm bbs bs T = *True*
Ifm bbs bs F = *False*
Ifm bbs bs (*Lt a*) = (*Inum bs a* < *0*)
Ifm bbs bs (*Gt a*) = (*Inum bs a* > *0*)
Ifm bbs bs (*Le a*) = (*Inum bs a* ≤ *0*)
Ifm bbs bs (*Ge a*) = (*Inum bs a* ≥ *0*)
Ifm bbs bs (*Eq a*) = (*Inum bs a* = *0*)
Ifm bbs bs (*NEq a*) = (*Inum bs a* ≠ *0*)
Ifm bbs bs (*Dvd i b*) = (*i dvd Inum bs b*)
Ifm bbs bs (*NDvd i b*) = (¬(*i dvd Inum bs b*))
Ifm bbs bs (*NOT p*) = (¬ (*Ifm bbs bs p*))
Ifm bbs bs (*And p q*) = (*Ifm bbs bs p* ∧ *Ifm bbs bs q*)
Ifm bbs bs (*Or p q*) = (*Ifm bbs bs p* ∨ *Ifm bbs bs q*)
Ifm bbs bs (*Imp p q*) = ((*Ifm bbs bs p*) ⟶ (*Ifm bbs bs q*))
Ifm bbs bs (*Iff p q*) = (*Ifm bbs bs p* = *Ifm bbs bs q*)
Ifm bbs bs (*E p*) = (∃ *x*. *Ifm bbs* (*x*#*bs*) *p*)
Ifm bbs bs (*A p*) = (∀ *x*. *Ifm bbs* (*x*#*bs*) *p*)

Ifm bbs bs (*Closed n*) = *bbs*!*n*
Ifm bbs bs (*NClosed n*) = (¬ *bbs*!*n*)

consts *prep* :: *fm* ⇒ *fm*
recdef *prep measure fmsize*
prep (*E T*) = *T*
prep (*E F*) = *F*
prep (*E* (*Or p q*)) = *Or* (*prep* (*E p*)) (*prep* (*E q*))
prep (*E* (*Imp p q*)) = *Or* (*prep* (*E* (*NOT p*))) (*prep* (*E q*))
prep (*E* (*Iff p q*)) = *Or* (*prep* (*E* (*And p q*))) (*prep* (*E* (*And* (*NOT p*) (*NOT q*))))
prep (*E* (*NOT* (*And p q*))) = *Or* (*prep* (*E* (*NOT p*))) (*prep* (*E*(*NOT q*)))
prep (*E* (*NOT* (*Imp p q*))) = *prep* (*E* (*And p* (*NOT q*)))
prep (*E* (*NOT* (*Iff p q*))) = *Or* (*prep* (*E* (*And p* (*NOT q*)))) (*prep* (*E*(*And* (*NOT p*) *q*)))
prep (*E p*) = *E* (*prep p*)
prep (*A* (*And p q*)) = *And* (*prep* (*A p*)) (*prep* (*A q*))
prep (*A p*) = *prep* (*NOT* (*E* (*NOT p*)))
prep (*NOT* (*NOT p*)) = *prep p*
prep (*NOT* (*And p q*)) = *Or* (*prep* (*NOT p*)) (*prep* (*NOT q*))
prep (*NOT* (*A p*)) = *prep* (*E* (*NOT p*))
prep (*NOT* (*Or p q*)) = *And* (*prep* (*NOT p*)) (*prep* (*NOT q*))
prep (*NOT* (*Imp p q*)) = *And* (*prep p*) (*prep* (*NOT q*))
prep (*NOT* (*Iff p q*)) = *Or* (*prep* (*And p* (*NOT q*))) (*prep* (*And* (*NOT p*) *q*))
prep (*NOT p*) = *NOT* (*prep p*)
prep (*Or p q*) = *Or* (*prep p*) (*prep q*)
prep (*And p q*) = *And* (*prep p*) (*prep q*)
prep (*Imp p q*) = *prep* (*Or* (*NOT p*) *q*)
prep (*Iff p q*) = *Or* (*prep* (*And p q*)) (*prep* (*And* (*NOT p*) (*NOT q*)))
prep p = *p*
(**hints** *simp add*: *fmsize-pos*)
lemma *prep*: *Ifm bbs bs* (*prep p*) = *Ifm bbs bs p*
⟨*proof*⟩

consts *qfree*:: *fm* ⇒ *bool*
recdef *qfree measure size*
qfree (*E p*) = *False*
qfree (*A p*) = *False*
qfree (*NOT p*) = *qfree p*
qfree (*And p q*) = (*qfree p* ∧ *qfree q*)
qfree (*Or p q*) = (*qfree p* ∧ *qfree q*)
qfree (*Imp p q*) = (*qfree p* ∧ *qfree q*)
qfree (*Iff p q*) = (*qfree p* ∧ *qfree q*)
qfree p = *True*

consts

numbound0:: *num* ⇒ *bool*
bound0:: *fm* ⇒ *bool*
subst0:: *num* ⇒ *fm* ⇒ *fm*
primrec
numbound0 (*C c*) = *True*
numbound0 (*Bound n*) = (*n*>*0*)
numbound0 (*CN n i a*) = (*n* >*0* ∧ *numbound0 a*)
numbound0 (*Neg a*) = *numbound0 a*
numbound0 (*Add a b*) = (*numbound0 a* ∧ *numbound0 b*)
numbound0 (*Sub a b*) = (*numbound0 a* ∧ *numbound0 b*)
numbound0 (*Mul i a*) = *numbound0 a*

lemma *numbound0-I*:
assumes *nb*: *numbound0 a*
shows *Inum* (*b*#*bs*) *a* = *Inum* (*b'*#*bs*) *a*
⟨*proof*⟩

primrec
bound0 T = *True*
bound0 F = *True*
bound0 (*Lt a*) = *numbound0 a*
bound0 (*Le a*) = *numbound0 a*
bound0 (*Gt a*) = *numbound0 a*
bound0 (*Ge a*) = *numbound0 a*
bound0 (*Eq a*) = *numbound0 a*
bound0 (*NEq a*) = *numbound0 a*
bound0 (*Dvd i a*) = *numbound0 a*
bound0 (*NDvd i a*) = *numbound0 a*
bound0 (*NOT p*) = *bound0 p*
bound0 (*And p q*) = (*bound0 p* ∧ *bound0 q*)
bound0 (*Or p q*) = (*bound0 p* ∧ *bound0 q*)
bound0 (*Imp p q*) = ((*bound0 p*) ∧ (*bound0 q*))
bound0 (*Iff p q*) = (*bound0 p* ∧ *bound0 q*)
bound0 (*E p*) = *False*
bound0 (*A p*) = *False*
bound0 (*Closed P*) = *True*
bound0 (*NClosed P*) = *True*
lemma *bound0-I*:
assumes *bp*: *bound0 p*
shows *Ifm bbs* (*b*#*bs*) *p* = *Ifm bbs* (*b'*#*bs*) *p*
⟨*proof*⟩

fun *numsubst0*:: *num* ⇒ *num* ⇒ *num* **where**
numsubst0 t (*C c*) = (*C c*)
| *numsubst0 t* (*Bound n*) = (*if n*=*0 then t else Bound n*)
| *numsubst0 t* (*CN 0 i a*) = *Add* (*Mul i t*) (*numsubst0 t a*)
| *numsubst0 t* (*CN n i a*) = *CN n i* (*numsubst0 t a*)
| *numsubst0 t* (*Neg a*) = *Neg* (*numsubst0 t a*)
| *numsubst0 t* (*Add a b*) = *Add* (*numsubst0 t a*) (*numsubst0 t b*)

| *numsubst0 t* (*Sub a b*) = *Sub* (*numsubst0 t a*) (*numsubst0 t b*)
| *numsubst0 t* (*Mul i a*) = *Mul i* (*numsubst0 t a*)

lemma *numsubst0-I*:
Inum (*b#bs*) (*numsubst0 a t*) = *Inum* ((*Inum* (*b#bs*) *a*)#*bs*) *t*
⟨*proof*⟩

lemma *numsubst0-I′*:
numbound0 a ⟹ *Inum* (*b#bs*) (*numsubst0 a t*) = *Inum* ((*Inum* (*b′#bs*) *a*)#*bs*) *t*
⟨*proof*⟩

primrec
subst0 t T = *T*
subst0 t F = *F*
subst0 t (*Lt a*) = *Lt* (*numsubst0 t a*)
subst0 t (*Le a*) = *Le* (*numsubst0 t a*)
subst0 t (*Gt a*) = *Gt* (*numsubst0 t a*)
subst0 t (*Ge a*) = *Ge* (*numsubst0 t a*)
subst0 t (*Eq a*) = *Eq* (*numsubst0 t a*)
subst0 t (*NEq a*) = *NEq* (*numsubst0 t a*)
subst0 t (*Dvd i a*) = *Dvd i* (*numsubst0 t a*)
subst0 t (*NDvd i a*) = *NDvd i* (*numsubst0 t a*)
subst0 t (*NOT p*) = *NOT* (*subst0 t p*)
subst0 t (*And p q*) = *And* (*subst0 t p*) (*subst0 t q*)
subst0 t (*Or p q*) = *Or* (*subst0 t p*) (*subst0 t q*)
subst0 t (*Imp p q*) = *Imp* (*subst0 t p*) (*subst0 t q*)
subst0 t (*Iff p q*) = *Iff* (*subst0 t p*) (*subst0 t q*)
subst0 t (*Closed P*) = (*Closed P*)
subst0 t (*NClosed P*) = (*NClosed P*)

lemma *subst0-I*: **assumes** *qfp*: *qfree p*
shows *Ifm bbs* (*b#bs*) (*subst0 a p*) = *Ifm bbs* ((*Inum* (*b#bs*) *a*)#*bs*) *p*
⟨*proof*⟩

consts
decrnum:: *num* ⇒ *num*
decr :: *fm* ⇒ *fm*

recdef *decrnum measure size*
decrnum (*Bound n*) = *Bound* (*n* − *1*)
decrnum (*Neg a*) = *Neg* (*decrnum a*)
decrnum (*Add a b*) = *Add* (*decrnum a*) (*decrnum b*)
decrnum (*Sub a b*) = *Sub* (*decrnum a*) (*decrnum b*)
decrnum (*Mul c a*) = *Mul c* (*decrnum a*)
decrnum (*CN n i a*) = (*CN* (*n* − *1*) *i* (*decrnum a*))
decrnum a = *a*

recdef *decr measure size*
decr (*Lt a*) = *Lt* (*decrnum a*)
decr (*Le a*) = *Le* (*decrnum a*)
decr (*Gt a*) = *Gt* (*decrnum a*)
decr (*Ge a*) = *Ge* (*decrnum a*)
decr (*Eq a*) = *Eq* (*decrnum a*)
decr (*NEq a*) = *NEq* (*decrnum a*)
decr (*Dvd i a*) = *Dvd i* (*decrnum a*)
decr (*NDvd i a*) = *NDvd i* (*decrnum a*)
decr (*NOT p*) = *NOT* (*decr p*)
decr (*And p q*) = *And* (*decr p*) (*decr q*)
decr (*Or p q*) = *Or* (*decr p*) (*decr q*)
decr (*Imp p q*) = *Imp* (*decr p*) (*decr q*)
decr (*Iff p q*) = *Iff* (*decr p*) (*decr q*)
decr p = *p*

lemma *decrnum*: **assumes** *nb*: *numbound0 t*
shows *Inum* (*x#bs*) *t* = *Inum bs* (*decrnum t*)
⟨*proof*⟩

lemma *decr*: **assumes** *nb*: *bound0 p*
shows *Ifm bbs* (*x#bs*) *p* = *Ifm bbs bs* (*decr p*)
⟨*proof*⟩

lemma *decr-qf*: *bound0 p* $\Longrightarrow$ *qfree* (*decr p*)
⟨*proof*⟩

consts
isatom :: *fm* $\Rightarrow$ *bool*
recdef *isatom measure size*
isatom T = *True*
isatom F = *True*
isatom (*Lt a*) = *True*
isatom (*Le a*) = *True*
isatom (*Gt a*) = *True*
isatom (*Ge a*) = *True*
isatom (*Eq a*) = *True*
isatom (*NEq a*) = *True*
isatom (*Dvd i b*) = *True*
isatom (*NDvd i b*) = *True*
isatom (*Closed P*) = *True*
isatom (*NClosed P*) = *True*
isatom p = *False*

lemma *numsubst0-numbound0*: **assumes** *nb*: *numbound0 t*
shows *numbound0* (*numsubst0 t a*)
⟨*proof*⟩

lemma *subst0-bound0*: **assumes** *qf*: *qfree p* **and** *nb*: *numbound0 t*

shows *bound0* (*subst0 t p*)
⟨*proof*⟩

lemma *bound0-qf*: *bound0 p* $\Longrightarrow$ *qfree p*
⟨*proof*⟩

constdefs *djf* :: (*'a* $\Rightarrow$ *fm*) $\Rightarrow$ *'a* $\Rightarrow$ *fm* $\Rightarrow$ *fm*
djf f p q $\equiv$ (*if q=T then T else if q=F then f p else*
(*let fp = f p in case fp of T* $\Rightarrow$ *T* | *F* $\Rightarrow$ *q* | - $\Rightarrow$ *Or* (*f p*) *q*))
constdefs *evaldjf* :: (*'a* $\Rightarrow$ *fm*) $\Rightarrow$ *'a list* $\Rightarrow$ *fm*
evaldjf f ps $\equiv$ *foldr* (*djf f*) *ps F*

lemma *djf-Or*: *Ifm bbs bs* (*djf f p q*) = *Ifm bbs bs* (*Or* (*f p*) *q*)
⟨*proof*⟩

lemma *evaldjf-ex*: *Ifm bbs bs* (*evaldjf f ps*) = ($\exists$ *p* $\in$ *set ps*. *Ifm bbs bs* (*f p*))
⟨*proof*⟩

lemma *evaldjf-bound0*:
assumes *nb*: $\forall$ *x*$\in$ *set xs*. *bound0* (*f x*)
shows *bound0* (*evaldjf f xs*)
⟨*proof*⟩

lemma *evaldjf-qf*:
assumes *nb*: $\forall$ *x*$\in$ *set xs*. *qfree* (*f x*)
shows *qfree* (*evaldjf f xs*)
⟨*proof*⟩

consts *disjuncts* :: *fm* $\Rightarrow$ *fm list*
recdef *disjuncts measure size*
disjuncts (*Or p q*) = (*disjuncts p*) @ (*disjuncts q*)
disjuncts F = []
disjuncts p = [*p*]

lemma *disjuncts*: ($\exists$ *q*$\in$ *set* (*disjuncts p*). *Ifm bbs bs q*) = *Ifm bbs bs p*
⟨*proof*⟩

lemma *disjuncts-nb*: *bound0 p* $\Longrightarrow$ $\forall$ *q*$\in$ *set* (*disjuncts p*). *bound0 q*
⟨*proof*⟩

lemma *disjuncts-qf*: *qfree p* $\Longrightarrow$ $\forall$ *q*$\in$ *set* (*disjuncts p*). *qfree q*
⟨*proof*⟩

constdefs *DJ* :: (*fm* $\Rightarrow$ *fm*) $\Rightarrow$ *fm* $\Rightarrow$ *fm*
DJ f p $\equiv$ *evaldjf f* (*disjuncts p*)

lemma *DJ*: **assumes** *fdj*: $\forall$ *p q*. *f* (*Or p q*) = *Or* (*f p*) (*f q*)
and *fF*: *f F* = *F*

shows *Ifm bbs bs* (*DJ f p*) = *Ifm bbs bs* (*f p*)
⟨*proof*⟩

lemma *DJ-qf*: **assumes**
fqf: ∀ *p*. *qfree p* ⟶ *qfree* (*f p*)
shows ∀ *p*. *qfree p* ⟶ *qfree* (*DJ f p*)
⟨*proof*⟩

lemma *DJ-qe*: **assumes** *qe*: ∀ *bs p*. *qfree p* ⟶ *qfree* (*qe p*) ∧ (*Ifm bbs bs* (*qe p*) = *Ifm bbs bs* (*E p*))
shows ∀ *bs p*. *qfree p* ⟶ *qfree* (*DJ qe p*) ∧ (*Ifm bbs bs* ((*DJ qe p*)) = *Ifm bbs bs* (*E p*))
⟨*proof*⟩

consts *bnds*:: *num* ⇒ *nat list*
lex-ns:: *nat list* × *nat list* ⇒ *bool*
recdef *bnds measure size*
bnds (*Bound n*) = [*n*]
bnds (*CN n c a*) = *n*#(*bnds a*)
bnds (*Neg a*) = *bnds a*
bnds (*Add a b*) = (*bnds a*)@(*bnds b*)
bnds (*Sub a b*) = (*bnds a*)@(*bnds b*)
bnds (*Mul i a*) = *bnds a*
bnds a = []
recdef *lex-ns measure* (λ (*xs*,*ys*). *length xs* + *length ys*)
lex-ns ([], *ms*) = *True*
lex-ns (*ns*, []) = *False*
lex-ns (*n*#*ns*, *m*#*ms*) = (*n*<*m* ∨ ((*n* = *m*) ∧ *lex-ns* (*ns*,*ms*)))
constdefs *lex-bnd* :: *num* ⇒ *num* ⇒ *bool*
lex-bnd t s ≡ *lex-ns* (*bnds t*, *bnds s*)

consts
numadd:: *num* × *num* ⇒ *num*
recdef *numadd measure* (λ (*t*,*s*). *num-size t* + *num-size s*)
numadd (*CN n1 c1 r1* ,*CN n2 c2 r2*) =
(*if n1*=*n2 then*
(*let c* = *c1* + *c2*
in (*if c*=*0 then numadd*(*r1*,*r2*) *else CN n1 c* (*numadd* (*r1*,*r2*))))
else if n1 ≤ *n2 then CN n1 c1* (*numadd* (*r1*,*Add* (*Mul c2* (*Bound n2*)) *r2*))
else CN n2 c2 (*numadd* (*Add* (*Mul c1* (*Bound n1*)) *r1*,*r2*)))
numadd (*CN n1 c1 r1*, *t*) = *CN n1 c1* (*numadd* (*r1*, *t*))
numadd (*t*,*CN n2 c2 r2*) = *CN n2 c2* (*numadd* (*t*,*r2*))
numadd (*C b1*, *C b2*) = *C* (*b1*+*b2*)
numadd (*a*,*b*) = *Add a b*

lemma *numadd*: $Inum\ bs\ (numadd\ (t,s)) = Inum\ bs\ (Add\ t\ s)$
⟨*proof*⟩

lemma *numadd-nb*: $[\![\ numbound0\ t\ ;\ numbound0\ s]\!] \Longrightarrow numbound0\ (numadd\ (t,s))$
⟨*proof*⟩

fun
  $nummul :: int \Rightarrow num \Rightarrow num$
where
  $nummul\ i\ (C\ j) = C\ (i * j)$
  $|\ nummul\ i\ (CN\ n\ c\ t) = CN\ n\ (c*i)\ (nummul\ i\ t)$
  $|\ nummul\ i\ t = Mul\ i\ t$

lemma *nummul*: $\bigwedge\ i.\ Inum\ bs\ (nummul\ i\ t) = Inum\ bs\ (Mul\ i\ t)$
⟨*proof*⟩

lemma *nummul-nb*: $\bigwedge\ i.\ numbound0\ t \Longrightarrow numbound0\ (nummul\ i\ t)$
⟨*proof*⟩

constdefs $numneg :: num \Rightarrow num$
  $numneg\ t \equiv nummul\ (-\ 1)\ t$

constdefs $numsub :: num \Rightarrow num \Rightarrow num$
  $numsub\ s\ t \equiv (if\ s = t\ then\ C\ 0\ else\ numadd\ (s,\ numneg\ t))$

lemma *numneg*: $Inum\ bs\ (numneg\ t) = Inum\ bs\ (Neg\ t)$
⟨*proof*⟩

lemma *numneg-nb*: $numbound0\ t \Longrightarrow numbound0\ (numneg\ t)$
⟨*proof*⟩

lemma *numsub*: $Inum\ bs\ (numsub\ a\ b) = Inum\ bs\ (Sub\ a\ b)$
⟨*proof*⟩

lemma *numsub-nb*: $[\![\ numbound0\ t\ ;\ numbound0\ s]\!] \Longrightarrow numbound0\ (numsub\ t\ s)$
⟨*proof*⟩

fun
  $simpnum :: num \Rightarrow num$
where
  $simpnum\ (C\ j) = C\ j$
  $|\ simpnum\ (Bound\ n) = CN\ n\ 1\ (C\ 0)$
  $|\ simpnum\ (Neg\ t) = numneg\ (simpnum\ t)$
  $|\ simpnum\ (Add\ t\ s) = numadd\ (simpnum\ t,\ simpnum\ s)$
  $|\ simpnum\ (Sub\ t\ s) = numsub\ (simpnum\ t)\ (simpnum\ s)$
  $|\ simpnum\ (Mul\ i\ t) = (if\ i = 0\ then\ C\ 0\ else\ nummul\ i\ (simpnum\ t))$
  $|\ simpnum\ t = t$

lemma *simpnum-ci*: *Inum bs* (*simpnum t*) = *Inum bs t*
⟨*proof*⟩

lemma *simpnum-numbound0*:
numbound0 t $\Longrightarrow$ *numbound0* (*simpnum t*)
⟨*proof*⟩

fun
not :: *fm* ⇒ *fm*
where
not (*NOT p*) = *p*
| *not T* = *F*
| *not F* = *T*
| *not p* = *NOT p*
lemma *not*: *Ifm bbs bs* (*not p*) = *Ifm bbs bs* (*NOT p*)
⟨*proof*⟩
lemma *not-qf*: *qfree p* $\Longrightarrow$ *qfree* (*not p*)
⟨*proof*⟩
lemma *not-bn*: *bound0 p* $\Longrightarrow$ *bound0* (*not p*)
⟨*proof*⟩

constdefs *conj* :: *fm* ⇒ *fm* ⇒ *fm*
conj p q ≡ (*if* (*p* = *F* ∨ *q*=*F*) *then F else if p*=*T then q else if q*=*T then p else And p q*)
lemma *conj*: *Ifm bbs bs* (*conj p q*) = *Ifm bbs bs* (*And p q*)
⟨*proof*⟩

lemma *conj-qf*: ⟦*qfree p* ; *qfree q*⟧ $\Longrightarrow$ *qfree* (*conj p q*)
⟨*proof*⟩
lemma *conj-nb*: ⟦*bound0 p* ; *bound0 q*⟧ $\Longrightarrow$ *bound0* (*conj p q*)
⟨*proof*⟩

constdefs *disj* :: *fm* ⇒ *fm* ⇒ *fm*
disj p q ≡ (*if* (*p* = *T* ∨ *q*=*T*) *then T else if p*=*F then q else if q*=*F then p else Or p q*)

lemma *disj*: *Ifm bbs bs* (*disj p q*) = *Ifm bbs bs* (*Or p q*)
⟨*proof*⟩
lemma *disj-qf*: ⟦*qfree p* ; *qfree q*⟧ $\Longrightarrow$ *qfree* (*disj p q*)
⟨*proof*⟩
lemma *disj-nb*: ⟦*bound0 p* ; *bound0 q*⟧ $\Longrightarrow$ *bound0* (*disj p q*)
⟨*proof*⟩

constdefs *imp* :: *fm* ⇒ *fm* ⇒ *fm*
imp p q ≡ (*if* (*p* = *F* ∨ *q*=*T*) *then T else if p*=*T then q else if q*=*F then not p else Imp p q*)
lemma *imp*: *Ifm bbs bs* (*imp p q*) = *Ifm bbs bs* (*Imp p q*)
⟨*proof*⟩
lemma *imp-qf*: ⟦*qfree p* ; *qfree q*⟧ $\Longrightarrow$ *qfree* (*imp p q*)

$\langle proof \rangle$
lemma *imp-nb*: $[\![bound0\ p\ ;\ bound0\ q]\!] \Longrightarrow bound0\ (imp\ p\ q)$
$\langle proof \rangle$

constdefs $iff :: fm \Rightarrow fm \Rightarrow fm$
$iff\ p\ q \equiv (if\ (p = q)\ then\ T\ else\ if\ (p = not\ q \vee not\ p = q)\ then\ F\ else$
$if\ p{=}F\ then\ not\ q\ else\ if\ q{=}F\ then\ not\ p\ else\ if\ p{=}T\ then\ q\ else\ if\ q{=}T\ then$
$p\ else$
$Iff\ p\ q)$
lemma *iff*: $Ifm\ bbs\ bs\ (iff\ p\ q) = Ifm\ bbs\ bs\ (Iff\ p\ q)$
$\langle proof \rangle$
lemma *iff-qf*: $[\![qfree\ p\ ;\ qfree\ q]\!] \Longrightarrow qfree\ (iff\ p\ q)$
$\langle proof \rangle$
lemma *iff-nb*: $[\![bound0\ p\ ;\ bound0\ q]\!] \Longrightarrow bound0\ (iff\ p\ q)$
$\langle proof \rangle$

function (*sequential*)
$simpfm :: fm \Rightarrow fm$
where
$simpfm\ (And\ p\ q) = conj\ (simpfm\ p)\ (simpfm\ q)$
$|\ simpfm\ (Or\ p\ q) = disj\ (simpfm\ p)\ (simpfm\ q)$
$|\ simpfm\ (Imp\ p\ q) = imp\ (simpfm\ p)\ (simpfm\ q)$
$|\ simpfm\ (Iff\ p\ q) = iff\ (simpfm\ p)\ (simpfm\ q)$
$|\ simpfm\ (NOT\ p) = not\ (simpfm\ p)$
$|\ simpfm\ (Lt\ a) = (let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (v < 0)\ then\ T$
$else\ F$
$|\ \text{-} \Rightarrow Lt\ a')$
$|\ simpfm\ (Le\ a) = (let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (v \leq 0)\ then\ T$
$else\ F\ |\ \text{-} \Rightarrow Le\ a')$
$|\ simpfm\ (Gt\ a) = (let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (v > 0)\ then\ T$
$else\ F\ |\ \text{-} \Rightarrow Gt\ a')$
$|\ simpfm\ (Ge\ a) = (let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (v \geq 0)\ then\ T$
$else\ F\ |\ \text{-} \Rightarrow Ge\ a')$
$|\ simpfm\ (Eq\ a) = (let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (v = 0)\ then\ T$
$else\ F\ |\ \text{-} \Rightarrow Eq\ a')$
$|\ simpfm\ (NEq\ a) = (let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (v \neq 0)\ then$
$T\ else\ F\ |\ \text{-} \Rightarrow NEq\ a')$
$|\ simpfm\ (Dvd\ i\ a) = (if\ i{=}0\ then\ simpfm\ (Eq\ a)$
$else\ if\ (abs\ i = 1)\ then\ T$
$else\ let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (i\ dvd\ v)\ then\ T\ else\ F$
$|\ \text{-} \Rightarrow Dvd\ i\ a')$
$|\ simpfm\ (NDvd\ i\ a) = (if\ i{=}0\ then\ simpfm\ (NEq\ a)$
$else\ if\ (abs\ i = 1)\ then\ F$
$else\ let\ a' = simpnum\ a\ in\ case\ a'\ of\ C\ v \Rightarrow if\ (\neg(i\ dvd\ v))\ then\ T\ else$
$F\ |\ \text{-} \Rightarrow NDvd\ i\ a')$
$|\ simpfm\ p = p$
$\langle proof \rangle$
termination $\langle proof \rangle$

lemma *simpfm*: *Ifm bbs bs* (*simpfm p*) = *Ifm bbs bs p*
⟨*proof*⟩

lemma *simpfm-bound0*: *bound0 p* ⟹ *bound0* (*simpfm p*)
⟨*proof*⟩

lemma *simpfm-qf*: *qfree p* ⟹ *qfree* (*simpfm p*)
⟨*proof*⟩

consts *qelim* :: *fm* ⇒ (*fm* ⇒ *fm*) ⇒ *fm*
recdef *qelim measure fmsize*
  qelim (*E p*) = (λ *qe*. *DJ qe* (*qelim p qe*))
  qelim (*A p*) = (λ *qe*. *not* (*qe* ((*qelim* (*NOT p*) *qe*))))
  qelim (*NOT p*) = (λ *qe*. *not* (*qelim p qe*))
  qelim (*And p q*) = (λ *qe*. *conj* (*qelim p qe*) (*qelim q qe*))
  qelim (*Or p q*) = (λ *qe*. *disj* (*qelim p qe*) (*qelim q qe*))
  qelim (*Imp p q*) = (λ *qe*. *imp* (*qelim p qe*) (*qelim q qe*))
  qelim (*Iff p q*) = (λ *qe*. *iff* (*qelim p qe*) (*qelim q qe*))
  qelim p = (λ *y*. *simpfm p*)

lemma *qelim-ci*:
  assumes *qe-inv*: ∀ *bs p*. *qfree p* ⟶ *qfree* (*qe p*) ∧ (*Ifm bbs bs* (*qe p*) = *Ifm bbs bs* (*E p*))
  shows ⋀ *bs*. *qfree* (*qelim p qe*) ∧ (*Ifm bbs bs* (*qelim p qe*) = *Ifm bbs bs p*)
⟨*proof*⟩

fun
  zsplit0 :: *num* ⇒ *int* × *num*
where
  zsplit0 (*C c*) = (*0*,*C c*)
  | *zsplit0* (*Bound n*) = (*if n*=*0 then* (*1*, *C 0*) *else* (*0*,*Bound n*))
  | *zsplit0* (*CN n i a*) =
      (*let* (*i'*,*a'*) = *zsplit0 a*
       in if n=*0 then* (*i*+*i'*, *a'*) *else* (*i'*,*CN n i a'*))
  | *zsplit0* (*Neg a*) = (*let* (*i'*,*a'*) = *zsplit0 a in* (−*i'*, *Neg a'*))
  | *zsplit0* (*Add a b*) = (*let* (*ia*,*a'*) = *zsplit0 a* ;
                            (*ib*,*b'*) = *zsplit0 b*
                            in (*ia*+*ib*, *Add a' b'*))
  | *zsplit0* (*Sub a b*) = (*let* (*ia*,*a'*) = *zsplit0 a* ;
                            (*ib*,*b'*) = *zsplit0 b*
                            in (*ia*−*ib*, *Sub a' b'*))
  | *zsplit0* (*Mul i a*) = (*let* (*i'*,*a'*) = *zsplit0 a in* (*i*∗*i'*, *Mul i a'*))

lemma *zsplit0-I*:
  shows ⋀ *n a*. *zsplit0 t* = (*n*,*a*) ⟹ (*Inum* ((*x*::*int*) #*bs*) (*CN 0 n a*) = *Inum*

$(x \# bs)\ t) \wedge numbound0\ a$
(**is** $\bigwedge n\ a.\ ?S\ t = (n,a) \Longrightarrow (?I\ x\ (CN\ 0\ n\ a) = ?I\ x\ t) \wedge ?N\ a$)
⟨*proof*⟩

consts
iszlfm :: *fm* ⇒ *bool*
recdef *iszlfm measure size*
iszlfm (*And p q*) = (*iszlfm p* ∧ *iszlfm q*)
iszlfm (*Or p q*) = (*iszlfm p* ∧ *iszlfm q*)
iszlfm (*Eq* (*CN 0 c e*)) = (*c*>*0* ∧ *numbound0 e*)
iszlfm (*NEq* (*CN 0 c e*)) = (*c*>*0* ∧ *numbound0 e*)
iszlfm (*Lt* (*CN 0 c e*)) = (*c*>*0* ∧ *numbound0 e*)
iszlfm (*Le* (*CN 0 c e*)) = (*c*>*0* ∧ *numbound0 e*)
iszlfm (*Gt* (*CN 0 c e*)) = (*c*>*0* ∧ *numbound0 e*)
iszlfm (*Ge* (*CN 0 c e*)) = (*c*>*0* ∧ *numbound0 e*)
iszlfm (*Dvd i* (*CN 0 c e*)) =
(*c*>*0* ∧ *i*>*0* ∧ *numbound0 e*)
iszlfm (*NDvd i* (*CN 0 c e*))=
(*c*>*0* ∧ *i*>*0* ∧ *numbound0 e*)
iszlfm p = (*isatom p* ∧ (*bound0 p*))

lemma *zlin-qfree*: *iszlfm p* ⟹ *qfree p*
⟨*proof*⟩

consts
zlfm :: *fm* ⇒ *fm*
recdef *zlfm measure fmsize*
zlfm (*And p q*) = *And* (*zlfm p*) (*zlfm q*)
zlfm (*Or p q*) = *Or* (*zlfm p*) (*zlfm q*)
zlfm (*Imp p q*) = *Or* (*zlfm* (*NOT p*)) (*zlfm q*)
zlfm (*Iff p q*) = *Or* (*And* (*zlfm p*) (*zlfm q*)) (*And* (*zlfm* (*NOT p*)) (*zlfm* (*NOT q*)))
zlfm (*Lt a*) = (*let* (*c*,*r*) = *zsplit0 a in*
if c=*0 then Lt r else*
if c>*0 then* (*Lt* (*CN 0 c r*)) *else* (*Gt* (*CN 0* (− *c*) (*Neg r*))))
zlfm (*Le a*) = (*let* (*c*,*r*) = *zsplit0 a in*
if c=*0 then Le r else*
if c>*0 then* (*Le* (*CN 0 c r*)) *else* (*Ge* (*CN 0* (− *c*) (*Neg r*))))
zlfm (*Gt a*) = (*let* (*c*,*r*) = *zsplit0 a in*
if c=*0 then Gt r else*
if c>*0 then* (*Gt* (*CN 0 c r*)) *else* (*Lt* (*CN 0* (− *c*) (*Neg r*))))
zlfm (*Ge a*) = (*let* (*c*,*r*) = *zsplit0 a in*
if c=*0 then Ge r else*
if c>*0 then* (*Ge* (*CN 0 c r*)) *else* (*Le* (*CN 0* (− *c*) (*Neg r*))))
zlfm (*Eq a*) = (*let* (*c*,*r*) = *zsplit0 a in*
if c=*0 then Eq r else*
if c>*0 then* (*Eq* (*CN 0 c r*)) *else* (*Eq* (*CN 0* (− *c*) (*Neg r*))))
zlfm (*NEq a*) = (*let* (*c*,*r*) = *zsplit0 a in*
if c=*0 then NEq r else*

if c>0 then (NEq (CN 0 c r)) else (NEq (CN 0 (− c) (Neg r))))
zlfm (Dvd i a) = (if i=0 then zlfm (Eq a)
else (let (c,r) = zsplit0 a in
if c=0 then (Dvd (abs i) r) else
if c>0 then (Dvd (abs i) (CN 0 c r))
else (Dvd (abs i) (CN 0 (− c) (Neg r)))))
zlfm (NDvd i a) = (if i=0 then zlfm (NEq a)
else (let (c,r) = zsplit0 a in
if c=0 then (NDvd (abs i) r) else
if c>0 then (NDvd (abs i) (CN 0 c r))
else (NDvd (abs i) (CN 0 (− c) (Neg r)))))
zlfm (NOT (And p q)) = Or (zlfm (NOT p)) (zlfm (NOT q))
zlfm (NOT (Or p q)) = And (zlfm (NOT p)) (zlfm (NOT q))
zlfm (NOT (Imp p q)) = And (zlfm p) (zlfm (NOT q))
zlfm (NOT (Iff p q)) = Or (And(zlfm p) (zlfm(NOT q))) (And (zlfm(NOT p)) (zlfm q))
zlfm (NOT (NOT p)) = zlfm p
zlfm (NOT T) = F
zlfm (NOT F) = T
zlfm (NOT (Lt a)) = zlfm (Ge a)
zlfm (NOT (Le a)) = zlfm (Gt a)
zlfm (NOT (Gt a)) = zlfm (Le a)
zlfm (NOT (Ge a)) = zlfm (Lt a)
zlfm (NOT (Eq a)) = zlfm (NEq a)
zlfm (NOT (NEq a)) = zlfm (Eq a)
zlfm (NOT (Dvd i a)) = zlfm (NDvd i a)
zlfm (NOT (NDvd i a)) = zlfm (Dvd i a)
zlfm (NOT (Closed P)) = NClosed P
zlfm (NOT (NClosed P)) = Closed P
zlfm p = p (**hints** *simp add*: *fmsize-pos*)

lemma *zlfm-I*:
assumes *qfp*: *qfree p*
shows (*Ifm bbs (i#bs) (zlfm p) = Ifm bbs (i# bs) p*) ∧ *iszlfm (zlfm p)*
(**is** (*?I (?l p) = ?I p*) ∧ *?L (?l p)*)
⟨*proof*⟩

consts
plusinf:: *fm* ⇒ *fm*
minusinf:: *fm* ⇒ *fm*
δ :: *fm* ⇒ *int*
dδ :: *fm* ⇒ *int* ⇒ *bool*

recdef *minusinf measure size*
minusinf (And p q) = And (minusinf p) (minusinf q)
minusinf (Or p q) = Or (minusinf p) (minusinf q)
minusinf (Eq (CN 0 c e)) = F
minusinf (NEq (CN 0 c e)) = T
minusinf (Lt (CN 0 c e)) = T

minusinf (*Le* (*CN 0 c e*)) = *T*
minusinf (*Gt* (*CN 0 c e*)) = *F*
minusinf (*Ge* (*CN 0 c e*)) = *F*
minusinf p = *p*

lemma *minusinf-qfree*: *qfree p* $\Longrightarrow$ *qfree* (*minusinf p*)
⟨*proof*⟩

recdef *plusinf measure size*
plusinf (*And p q*) = *And* (*plusinf p*) (*plusinf q*)
plusinf (*Or p q*) = *Or* (*plusinf p*) (*plusinf q*)
plusinf (*Eq* (*CN 0 c e*)) = *F*
plusinf (*NEq* (*CN 0 c e*)) = *T*
plusinf (*Lt* (*CN 0 c e*)) = *F*
plusinf (*Le* (*CN 0 c e*)) = *F*
plusinf (*Gt* (*CN 0 c e*)) = *T*
plusinf (*Ge* (*CN 0 c e*)) = *T*
plusinf p = *p*

recdef δ *measure size*
δ (*And p q*) = *ilcm* (δ *p*) (δ *q*)
δ (*Or p q*) = *ilcm* (δ *p*) (δ *q*)
δ (*Dvd i* (*CN 0 c e*)) = *i*
δ (*NDvd i* (*CN 0 c e*)) = *i*
δ *p* = *1*

recdef *d*δ *measure size*
dδ (*And p q*) = (λ *d*. *d*δ *p d* $\wedge$ *d*δ *q d*)
dδ (*Or p q*) = (λ *d*. *d*δ *p d* $\wedge$ *d*δ *q d*)
dδ (*Dvd i* (*CN 0 c e*)) = (λ *d*. *i dvd d*)
dδ (*NDvd i* (*CN 0 c e*)) = (λ *d*. *i dvd d*)
dδ *p* = (λ *d*. *True*)

lemma *delta-mono*:
assumes *lin*: *iszlfm p*
and *d*: *d dvd d'*
and *ad*: *d*δ *p d*
shows *d*δ *p d'*
⟨*proof*⟩

lemma δ : **assumes** *lin*:*iszlfm p*
shows *d*δ *p* (δ *p*) $\wedge$ δ *p* >*0*
⟨*proof*⟩

consts
aβ :: *fm* $\Rightarrow$ *int* $\Rightarrow$ *fm*
dβ :: *fm* $\Rightarrow$ *int* $\Rightarrow$ *bool*
ζ :: *fm* $\Rightarrow$ *int*

β :: *fm* $\Rightarrow$ *num list*
α :: *fm* $\Rightarrow$ *num list*

recdef $a\beta$ *measure size*
$a\beta$ (*And p q*) = (λ *k*. *And* ($a\beta$ *p k*) ($a\beta$ *q k*))
$a\beta$ (*Or p q*) = (λ *k*. *Or* ($a\beta$ *p k*) ($a\beta$ *q k*))
$a\beta$ (*Eq* (*CN 0 c e*)) = (λ *k*. *Eq* (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*NEq* (*CN 0 c e*)) = (λ *k*. *NEq* (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*Lt* (*CN 0 c e*)) = (λ *k*. *Lt* (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*Le* (*CN 0 c e*)) = (λ *k*. *Le* (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*Gt* (*CN 0 c e*)) = (λ *k*. *Gt* (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*Ge* (*CN 0 c e*)) = (λ *k*. *Ge* (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*Dvd i* (*CN 0 c e*)) =(λ *k*. *Dvd* ((*k div c*)$*$*i*) (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ (*NDvd i* (*CN 0 c e*))=(λ *k*. *NDvd* ((*k div c*)$*$*i*) (*CN 0 1* (*Mul* (*k div c*) *e*)))
$a\beta$ *p* = (λ *k*. *p*)

recdef $d\beta$ *measure size*
$d\beta$ (*And p q*) = (λ *k*. ($d\beta$ *p k*) $\wedge$ ($d\beta$ *q k*))
$d\beta$ (*Or p q*) = (λ *k*. ($d\beta$ *p k*) $\wedge$ ($d\beta$ *q k*))
$d\beta$ (*Eq* (*CN 0 c e*)) = (λ *k*. *c dvd k*)
$d\beta$ (*NEq* (*CN 0 c e*)) = (λ *k*. *c dvd k*)
$d\beta$ (*Lt* (*CN 0 c e*)) = (λ *k*. *c dvd k*)
$d\beta$ (*Le* (*CN 0 c e*)) = (λ *k*. *c dvd k*)
$d\beta$ (*Gt* (*CN 0 c e*)) = (λ *k*. *c dvd k*)
$d\beta$ (*Ge* (*CN 0 c e*)) = (λ *k*. *c dvd k*)
$d\beta$ (*Dvd i* (*CN 0 c e*)) =(λ *k*. *c dvd k*)
$d\beta$ (*NDvd i* (*CN 0 c e*))=(λ *k*. *c dvd k*)
$d\beta$ *p* = (λ *k*. *True*)

recdef ζ *measure size*
ζ (*And p q*) = *ilcm* (ζ *p*) (ζ *q*)
ζ (*Or p q*) = *ilcm* (ζ *p*) (ζ *q*)
ζ (*Eq* (*CN 0 c e*)) = *c*
ζ (*NEq* (*CN 0 c e*)) = *c*
ζ (*Lt* (*CN 0 c e*)) = *c*
ζ (*Le* (*CN 0 c e*)) = *c*
ζ (*Gt* (*CN 0 c e*)) = *c*
ζ (*Ge* (*CN 0 c e*)) = *c*
ζ (*Dvd i* (*CN 0 c e*)) = *c*
ζ (*NDvd i* (*CN 0 c e*))= *c*
ζ *p* = *1*

recdef β *measure size*
β (*And p q*) = (β *p* @ β *q*)
β (*Or p q*) = (β *p* @ β *q*)
β (*Eq* (*CN 0 c e*)) = [*Sub* (*C* $-$*1*) *e*]
β (*NEq* (*CN 0 c e*)) = [*Neg e*]
β (*Lt* (*CN 0 c e*)) = []
β (*Le* (*CN 0 c e*)) = []

β (*Gt* (*CN 0 c e*)) = [*Neg e*]
β (*Ge* (*CN 0 c e*)) = [*Sub* (*C* −*1*) *e*]
β *p* = []

recdef α *measure size*
α (*And p q*) = (α *p* @ α *q*)
α (*Or p q*) = (α *p* @ α *q*)
α (*Eq* (*CN 0 c e*)) = [*Add* (*C* −*1*) *e*]
α (*NEq* (*CN 0 c e*)) = [*e*]
α (*Lt* (*CN 0 c e*)) = [*e*]
α (*Le* (*CN 0 c e*)) = [*Add* (*C* −*1*) *e*]
α (*Gt* (*CN 0 c e*)) = []
α (*Ge* (*CN 0 c e*)) = []
α *p* = []
consts *mirror* :: *fm* ⇒ *fm*
recdef *mirror measure size*
mirror (*And p q*) = *And* (*mirror p*) (*mirror q*)
mirror (*Or p q*) = *Or* (*mirror p*) (*mirror q*)
mirror (*Eq* (*CN 0 c e*)) = *Eq* (*CN 0 c* (*Neg e*))
mirror (*NEq* (*CN 0 c e*)) = *NEq* (*CN 0 c* (*Neg e*))
mirror (*Lt* (*CN 0 c e*)) = *Gt* (*CN 0 c* (*Neg e*))
mirror (*Le* (*CN 0 c e*)) = *Ge* (*CN 0 c* (*Neg e*))
mirror (*Gt* (*CN 0 c e*)) = *Lt* (*CN 0 c* (*Neg e*))
mirror (*Ge* (*CN 0 c e*)) = *Le* (*CN 0 c* (*Neg e*))
mirror (*Dvd i* (*CN 0 c e*)) = *Dvd i* (*CN 0 c* (*Neg e*))
mirror (*NDvd i* (*CN 0 c e*)) = *NDvd i* (*CN 0 c* (*Neg e*))
mirror p = *p*

lemma *dvd1-eq1*: *x* >*0* ⟹ (*x*::*int*) *dvd 1* = (*x* = *1*)
⟨*proof*⟩

lemma *minusinf-inf*:
assumes *linp*: *iszlfm p*
and *u*: *d*β *p 1*
shows ∃ (*z*::*int*). ∀ *x* < *z*. *Ifm bbs* (*x*#*bs*) (*minusinf p*) = *Ifm bbs* (*x*#*bs*) *p*
(**is** *?P p* **is** ∃ (*z*::*int*). ∀ *x* < *z*. *?I x* (*?M p*) = *?I x p*)
⟨*proof*⟩

lemma *minusinf-repeats*:
assumes *d*: *d*δ *p d* **and** *linp*: *iszlfm p*
shows *Ifm bbs* ((*x* − *k*∗*d*)#*bs*) (*minusinf p*) = *Ifm bbs* (*x* #*bs*) (*minusinf p*)
⟨*proof*⟩

lemma *minusinf-ex*:
assumes *lin*: *iszlfm p* **and** *u*: *d*β *p 1*
and *exmi*: ∃ (*x*::*int*). *Ifm bbs* (*x*#*bs*) (*minusinf p*) (**is** ∃ *x*. *?P1 x*)
shows ∃ (*x*::*int*). *Ifm bbs* (*x*#*bs*) *p* (**is** ∃ *x*. *?P x*)
⟨*proof*⟩

lemma *minusinf-bex*:
assumes *lin*: *iszlfm p*
shows $(\exists\ (x{::}int).\ Ifm\ bbs\ (x\#bs)\ (minusinf\ p)) =$
$(\exists\ (x{::}int) \in \{1..\delta\ p\}.\ Ifm\ bbs\ (x\#bs)\ (minusinf\ p))$
(**is** $(\exists\ x.\ ?P\ x) = -$)
⟨*proof*⟩

lemma *mirrorαβ*:
assumes *lp*: *iszlfm p*
shows $(Inum\ (i\#bs))$ ‘ $set\ (\alpha\ p) = (Inum\ (i\#bs))$ ‘ $set\ (\beta\ (mirror\ p))$
⟨*proof*⟩

lemma *mirror*:
assumes *lp*: *iszlfm p*
shows $Ifm\ bbs\ (x\#bs)\ (mirror\ p) = Ifm\ bbs\ ((-\ x)\#bs)\ p$
⟨*proof*⟩

lemma *mirror-l*: $iszlfm\ p \wedge d\beta\ p\ 1$
$\Longrightarrow iszlfm\ (mirror\ p) \wedge d\beta\ (mirror\ p)\ 1$
⟨*proof*⟩

lemma *mirror-δ*: $iszlfm\ p \Longrightarrow \delta\ (mirror\ p) = \delta\ p$
⟨*proof*⟩

lemma *β-numbound0*: **assumes** *lp*: *iszlfm p*
shows $\forall\ b \in set\ (\beta\ p).\ numbound0\ b$
⟨*proof*⟩

lemma *dβ-mono*:
assumes *linp*: *iszlfm p*
and *dr*: $d\beta\ p\ l$
and *d*: $l\ dvd\ l'$
shows $d\beta\ p\ l'$
⟨*proof*⟩

lemma *α-l*: **assumes** *lp*: *iszlfm p*
shows $\forall\ b \in set\ (\alpha\ p).\ numbound0\ b$
⟨*proof*⟩

lemma *ζ*:
assumes *linp*: *iszlfm p*
shows $\zeta\ p > 0 \wedge d\beta\ p\ (\zeta\ p)$
⟨*proof*⟩

lemma *aβ*: **assumes** *linp*: *iszlfm p* **and** *d*: $d\beta\ p\ l$ **and** *lp*: $l > 0$
shows $iszlfm\ (a\beta\ p\ l) \wedge d\beta\ (a\beta\ p\ l)\ 1 \wedge (Ifm\ bbs\ (l{*}x\ \#bs)\ (a\beta\ p\ l) = Ifm\ bbs$

$(x\#bs)\ p)$
$\langle proof \rangle$

lemma $a\beta$-ex: **assumes** $linp$: $iszlfm\ p$ **and** d: $d\beta\ p\ l$ **and** lp: $l>0$
shows $(\exists\ x.\ l\ dvd\ x \wedge Ifm\ bbs\ (x\ \#bs)\ (a\beta\ p\ l)) = (\exists\ (x{::}int).\ Ifm\ bbs\ (x\#bs)\ p)$
(**is** $(\exists\ x.\ l\ dvd\ x \wedge ?P\ x) = (\exists\ x.\ ?P'\ x)$)
$\langle proof \rangle$

lemma β:
assumes lp: $iszlfm\ p$
and u: $d\beta\ p\ 1$
and d: $d\delta\ p\ d$
and dp: $d > 0$
and nob: $\neg(\exists(j{::}int) \in \{1\ ..\ d\}.\ \exists\ b\in (Inum\ (a\#bs))\ \text{`}\ set(\beta\ p).\ x = b + j)$
and p: $Ifm\ bbs\ (x\#bs)\ p$ (**is** $?P\ x$)
shows $?P\ (x - d)$
$\langle proof \rangle$

lemma β':
assumes lp: $iszlfm\ p$
and u: $d\beta\ p\ 1$
and d: $d\delta\ p\ d$
and dp: $d > 0$
shows $\forall\ x.\ \neg(\exists(j{::}int) \in \{1\ ..\ d\}.\ \exists\ b\in set(\beta\ p).\ Ifm\ bbs\ ((Inum\ (a\#bs)\ b + j)\ \#bs)\ p) \longrightarrow Ifm\ bbs\ (x\#bs)\ p \longrightarrow Ifm\ bbs\ ((x - d)\#bs)\ p$ (**is** $\forall\ x.\ ?b \longrightarrow ?P\ x \longrightarrow ?P\ (x - d)$)
$\langle proof \rangle$
lemma $cpmi$-eq: $0 < D \Longrightarrow (EX\ z{::}int.\ ALL\ x.\ x < z\ {-}{-}{>}\ (P\ x = P1\ x))$
$=={>}\ ALL\ x.\tilde{}(EX\ (j{::}int) : \{1..D\}.\ EX\ (b{::}int) : B.\ P(b{+}j))\ {-}{-}{>}\ P\ (x)\ {-}{-}{>}\ P\ (x - D)$
$=={>}\ (ALL\ (x{::}int).\ ALL\ (k{::}int).\ ((P1\ x){=}\ (P1\ (x{-}k{*}D))))$
$=={>}\ (EX\ (x{::}int).\ P(x)) = ((EX\ (j{::}int) : \{1..D\}\ .\ (P1(j)))\ |\ (EX\ (j{::}int) : \{1..D\}.\ EX\ (b{::}int) : B.\ P\ (b{+}j)))$
$\langle proof \rangle$

theorem cp-thm:
assumes lp: $iszlfm\ p$
and u: $d\beta\ p\ 1$
and d: $d\delta\ p\ d$
and dp: $d > 0$
shows $(\exists\ (x{::}int).\ Ifm\ bbs\ (x\ \#bs)\ p) = (\exists\ j\in \{1..\ d\}.\ Ifm\ bbs\ (j\ \#bs)\ (minusinf\ p) \vee (\exists\ b \in set\ (\beta\ p).\ Ifm\ bbs\ ((Inum\ (i\#bs)\ b + j)\ \#bs)\ p))$
(**is** $(\exists\ (x{::}int).\ ?P\ (x)) = (\exists\ j\in\ ?D.\ ?M\ j \vee (\exists\ b\in\ ?B.\ ?P\ (?I\ b + j))))$
$\langle proof \rangle$

lemma $mirror$-ex:
assumes lp: $iszlfm\ p$

shows ($\exists$ *x*. *Ifm bbs* (*x#bs*) (*mirror p*)) = ($\exists$ *x*. *Ifm bbs* (*x#bs*) *p*)
(**is** ($\exists$ *x*. *?I x ?mp*) = ($\exists$ *x*. *?I x p*))
⟨*proof*⟩

lemma *cp-thm'*:
assumes *lp*: *iszlfm p*
and *up*: *dβ p 1* **and** *dd*: *dδ p d* **and** *dp*: *d* > *0*
shows ($\exists$ *x*. *Ifm bbs* (*x#bs*) *p*) = (($\exists$ *j*∈ {*1* .. *d*}. *Ifm bbs* (*j#bs*) (*minusinf p*)) ∨ ($\exists$ *j*∈ {*1*.. *d*}. $\exists$ *b*∈ (*Inum* (*i#bs*)) ' *set* (*β p*). *Ifm bbs* ((*b*+*j*)*#bs*) *p*))
⟨*proof*⟩

constdefs *unit*:: *fm* ⇒ *fm* × *num list* × *int*
unit p ≡ (*let p'* = *zlfm p* ; *l* = *ζ p'* ; *q* = *And* (*Dvd l* (*CN 0 1* (*C 0*))) (*aβ p' l*); *d* = *δ q*;
B = *remdups* (*map simpnum* (*β q*)) ; *a* = *remdups* (*map simpnum* (*α q*))
in if length B ≤ *length a then* (*q*,*B*,*d*) *else* (*mirror q*, *a*,*d*))

lemma *unit*: **assumes** *qf*: *qfree p*
shows $\bigwedge$ *q B d*. *unit p* = (*q*,*B*,*d*) ⟹ (($\exists$ *x*. *Ifm bbs* (*x#bs*) *p*) = ($\exists$ *x*. *Ifm bbs* (*x#bs*) *q*)) ∧ (*Inum* (*i#bs*)) ' *set B* = (*Inum* (*i#bs*)) ' *set* (*β q*) ∧ *dβ q 1* ∧ *dδ q d* ∧ *d* >*0* ∧ *iszlfm q* ∧ (∀ *b*∈ *set B*. *numbound0 b*)
⟨*proof*⟩

constdefs *cooper* :: *fm* ⇒ *fm*
cooper p ≡
(*let* (*q*,*B*,*d*) = *unit p*; *js* = *iupt 1 d*;
mq = *simpfm* (*minusinf q*);
md = *evaldjf* (λ *j*. *simpfm* (*subst0* (*C j*) *mq*)) *js*
in if md = *T then T else*
(*let qd* = *evaldjf* (λ (*b*,*j*). *simpfm* (*subst0* (*Add b* (*C j*)) *q*))
[(*b*,*j*). *b*←*B*,*j*←*js*]
in decr (*disj md qd*)))
lemma *cooper*: **assumes** *qf*: *qfree p*
shows (($\exists$ *x*. *Ifm bbs* (*x#bs*) *p*) = (*Ifm bbs bs* (*cooper p*))) ∧ *qfree* (*cooper p*)
(**is** (*?lhs* = *?rhs*) ∧ -)
⟨*proof*⟩

constdefs *pa*:: *fm* ⇒ *fm*
pa ≡ (λ *p*. *qelim* (*prep p*) *cooper*)

theorem *mirqe*: (*Ifm bbs bs* (*pa p*) = *Ifm bbs bs p*) ∧ *qfree* (*pa p*)
⟨*proof*⟩

definition
cooper-test :: *unit* ⇒ *fm*
where

cooper-test u = *pa* (*E* (*A* (*Imp* (*Ge* (*Sub* (*Bound 0*) (*Bound 1*)))
(*E* (*E* (*Eq* (*Sub* (*Add* (*Mul 3* (*Bound 1*)) (*Mul 5* (*Bound 0*)))
(*Bound 2*))))))))

code-reserved *SML oo*
export-code *pa cooper-test* **in** *SML* **module-name** *GeneratedCooper*

⟨*ML*⟩

lemma ∃ (*j*::*int*). ∀ *x*≥*j*. (∃ *a b*. *x* = *3*∗*a*+*5*∗*b*)
⟨*proof*⟩

lemma *ALL* (*x*::*int*) >=*8*. *EX i j*. *5*∗*i* + *3*∗*j* = *x* ⟨*proof*⟩
theorem (∀(*y*::*int*). *3 dvd y*) ==> ∀(*x*::*int*). *b* < *x* −−> *a* ≤ *x*
⟨*proof*⟩

theorem !! (*y*::*int*) (*z*::*int*) (*n*::*int*). *3 dvd z* ==> *2 dvd* (*y*::*int*) ==>
(∃(*x*::*int*). *2*∗*x* = *y*) & (∃(*k*::*int*). *3*∗*k* = *z*)
⟨*proof*⟩

theorem !! (*y*::*int*) (*z*::*int*) *n*. *Suc*(*n*::*nat*) < *6* ==> *3 dvd z* ==>
2 dvd (*y*::*int*) ==> (∃(*x*::*int*). *2*∗*x* = *y*) & (∃(*k*::*int*). *3*∗*k* = *z*)
⟨*proof*⟩

theorem ∀(*x*::*nat*). ∃(*y*::*nat*). (*0*::*nat*) ≤ *5* −−> *y* = *5* + *x*
⟨*proof*⟩

lemma *ALL* (*x*::*int*) >=*8*. *EX i j*. *5*∗*i* + *3*∗*j* = *x* ⟨*proof*⟩
lemma *ALL* (*y*::*int*) (*z*::*int*) (*n*::*int*). *3 dvd z* −−> *2 dvd* (*y*::*int*) −−> (*EX* (*x*::*int*). *2*∗*x* = *y*) & (*EX* (*k*::*int*). *3*∗*k* = *z*) ⟨*proof*⟩
lemma *ALL*(*x*::*int*) *y*. *x* < *y* −−> *2* ∗ *x* + *1* < *2* ∗ *y* ⟨*proof*⟩
lemma *ALL*(*x*::*int*) *y*. *2* ∗ *x* + *1* ~= *2* ∗ *y* ⟨*proof*⟩
lemma *EX*(*x*::*int*) *y*. *0* < *x* & *0* <= *y* & *3* ∗ *x* − *5* ∗ *y* = *1* ⟨*proof*⟩
lemma ~ (*EX*(*x*::*int*) (*y*::*int*) (*z*::*int*). *4*∗*x* + (−*6*::*int*)∗*y* = *1*) ⟨*proof*⟩
lemma *ALL*(*x*::*int*). (*2 dvd x*) −−> (*EX*(*y*::*int*). *x* = *2*∗*y*) ⟨*proof*⟩
lemma *ALL*(*x*::*int*). (*2 dvd x*) −−> (*EX*(*y*::*int*). *x* = *2*∗*y*) ⟨*proof*⟩
lemma *ALL*(*x*::*int*). (*2 dvd x*) = (*EX*(*y*::*int*). *x* = *2*∗*y*) ⟨*proof*⟩
lemma *ALL*(*x*::*int*). ((*2 dvd x*) = (*ALL*(*y*::*int*). *x* ~= *2*∗*y* + *1*)) ⟨*proof*⟩
lemma ~ (*ALL*(*x*::*int*). ((*2 dvd x*) = (*ALL*(*y*::*int*). *x* ~= *2*∗*y*+*1*) | (*EX*(*q*::*int*) (*u*::*int*) *i*. *3*∗*i* + *2*∗*q* − *u* < *17*) −−> *0* < *x* | ((~ *3 dvd x*) &(*x* + *8* = *0*))))
⟨*proof*⟩
lemma ~ (*ALL*(*i*::*int*). *4* <= *i* −−> (*EX x y*. *0* <= *x* & *0* <= *y* & *3* ∗ *x* + *5* ∗ *y* = *i*))
⟨*proof*⟩
lemma *EX j*. *ALL* (*x*::*int*) >= *j*. *EX i j*. *5*∗*i* + *3*∗*j* = *x* ⟨*proof*⟩

theorem (∀(*y*::*int*). *3 dvd y*) ==> ∀(*x*::*int*). *b* < *x* −−> *a* ≤ *x*

$\langle proof \rangle$

theorem !! $(y::int)$ $(z::int)$ $(n::int)$. 3 dvd z ==> 2 dvd $(y::int)$ ==>
$(\exists (x::int).\ 2*x = y)$ & $(\exists (k::int).\ 3*k = z)$
$\langle proof \rangle$

theorem !! $(y::int)$ $(z::int)$ n. $Suc(n::nat) < 6$ ==> 3 dvd z ==>
2 dvd $(y::int)$ ==> $(\exists (x::int).\ 2*x = y)$ & $(\exists (k::int).\ 3*k = z)$
$\langle proof \rangle$

theorem $\forall (x::nat).\ \exists (y::nat).\ (0::nat) \leq 5$ --> $y = 5 + x$
$\langle proof \rangle$

theorem $\forall (x::nat).\ \exists (y::nat).\ y = 5 + x \mid x\ div\ 6 + 1 = 2$
$\langle proof \rangle$

theorem $\exists (x::int).\ 0 < x$
$\langle proof \rangle$

theorem $\forall (x::int)\ y.\ x < y$ --> $2 * x + 1 < 2 * y$
$\langle proof \rangle$

theorem $\forall (x::int)\ y.\ 2 * x + 1 \neq 2 * y$
$\langle proof \rangle$

theorem $\exists (x::int)\ y.\ 0 < x$ & $0 \leq y$ & $3 * x - 5 * y = 1$
$\langle proof \rangle$

theorem ~ $(\exists (x::int)\ (y::int)\ (z::int).\ 4*x + (-6::int)*y = 1)$
$\langle proof \rangle$

theorem ~ $(\exists (x::int).\ False)$
$\langle proof \rangle$

theorem $\forall (x::int).\ (2\ dvd\ x)$ --> $(\exists (y::int).\ x = 2*y)$
$\langle proof \rangle$

theorem $\forall (x::int).\ (2\ dvd\ x)$ --> $(\exists (y::int).\ x = 2*y)$
$\langle proof \rangle$

theorem $\forall (x::int).\ (2\ dvd\ x) = (\exists (y::int).\ x = 2*y)$
$\langle proof \rangle$

theorem $\forall (x::int).\ ((2\ dvd\ x) = (\forall (y::int).\ x \neq 2*y + 1))$
$\langle proof \rangle$

theorem ~ $(\forall (x::int).$
$((2\ dvd\ x) = (\forall (y::int).\ x \neq 2*y+1) \mid$
$(\exists (q::int)\ (u::int)\ i.\ 3*i + 2*q - u < 17)$

$--> 0 < x \mid ((\sim 3\ dvd\ x)\ \&(x + 8 = 0))))$
⟨proof⟩

theorem $\sim (\forall (i::int).\ 4 \leq i --> (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
⟨proof⟩

theorem $\forall (i::int).\ 8 \leq i --> (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
⟨proof⟩

theorem $\exists (j::int).\ \forall i.\ j \leq i --> (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
⟨proof⟩

theorem $\sim (\forall j\ (i::int).\ j \leq i --> (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
⟨proof⟩

theorem $(\exists m::nat.\ n = 2 * m) --> (n + 1)\ div\ 2 = n\ div\ 2$
⟨proof⟩

end

43 Generic reflection and reification

theory *Reflection*
imports *Main*
uses *reflection-data.ML* (*reflection.ML*)
begin

⟨ML⟩

lemma *ext2*: $(\forall x.\ f\ x = g\ x) \Longrightarrow f = g$
⟨proof⟩

⟨ML⟩
end

44 Implementation of finite sets by lists

theory *Executable-Set*
imports *Main*
begin

44.1 Definitional rewrites

lemma [*code target*: *Set*]:

$A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
⟨*proof*⟩

lemma [*code*]:
$a \in A \longleftrightarrow (\exists x \in A.\ x = a)$
⟨*proof*⟩

definition
filter-set :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'a\ set$ **where**
filter-set $P\ xs = \{x \in xs.\ P\ x\}$

44.2 Operations on lists

44.2.1 Basic definitions

definition
flip :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
flip $f\ a\ b = f\ b\ a$

definition
member :: $'a\ list \Rightarrow 'a \Rightarrow bool$ **where**
member $xs\ x \longleftrightarrow x \in set\ xs$

definition
insertl :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
insertl $x\ xs = (if\ member\ xs\ x\ then\ xs\ else\ x\#xs)$

lemma [*code target*: *List*]: *member* $[]\ y \longleftrightarrow False$
and [*code target*: *List*]: *member* $(x\#xs)\ y \longleftrightarrow y = x \vee member\ xs\ y$
⟨*proof*⟩

fun
drop-first :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
drop-first $f\ [] = []$
| *drop-first* $f\ (x\#xs) = (if\ f\ x\ then\ xs\ else\ x\ \#\ drop\text{-}first\ f\ xs)$
declare *drop-first.simps* [*code del*]
declare *drop-first.simps* [*code target*: *List*]

declare *remove1.simps* [*code del*]
lemma [*code target*: *List*]:
remove1 $x\ xs = (if\ member\ xs\ x\ then\ drop\text{-}first\ (\lambda y.\ y = x)\ xs\ else\ xs)$
⟨*proof*⟩

lemma *member-nil* [*simp*]:
member $[] = (\lambda x.\ False)$
⟨*proof*⟩

lemma *member-insertl* [*simp*]:
$x \in set\ (insertl\ x\ xs)$
⟨*proof*⟩

lemma *insertl-member* [*simp*]:
  fixes *xs x*
  assumes *member*: *member xs x*
  shows *insertl x xs = xs*
  ⟨*proof*⟩

lemma *insertl-not-member* [*simp*]:
  fixes *xs x*
  assumes *member*: ¬ (*member xs x*)
  shows *insertl x xs = x # xs*
  ⟨*proof*⟩

lemma *foldr-remove1-empty* [*simp*]:
  foldr remove1 xs [] = []
  ⟨*proof*⟩

44.2.2 Derived definitions

function *unionl* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  unionl [] *ys* = *ys*
| *unionl xs ys* = *foldr insertl xs ys*
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *unionl-def* = *unionl.simps*(*2*)

function *intersect* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  intersect [] *ys* = []
| *intersect xs* [] = []
| *intersect xs ys* = *filter* (*member xs*) *ys*
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *intersect-def* = *intersect.simps*(*3*)

function *subtract* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
  subtract [] *ys* = *ys*
| *subtract xs* [] = []
| *subtract xs ys* = *foldr remove1 xs ys*
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *subtract-def* = *subtract.simps*(*3*)

function *map-distinct* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list*

where
  map-distinct f [] = []
| *map-distinct f xs* = *foldr* (*insertl o f*) *xs* []
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *map-distinct-def* = *map-distinct.simps*(*2*)

function *unions* :: *'a list list* ⇒ *'a list*
where
  unions [] = []
| *unions xs* = *foldr unionl xs* []
⟨*proof*⟩
termination ⟨*proof*⟩

lemmas *unions-def* = *unions.simps*(*2*)

consts *intersects* :: *'a list list* ⇒ *'a list*
primrec
  intersects (*x*#*xs*) = *foldr intersect xs x*

definition
  map-union :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
  map-union xs f = *unions* (*map f xs*)

definition
  map-inter :: *'a list* ⇒ (*'a* ⇒ *'b list*) ⇒ *'b list* **where**
  map-inter xs f = *intersects* (*map f xs*)

44.3 Isomorphism proofs

lemma *iso-member*:
  member xs x ⟷ *x* ∈ *set xs*
  ⟨*proof*⟩

lemma *iso-insert*:
  set (*insertl x xs*) = *insert x* (*set xs*)
  ⟨*proof*⟩

lemma *iso-remove1*:
  assumes *distnct*: *distinct xs*
  shows *set* (*remove1 x xs*) = *set xs* − {*x*}
  ⟨*proof*⟩

lemma *iso-union*:
  set (*unionl xs ys*) = *set xs* ∪ *set ys*
  ⟨*proof*⟩

lemma *iso-intersect*:

set (*intersect xs ys*) = *set xs* ∩ *set ys*
⟨*proof*⟩

definition
subtract′ :: *′a list* ⇒ *′a list* ⇒ *′a list* **where**
subtract′ = *flip subtract*

lemma *iso-subtract*:
fixes *ys*
assumes *distnct*: *distinct ys*
shows *set* (*subtract′ ys xs*) = *set ys* − *set xs*
and *distinct* (*subtract′ ys xs*)
⟨*proof*⟩

lemma *iso-map-distinct*:
set (*map-distinct f xs*) = *image f* (*set xs*)
⟨*proof*⟩

lemma *iso-unions*:
set (*unions xss*) = $\bigcup$ *set* (*map set xss*)
⟨*proof*⟩

lemma *iso-intersects*:
set (*intersects* (*xs#xss*)) = $\bigcap$ *set* (*map set* (*xs#xss*))
⟨*proof*⟩

lemma *iso-UNION*:
set (*map-union xs f*) = *UNION* (*set xs*) (*set o f*)
⟨*proof*⟩

lemma *iso-INTER*:
set (*map-inter* (*x#xs*) *f*) = *INTER* (*set* (*x#xs*)) (*set o f*)
⟨*proof*⟩

definition
Blall :: *′a list* ⇒ (*′a* ⇒ *bool*) ⇒ *bool* **where**
Blall = *flip list-all*
definition
Blex :: *′a list* ⇒ (*′a* ⇒ *bool*) ⇒ *bool* **where**
Blex = *flip list-ex*

lemma *iso-Ball*:
Blall xs f = *Ball* (*set xs*) *f*
⟨*proof*⟩

lemma *iso-Bex*:
Blex xs f = *Bex* (*set xs*) *f*
⟨*proof*⟩

lemma *iso-filter*:
  set (*filter P xs*) = *filter-set P* (*set xs*)
  ⟨*proof*⟩

44.4 code generator setup

⟨*ML*⟩

44.4.1 type serializations

types-code
  set (- *list*)
attach (*term-of*) ⟪
fun term-of-set f T [] = *Const* ({}, *Type* (*set*, [*T*]))
  | *term-of-set f T* (*x* :: *xs*) = *Const* (*insert*,
      T −−> *Type* (*set*, [*T*]) −−> *Type* (*set*, [*T*])) $ *f x* $ *term-of-set f T xs*;
⟫
attach (*test*) ⟪
fun gen-set′ aG i j = *frequency*
  [(*i*, *fn* () => *aG j* :: *gen-set′ aG* (*i*−*1*) *j*), (*1*, *fn* () => [])] ()
and gen-set aG i = *gen-set′ aG i i*;
⟫

44.4.2 const serializations

consts-code
  {} ({∗[]∗})
  insert ({∗*insertl*∗})
  op ∪ ({∗*unionl*∗})
  op ∩ ({∗*intersect*∗})
  op − :: ′*a set* ⇒ ′*a set* ⇒ ′*a set* ({∗ *flip subtract* ∗})
  image ({∗*map-distinct*∗})
  Union ({∗*unions*∗})
  Inter ({∗*intersects*∗})
  UNION ({∗*map-union*∗})
  INTER ({∗*map-inter*∗})
  Ball ({∗*Blall*∗})
  Bex ({∗*Blex*∗})
  filter-set ({∗*filter*∗})

end

theory *NBE* **imports** *Main Executable-Set* **begin**

axiomatization where *unproven*: *PROP A*

declare *Let-def* [*simp*]

consts-code *undefined* ((*raise Match*))

types *lam-var-name* = *nat*
ml-var-name = *nat*
const-name = *nat*

datatype *tm* = *Ct const-name* | *Vt lam-var-name* | *Lam tm* | *At tm tm*
| *term-of ml*
and *ml* =
C const-name ml list | *V lam-var-name ml list*
| *Fun ml ml list nat*
| *apply ml ml*

| *V-ML ml-var-name* | *A-ML ml ml list* | *Lam-ML ml*
| *CC const-name*

lemma [*simp*]: $x \in set\ vs \Longrightarrow size\ x < Suc\ (ml\text{-}list\text{-}size1\ vs)$
⟨*proof*⟩
lemma [*simp*]: $x \in set\ vs \Longrightarrow size\ x < Suc\ (ml\text{-}list\text{-}size2\ vs)$
⟨*proof*⟩
lemma [*simp*]:$x \in set\ vs \Longrightarrow size\ x < Suc\ (size\ v + ml\text{-}list\text{-}size3\ vs)$
⟨*proof*⟩
lemma [*simp*]: $x \in set\ vs \Longrightarrow size\ x < Suc\ (size\ v + ml\text{-}list\text{-}size4\ vs)$
⟨*proof*⟩

locale *Vars* =
fixes *r s t*:: *tm*
and *rs ss ts* :: *tm list*
and *u v w* :: *ml*
and *us vs ws* :: *ml list*
and *nm* :: *const-name*
and *x* :: *lam-var-name*
and *X* :: *ml-var-name*

inductive-set *Pure-tms* :: *tm set*
where
Ct s : *Pure-tms*
| *Vt x* : *Pure-tms*
| *t* : *Pure-tms* ==> *Lam t* : *Pure-tms*
| *s* : *Pure-tms* ==> *t* : *Pure-tms* ==> *At s t* : *Pure-tms*

consts
R :: (*const-name* ∗ *tm list* ∗ *tm*)*set*
compR :: (*const-name* ∗ *ml list* ∗ *ml*)*set*

fun
lift-tm :: *nat* ⇒ *tm* ⇒ *tm* (*lift*) **and**
lift-ml :: *nat* ⇒ *ml* ⇒ *ml* (*lift*)

where
$lift\ i\ (Ct\ nm) = Ct\ nm\ |$
$lift\ i\ (Vt\ x) = Vt(if\ x < i\ then\ x\ else\ x{+}1)\ |$
$lift\ i\ (Lam\ t) = Lam\ (lift\ (i{+}1)\ t)\ |$
$lift\ i\ (At\ s\ t) = At\ (lift\ i\ s)\ (lift\ i\ t)\ |$
$lift\ i\ (term\text{-}of\ v) = term\text{-}of\ (lift\ i\ v)\ |$

$lift\ i\ (C\ nm\ vs) = C\ nm\ (map\ (lift\ i)\ vs)\ |$
$lift\ i\ (V\ x\ vs) = V\ (if\ x < i\ then\ x\ else\ x{+}1)\ (map\ (lift\ i)\ vs)\ |$
$lift\ i\ (Fun\ v\ vs\ n) = Fun\ (lift\ i\ v)\ (map\ (lift\ i)\ vs)\ n\ |$
$lift\ i\ (apply\ u\ v) = apply\ (lift\ i\ u)\ (lift\ i\ v)\ |$
$lift\ i\ (V\text{-}ML\ X) = V\text{-}ML\ X\ |$
$lift\ i\ (A\text{-}ML\ v\ vs) = A\text{-}ML\ (lift\ i\ v)\ (map\ (lift\ i)\ vs)\ |$
$lift\ i\ (Lam\text{-}ML\ v) = Lam\text{-}ML\ (lift\ i\ v)\ |$
$lift\ i\ (CC\ nm) = CC\ nm$

fun
$lift\text{-}tm\text{-}ML :: nat \Rightarrow tm \Rightarrow tm\ (lift_{ML})$ **and**
$lift\text{-}ml\text{-}ML :: nat \Rightarrow ml \Rightarrow ml\ (lift_{ML})$
where
$lift_{ML}\ i\ (Ct\ nm) = Ct\ nm\ |$
$lift_{ML}\ i\ (Vt\ x) = Vt\ x\ |$
$lift_{ML}\ i\ (Lam\ t) = Lam\ (lift_{ML}\ i\ t)\ |$
$lift_{ML}\ i\ (At\ s\ t) = At\ (lift_{ML}\ i\ s)\ (lift_{ML}\ i\ t)\ |$
$lift_{ML}\ i\ (term\text{-}of\ v) = term\text{-}of\ (lift_{ML}\ i\ v)\ |$

$lift_{ML}\ i\ (C\ nm\ vs) = C\ nm\ (map\ (lift_{ML}\ i)\ vs)\ |$
$lift_{ML}\ i\ (V\ x\ vs) = V\ x\ (map\ (lift_{ML}\ i)\ vs)\ |$
$lift_{ML}\ i\ (Fun\ v\ vs\ n) = Fun\ (lift_{ML}\ i\ v)\ (map\ (lift_{ML}\ i)\ vs)\ n\ |$
$lift_{ML}\ i\ (apply\ u\ v) = apply\ (lift_{ML}\ i\ u)\ (lift_{ML}\ i\ v)\ |$
$lift_{ML}\ i\ (V\text{-}ML\ X) = V\text{-}ML\ (if\ X < i\ then\ X\ else\ X{+}1)\ |$
$lift_{ML}\ i\ (A\text{-}ML\ v\ vs) = A\text{-}ML\ (lift_{ML}\ i\ v)\ (map\ (lift_{ML}\ i)\ vs)\ |$
$lift_{ML}\ i\ (Lam\text{-}ML\ v) = Lam\text{-}ML\ (lift_{ML}\ (i{+}1)\ v)\ |$
$lift_{ML}\ i\ (CC\ nm) = CC\ nm$

constdefs
$cons :: tm \Rightarrow (nat \Rightarrow tm) \Rightarrow (nat \Rightarrow tm)$ (**infix** $\#\#\ 65$)
$t\#\#f \equiv \lambda i.\ case\ i\ of\ 0 \Rightarrow t\ |\ Suc\ j \Rightarrow lift\ 0\ (f\ j)$
$cons\text{-}ML :: ml \Rightarrow (nat \Rightarrow ml) \Rightarrow (nat \Rightarrow ml)$ (**infix** $\#\#\ 65$)
$v\#\#f \equiv \lambda i.\ case\ i\ of\ 0 \Rightarrow v{::}ml\ |\ Suc\ j \Rightarrow lift_{ML}\ 0\ (f\ j)$

consts $subst :: (nat \Rightarrow tm) \Rightarrow tm \Rightarrow tm$
primrec
$subst\ f\ (Ct\ nm) = Ct\ nm$
$subst\ f\ (Vt\ x) = f\ x$
$subst\ f\ (Lam\ t) = Lam\ (subst\ (Vt\ 0\ \#\#\ f)\ t)$
$subst\ f\ (At\ s\ t) = At\ (subst\ f\ s)\ (subst\ f\ t)$

lemma *size-lift*[*simp*]: **shows**
size(*lift i t*) = *size*(*t*::*tm*) **and** *size*(*lift i* (*v*::*ml*)) = *size v*
and *ml-list-size1* (*map* (*lift i*) *vs*) = *ml-list-size1 vs*
and *ml-list-size2* (*map* (*lift i*) *vs*) = *ml-list-size2 vs*
and *ml-list-size3* (*map* (*lift i*) *vs*) = *ml-list-size3 vs*
and *ml-list-size4* (*map* (*lift i*) *vs*) = *ml-list-size4 vs*
⟨*proof*⟩

lemma *size-lift-ML*[*simp*]: **shows**
size($lift_{ML}$ *i t*) = *size*(*t*::*tm*) **and** *size*($lift_{ML}$ *i* (*v*::*ml*)) = *size v*
and *ml-list-size1* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size1 vs*
and *ml-list-size2* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size2 vs*
and *ml-list-size3* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size3 vs*
and *ml-list-size4* (*map* ($lift_{ML}$ *i*) *vs*) = *ml-list-size4 vs*
⟨*proof*⟩

fun
subst-ml-ML :: (*nat* ⇒ *ml*) ⇒ *ml* ⇒ *ml* ($subst_{ML}$) **and**
subst-tm-ML :: (*nat* ⇒ *ml*) ⇒ *tm* ⇒ *tm* ($subst_{ML}$)
where
$subst_{ML}$ *f* (*Ct nm*) = *Ct nm* |
$subst_{ML}$ *f* (*Vt x*) = *Vt x* |
$subst_{ML}$ *f* (*Lam t*) = *Lam* ($subst_{ML}$ (*lift 0 o f*) *t*) |
$subst_{ML}$ *f* (*At s t*) = *At* ($subst_{ML}$ *f s*) ($subst_{ML}$ *f t*) |
$subst_{ML}$ *f* (*term-of v*) = *term-of* ($subst_{ML}$ *f v*) |

$subst_{ML}$ *f* (*C nm vs*) = *C nm* (*map* ($subst_{ML}$ *f*) *vs*) |
$subst_{ML}$ *f* (*V x vs*) = *V x* (*map* ($subst_{ML}$ *f*) *vs*) |
$subst_{ML}$ *f* (*Fun v vs n*) = *Fun* ($subst_{ML}$ *f v*) (*map* ($subst_{ML}$ *f*) *vs*) *n* |
$subst_{ML}$ *f* (*apply u v*) = *apply* ($subst_{ML}$ *f u*) ($subst_{ML}$ *f v*) |
$subst_{ML}$ *f* (*V-ML X*) = *f X* |
$subst_{ML}$ *f* (*A-ML v vs*) = *A-ML* ($subst_{ML}$ *f v*) (*map* ($subst_{ML}$ *f*) *vs*) |
$subst_{ML}$ *f* (*Lam-ML v*) = *Lam-ML* ($subst_{ML}$ (*V-ML 0 ## f*) *v*) |
$subst_{ML}$ *f* (*CC nm*) = *CC nm*

lemmas [*code*] = *lift-tm-ML.simps lift-ml-ML.simps*
lemmas [*code*] = *lift-tm.simps lift-ml.simps*
lemmas [*code*] = *subst-tm-ML.simps subst-ml-ML.simps*

abbreviation
subst-decr :: *nat* ⇒ *tm* ⇒ *nat* ⇒ *tm* **where**
subst-decr k t == %*n. if n*<*k then Vt n else if n*=*k then t else Vt*(*n* − *1*)
abbreviation
subst-decr-ML :: *nat* ⇒ *ml* ⇒ *nat* ⇒ *ml* **where**
subst-decr-ML k v == %*n. if n*<*k then V-ML n else if n*=*k then v else V-ML*(*n* − *1*)
abbreviation

$subst1 :: tm \Rightarrow tm \Rightarrow nat \Rightarrow tm$ ((-/[-'/-]) [300, 0, 0] 300) **where**
$s[t/k] == subst\ (subst\text{-}decr\ k\ t)\ s$
abbreviation
$subst1\text{-}ML :: ml \Rightarrow ml \Rightarrow nat \Rightarrow ml$ ((-/[-'/-]) [300, 0, 0] 300) **where**
$u[v/k] == subst_{ML}\ (subst\text{-}decr\text{-}ML\ k\ v)\ u$

lemma *size-subst-ML*[*simp*]: **shows**
$(!x.\ size(f\ x) = 0) \longrightarrow size(subst_{ML}\ f\ t) = size(t::tm)$ **and**
$(!x.\ size(f\ x) = 0) \longrightarrow size(subst_{ML}\ f\ (v::ml)) = size\ v$
and $(!x.\ size(f\ x) = 0) \longrightarrow ml\text{-}list\text{-}size1\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size1\ vs$
and $(!x.\ size(f\ x) = 0) \longrightarrow ml\text{-}list\text{-}size2\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size2\ vs$
and $(!x.\ size(f\ x) = 0) \longrightarrow ml\text{-}list\text{-}size3\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size3\ vs$
and $(!x.\ size(f\ x) = 0) \longrightarrow ml\text{-}list\text{-}size4\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size4\ vs$
$\langle proof \rangle$

lemma *lift-lift*: **includes** *Vars* **shows**
$i < k+1 \Longrightarrow lift\ (Suc\ k)\ (lift\ i\ t) = lift\ i\ (lift\ k\ t)$
and $i < k+1 \Longrightarrow lift\ (Suc\ k)\ (lift\ i\ v) = lift\ i\ (lift\ k\ v)$
$\langle proof \rangle$

corollary *lift-o-lift*: **shows**
$i < k+1 \Longrightarrow lift\text{-}tm\ (Suc\ k)\ o\ (lift\text{-}tm\ i) = lift\text{-}tm\ i\ o\ lift\text{-}tm\ k$ **and**
$i < k+1 \Longrightarrow lift\text{-}ml\ (Suc\ k)\ o\ (lift\text{-}ml\ i) = lift\text{-}ml\ i\ o\ lift\text{-}ml\ k$
$\langle proof \rangle$

lemma *lift-lift-ML*: **includes** *Vars* **shows**
$i < k+1 \Longrightarrow lift_{ML}\ (Suc\ k)\ (lift_{ML}\ i\ t) = lift_{ML}\ i\ (lift_{ML}\ k\ t)$
and $i < k+1 \Longrightarrow lift_{ML}\ (Suc\ k)\ (lift_{ML}\ i\ v) = lift_{ML}\ i\ (lift_{ML}\ k\ v)$
$\langle proof \rangle$

lemma *lift-lift-ML-comm*: **includes** *Vars* **shows**
$lift\ j\ (lift_{ML}\ i\ t) = lift_{ML}\ i\ (lift\ j\ t)$ **and**
$lift\ j\ (lift_{ML}\ i\ v) = lift_{ML}\ i\ (lift\ j\ v)$
$\langle proof \rangle$

lemma [*simp*]:
$V\text{-}ML\ 0\ \#\#\ subst\text{-}decr\text{-}ML\ k\ v = subst\text{-}decr\text{-}ML\ (Suc\ k)\ (lift_{ML}\ 0\ v)$
$\langle proof \rangle$

lemma [*simp*]: $lift\ 0\ o\ subst\text{-}decr\text{-}ML\ k\ v = subst\text{-}decr\text{-}ML\ k\ (lift\ 0\ v)$
$\langle proof \rangle$

lemma *subst-lift-id*[*simp*]: **includes** *Vars* **shows**
$subst_{ML}\ (subst\text{-}decr\text{-}ML\ k\ v)\ (lift_{ML}\ k\ t) = t$ **and** $(lift_{ML}\ k\ u)[v/k] = u$
$\langle proof \rangle$

inductive-set

tRed :: (*tm* ∗ *tm*)*set*
and *tred* :: [*tm*, *tm*] => *bool* (**infixl** → *50*)
where
$s \rightarrow t == (s,\ t) \in tRed$
| *At* (*Lam t*) *s* → *t*[*s*/*0*]
| (*nm*,*ts*,*t*) : *R* ==> *foldl At* (*Ct nm*) (*map* (*subst rs*) *ts*) → *subst rs t*
| $t \rightarrow t'$ ==> *Lam t* → *Lam* t'
| $s \rightarrow s'$ ==> *At s t* → *At* s' *t*
| $t \rightarrow t'$ ==> *At s t* → *At s* t'

abbreviation
treds :: [*tm*, *tm*] => *bool* (**infixl** →∗ *50*) **where**
s →∗ *t* == (*s*, *t*) ∈ *tRed*ˆ∗

inductive-set
tRed-list :: (*tm list* ∗ *tm list*) *set*
and *treds-list* :: [*tm list*, *tm list*] ⇒ *bool* (**infixl** →∗ *50*)
where
ss →∗ *ts* == (*ss*, *ts*) ∈ *tRed-list*
| [] →∗ []
| *ts* →∗ *ts*′ ==> *t* →∗ *t*′ ==> *t*#*ts* →∗ *t*′#*ts*′

declare *tRed-list.intros*[*simp*]

lemma *tRed-list-refl*[*simp*]: **includes** *Vars* **shows** *ts* →∗ *ts*
⟨*proof*⟩

fun *ML-closed* :: *nat* ⇒ *ml* ⇒ *bool*
and *ML-closed-t* :: *nat* ⇒ *tm* ⇒ *bool* **where**
ML-closed i (*C nm vs*) = (*ALL v*:*set vs*. *ML-closed i v*) |
ML-closed i (*V nm vs*) = (*ALL v*:*set vs*. *ML-closed i v*) |
ML-closed i (*Fun f vs n*) = (*ML-closed i f* & (*ALL v*:*set vs*. *ML-closed i v*)) |
ML-closed i (*A-ML v vs*) = (*ML-closed i v* & (*ALL v*:*set vs*. *ML-closed i v*)) |
ML-closed i (*apply v w*) = (*ML-closed i v* & *ML-closed i w*) |
ML-closed i (*CC nm*) = *True* |
ML-closed i (*V-ML X*) = (*X*<*i*) |
ML-closed i (*Lam-ML v*) = *ML-closed* (*i*+*1*) *v* |
ML-closed-t i (*term-of v*) = *ML-closed i v* |
ML-closed-t i (*At r s*) = (*ML-closed-t i r* & *ML-closed-t i s*) |
ML-closed-t i (*Lam t*) = (*ML-closed-t i t*) |
ML-closed-t i v = *True*
thm *ML-closed.simps ML-closed-t.simps*

inductive-set
Red :: (*ml* ∗ *ml*)*set*
and *Redt* :: (*tm* ∗ *tm*)*set*
and *Redl* :: (*ml list* ∗ *ml list*)*set*
and *red* :: [*ml*, *ml*] => *bool* (**infixl** ⇒ *50*)

and *redl* :: [*ml list*, *ml list*] => *bool* (**infixl** ⇒ *50*)
and *redt* :: [*tm*, *tm*] => *bool* (**infixl** ⇒ *50*)
and *reds* :: [*ml*, *ml*] => *bool* (**infixl** ⇒∗ *50*)
and *redts* :: [*tm*, *tm*] => *bool* (**infixl** ⇒∗ *50*)
where
s ⇒ *t* == (*s*, *t*) ∈ *Red*
| *s* ⇒ *t* == (*s*, *t*) ∈ *Redl*
| *s* ⇒ *t* == (*s*, *t*) ∈ *Redt*
| *s* ⇒∗ *t* == (*s*, *t*) ∈ *Red*ˆ∗
| *s* ⇒∗ *t* == (*s*, *t*) ∈ *Redt*ˆ∗

| *A-ML* (*Lam-ML u*) [*v*] ⇒ *u*[*v*/*0*]

| (*nm*,*vs*,*v*) : *compR* ==> *ALL i*. *ML-closed 0* (*f i*) ⟹ *A-ML* (*CC nm*) (*map* ($subst_{ML}$ *f*) *vs*) ⇒ $subst_{ML}$ *f v*

| *apply-Fun1*: *apply* (*Fun f vs* (*Suc 0*)) *v* ⇒ *A-ML f* (*vs* @ [*v*])
| *apply-Fun2*: *n* > *0* ==>
apply (*Fun f vs* (*Suc n*)) *v* ⇒ *Fun f* (*vs* @ [*v*]) *n*
| *apply-C*: *apply* (*C nm vs*) *v* ⇒ *C nm* (*vs* @ [*v*])
| *apply-V*: *apply* (*V x vs*) *v* ⇒ *V x* (*vs* @ [*v*])

| *term-of-C*: *term-of* (*C nm vs*) ⇒ *foldl At* (*Ct nm*) (*map term-of vs*)
| *term-of-V*: *term-of* (*V x vs*) ⇒ *foldl At* (*Vt x*) (*map term-of vs*)
| *term-of-Fun*: *term-of*(*Fun vf vs n*) ⇒
Lam (*term-of* ((*apply* (*lift 0* (*Fun vf vs n*)) (*V-ML 0*))[*V 0* []/*0*]))

| *ctxt-Lam*: *t* ⇒ *t′* ==> *Lam t* ⇒ *Lam t′*
| *ctxt-At1*: *s* ⇒ *s′* ==> *At s t* ⇒ *At s′ t*
| *ctxt-At2*: *t* ⇒ *t′* ==> *At s t* ⇒ *At s t′*
| *ctxt-term-of*: *v* ⇒ *v′* ==> *term-of v* ⇒ *term-of v′*
| *ctxt-C*: *vs* ⇒ *vs′* ==> *C nm vs* ⇒ *C nm vs′*
| *ctxt-V*: *vs* ⇒ *vs′* ==> *V x vs* ⇒ *V x vs′*
| *ctxt-Fun1*: *f* ⇒ *f′* ==> *Fun f vs n* ⇒ *Fun f′ vs n*
| *ctxt-Fun3*: *vs* ⇒ *vs′* ==> *Fun f vs n* ⇒ *Fun f vs′ n*
| *ctxt-apply1*: *s* ⇒ *s′* ==> *apply s t* ⇒ *apply s′ t*
| *ctxt-apply2*: *t* ⇒ *t′* ==> *apply s t* ⇒ *apply s t′*
| *ctxt-A-ML1*: *f* ⇒ *f′* ==> *A-ML f vs* ⇒ *A-ML f′ vs*
| *ctxt-A-ML2*: *vs* ⇒ *vs′* ==> *A-ML f vs* ⇒ *A-ML f vs′*
| *ctxt-list1*: *v* ⇒ *v′* ==> *v*#*vs* ⇒ *v′*#*vs*
| *ctxt-list2*: *vs* ⇒ *vs′* ==> *v*#*vs* ⇒ *v*#*vs′*

consts
ar :: *const-name* ⇒ *nat*

axioms
ar-pos: *ar nm* > *0*

types *env* = *ml list*

consts *eval* :: $tm \Rightarrow env \Rightarrow ml$
primrec
$eval\ (Vt\ x)\ e = e!x$
$eval\ (Ct\ nm)\ e = Fun\ (CC\ nm)\ []\ (ar\ nm)$
$eval\ (At\ s\ t)\ e = apply\ (eval\ s\ e)\ (eval\ t\ e)$
$eval\ (Lam\ t)\ e = Fun\ (Lam\text{-}ML\ (eval\ t\ ((V\text{-}ML\ 0)\ \#\ map\ (lift_{ML}\ 0)\ e)))\ []\ 1$

fun $size'$ :: $ml \Rightarrow nat$ **where**
$size'\ (C\ nm\ vs) = (\sum v{\leftarrow}vs.\ size'\ v)+1\ |$
$size'\ (V\ nm\ vs) = (\sum v{\leftarrow}vs.\ size'\ v)+1\ |$
$size'\ (Fun\ f\ vs\ n) = (size'\ f + (\sum v{\leftarrow}vs.\ size'\ v))+1\ |$
$size'\ (A\text{-}ML\ v\ vs) = (size'\ v + (\sum v{\leftarrow}vs.\ size'\ v))+1\ |$
$size'\ (apply\ v\ w) = (size'\ v + size'\ w)+1\ |$
$size'\ (CC\ nm) = 1\ |$
$size'\ (V\text{-}ML\ X) = 1\ \ |$
$size'\ (Lam\text{-}ML\ v) = size'\ v + 1$

lemma *listsum-size'*[*simp*]:
$v \in set\ vs \Longrightarrow size'\ v < Suc(listsum\ (map\ size'\ vs))$
⟨*proof*⟩

corollary *cor-listsum-size'*[*simp*]:
$v \in set\ vs \Longrightarrow size'\ v < Suc(m + listsum\ (map\ size'\ vs))$
⟨*proof*⟩

lemma
size-subst-ML[*simp*]: **includes** *Vars* **assumes** A: $!i.\ size(f\ i) = 0$
shows $size(subst_{ML}\ f\ t) = size(t)$
and $size(subst_{ML}\ f\ v) = size(v)$
and $ml\text{-}list\text{-}size1\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size1\ vs$
and $ml\text{-}list\text{-}size2\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size2\ vs$
and $ml\text{-}list\text{-}size3\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size3\ vs$
and $ml\text{-}list\text{-}size4\ (map\ (subst_{ML}\ f)\ vs) = ml\text{-}list\text{-}size4\ vs$
⟨*proof*⟩

lemma [*simp*]:
$\forall\ i\ j.\ size'(f\ i) = size'(V\text{-}ML\ j) \Longrightarrow size'\ (subst_{ML}\ f\ v) = size'\ v$
⟨*proof*⟩

lemma [*simp*]: $size'\ (lift\ i\ v) = size'\ v$
⟨*proof*⟩

function *kernel* :: $ml \Rightarrow tm$ (-! *300*) **where**
$(C\ nm\ vs)! = foldl\ At\ (Ct\ nm)\ (map\ kernel\ vs)\ |$
$(Lam\text{-}ML\ v)! = Lam\ (((lift\ 0\ v)[V\ 0\ []/0])!)\ |$

$(Fun\ f\ vs\ n)! = foldl\ At\ (f!)\ (map\ kernel\ vs)$ |
$(A\text{-}ML\ v\ vs)! = foldl\ At\ (v!)\ (map\ kernel\ vs)$ |
$(apply\ v\ w)! = At\ (v!)\ (w!)$ |
$(CC\ nm)! = Ct\ nm$ |
$(V\ x\ vs)! = foldl\ At\ (Vt\ x)\ (map\ kernel\ vs)$ |
$(V\text{-}ML\ X)! = undefined$
⟨proof⟩
termination ⟨proof⟩

consts *kernelt* :: $tm \Rightarrow tm$ (-! *300*)
primrec
$(Ct\ nm)! = Ct\ nm$
$(term\text{-}of\ v)! = v!$
$(Vt\ x)! = Vt\ x$
$(At\ s\ t)! = At\ (s!)\ (t!)$
$(Lam\ t)! = Lam\ (t!)$

abbreviation
  kernels :: *ml list* ⇒ *tm list* (-! *300*) **where**
  vs ! == *map kernel vs*

axioms
compiler-correct:
$(nm,\ vs,\ v) : compR$ ==> $ALL\ i.\ ML\text{-}closed\ 0\ (f\ i) \Longrightarrow (nm,\ (map\ (subst_{ML}\ f)\ vs)!,\ (subst_{ML}\ f\ v)!) : R$

consts
  free-vars :: $tm \Rightarrow lam\text{-}var\text{-}name\ set$
primrec
$free\text{-}vars\ (Ct\ nm) = \{\}$
$free\text{-}vars\ (Vt\ x) = \{x\}$
$free\text{-}vars\ (Lam\ t) = \{i.\ EX\ j : free\text{-}vars\ t.\ j = i+1\}$
$free\text{-}vars\ (At\ s\ t) = free\text{-}vars\ s \cup free\text{-}vars\ t$

lemma [*simp*]: $t : Pure\text{-}tms \Longrightarrow lift_{ML}\ k\ t = t$
⟨proof⟩

lemma *kernel-pure*: **includes** *Vars* **assumes** $t : Pure\text{-}tms$ **shows** $t! = t$
⟨proof⟩

lemma *lift-eval*:
 $t : Pure\text{-}tms \Longrightarrow ALL\ e\ k.\ (ALL\ i : free\text{-}vars\ t.\ i < size\ e)$ −−> $lift\ k\ (eval\ t\ e) = eval\ t\ (map\ (lift\ k)\ e)$
⟨proof⟩

lemma *lift-ML-eval*[*rule-format*]:
 $t : Pure\text{-}tms \Longrightarrow ALL\ e\ k.\ (ALL\ i : free\text{-}vars\ t.\ i < size\ e)$ −−> $lift_{ML}\ k\ (eval$

$t\ e) = eval\ t\ (map\ (lift_{ML}\ k)\ e)$
⟨proof⟩

lemma [*simp*]: **includes** *Vars* **shows** $(v\ \#\#\ f)\ 0 = v$
⟨proof⟩

lemma [*simp*]: **includes** *Vars* **shows** $(v\ \#\#\ f)\ (Suc\ n) = lift_{ML}\ 0\ (f\ n)$
⟨proof⟩

lemma *lift-o-shift*: $lift\ k\ o\ (V\text{-}ML\ 0\ \#\#\ f) = (V\text{-}ML\ 0\ \#\#\ (lift\ k \circ f))$
⟨proof⟩

lemma *lift-subst-ML*: **shows**
$lift\text{-}tm\ k\ (subst_{ML}\ f\ t) = subst_{ML}\ (lift\text{-}ml\ k\ o\ f)\ (lift\text{-}tm\ k\ t)$ **and**
$lift\text{-}ml\ k\ (subst_{ML}\ f\ v) = subst_{ML}\ (lift\text{-}ml\ k\ o\ f)\ (lift\text{-}ml\ k\ v)$
⟨proof⟩

corollary *lift-subst-ML1*: $\forall\ v\ k.\ lift\text{-}ml\ 0\ (u[v/k]) = (lift\text{-}ml\ 0\ u)[lift\ 0\ v/k]$
⟨proof⟩

lemma *lift-ML-lift-ML*: **includes** *Vars* **shows**
$i < k+1 \Longrightarrow lift_{ML}\ (Suc\ k)\ (lift_{ML}\ i\ t) = lift_{ML}\ i\ (lift_{ML}\ k\ t)$
and $i < k+1 \Longrightarrow lift_{ML}\ (Suc\ k)\ (lift_{ML}\ i\ v) = lift_{ML}\ i\ (lift_{ML}\ k\ v)$
⟨proof⟩

corollary *lift-ML-o-lift-ML*: **shows**
$i < k+1 \Longrightarrow lift\text{-}tm\text{-}ML\ (Suc\ k)\ o\ (lift\text{-}tm\text{-}ML\ i) = lift\text{-}tm\text{-}ML\ i\ o\ lift\text{-}tm\text{-}ML\ k$
and
$i < k+1 \Longrightarrow lift\text{-}ml\text{-}ML\ (Suc\ k)\ o\ (lift\text{-}ml\text{-}ML\ i) = lift\text{-}ml\text{-}ML\ i\ o\ lift\text{-}ml\text{-}ML\ k$
⟨proof⟩

abbreviation *insrt* **where**
$insrt\ k\ f == (\%i.\ if\ i{<}k\ then\ lift\text{-}ml\text{-}ML\ k\ (f\ i)\ else\ if\ i{=}k\ then\ V\text{-}ML\ k\ else\ lift\text{-}ml\text{-}ML\ k\ (f(i - 1)))$

lemma *subst-insrt-lift*: **includes** *Vars* **shows**
$subst_{ML}\ (insrt\ k\ f)\ (lift_{ML}\ k\ t) = lift_{ML}\ k\ (subst_{ML}\ f\ t)$ **and**
$subst_{ML}\ (insrt\ k\ f)\ (lift_{ML}\ k\ v) = lift_{ML}\ k\ (subst_{ML}\ f\ v)$
⟨proof⟩

corollary *subst-cons-lift*: **includes** *Vars* **shows**
$subst_{ML}\ (V\text{-}ML\ 0\ \#\#\ f)\ o\ (lift\text{-}ml\text{-}ML\ 0) = lift\text{-}ml\text{-}ML\ 0\ o\ (subst\text{-}ml\text{-}ML\ f)$
⟨proof⟩

lemma *subst-eval*[*rule-format*]: $t : Pure\text{-}tms \Longrightarrow$
$ALL\ f\ e.\ (ALL\ i : free\text{-}vars\ t.\ i < size\ e) \longrightarrow subst_{ML}\ f\ (eval\ t\ e) = eval\ t\ (map\ (subst_{ML}\ f)\ e)$
⟨proof⟩

theorem *kernel-eval*[*rule-format*]: **includes** *Vars* **shows**
t : *Pure-tms* $==>$
ALL e. (*ALL* i : *free-vars* t. $i <$ *size* e) $\longrightarrow$ (*ALL* $i <$ *size* e. $e!i = V\ i$ []) $-->$
(*eval* t e)! = t!
⟨*proof*⟩

lemma *map-eq-iff-nth*:
(*map* f *xs* = *map* g *xs*) = (!i<*size* *xs*. f(*xs*!i) = g(*xs*!i))
⟨*proof*⟩

lemma [*simp*]: **includes** *Vars* **shows** *ML-closed* k $v \Longrightarrow$ $lift_{ML}$ k $v = v$
⟨*proof*⟩
lemma [*simp*]: **includes** *Vars* **shows** *ML-closed* *0* $v \Longrightarrow$ $subst_{ML}$ f $v = v$
⟨*proof*⟩
lemma [*simp*]: **includes** *Vars* **shows** *ML-closed* k $v \Longrightarrow$ *ML-closed* k (*lift* m v)
⟨*proof*⟩

lemma *red-Lam*[*simp*]: **includes** *Vars* **shows** $t \rightarrow* t' ==>$ *Lam* $t \rightarrow*$ *Lam* t'
⟨*proof*⟩

lemma *red-At1*[*simp*]: **includes** *Vars* **shows** $t \rightarrow* t' ==>$ *At* t $s \rightarrow*$ *At* t' s
⟨*proof*⟩

lemma *red-At2*[*simp*]: **includes** *Vars* **shows** $t \rightarrow* t' ==>$ *At* s $t \rightarrow*$ *At* s t'
⟨*proof*⟩

lemma *tRed-list-foldl-At*:
ts $\rightarrow*$ *ts'* $\Longrightarrow$ $s \rightarrow* s' \Longrightarrow$ *foldl* *At* s *ts* $\rightarrow*$ *foldl* *At* s' *ts'*
⟨*proof*⟩

lemma [*trans*]: $s = t \Longrightarrow t \rightarrow t' \Longrightarrow s \rightarrow t'$
⟨*proof*⟩

lemma *subst-foldl*[*simp*]:
subst f (*foldl* *At* s *ts*) = *foldl* *At* (*subst* f s) (*map* (*subst* f) *ts*)
⟨*proof*⟩

lemma *foldl-At-size*: *size* *ts* = *size* *ts'* $\Longrightarrow$
foldl *At* s *ts* = *foldl* *At* s' *ts'* $\longleftrightarrow$ $s = s'$ & *ts* = *ts'*
⟨*proof*⟩

consts *depth-At* :: *tm* $\Rightarrow$ *nat*
primrec
depth-At(*Ct* *cn*) = *0*

depth-At(*Vt x*) = *0*
depth-At(*Lam t*) = *0*
depth-At(*At s t*) = *depth-At s* + *1*
depth-At(*term-of v*) = *0*

lemma *depth-At-foldl*:
depth-At(*foldl At s ts*) = *depth-At s* + *size ts*
⟨*proof*⟩

lemma *foldl-At-eq-length*:
foldl At s ts = *foldl At s ts'* ⟹ *length ts* = *length ts'*
⟨*proof*⟩

lemma *foldl-At-eq*[*simp*]: *foldl At s ts* = *foldl At s ts'* ⟷ *ts* = *ts'*
⟨*proof*⟩

lemma [*simp*]: *foldl At s ts* ! = *foldl At* (*s*!) (*map kernelt ts*)
⟨*proof*⟩

lemma [*simp*]: (*kernelt* ∘ *term-of*) = *kernel*
⟨*proof*⟩

lemma *shift-subst-decr*:
Vt 0 ## *subst-decr k t* = *subst-decr* (*Suc k*) (*lift 0 t*)
⟨*proof*⟩

lemma [*simp*]: *lift k* (*foldl At s ts*) = *foldl At* (*lift k s*) (*map* (*lift k*) *ts*)
⟨*proof*⟩

44.5 Horrible detour

definition *liftn n* == *lift-ml 0* ^ *n*

lemma [*simp*]: *liftn n* (*C i vs*) = *C i* (*map* (*liftn n*) *vs*)
⟨*proof*⟩

lemma [*simp*]: *liftn n* (*CC nm*) = *CC nm*
⟨*proof*⟩

lemma [*simp*]: *liftn n* (*apply v w*) = *apply* (*liftn n v*) (*liftn n w*)
⟨*proof*⟩

lemma [*simp*]: *liftn n* (*A-ML v vs*) = *A-ML* (*liftn n v*) (*map* (*liftn n*) *vs*)
⟨*proof*⟩

lemma [*simp*]:
liftn n (*Fun v vs i*) = *Fun* (*liftn n v*) (*map* (*liftn n*) *vs*) *i*
⟨*proof*⟩

lemma [*simp*]: *liftn n (Lam-ML v) = Lam-ML (liftn n v)*
⟨*proof*⟩

lemma *liftn-liftn-add*: *liftn m (liftn n v) = liftn (m+n) v*
⟨*proof*⟩

lemma [*simp*]: *liftn n (V-ML k) = V-ML k*
⟨*proof*⟩

lemma *liftn-lift-ML-comm*: *liftn n* ($lift_{ML}$ *0 v*) = $lift_{ML}$ *0 (liftn n v)*
⟨*proof*⟩

lemma *liftn-cons*: *liftn n ((V-ML 0 ## f) x) = (V-ML 0 ## (liftn n o f)) x*
⟨*proof*⟩

End of horrible detour

lemma *kernel-subst1*:
ML-closed 1 u ⟹ ML-closed 0 v ⟹ kernel(u[v/0]) = (kernel((lift 0 u)[V 0 []/0]))[kernel v/0]
⟨*proof*⟩

lemma includes *Vars* **shows** *foldl-Pure*[*simp*]:
t : Pure-tms ⟹ ∀ t∈set ts. t : Pure-tms ⟹
(!!s t. s : Pure-tms ⟹ t : Pure-tms ⟹ f s t : Pure-tms) ⟹
foldl f t ts ∈ Pure-tms
⟨*proof*⟩

declare *Pure-tms.intros*[*simp*]

lemma includes *Vars* **shows** *ML-closed 0 v ⟹ kernel v : Pure-tms*
⟨*proof*⟩

lemma *subst-Vt*: **includes** *Vars* **shows** *subst Vt = id*
⟨*proof*⟩

theorem *Red-sound*: **includes** *Vars*
shows *v ⇒ v′ ⟹ ML-closed 0 v ⟹ v! →∗ v′! & ML-closed 0 v′*
and *t ⇒ t′ ⟹ ML-closed-t 0 t ⟹ kernelt t →∗ kernelt t′ & ML-closed-t 0 t′*
and *(vs :: ml list) ⇒ vs′ ⟹ !v : set vs . ML-closed 0 v ⟹ map kernel vs →∗ map kernel vs′ & (! v′:set vs′. ML-closed 0 v′)*
⟨*proof*⟩

inductive-cases *tRedE*: *Ct n → u*
thm *tRedE*

lemma [*simp*]: *Ct n = foldl At t ts ⟷ t = Ct n & ts = []*
⟨*proof*⟩

corollary *kernel-inv*: **includes** *Vars* **shows**
$(t :: tm) \Rightarrow* t' ==> ML\text{-}closed\text{-}t\ 0\ t ==> t! \rightarrow* t'!$
⟨*proof*⟩

theorem includes *Vars*
assumes t: t : *Pure-tms* **and** t': t' : *Pure-tms* **and**
closed: *free-vars* $t = \{\}$ **and** *reds*: *term-of* (*eval* t []) $\Rightarrow* t'$
shows $t \rightarrow* t'$
⟨*proof*⟩

end

45 Installing an oracle for SVC (Stanford Validity Checker)

theory *SVC-Oracle*
imports *Main*
uses *svc-funcs.ML*
begin

consts
iff-keep :: [*bool*, *bool*] => *bool*
iff-unfold :: [*bool*, *bool*] => *bool*

hide *const iff-keep iff-unfold*

⟨*ML*⟩

end

46 Examples for the 'refute' command

theory *Refute-Examples* **imports** *Main*
begin

refute-params [*satsolver=dpll*]

lemma $P \wedge Q$
⟨*proof*⟩
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

$\langle proof \rangle$

46.1 Examples and Test Cases

46.1.1 Propositional logic

lemma *True*
  refute
  $\langle proof \rangle$

lemma *False*
  refute
$\langle proof \rangle$

lemma P
  refute
$\langle proof \rangle$

lemma $\sim P$
  refute
$\langle proof \rangle$

lemma $P \ \& \ Q$
  refute
$\langle proof \rangle$

lemma $P \mid Q$
  refute
$\langle proof \rangle$

lemma $P \longrightarrow Q$
  refute
$\langle proof \rangle$

lemma $(P::bool) = Q$
  refute
$\langle proof \rangle$

lemma $(P \mid Q) \longrightarrow (P \ \& \ Q)$
  refute
$\langle proof \rangle$

46.1.2 Predicate logic

lemma $P\ x\ y\ z$
  refute
$\langle proof \rangle$

lemma $P\ x\ y \longrightarrow P\ y\ x$
  refute

⟨*proof*⟩

lemma $P\ (f\ (f\ x)) \longrightarrow P\ x \longrightarrow P\ (f\ x)$
 refute
⟨*proof*⟩

46.1.3 Equality

lemma $P = \mathit{True}$
 refute
⟨*proof*⟩

lemma $P = \mathit{False}$
 refute
⟨*proof*⟩

lemma $x = y$
 refute
⟨*proof*⟩

lemma $f\ x = g\ x$
 refute
⟨*proof*⟩

lemma $(f::'a \Rightarrow 'b) = g$
 refute
⟨*proof*⟩

lemma $(f::('d \Rightarrow 'd) \Rightarrow ('c \Rightarrow 'd)) = g$
 refute
⟨*proof*⟩

lemma $\mathit{distinct}\ [a,b]$
 refute
 ⟨*proof*⟩
 refute
⟨*proof*⟩

46.1.4 First-Order Logic

lemma $\exists\, x.\ P\ x$
 refute
⟨*proof*⟩

lemma $\forall\, x.\ P\ x$
 refute
⟨*proof*⟩

lemma $EX!\ x.\ P\ x$
 refute

$\langle proof \rangle$

lemma *Ex P*
  refute
$\langle proof \rangle$

lemma *All P*
  refute
$\langle proof \rangle$

lemma *Ex1 P*
  refute
$\langle proof \rangle$

lemma $(\exists x.\ P\ x) \longrightarrow (\forall x.\ P\ x)$
  refute
$\langle proof \rangle$

lemma $(\forall x.\ \exists y.\ P\ x\ y) \longrightarrow (\exists y.\ \forall x.\ P\ x\ y)$
  refute
$\langle proof \rangle$

lemma $(\exists x.\ P\ x) \longrightarrow (EX!\ x.\ P\ x)$
  refute
$\langle proof \rangle$

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*]
  $\langle proof \rangle$

"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
$\langle proof \rangle$

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
$\langle proof \rangle$

"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
$\langle proof \rangle$

The "Drinker's theorem" ...

lemma $\exists x.\ f\ x = g\ x \longrightarrow f = g$

refute [*maxsize=4*]
⟨*proof*⟩

... and an incorrect version of it

lemma $(\exists x.\ f\ x = g\ x) \longrightarrow f = g$
refute
⟨*proof*⟩

"Every function has a fixed point."

lemma $\exists x.\ f\ x = x$
refute
⟨*proof*⟩

"Function composition is commutative."

lemma $f\ (g\ x) = g\ (f\ x)$
refute
⟨*proof*⟩

"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
⟨*proof*⟩

46.1.5 Higher-Order Logic

lemma $\exists P.\ P$
refute
⟨*proof*⟩

lemma $\forall P.\ P$
refute
⟨*proof*⟩

lemma $EX!\ P.\ P$
refute
⟨*proof*⟩

lemma $EX!\ P.\ P\ x$
refute
⟨*proof*⟩

lemma $P\ Q \mid Q\ x$
refute
⟨*proof*⟩

lemma $x \neq All$
refute
⟨*proof*⟩

lemma $x \neq Ex$
  refute
$\langle proof \rangle$

lemma $x \neq Ex1$
  refute
$\langle proof \rangle$

"The transitive closure 'T' of an arbitrary relation 'P' is non-empty."

constdefs
  $trans :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  $trans\ P == (ALL\ x\ y\ z.\ P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z)$
  $subset :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  $subset\ P\ Q == (ALL\ x\ y.\ P\ x\ y \longrightarrow Q\ x\ y)$
  $trans\text{-}closure :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  $trans\text{-}closure\ P\ Q == (subset\ Q\ P)\ \&\ (trans\ P)\ \&\ (ALL\ R.\ subset\ Q\ R \longrightarrow trans\ R \longrightarrow subset\ P\ R)$

lemma $trans\text{-}closure\ T\ P \longrightarrow (\exists\, x\ y.\ T\ x\ y)$
  refute
$\langle proof \rangle$

"The union of transitive closures is equal to the transitive closure of unions."

lemma $(\forall\, x\ y.\ (P\ x\ y\ |\ R\ x\ y) \longrightarrow T\ x\ y) \longrightarrow trans\ T \longrightarrow (\forall\, Q.\ (\forall\, x\ y.\ (P\ x\ y\ |\ R\ x\ y) \longrightarrow Q\ x\ y) \longrightarrow trans\ Q \longrightarrow subset\ T\ Q)$
    $\longrightarrow trans\text{-}closure\ TP\ P$
    $\longrightarrow trans\text{-}closure\ TR\ R$
    $\longrightarrow (T\ x\ y = (TP\ x\ y\ |\ TR\ x\ y))$
  refute
$\langle proof \rangle$

"Every surjective function is invertible."

lemma $(\forall\, y.\ \exists\, x.\ y = f\ x) \longrightarrow (\exists\, g.\ \forall\, x.\ g\ (f\ x) = x)$
  refute
$\langle proof \rangle$

"Every invertible function is surjective."

lemma $(\exists\, g.\ \forall\, x.\ g\ (f\ x) = x) \longrightarrow (\forall\, y.\ \exists\, x.\ y = f\ x)$
  refute
$\langle proof \rangle$

Every point is a fixed point of some function.

lemma $\exists\, f.\ f\ x = x$
  refute [$maxsize=4$]
  $\langle proof \rangle$

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
⟨*proof*⟩

... 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*]
⟨*proof*⟩

lemma $(\forall x.\ EX!y.\ P\ x\ y) \longrightarrow (EX!f.\ \forall x.\ P\ x\ (f\ x))$
refute [*maxsize=2*]
⟨*proof*⟩

46.1.6 Meta-logic

lemma $!!x.\ P\ x$
refute
⟨*proof*⟩

lemma $f\ x == g\ x$
refute
⟨*proof*⟩

lemma $P \Longrightarrow Q$
refute
⟨*proof*⟩

lemma $[\![\ P;\ Q;\ R\ ]\!] \Longrightarrow S$
refute
⟨*proof*⟩

lemma $(x == all) \Longrightarrow False$
refute
⟨*proof*⟩

lemma $(x == (op\ ==)) \Longrightarrow False$
refute
⟨*proof*⟩

lemma $(x == (op \Longrightarrow)) \Longrightarrow False$
refute
⟨*proof*⟩

46.1.7 Schematic variables

lemma $?P$
refute
⟨*proof*⟩

lemma $x = ?y$
refute

$\langle proof \rangle$

46.1.8 Abstractions

lemma $(\lambda x.\ x) = (\lambda x.\ y)$
refute
$\langle proof \rangle$

lemma $(\lambda f.\ f\ x) = (\lambda f.\ True)$
refute
$\langle proof \rangle$

lemma $(\lambda x.\ x) = (\lambda y.\ y)$
refute
$\langle proof \rangle$

46.1.9 Sets

lemma $P\ (A::'a\ set)$
refute
$\langle proof \rangle$

lemma $P\ (A::'a\ set\ set)$
refute
$\langle proof \rangle$

lemma $\{x.\ P\ x\} = \{y.\ P\ y\}$
refute
$\langle proof \rangle$

lemma $x : \{x.\ P\ x\}$
refute
$\langle proof \rangle$

lemma $P\ op:$
refute
$\langle proof \rangle$

lemma $P\ (op:\ x)$
refute
$\langle proof \rangle$

lemma $P\ Collect$
refute
$\langle proof \rangle$

lemma $A\ Un\ B = A\ Int\ B$
refute
$\langle proof \rangle$

lemma $(A\ Int\ B)\ Un\ C = (A\ Un\ C)\ Int\ B$
  refute
⟨*proof*⟩

lemma $Ball\ A\ P \longrightarrow Bex\ A\ P$
  refute
⟨*proof*⟩

46.1.10 arbitrary

lemma *arbitrary*
  refute
⟨*proof*⟩

lemma $P\ arbitrary$
  refute
⟨*proof*⟩

lemma $arbitrary\ x$
  refute
⟨*proof*⟩

lemma $arbitrary\ arbitrary$
  refute
⟨*proof*⟩

46.1.11 The

lemma $The\ P$
  refute
⟨*proof*⟩

lemma $P\ The$
  refute
⟨*proof*⟩

lemma $P\ (The\ P)$
  refute
⟨*proof*⟩

lemma $(THE\ x.\ x{=}y) = z$
  refute
⟨*proof*⟩

lemma $Ex\ P \longrightarrow P\ (The\ P)$
  refute
⟨*proof*⟩

46.1.12 Eps

lemma *Eps P*
  refute
⟨*proof*⟩

lemma *P Eps*
  refute
⟨*proof*⟩

lemma *P* (*Eps P*)
  refute
⟨*proof*⟩

lemma (*SOME x. x=y*) = *z*
  refute
⟨*proof*⟩

lemma *Ex P* $\longrightarrow$ *P* (*Eps P*)
  refute [*maxsize=3*]
  ⟨*proof*⟩

46.1.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of *'a*:

typedef *'a myTdef* = *insert* (*arbitrary*::*'a*) (*arbitrary*::*'a set*)
  ⟨*proof*⟩

lemma (*x*::*'a myTdef*) = *y*
  refute
⟨*proof*⟩

typedecl *myTdecl*

typedef *'a T-bij* = $\{(f::'a \Rightarrow 'a).\ \forall y.\ \exists !x.\ f\ x = y\}$
  ⟨*proof*⟩

lemma *P* (*f*::(*myTdecl myTdef*) *T-bij*)
  refute
⟨*proof*⟩

46.1.14 Inductive datatypes

With *quick-and-dirty* set, the datatype package does not generate certain axioms for recursion operators. Without these axioms, refute may find spurious countermodels.

unit

lemma *P* (*x*::*unit*)

refute
⟨*proof*⟩

lemma $\forall x{::}unit.\ P\ x$
refute
⟨*proof*⟩

lemma $P\ ()$
refute
⟨*proof*⟩

lemma $unit\text{-}rec\ u\ x = u$
refute
⟨*proof*⟩

lemma $P\ (unit\text{-}rec\ u\ x)$
refute
⟨*proof*⟩

lemma $P\ (case\ x\ of\ ()\Rightarrow u)$
refute
⟨*proof*⟩

option

lemma $P\ (x{::}'a\ option)$
refute
⟨*proof*⟩

lemma $\forall x{::}'a\ option.\ P\ x$
refute
⟨*proof*⟩

lemma $P\ None$
refute
⟨*proof*⟩

lemma $P\ (Some\ x)$
refute
⟨*proof*⟩

lemma $option\text{-}rec\ n\ s\ None = n$
refute
⟨*proof*⟩

lemma $option\text{-}rec\ n\ s\ (Some\ x) = s\ x$
refute [*maxsize=4*]
⟨*proof*⟩

lemma $P\ (option\text{-}rec\ n\ s\ x)$

refute
⟨*proof*⟩

lemma P (*case* x *of* *None* $\Rightarrow$ n | *Some* u $\Rightarrow$ s u)
refute
⟨*proof*⟩

*

lemma P $(x::'a*'b)$
refute
⟨*proof*⟩

lemma $\forall x::'a*'b.\ P\ x$
refute
⟨*proof*⟩

lemma P $(x,\ y)$
refute
⟨*proof*⟩

lemma P (*fst* x)
refute
⟨*proof*⟩

lemma P (*snd* x)
refute
⟨*proof*⟩

lemma P *Pair*
refute
⟨*proof*⟩

lemma *prod-rec* p $(a,\ b) = p\ a\ b$
refute [*maxsize=2*]
⟨*proof*⟩

lemma P (*prod-rec* p x)
refute
⟨*proof*⟩

lemma P (*case* x *of* *Pair* a b $\Rightarrow$ p a b)
refute
⟨*proof*⟩

+

lemma P $(x::'a+'b)$
refute
⟨*proof*⟩

lemma $\forall x::'a+'b.\ P\ x$
  refute
$\langle proof \rangle$

lemma $P\ (Inl\ x)$
  refute
$\langle proof \rangle$

lemma $P\ (Inr\ x)$
  refute
$\langle proof \rangle$

lemma $P\ Inl$
  refute
$\langle proof \rangle$

lemma $sum\text{-}rec\ l\ r\ (Inl\ x) = l\ x$
  refute $[maxsize=3]$
  $\langle proof \rangle$

lemma $sum\text{-}rec\ l\ r\ (Inr\ x) = r\ x$
  refute $[maxsize=3]$
  $\langle proof \rangle$

lemma $P\ (sum\text{-}rec\ l\ r\ x)$
  refute
$\langle proof \rangle$

lemma $P\ (case\ x\ of\ Inl\ a \Rightarrow l\ a\ |\ Inr\ b \Rightarrow r\ b)$
  refute
$\langle proof \rangle$

Non-recursive datatypes

datatype $T1 = A\ |\ B$

lemma $P\ (x::T1)$
  refute
$\langle proof \rangle$

lemma $\forall x::T1.\ P\ x$
  refute
$\langle proof \rangle$

lemma $P\ A$
  refute
$\langle proof \rangle$

lemma $P\ B$
  refute

⟨*proof*⟩

lemma $T1\text{-}rec\ a\ b\ A = a$
  refute
  ⟨*proof*⟩

lemma $T1\text{-}rec\ a\ b\ B = b$
  refute
  ⟨*proof*⟩

lemma $P\ (T1\text{-}rec\ a\ b\ x)$
  refute
⟨*proof*⟩

lemma $P\ (case\ x\ of\ A \Rightarrow a \mid B \Rightarrow b)$
  refute
⟨*proof*⟩

datatype $'a\ T2 = C\ T1 \mid D\ 'a$

lemma $P\ (x{::}'a\ T2)$
  refute
⟨*proof*⟩

lemma $\forall x{::}'a\ T2.\ P\ x$
  refute
⟨*proof*⟩

lemma $P\ D$
  refute
⟨*proof*⟩

lemma $T2\text{-}rec\ c\ d\ (C\ x) = c\ x$
  refute [*maxsize=4*]
  ⟨*proof*⟩

lemma $T2\text{-}rec\ c\ d\ (D\ x) = d\ x$
  refute [*maxsize=4*]
  ⟨*proof*⟩

lemma $P\ (T2\text{-}rec\ c\ d\ x)$
  refute
⟨*proof*⟩

lemma $P\ (case\ x\ of\ C\ u \Rightarrow c\ u \mid D\ v \Rightarrow d\ v)$
  refute
⟨*proof*⟩

datatype $('a,'b)\ T3 = E\ 'a \Rightarrow 'b$

lemma $P\ (x::('a,'b)\ T3)$
  refute
⟨*proof*⟩

lemma $\forall x::('a,'b)\ T3.\ P\ x$
  refute
⟨*proof*⟩

lemma $P\ E$
  refute
⟨*proof*⟩

lemma $T3\text{-}rec\ e\ (E\ x) = e\ x$
  refute [*maxsize=2*]
  ⟨*proof*⟩

lemma $P\ (T3\text{-}rec\ e\ x)$
  refute
⟨*proof*⟩

lemma $P\ (case\ x\ of\ E\ f \Rightarrow e\ f)$
  refute
⟨*proof*⟩

Recursive datatypes

nat

lemma $P\ (x::nat)$
  refute
⟨*proof*⟩

lemma $\forall x::nat.\ P\ x$
  refute
⟨*proof*⟩

lemma $P\ (Suc\ 0)$
  refute
⟨*proof*⟩

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
⟨*proof*⟩

lemma $nat\text{-}rec\ zero\ suc\ 0 = zero$
  refute
  ⟨*proof*⟩

lemma $nat\text{-}rec\ zero\ suc\ (Suc\ x) = suc\ x\ (nat\text{-}rec\ zero\ suc\ x)$

refute [*maxsize=2*]
⟨*proof*⟩

lemma *P* (*nat-rec zero suc x*)
refute
⟨*proof*⟩

lemma *P* (*case x of 0* ⇒ *zero* | *Suc n* ⇒ *suc n*)
refute
⟨*proof*⟩

'a list

lemma *P* (*xs*::*'a list*)
refute
⟨*proof*⟩

lemma ∀ *xs*::*'a list*. *P xs*
refute
⟨*proof*⟩

lemma *P* [*x*, *y*]
refute
⟨*proof*⟩

lemma *list-rec nil cons* [] = *nil*
refute [*maxsize=3*]
⟨*proof*⟩

lemma *list-rec nil cons* (*x*#*xs*) = *cons x xs* (*list-rec nil cons xs*)
refute [*maxsize=2*]
⟨*proof*⟩

lemma *P* (*list-rec nil cons xs*)
refute
⟨*proof*⟩

lemma *P* (*case x of Nil* ⇒ *nil* | *Cons a b* ⇒ *cons a b*)
refute
⟨*proof*⟩

lemma (*xs*::*'a list*) = *ys*
refute
⟨*proof*⟩

lemma *a* # *xs* = *b* # *xs*
refute
⟨*proof*⟩

datatype *BitList* = *BitListNil* | *Bit0 BitList* | *Bit1 BitList*

lemma P $(x::BitList)$
  refute
⟨*proof*⟩

lemma $\forall x::BitList.\ P\ x$
  refute
⟨*proof*⟩

lemma P $(Bit0\ (Bit1\ BitListNil))$
  refute
⟨*proof*⟩

lemma $BitList\text{-}rec\ nil\ bit0\ bit1\ BitListNil = nil$
  refute $[maxsize=4]$
  ⟨*proof*⟩

lemma $BitList\text{-}rec\ nil\ bit0\ bit1\ (Bit0\ xs) = bit0\ xs\ (BitList\text{-}rec\ nil\ bit0\ bit1\ xs)$
  refute $[maxsize=2]$
  ⟨*proof*⟩

lemma $BitList\text{-}rec\ nil\ bit0\ bit1\ (Bit1\ xs) = bit1\ xs\ (BitList\text{-}rec\ nil\ bit0\ bit1\ xs)$
  refute $[maxsize=2]$
  ⟨*proof*⟩

lemma P $(BitList\text{-}rec\ nil\ bit0\ bit1\ x)$
  refute
⟨*proof*⟩

datatype $'a\ BinTree = Leaf\ 'a\ |\ Node\ 'a\ BinTree\ 'a\ BinTree$

lemma P $(x::'a\ BinTree)$
  refute
⟨*proof*⟩

lemma $\forall x::'a\ BinTree.\ P\ x$
  refute
⟨*proof*⟩

lemma P $(Node\ (Leaf\ x)\ (Leaf\ y))$
  refute
⟨*proof*⟩

lemma $BinTree\text{-}rec\ l\ n\ (Leaf\ x) = l\ x$
  refute $[maxsize=1]$
  ⟨*proof*⟩

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 $[maxsize=1]$

⟨*proof*⟩

lemma P (*BinTree-rec l n x*)
refute
⟨*proof*⟩

lemma P (*case x of Leaf a* ⇒ *l a* | *Node a b* ⇒ *n a b*)
refute
⟨*proof*⟩

Mutually recursive datatypes

datatype *'a aexp* = *Number 'a* | *ITE 'a bexp 'a aexp 'a aexp*
and *'a bexp* = *Equal 'a aexp 'a aexp*

lemma P (*x*::*'a aexp*)
refute
⟨*proof*⟩

lemma ∀ *x*::*'a aexp*. *P x*
refute
⟨*proof*⟩

lemma P (*ITE* (*Equal* (*Number x*) (*Number y*)) (*Number x*) (*Number y*))
refute
⟨*proof*⟩

lemma P (*x*::*'a bexp*)
refute
⟨*proof*⟩

lemma ∀ *x*::*'a bexp*. *P x*
refute
⟨*proof*⟩

lemma *aexp-bexp-rec-1 number ite equal* (*Number x*) = *number x*
refute [*maxsize=1*]
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma P (*aexp-bexp-rec-1 number ite equal x*)
refute
⟨*proof*⟩

lemma P (*case x of Number a* ⇒ *number a* | *ITE b a1 a2* ⇒ *ite b a1 a2*)

refute
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *P* (*aexp-bexp-rec-2 number ite equal x*)
refute
⟨*proof*⟩

lemma *P* (*case x of Equal a1 a2* ⇒ *equal a1 a2*)
refute
⟨*proof*⟩

datatype *X* = *A* | *B X* | *C Y*
and *Y* = *D X* | *E Y* | *F*

lemma *P* (*x*::*X*)
refute
⟨*proof*⟩

lemma *P* (*y*::*Y*)
refute
⟨*proof*⟩

lemma *P* (*B* (*B A*))
refute
⟨*proof*⟩

lemma *P* (*B* (*C F*))
refute
⟨*proof*⟩

lemma *P* (*C* (*D A*))
refute
⟨*proof*⟩

lemma *P* (*C* (*E F*))
refute
⟨*proof*⟩

lemma *P* (*D* (*B A*))
refute
⟨*proof*⟩

lemma *P* (*D* (*C F*))
refute

⟨*proof*⟩

lemma P $(E$ $(D$ $A))$
 refute
⟨*proof*⟩

lemma P $(E$ $(E$ $F))$
 refute
⟨*proof*⟩

lemma P $(C$ $(D$ $(C$ $F)))$
 refute
⟨*proof*⟩

lemma *X-Y-rec-1* a b c d e f $A = a$
 refute [*maxsize=3*]
 ⟨*proof*⟩

lemma *X-Y-rec-1* a b c d e f $(B$ $x) = b$ x (*X-Y-rec-1* a b c d e f x)
 refute [*maxsize=1*]
 ⟨*proof*⟩

lemma *X-Y-rec-1* a b c d e f $(C$ $y) = c$ y (*X-Y-rec-2* a b c d e f y)
 refute [*maxsize=1*]
 ⟨*proof*⟩

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*]
 ⟨*proof*⟩

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*]
 ⟨*proof*⟩

lemma *X-Y-rec-2* a b c d e f $F = f$
 refute [*maxsize=3*]
 ⟨*proof*⟩

lemma P (*X-Y-rec-1* a b c d e f x)
 refute
⟨*proof*⟩

lemma P (*X-Y-rec-2* a b c d e f y)
 refute
⟨*proof*⟩

Other datatype examples

Indirect recursion is implemented via mutual recursion.

datatype *XOpt* = *CX XOpt option* | *DX bool* $\Rightarrow$ *XOpt option*

lemma *P* (*x*::*XOpt*)
  refute
⟨*proof*⟩

lemma *P* (*CX None*)
  refute
⟨*proof*⟩

lemma *P* (*CX* (*Some* (*CX None*)))
  refute
⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *XOpt-rec-1 cx dx n1 s1 n2 s2* (*DX x*) = *dx x* (λ*b*. *XOpt-rec-3 cx dx n1 s1 n2 s2* (*x b*))
  refute [*maxsize=1*]
  ⟨*proof*⟩

lemma *XOpt-rec-2 cx dx n1 s1 n2 s2 None* = *n1*
  refute [*maxsize=2*]
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *XOpt-rec-3 cx dx n1 s1 n2 s2 None* = *n2*
  refute [*maxsize=2*]
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *P* (*XOpt-rec-1 cx dx n1 s1 n2 s2 x*)
  refute
⟨*proof*⟩

lemma *P* (*XOpt-rec-2 cx dx n1 s1 n2 s2 x*)
  refute
⟨*proof*⟩

lemma *P* (*XOpt-rec-3 cx dx n1 s1 n2 s2 x*)
  refute
⟨*proof*⟩

datatype *'a YOpt* = *CY* (*'a* ⇒ *'a YOpt*) *option*

lemma *P* (*x*::*'a YOpt*)
  refute
⟨*proof*⟩

lemma *P* (*CY None*)
  refute
⟨*proof*⟩

lemma *P* (*CY* (*Some* (*λa. CY None*)))
  refute
⟨*proof*⟩

lemma *YOpt-rec-1 cy n s* (*CY x*) = *cy x* (*YOpt-rec-2 cy n s x*)
  refute [*maxsize=1*]
  ⟨*proof*⟩

lemma *YOpt-rec-2 cy n s None* = *n*
  refute [*maxsize=2*]
  ⟨*proof*⟩

lemma *YOpt-rec-2 cy n s* (*Some x*) = *s x* (*λa. YOpt-rec-1 cy n s* (*x a*))
  refute [*maxsize=1*]
  ⟨*proof*⟩

lemma *P* (*YOpt-rec-1 cy n s x*)
  refute
⟨*proof*⟩

lemma *P* (*YOpt-rec-2 cy n s x*)
  refute
⟨*proof*⟩

datatype *Trie* = *TR Trie list*

lemma *P* (*x*::*Trie*)
  refute
⟨*proof*⟩

lemma ∀ *x*::*Trie*. *P x*
  refute
⟨*proof*⟩

lemma *P* (*TR* [*TR* []])

refute
⟨*proof*⟩

lemma *Trie-rec-1 tr nil cons* (*TR x*) = *tr x* (*Trie-rec-2 tr nil cons x*)
refute [*maxsize=1*]
⟨*proof*⟩

lemma *Trie-rec-2 tr nil cons* [] = *nil*
refute [*maxsize=3*]
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *P* (*Trie-rec-1 tr nil cons x*)
refute
⟨*proof*⟩

lemma *P* (*Trie-rec-2 tr nil cons x*)
refute
⟨*proof*⟩

datatype *InfTree* = *Leaf* | *Node nat* ⇒ *InfTree*

lemma *P* (*x*::*InfTree*)
refute
⟨*proof*⟩

lemma ∀ *x*::*InfTree*. *P x*
refute
⟨*proof*⟩

lemma *P* (*Node* (*λn. Leaf*))
refute
⟨*proof*⟩

lemma *InfTree-rec leaf node Leaf* = *leaf*
refute [*maxsize=2*]
⟨*proof*⟩

lemma *InfTree-rec leaf node* (*Node x*) = *node x* (*λn. InfTree-rec leaf node* (*x n*))
refute [*maxsize=1*]
⟨*proof*⟩

lemma *P* (*InfTree-rec leaf node x*)
refute
⟨*proof*⟩

datatype $'a$ *lambda* = *Var* $'a$ | *App* $'a$ *lambda* $'a$ *lambda* | *Lam* $'a \Rightarrow 'a$ *lambda*

lemma P (x::$'a$ *lambda*)
  refute
⟨*proof*⟩

lemma $\forall\, x$::$'a$ *lambda*. $P\ x$
  refute
⟨*proof*⟩

lemma P (*Lam* (λa. *Var* a))
  refute
⟨*proof*⟩

lemma *lambda-rec var app lam* (*Var* x) = *var* x
  refute [*maxsize=1*]
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *lambda-rec var app lam* (*Lam* x) = *lam* x (λa. *lambda-rec var app lam* (x a))
  refute [*maxsize=1*]
  ⟨*proof*⟩

lemma P (*lambda-rec* v a l x)
  refute
⟨*proof*⟩

Taken from "Inductive datatypes in HOL", p.8:

datatype ($'a$, $'b$) T = C $'a \Rightarrow$ *bool* | D $'b$ *list*
datatype $'c$ U = E ($'c$, $'c$ U) T

lemma P (x::$'c$ U)
  refute
⟨*proof*⟩

lemma $\forall\, x$::$'c$ U. $P\ x$
  refute
⟨*proof*⟩

lemma P (E (C (λa. *True*)))
  refute
⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *U-rec-2 e c d nil cons* (*C x*) = *c x*
  refute [*maxsize=1*]
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *U-rec-3 e c d nil cons* [] = *nil*
  refute [*maxsize=2*]
  ⟨*proof*⟩

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*]
  ⟨*proof*⟩

lemma *P* (*U-rec-1 e c d nil cons x*)
  refute
⟨*proof*⟩

lemma *P* (*U-rec-2 e c d nil cons x*)
  refute
⟨*proof*⟩

lemma *P* (*U-rec-3 e c d nil cons x*)
  refute
⟨*proof*⟩

46.1.15 Records

record (*'a*, *'b*) *point* =
  xpos :: *'a*
  ypos :: *'b*

lemma (*x*::(*'a*, *'b*) *point*) = *y*
  refute
⟨*proof*⟩

record (*'a*, *'b*, *'c*) *extpoint* = (*'a*, *'b*) *point* +
  ext :: *'c*

lemma (*x*::(*'a*, *'b*, *'c*) *extpoint*) = *y*
  refute
⟨*proof*⟩

46.1.16 Inductively defined sets

inductive-set *arbitrarySet* :: *'a set*
where
  arbitrary : *arbitrarySet*

lemma *x* : *arbitrarySet*
  refute
⟨*proof*⟩

inductive-set *evenCard* :: *'a set set*
where
  {} : *evenCard*
| ⟦ *S* : *evenCard*; *x* ∉ *S*; *y* ∉ *S*; *x* ≠ *y* ⟧ ⟹ *S* ∪ {*x*, *y*} : *evenCard*

lemma *S* : *evenCard*
  refute
⟨*proof*⟩

inductive-set
  even :: *nat set*
  and *odd* :: *nat set*
where
  0 : *even*
| *n* : *even* ⟹ *Suc n* : *odd*
| *n* : *odd* ⟹ *Suc n* : *even*

lemma *n* : *odd*

⟨*proof*⟩

consts *f* :: *'a* ⇒ *'a*

inductive-set
  a-even :: *'a set*
  and *a-odd* :: *'a set*
where
  arbitrary : *a-even*
| *x* : *a-even* ⟹ *f x* : *a-odd*
| *x* : *a-odd* ⟹ *f x* : *a-even*

lemma *x* : *a-odd*
  refute — finds a model of size 2, as expected
⟨*proof*⟩

46.1.17 Examples involving special functions

lemma *card x* = *0*
  refute
⟨*proof*⟩

lemma *finite x*
  refute — no finite countermodel exists
⟨*proof*⟩

lemma $(x::nat) + y = 0$
  refute
⟨*proof*⟩

lemma $(x::nat) = x + x$
  refute
⟨*proof*⟩

lemma $(x::nat) - y + y = x$
  refute
⟨*proof*⟩

lemma $(x::nat) = x * x$
  refute
⟨*proof*⟩

lemma $(x::nat) < x + y$
  refute
⟨*proof*⟩

lemma *xs* @ [] = *ys* @ []
  refute
⟨*proof*⟩

lemma *xs* @ *ys* = *ys* @ *xs*
  refute
⟨*proof*⟩

lemma *f* (*lfp f*) = *lfp f*
  refute
⟨*proof*⟩

lemma *f* (*gfp f*) = *gfp f*
  refute
⟨*proof*⟩

lemma *lfp f* = *gfp f*
  refute
⟨*proof*⟩

46.1.18 Axiomatic type classes and overloading

A type class without axioms:

axclass *classA*

lemma $P\ (x::'a::classA)$
  refute
$\langle proof \rangle$

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
$\langle proof \rangle$

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
$\langle proof \rangle$

lemma $\exists\, x\ y.\ (x::'a::classC) \neq y$
  refute — no countermodel exists
$\langle proof \rangle$

A type class for which a constant is defined:

consts
  classD-const :: $'a \Rightarrow 'a$

axclass *classD* < *type*
  classD-ax: *classD-const* (*classD-const* x) = *classD-const* x

lemma $P\ (x::'a::classD)$
  refute
$\langle proof \rangle$

A type class with multiple superclasses:

axclass *classE* < *classC*, *classD*

lemma $P\ (x::'a::classE)$
  refute
$\langle proof \rangle$

lemma $P\ (x::'a::\{classB,\ classE\})$
  refute
$\langle proof \rangle$

OFCLASS:

lemma *OFCLASS*(*'a*::*type*, *type-class*)
refute — no countermodel exists
⟨*proof*⟩

lemma *OFCLASS*(*'a*::*classC*, *type-class*)
refute — no countermodel exists
⟨*proof*⟩

lemma *OFCLASS*(*'a*, *classB-class*)
refute — no countermodel exists
⟨*proof*⟩

lemma *OFCLASS*(*'a*::*type*, *classC-class*)
refute
⟨*proof*⟩

Overloading:

consts *inverse* :: *'a* ⇒ *'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
⟨*proof*⟩

lemma *P* (*inverse* (*S*::*'a set*))
refute
⟨*proof*⟩

lemma *P* (*inverse* (*p*::*'a*×*'b*))
refute
⟨*proof*⟩

refute-params [*satsolver*=*auto*]

end

47 Examples for the 'quickcheck' command

theory *Quickcheck-Examples* **imports** *Main* **begin**

The 'quickcheck' command allows to find counterexamples by evaluating formulae under an assignment of free variables to random values. In contrast to 'refute', it can deal with inductive datatypes, but cannot handle quantifiers.

47.1 Lists

theorem *map g* (*map f xs*) = *map* (*g o f*) *xs*
  quickcheck
  ⟨*proof*⟩

theorem *map g* (*map f xs*) = *map* (*f o g*) *xs*
  quickcheck
  ⟨*proof*⟩

theorem *rev* (*xs* @ *ys*) = *rev ys* @ *rev xs*
  quickcheck
  ⟨*proof*⟩

theorem *rev* (*xs* @ *ys*) = *rev xs* @ *rev ys*
  quickcheck
  ⟨*proof*⟩

theorem *rev* (*rev xs*) = *xs*
  quickcheck
  ⟨*proof*⟩

theorem *rev xs* = *xs*
  quickcheck
  ⟨*proof*⟩

consts
  occurs :: *'a* ⇒ *'a list* ⇒ *nat*
primrec
  occurs a [] = *0*
  occurs a (*x#xs*) = (*if* (*x=a*) *then Suc*(*occurs a xs*) *else occurs a xs*)

consts
  del1 :: *'a* ⇒ *'a list* ⇒ *'a list*
primrec
  del1 a [] = []
  del1 a (*x#xs*) = (*if* (*x=a*) *then xs else* (*x#del1 a xs*))

lemma *Suc* (*occurs a* (*del1 a xs*)) = *occurs a xs*
  — Wrong. Precondition needed.
  quickcheck
  ⟨*proof*⟩

lemma *xs* ~= [] ⟶ *Suc* (*occurs a* (*del1 a xs*)) = *occurs a xs*
  quickcheck
    — Also wrong.
  ⟨*proof*⟩

lemma *0* < *occurs a xs* ⟶ *Suc* (*occurs a* (*del1 a xs*)) = *occurs a xs*

quickcheck
⟨*proof*⟩

consts
$replace :: 'a \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
primrec
$replace\ a\ b\ []\ =\ []$
$replace\ a\ b\ (x\#xs) = (if\ (x=a)\ then\ (b\#(replace\ a\ b\ xs))$
$\qquad\qquad else\ (x\#(replace\ a\ b\ xs)))$

lemma $occurs\ a\ xs = occurs\ b\ (replace\ a\ b\ xs)$
quickcheck
— Wrong. Precondition needed.
⟨*proof*⟩

lemma $occurs\ b\ xs = 0 \lor a=b \longrightarrow occurs\ a\ xs = occurs\ b\ (replace\ a\ b\ xs)$
quickcheck
⟨*proof*⟩

47.2 Trees

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

consts
$leaves :: 'a\ tree \Rightarrow 'a\ list$
primrec
$leaves\ Twig = []$
$leaves\ (Leaf\ a) = [a]$
$leaves\ (Branch\ l\ r) = (leaves\ l)\ @\ (leaves\ r)$

consts
$plant :: 'a\ list \Rightarrow 'a\ tree$
primrec
$plant\ [] = Twig$
$plant\ (x\#xs) = Branch\ (Leaf\ x)\ (plant\ xs)$

consts
$mirror :: 'a\ tree \Rightarrow 'a\ tree$
primrec
$mirror\ (Twig) = Twig$
$mirror\ (Leaf\ a) = Leaf\ a$
$mirror\ (Branch\ l\ r) = Branch\ (mirror\ r)\ (mirror\ l)$

theorem $plant\ (rev\ (leaves\ xt)) = mirror\ xt$
quickcheck
— Wrong!
⟨*proof*⟩

theorem $plant((leaves\ xt)\ @\ (leaves\ yt)) = Branch\ xt\ yt$

quickcheck
— Wrong!
⟨*proof*⟩

datatype *'a ntree* = *Tip 'a* | *Node 'a 'a ntree 'a ntree*

consts
inOrder :: *'a ntree* ⇒ *'a list*
primrec
inOrder (*Tip a*)= [*a*]
inOrder (*Node f x y*) = (*inOrder x*)@[*f*]@(*inOrder y*)

consts
root :: *'a ntree* ⇒ *'a*
primrec
root (*Tip a*) = *a*
root (*Node f x y*) = *f*

theorem *hd*(*inOrder xt*) = *root xt*
quickcheck
— Wrong!
⟨*proof*⟩

end

References

[1] M. J. C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[2] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

[3] K. McMillan. Lecture notes on verification of digital and hybrid systems. NATO summer school, http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial/toc.html.

[4] K. McMillan. *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[5] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in Higher-Order Logic. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *LNCS*, 1998.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.

[7] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[8] L. C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[9] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.

[10] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, September 2001. Submitted.

[11] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2001. Part of the Isabelle distribution, http://isabelle.in.tum.de/doc/isar-ref.pdf.

[12] M. Wenzel. Miscellaneous Isabelle/Isar examples for higher-order logic. Part of the Isabelle distribution, http://isabelle.in.tum.de/library/HOL/Isar_examples/document.pdf, 2001.