Miscellaneous HOL Examples

1st October 2005

Contents

1 Foundations of HOL

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

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

1.1 Pure Logic

classes *type*
defaultsort *type*

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

1.1.1 Basic logical connectives

judgment
Trueprop :: *o* ⇒ *prop* (- *5*)

consts

imp :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\longrightarrow$ *25*)
All :: $('a \Rightarrow o) \Rightarrow o$ (**binder** $\forall$ *10*)

axioms
impI [*intro*]: $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$
impE [*dest*, *trans*]: $A \longrightarrow B \Longrightarrow A \Longrightarrow B$
allI [*intro*]: $(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$
allE [*dest*]: $\forall x.\ P\ x \Longrightarrow P\ a$

1.1.2 Extensional equality

consts
equal :: $'a \Rightarrow 'a \Rightarrow o$ (**infixl** $=$ *50*)

axioms
refl [*intro*]: $x = x$
subst: $x = y \Longrightarrow P\ x \Longrightarrow P\ y$
ext [*intro*]: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow f = g$
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

constdefs
false :: o ($\bot$)
$\bot \equiv \forall A.\ A$
true :: o ($\top$)
$\top \equiv \bot \longrightarrow \bot$
not :: $o \Rightarrow o$ ($\neg$ - [*40*] *40*)
not $\equiv \lambda A.\ A \longrightarrow \bot$
conj :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\wedge$ *35*)
conj $\equiv \lambda A\ B.\ \forall C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C$
disj :: $o \Rightarrow o \Rightarrow o$ (**infixr** $\vee$ *30*)

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

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

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-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 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 $k :: nat \Rightarrow nat$

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 $part :: ('a \Rightarrow bool) * 'a\ list * 'a\ list * 'a\ list \Rightarrow 'a\ list * 'a\ list$
recdef $part$ *measure* $(\lambda(P, l, l1, l2).\ size\ l)$
$part\ (P, [], l1, l2) = (l1, l2)$
$part\ (P, h \# rst, l1, l2) =$
$(if\ P\ h\ then\ part\ (P, rst, h \# l1, l2)$
$else\ part\ (P, rst, l1, h \# l2))$

consts $fqsort :: ('a \Rightarrow 'a \Rightarrow bool) * 'a\ list \Rightarrow 'a\ list$
recdef (**permissive**) $fqsort$ *measure* $(size\ o\ snd)$
$fqsort\ (ord, []) = []$
$fqsort\ (ord, x \# rst) =$
$(let\ (less, more) = part\ ((\lambda y.\ ord\ y\ x), rst, ([], []))$
$in\ fqsort\ (ord, less)\ @\ [x]\ @\ fqsort\ (ord, more))$

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 $\lambda x.\ mapf\ x$ instead of *mapf*.

consts $mapf :: nat \Rightarrow nat\ list$
recdef (**permissive**) $mapf$ *measure* $(\lambda m.\ m)$
$mapf\ 0 = []$
$mapf\ (Suc\ n) = concat\ (map\ (\lambda x.\ mapf\ x)\ (replicate\ n\ n))$
(**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

3 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.

constdefs *indinv* :: $('a * 'a)$ *set* => $('a$ => $'b$ => *bool*) => (($'a$ => $'b$) => ($'a$ => $'b$)) => *bool*
indinv r S F == $\forall f\ x.\ (\forall y.\ (y,x) : r$ −−> $S\ y\ (f\ y))$ −−> $S\ x\ (F\ f\ x)$

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

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

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: $\forall f.\ \forall x{\in}D.\ (\forall y{\in}D.\ (y,x) \in r$ −−> $S\ y\ (wfrec\ r\ F\ y)$ & $f\ y = wfrec\ r\ F\ y)$
−−> $S\ x\ (wfrec\ r\ F\ x)$ & $F\ f\ x = wfrec\ r\ F\ x$ **and**
D: $x{\in}D$
shows $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x$ & $S\ x\ (wfrec\ r\ F\ x)$
⟨*proof*⟩

theorem *ind-fixpoint-lemma*:

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

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

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

end

4 Example use if an inductive invariant to solve termination conditions

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

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

consts $g :: nat \Rightarrow nat$

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

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

recdef-tc *g-tc*[*simp*]: *g*
⟨*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 $g' :: nat \Rightarrow nat$
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*: $\exists k.\ n = Suc\ (Suc\ (2{*}k)) \Longrightarrow g\text{-}even\ n = 3$
⟨*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*: $\exists k.\ n = 2{*}k + 4 \Longrightarrow g\text{-}even\ n = g\text{-}even\ (g\text{-}even\ (n - 2) - 1)$
⟨*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

5 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*.

constdefs
SC :: *nat list* => *nat*
SC l == *Suc* (*zeroHd l*)

```
  CONST :: nat => nat list => nat
  CONST k l == k

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

  COMP :: (nat list => nat) => (nat list => nat) list => nat list => nat
  COMP g fs l == g (map (λf. f l) fs)

  PREC :: (nat list => nat) => (nat list => nat) => nat list => nat
  PREC f g l ==
    case l of
      [] => 0
    | x # l' => nat-rec (f l') (λy r. g (r # y # l')) x
  — Note that g is applied first to PREC f g y and then to y!

consts PRIMREC :: (nat list => nat) set
inductive PRIMREC
  intros
    SC: SC ∈ PRIMREC
    CONST: CONST k ∈ PRIMREC
    PROJ: PROJ i ∈ PRIMREC
      COMP: g ∈ PRIMREC ==> fs ∈ lists PRIMREC ==> COMP g fs ∈
PRIMREC
    PREC: f ∈ PRIMREC ==> g ∈ PRIMREC ==> PREC f g ∈ PRIMREC
```

Useful special cases of evaluation

```
lemma SC [simp]: SC (x # l) = Suc x
  ⟨proof⟩

lemma CONST [simp]: CONST k l = k
  ⟨proof⟩

lemma PROJ-0 [simp]: PROJ 0 (x # l) = x
  ⟨proof⟩

lemma COMP-1 [simp]: COMP g [f] l = g [f l]
  ⟨proof⟩

lemma PREC-0 [simp]: PREC f g (0 # l) = f l
  ⟨proof⟩

lemma PREC-Suc [simp]: PREC f g (Suc x # l) = g (PREC f g (x # l) # x #
l)
  ⟨proof⟩
```

PROPERTY A 4

```
lemma less-ack2 [iff]: j < ack (i, j)
```

⟨*proof*⟩

PROPERTY A 5-, the single-step lemma

lemma *ack-less-ack-Suc2* [*iff*]: $ack(i, j) < ack\ (i, Suc\ j)$
⟨*proof*⟩

PROPERTY A 5, monotonicity for $<$

lemma *ack-less-mono2*: $j < k ==> ack\ (i, j) < ack\ (i, k)$
⟨*proof*⟩

PROPERTY A 5', monotonicity for $\leq$

lemma *ack-le-mono2*: $j \leq k ==> ack\ (i, j) \leq ack\ (i, k)$
⟨*proof*⟩

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: $ack\ (i, Suc\ j) \leq ack\ (Suc\ i, j)$
⟨*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 *CONST* 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*: $i < j ==> ack\ (i, k) < ack\ (j, k)$
⟨*proof*⟩

PROPERTY A 7', monotonicity for $\leq$

lemma *ack-le-mono1*: $i \leq j ==> ack\ (i, k) \leq ack\ (j, k)$
⟨*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*: $i < ack\ (k,\ j) ==> i + j < ack\ (4 + k,\ j)$
⟨*proof*⟩

Inductive definition of the *PR* functions

MAIN RESULT

lemma *SC-case*: $SC\ l < ack\ (1,\ list\text{-}add\ l)$
⟨*proof*⟩

lemma *CONST-case*: $CONST\ k\ l < ack\ (k,\ list\text{-}add\ l)$
⟨*proof*⟩

lemma *PROJ-case* [*rule-format*]: $\forall\, i.\ PROJ\ i\ l < ack\ (0,\ list\text{-}add\ l)$
⟨*proof*⟩

COMP case

lemma *COMP-map-aux*: $fs \in lists\ (PRIMREC \cap \{f.\ \exists\, kf.\ \forall\, l.\ f\ l < ack\ (kf,\ list\text{-}add\ l)\})$
$==> \exists\, k.\ \forall\, l.\ list\text{-}add\ (map\ (\lambda f.\ f\ l)\ fs) < ack\ (k,\ list\text{-}add\ l)$
⟨*proof*⟩

lemma *COMP-case*:
$\forall\, l.\ g\ l < ack\ (kg,\ list\text{-}add\ l) ==>$
$fs \in lists(PRIMREC\ Int\ \{f.\ \exists\, kf.\ \forall\, l.\ f\ l < ack(kf,\ list\text{-}add\ l)\})$
$==> \exists\, k.\ \forall\, l.\ COMP\ g\ fs\ \ l < ack(k,\ list\text{-}add\ l)$
⟨*proof*⟩

PREC case

lemma *PREC-case-aux*:
$\forall\, l.\ f\ l + list\text{-}add\ l < ack\ (kf,\ list\text{-}add\ l) ==>$
$\forall\, l.\ g\ l + list\text{-}add\ l < ack\ (kg,\ list\text{-}add\ l) ==>$
$PREC\ f\ g\ l + list\text{-}add\ l < ack\ (Suc\ (kf + kg),\ list\text{-}add\ l)$
⟨*proof*⟩

lemma *PREC-case*:
$\forall\, l.\ f\ l < ack\ (kf,\ list\text{-}add\ l) ==>$
$\forall\, l.\ g\ l < ack\ (kg,\ list\text{-}add\ l) ==>$
$\exists\, k.\ \forall\, l.\ PREC\ f\ g\ l < ack\ (k,\ list\text{-}add\ l)$

⟨*proof*⟩

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

lemma *ack-not-PRIMREC*: $(\lambda l.\ case\ l\ of\ []\ \Rightarrow\ 0\ |\ x\ \#\ l' \Rightarrow ack\ (x,\ x)) \notin PRIMREC$
⟨*proof*⟩

end

6 Using locales in Isabelle/Isar – outdated version!

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

6.1 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.

6.2 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** · *70*)

and *inv* :: $'a \Rightarrow 'a$ $((\text{-}^{-1})\ [1000]\ 999)$
and *one* :: $'a$ $(\mathbf{1})$
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.

6.3 Explicit structures referenced implicitly

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

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

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

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

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

locale *semigroup* =
  fixes *S* (**structure**)
  assumes *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

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

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

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

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

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

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" (ι).

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.

6.4 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* ≡ *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* ⇒ *'b group* ⇒ (*'a* × *'b*) *group*
  group-product G H ≡
    semigroup.extend
      (*semigroup-product* (*semigroup.truncate G*) (*semigroup.truncate H*))
      (*group.fields* (λ*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* ≡ *group-product G H*
  shows *group I*
⟨*proof*⟩

theorem *group-product*: *group G* ⟹ *group H* ⟹ *group* (*group-product G H*)
  ⟨*proof*⟩

end

7 Using extensible records in HOL – points and coloured points

theory *Records* **imports** *Main* **begin**

7.1 Points

record *point* =
  xpos :: *nat*
  ypos :: *nat*

Apart many other things, above record declaration produces the following theorems:

thm *point.simps*
thm *point.iffs*
thm *point.defs*

The set of theorems *point.simps* is added automatically to the standard simpset, *point.iffs* is added to the Classical Reasoner and Simplifier context.

Record declarations define new types and type abbreviations:

point = (|*xpos* :: *nat*, *ypos* :: *nat*|) = () *point-ext-type*
'a point-scheme = (|*xpos* :: *nat*, *ypos* :: *nat*, ... :: *'a*|) = *'a point-ext-type*

consts *foo1* :: *point*
consts *foo2* :: (| *xpos* :: *nat*, *ypos* :: *nat* |)
consts *foo3* :: *'a* => *'a point-scheme*
consts *foo4* :: *'a* => (| *xpos* :: *nat*, *ypos* :: *nat*, ... :: *'a* |)

7.1.1 Introducing concrete records and record schemes

defs
foo1-def: *foo1* == (| *xpos* = *1*, *ypos* = *0* |)
foo3-def: *foo3 ext* == (| *xpos* = *1*, *ypos* = *0*, ... = *ext* |)

7.1.2 Record selection and record update

constdefs
getX :: *'a point-scheme* => *nat*
getX r == *xpos r*
setX :: *'a point-scheme* => *nat* => *'a point-scheme*
setX r n == *r* (| *xpos* := *n* |)

7.1.3 Some lemmas about records

Basic simplifications.

lemma *point.make n p* = (| *xpos* = *n*, *ypos* = *p* |)
⟨*proof*⟩

lemma *xpos* (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) = *m*
⟨*proof*⟩

lemma (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) (| *xpos*:= *0* |) = (| *xpos* = *0*, *ypos* = *n*, ... = *p* |)
⟨*proof*⟩

Equality of records.

lemma *n* = *n'* ==> *p* = *p'* ==> (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |)
— introduction of concrete record equality
⟨*proof*⟩

lemma (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |) ==> *n* = *n'*
— elimination of concrete record equality
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
— introduction of abstract record equality
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) = *r* (| *xpos* := *n'* |) ==> *n* = *n'*
— elimination of abstract record equality (manual proof)
⟨*proof*⟩

Surjective pairing

lemma *r* = (| *xpos* = *xpos r*, *ypos* = *ypos r* |)
⟨*proof*⟩

lemma *r* = (| *xpos* = *xpos r*, *ypos* = *ypos r*, ... = *point.more r* |)
⟨*proof*⟩

Representation of records by cases or (degenerate) induction.

lemma *r*(| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
⟨*proof*⟩

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
⟨*proof*⟩

Concrete records are type instances of record schemes.

constdefs
foo5 :: *nat*
foo5 == *getX* (| *xpos* = *1*, *ypos* = *0* |)

Manipulating the "..." (more) part.

constdefs
incX :: *'a point-scheme* => *'a point-scheme*
incX r == (| *xpos* = *xpos r* + *1*, *ypos* = *ypos r*, ... = *point.more r* |)

lemma *incX r = setX r (Suc (getX r))*
⟨*proof*⟩

An alternative definition.

constdefs
incX′ :: 'a point-scheme => 'a point-scheme
incX′ r == r (| xpos := xpos r + 1 |)

7.2 Coloured points: record extension

datatype *colour = Red | Green | Blue*

record *cpoint = point +*
colour :: colour

The record declaration defines a new type constructure and abbreviations:

cpoint = (| xpos :: nat, ypos :: nat, colour :: colour |) =
() cpoint-ext-type point-ext-type
'a cpoint-scheme = (| xpos :: nat, ypos :: nat, colour :: colour, ... :: 'a |) =
'a cpoint-ext-type point-ext-type

consts *foo6 :: cpoint*
consts *foo7 :: (| xpos :: nat, ypos :: nat, colour :: colour |)*
consts *foo8 :: 'a cpoint-scheme*
consts *foo9 :: (| xpos :: nat, ypos :: nat, colour :: colour, ... :: 'a |)*

Functions on *point* schemes work for *cpoints* as well.

constdefs
foo10 :: nat
foo10 == getX (| xpos = 2, ypos = 0, colour = Blue |)

7.2.1 Non-coercive structural subtyping

Term *foo11* has type *cpoint*, not type *point* — Great!

constdefs
foo11 :: cpoint
foo11 == setX (| xpos = 2, ypos = 0, colour = Blue |) 0

7.3 Other features

Field names contribute to record identity.

record *point′ =*
xpos′ :: nat
ypos′ :: nat

May not apply *getX* to (| *xpos′ = 2*, *ypos′ = 0* |) – type error.

Polymorphic records.

record *'a point''* = *point* +
  content :: *'a*

types *cpoint''* = *colour point''*

end

8 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*

constdefs
  monoid :: (| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, ... :: *'b* |) => *bool*
  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*

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

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

end

9 String examples

theory *StringEx* **imports** *Main* **begin**

lemma *hd ''ABCD''* = *CHR ''A''*
  ⟨*proof*⟩

lemma *hd ''ABCD''* ≠ *CHR ''B''*
  ⟨*proof*⟩

lemma *''ABCD'' ≠ ''ABCX''*
⟨*proof*⟩

lemma *''ABCD'' = ''ABCD''*
⟨*proof*⟩

lemma *''ABCDEFGHIJKLMNOPQRSTUVWXYZ'' ≠*
''ABCDEFGHIJKLMNOPQRSTUVWXY''
⟨*proof*⟩

lemma *set ''Foobar'' = {CHR ''F'', CHR ''a'', CHR ''b'', CHR ''o'', CHR ''r''}*
⟨*proof*⟩

lemma *set ''Foobar'' = ?X*
⟨*proof*⟩

end

10 Binary arithmetic examples

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

10.1 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*⟩

10.2 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*⟩

10.3 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 $(999999::int) \leq (1000001 + 1) - 2$
⟨*proof*⟩

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

lemma (*1000006*::*int*) *mod 10 = 6*
⟨*proof*⟩

Division by shifting

lemma *10000000 div 2 = (5000000*::*int*)
⟨*proof*⟩

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

lemma *10000001 mod 2 = (1::int)*
⟨*proof*⟩

lemma *10000055 div 32 = (312501::int)*
⟨*proof*⟩

lemma *10000055 mod 32 = (23::int)*
⟨*proof*⟩

lemma *100094 div 144 = (695::int)*
⟨*proof*⟩

lemma *100094 mod 144 = (14::int)*
⟨*proof*⟩

Powers

lemma *2 ^ 10 = (1024::int)*
⟨*proof*⟩

lemma *−3 ^ 7 = (−2187::int)*
⟨*proof*⟩

lemma *13 ^ 7 = (62748517::int)*
⟨*proof*⟩

lemma *3 ^ 15 = (14348907::int)*
⟨*proof*⟩

lemma *−5 ^ 11 = (−48828125::int)*
⟨*proof*⟩

10.4 The Natural Numbers

Successor

lemma *Suc 99999 = 100000*
⟨*proof*⟩

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 ^ 14 = (4782969::nat)*
⟨*proof*⟩

lemma *5 ^ 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*⟩

The proofs about arithmetic yielding normal forms have been deleted: they are irrelevant with the new treatment of numerals.

end

11 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).

11.1 Proof text

theorem *tnd*: $A \vee \neg A$
⟨*proof*⟩

11.2 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 · - ·
                 (HOL.trans · - · - · - · - ·
                   (back-subst · (λu. u = (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)))
```

11.3 Proof script

theorem *tnd'*: $A \vee \neg A$
⟨*proof*⟩

11.4 Proof term of script

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

end

12 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

13 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

14 Properly nested products

theory *Tuple* **imports** *HOL* **begin**

14.1 Abstract syntax

typedecl *unit*
typedecl ('*a*, '*b*) *prod*

consts
Pair :: '*a* => '*b* => ('*a*, '*b*) *prod*
fst :: ('*a*, '*b*) *prod* => '*a*
snd :: ('*a*, '*b*) *prod* => '*b*
split :: ('*a* => '*b* => '*c*) => ('*a*, '*b*) *prod* => '*c*
Unity :: *unit* ('())

14.2 Concrete syntax

14.2.1 Tuple types

nonterminals
tuple-type-args
syntax
-tuple-type-arg :: *type* => *tuple-type-args* (- [*21*] *21*)
-tuple-type-args :: *type* => *tuple-type-args* => *tuple-type-args* (- *∗*/ - [*21*, *20*] *20*)
-tuple-type :: *type* => *tuple-type-args* => *type* ((- *∗*/ -) [*21*, *20*] *20*)

syntax (*xsymbols*)
-tuple-type-args :: *type* => *tuple-type-args* => *tuple-type-args* (- ×/ - [*21*, *20*] *20*)
-tuple-type :: *type* => *tuple-type-args* => *type* ((- ×/ -) [*21*, *20*] *20*)

syntax (*HTML* **output**)
-tuple-type-args :: *type* => *tuple-type-args* => *tuple-type-args* (- ×/ - [*21*, *20*] *20*)
-tuple-type :: *type* => *tuple-type-args* => *type* ((- ×/ -) [*21*, *20*] *20*)

translations
(*type*) *'a* * *'b* == (*type*) (*'a*, (*'b*, *unit*) *prod*) *prod*
(*type*) (*'a*, (*'b*, *'cs*) *-tuple-type-args*) *-tuple-type* ==
(*type*) (*'a*, (*'b*, *'cs*) *-tuple-type*) *prod*

14.2.2 Tuples

nonterminals
tuple-args
syntax
-tuple :: *'a* => *tuple-args* => *'b* ((*1* *'(-,/ -')))
-tuple-arg :: *'a* => *tuple-args* (-)
-tuple-args :: *'a* => *tuple-args* => *tuple-args* (-,/ -)
translations
(*x*, *y*) == *Pair x* (*Pair y* ())
-tuple x (*-tuple-args y zs*) == *Pair x* (*-tuple y zs*)

14.2.3 Tuple patterns

nonterminals *tuple-pat-args*
— extends pre-defined type "pttrn" syntax used in abstractions
syntax
-tuple-pat-arg :: *pttrn* => *tuple-pat-args* (-)
-tuple-pat-args :: *pttrn* => *tuple-pat-args* => *tuple-pat-args* (-,/ -)
-tuple-pat :: *pttrn* => *tuple-pat-args* => *pttrn* (*'(-,/ -')*)

translations
%(*x*,*y*). *b* => *split* (%*x*. *split* (%*y*. (*-K b*) :: *unit* => -))
%(*x*,*y*). *b* <= *split* (%*x*. *split* (%*y*. *-K b*))
-abs (*-tuple-pat x* (*-tuple-pat-args y zs*)) *b* == *split* (%*x*. (*-abs* (*-tuple-pat y zs*) *b*))

-abs (*Pair x* (*Pair y* ())) *b* => %(*x*,*y*). *b*
-abs (*Pair x* (*-abs* (*-tuple-pat y zs*) *b*)) => *-abs* (*-tuple-pat x* (*-tuple-pat-args y zs*)) *b*

⟨*ML*⟩

end

15 Summing natural numbers

theory *NatSum* **imports** *Main* **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*] =
left-distrib right-distrib
left-diff-distrib right-diff-distrib — for true subtraction
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*⟩

Sum of fourth powers: three versions.

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

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)*
⟨*proof*⟩

lemma *of-nat-sum-of-fourth-powers*:
*30 * of-nat (∑ i=0..<m. i * i * i * i) =*
*of-nat m * (of-nat m − 1) * (of-nat (2 * m) − 1) *
(of-nat (3 * m * m) − of-nat (3 * m) − (1::int))*
⟨*proof*⟩

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

lemma *sum-of-2-powers*: *(∑ i=0..<n. 2^i) = 2^n − (1::nat)*
⟨*proof*⟩

lemma *sum-of-3-powers*: *2 * (∑ i=0..<n. 3^i) = 3^n − (1::nat)*
⟨*proof*⟩

lemma *sum-of-powers*: *0 < k ==> (k − 1) * (∑ i=0..<n. k^i) = k^n − (1::nat)*
⟨*proof*⟩

end

16 Higher-Order Logic: Intuitionistic predicate calculus problems

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

lemma *(~~(P&Q)) = ((~~P) & (~~Q))*
⟨*proof*⟩

lemma *~~ ((~P −−> Q) −−> (~P −−> ~Q) −−> P)*
⟨*proof*⟩

lemma *(~~(P−−>Q)) = (~~P −−> ~~Q)*
⟨*proof*⟩

lemma *(~~~P) = (~P)*
⟨*proof*⟩

lemma *~~((P −−> Q | R) −−> (P−−>Q) | (P−−>R))*
⟨*proof*⟩

lemma $(P{=}Q) = (Q{=}P)$
⟨*proof*⟩

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

lemma $(((G \dashrightarrow A) \dashrightarrow J) \dashrightarrow D \dashrightarrow E) \dashrightarrow (((H \dashrightarrow B) \dashrightarrow I) \dashrightarrow C \dashrightarrow J)$
$\dashrightarrow (A \dashrightarrow H) \dashrightarrow F \dashrightarrow G \dashrightarrow (((C \dashrightarrow B) \dashrightarrow I) \dashrightarrow D) \dashrightarrow (A \dashrightarrow C)$
$\dashrightarrow (((F \dashrightarrow A) \dashrightarrow B) \dashrightarrow I) \dashrightarrow E$
⟨*proof*⟩

lemma $P \dashrightarrow \sim\sim P$
⟨*proof*⟩

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

lemma $\sim\sim P \;\&\; \sim\sim(P \dashrightarrow Q) \dashrightarrow \sim\sim Q$
⟨*proof*⟩

lemma $((P{=}Q) \dashrightarrow P\&Q\&R) \;\&$
$((Q{=}R) \dashrightarrow P\&Q\&R) \;\&$
$((R{=}P) \dashrightarrow P\&Q\&R) \dashrightarrow P\&Q\&R$
⟨*proof*⟩

lemma $((P{=}Q) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((Q{=}R) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((R{=}S) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((S{=}T) \dashrightarrow P\&Q\&R\&S\&T) \;\&$
$((T{=}P) \dashrightarrow P\&Q\&R\&S\&T) \dashrightarrow P\&Q\&R\&S\&T$
⟨*proof*⟩

lemma $(ALL\ x.\ EX\ y.\ ALL\ z.\ p(x) \;\&\; q(y) \;\&\; r(z)) =$
$(ALL\ z.\ EX\ y.\ ALL\ x.\ p(x) \;\&\; q(y) \;\&\; r(z))$
⟨*proof*⟩

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

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 $\sim\sim(P\ |\ \sim P)$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma $\sim\sim(((P \;\&\; (Q\dashrightarrow R))\dashrightarrow S) = ((\sim P \mid Q \mid S) \;\&\; (\sim P \mid \sim R \mid S)))$
⟨*proof*⟩

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

lemma $(EX\ x.\ P(x)\dashrightarrow Q) \;\dashrightarrow\; (ALL\ x.\ P(x)) \dashrightarrow Q$
⟨*proof*⟩

lemma $((ALL\ x.\ P(x))\dashrightarrow Q) \dashrightarrow \sim (ALL\ x.\ P(x) \;\&\; \sim Q)$
⟨*proof*⟩

lemma $((ALL\ x.\ \sim P(x))\dashrightarrow Q) \;\dashrightarrow\; \sim (ALL\ x.\ \sim (P(x)|Q))$
⟨*proof*⟩

lemma $(ALL\ x.\ P(x)) \mid Q \;\dashrightarrow\; (ALL\ x.\ P(x) \mid Q)$
⟨*proof*⟩

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

lemma $\sim\sim(EX\ x.\ ALL\ y\ z.\ (P(y)\text{--}\!>Q(z))\ \text{--}\!>\ (P(x)\text{--}\!>Q(x)))$
⟨*proof*⟩

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

lemma $(EX\ x.\ P\text{--}\!>Q(x))\ \&\ (EX\ x.\ Q(x)\text{--}\!>P)\ \text{--}\!>\ \sim\sim(EX\ x.\ P{=}Q(x))$
⟨*proof*⟩

lemma $(ALL\ x.\ P\ =\ Q(x))\ \ \text{--}\!>\ \ (P\ =\ (ALL\ x.\ Q(x)))$
⟨*proof*⟩

lemma $\sim\sim\ ((ALL\ x.\ P\ |\ Q(x))\ \ =\ \ (P\ |\ (ALL\ x.\ Q(x))))$
⟨*proof*⟩

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

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

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

$--> (ALL\ x.\ P(x)\ \&\ L(x)\ -->\ M(x))$
⟨*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*) & ~ *h x y*))) &
(*EX x. j*(*x*) & (*ALL y. g*(*y*) −−> *h x y*))
−−> (*EX x. j*(*x*) & ~*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

17 Classical Predicate Calculus Problems

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

17.1 Traditional Classical Reasoner

The machine "griffon" mentioned below is a 2.5GHz Power Mac G5.

Taken from *FOL/Classical.thy*. When porting examples from first-order logic, beware of the precedence of = versus ↔.

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

If and only if

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

lemma $\sim (P = (\sim P))$
⟨*proof*⟩

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.

17.1.1 Pelletier's examples

1

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

2

lemma $(\sim \sim P) = P$
⟨*proof*⟩

3

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

4

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

5

lemma $((P|Q) \dashrightarrow (P|R)) \dashrightarrow (P|(Q \dashrightarrow R))$
⟨*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 $((P|Q) \ \& \ (\sim P|Q) \ \& \ (P| \sim Q)) --> \sim (\sim P \mid \sim Q)$
⟨*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*⟩

17.1.2 Classical Logic: examples with quantifiers

lemma $(\forall x.\ P(x)\ \&\ Q(x)) = ((\forall x.\ P(x))\ \&\ (\forall x.\ Q(x)))$
⟨*proof*⟩

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

lemma $(\exists x.\ P(x){-}{-}{>}Q) = ((\forall x.\ P(x))\ {-}{-}{>}\ Q)$
⟨*proof*⟩

lemma $((\forall x.\ P(x))\mid Q)\ =\ (\forall x.\ P(x)\mid Q)$
⟨*proof*⟩

From Wishnu Prasetya

lemma $(\forall s.\ q(s)\ {-}{-}{>}\ r(s))\ \&\ {\sim}r(s)\ \&\ (\forall s.\ {\sim}r(s)\ \&\ {\sim}q(s)\ {-}{-}{>}\ p(t)\mid q(t))$
$\quad {-}{-}{>}\ p(t)\mid r(t)$
⟨*proof*⟩

17.1.3 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*⟩

17.1.4 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 $(\forall x\ y.\ \exists z.\ \forall w.\ (P(x)\&Q(y)\text{--}{>}R(z)\&S(w)))$
$\quad\text{--}{>}\ (\exists x\ y.\ P(x)\ \&\ Q(y))\ \text{--}{>}\ (\exists z.\ R(z))$
⟨*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 $(\forall x.\ P = Q(x))\ \text{--}{>}\ (P = (\forall x.\ Q(x)))$
⟨*proof*⟩

Problem 23

lemma $(\forall x.\ P \mid Q(x))\ =\ (P \mid (\forall x.\ Q(x)))$
⟨*proof*⟩

Problem 24

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

Problem 25

lemma $(\exists x.\ P(x))\ \&$
$\qquad(\forall x.\ L(x)\ \text{--}{>}\ \sim\ (M(x)\ \&\ R(x)))\ \&$
$\qquad(\forall x.\ P(x)\ \text{--}{>}\ (M(x)\ \&\ L(x)))\ \&$
$\qquad((\forall x.\ P(x)\text{--}{>}Q(x))\ \mid\ (\exists x.\ P(x)\&R(x)))$
$\quad\text{--}{>}\ (\exists x.\ Q(x)\&P(x))$
⟨*proof*⟩

Problem 26

lemma $((\exists x.\ p(x)) = (\exists x.\ q(x)))\ \&$
$\quad(\forall x.\ \forall y.\ p(x)\ \&\ q(y)\ \text{--}{>}\ (r(x) = s(y)))$
$\ \text{--}{>}\ ((\forall x.\ p(x)\text{--}{>}r(x)) = (\forall x.\ q(x)\text{--}{>}s(x)))$
⟨*proof*⟩

Problem 27

lemma $(\exists x.\ P(x)\ \&\ \sim Q(x))\ \&$
$\qquad(\forall x.\ P(x)\ \text{--}{>}\ R(x))\ \&$
$\qquad(\forall x.\ M(x)\ \&\ L(x)\ \text{--}{>}\ P(x))\ \&$
$\qquad((\exists x.\ R(x)\ \&\ \sim\ Q(x))\ \text{--}{>}\ (\forall x.\ L(x)\ \text{--}{>}\ \sim\ R(x)))$
$\quad\text{--}{>}\ (\forall x.\ M(x)\ \text{--}{>}\ \sim L(x))$
⟨*proof*⟩

Problem 28. AMENDED

lemma $(\forall x.\ P(x) \longrightarrow (\forall x.\ Q(x))) \ \&$
$\quad ((\forall x.\ Q(x) | R(x)) \longrightarrow (\exists x.\ Q(x) \& S(x))) \ \&$
$\quad ((\exists x.\ S(x)) \longrightarrow (\forall x.\ L(x) \longrightarrow M(x)))$
$\quad \longrightarrow (\forall x.\ P(x) \ \& \ L(x) \longrightarrow M(x))$
⟨*proof*⟩

Problem 29. Essentially the same as Principia Mathematica *11.71

lemma $(\exists x.\ F(x)) \ \& \ (\exists y.\ G(y))$
$\quad \longrightarrow (\ ((\forall x.\ F(x) \longrightarrow H(x)) \ \& \ (\forall y.\ G(y) \longrightarrow J(y))) \ =$
$\qquad (\forall x\ y.\ F(x) \ \& \ G(y) \longrightarrow H(x) \ \& \ J(y)))$
⟨*proof*⟩

Problem 30

lemma $(\forall x.\ P(x) \mid Q(x) \longrightarrow \sim R(x)) \ \&$
$\quad (\forall x.\ (Q(x) \longrightarrow \sim S(x)) \longrightarrow P(x) \ \& \ R(x))$
$\quad \longrightarrow (\forall x.\ S(x))$
⟨*proof*⟩

Problem 31

lemma $\sim(\exists x.\ P(x) \ \& \ (Q(x) \mid R(x))) \ \&$
$\quad (\exists x.\ L(x) \ \& \ P(x)) \ \&$
$\quad (\forall x.\ \sim R(x) \longrightarrow M(x))$
$\quad \longrightarrow (\exists x.\ L(x) \ \& \ M(x))$
⟨*proof*⟩

Problem 32

lemma $(\forall x.\ P(x) \ \& \ (Q(x) | R(x)) \longrightarrow S(x)) \ \&$
$\quad (\forall x.\ S(x) \ \& \ R(x) \longrightarrow L(x)) \ \&$
$\quad (\forall x.\ M(x) \longrightarrow R(x))$
$\quad \longrightarrow (\forall x.\ P(x) \ \& \ M(x) \longrightarrow L(x))$
⟨*proof*⟩

Problem 33

lemma $(\forall x.\ P(a) \ \& \ (P(x) \longrightarrow P(b)) \longrightarrow P(c)) \ =$
$\quad (\forall x.\ (\sim P(a) \mid P(x) \mid P(c)) \ \& \ (\sim P(a) \mid \sim P(b) \mid P(c)))$
⟨*proof*⟩

Problem 34 AMENDED (TWICE!!)

Andrews's challenge

lemma $((\exists x.\ \forall y.\ p(x) = p(y)) \ =$
$\qquad ((\exists x.\ q(x)) = (\forall y.\ p(y)))) \ =$
$\qquad ((\exists x.\ \forall y.\ q(x) = q(y)) \ =$
$\qquad ((\exists x.\ p(x)) = (\forall y.\ q(y))))$
⟨*proof*⟩

Problem 35

lemma $\exists x\ y.\ P\ x\ y \longrightarrow \ (\forall u\ v.\ P\ u\ v)$

⟨*proof*⟩

Problem 36

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&$
$(\forall x.\ \exists y.\ G\ x\ y)\ \&$
$(\forall x\ y.\ J\ x\ y \mid G\ x\ y \longrightarrow$
$(\forall z.\ J\ y\ z \mid G\ y\ z \longrightarrow H\ x\ z))$
$\longrightarrow (\forall x.\ \exists y.\ H\ x\ y)$
⟨*proof*⟩

Problem 37

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

Problem 38

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

Problem 39

lemma $\sim (\exists x.\ \forall y.\ F\ y\ x = (\sim\ F\ y\ y))$
⟨*proof*⟩

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
$\longrightarrow\ \sim (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = (\sim\ F\ z\ x))$
⟨*proof*⟩

Problem 41

lemma $(\forall z.\ \exists y.\ \forall x.\ f\ x\ y = (f\ x\ z\ \&\ \sim f\ x\ x))$
$\longrightarrow \sim (\exists z.\ \forall x.\ f\ x\ z)$
⟨*proof*⟩

Problem 42

lemma $\sim (\exists y.\ \forall x.\ p\ x\ y = (\sim (\exists z.\ p\ x\ z\ \&\ p\ z\ x)))$
⟨*proof*⟩

Problem 43!!

lemma $(\forall x::'a.\ \forall y::'a.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y::bool)))$
$\longrightarrow (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x::bool)))$

⟨*proof*⟩

Problem 44

lemma $(\forall x.\ f(x)\ \text{--}\text{>}$
$(\exists y.\ g(y)\ \&\ h\ x\ y\ \&\ (\exists y.\ g(y)\ \&\ \sim\ h\ x\ y)))\ \ \&$
$(\exists x.\ j(x)\ \&\ (\forall y.\ g(y)\ \text{--}\text{>}\ h\ x\ y))$
$\text{--}\text{>}\ (\exists x.\ j(x)\ \&\ \sim f(x))$
⟨*proof*⟩

Problem 45

lemma $(\forall x.\ f(x)\ \&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ \text{--}\text{>}\ j\ x\ y)$
$\text{--}\text{>}\ (\forall y.\ g(y)\ \&\ h\ x\ y\ \text{--}\text{>}\ k(y)))\ \&$
$\sim\ (\exists y.\ l(y)\ \&\ k(y))\ \&$
$(\exists x.\ f(x)\ \&\ (\forall y.\ h\ x\ y\ \text{--}\text{>}\ l(y))$
$\&\ (\forall y.\ g(y)\ \&\ h\ x\ y\ \text{--}\text{>}\ j\ x\ y))$
$\text{--}\text{>}\ (\exists x.\ f(x)\ \&\ \sim\ (\exists y.\ g(y)\ \&\ h\ x\ y))$
⟨*proof*⟩

17.1.5 Problems (mainly) involving equality or functions

Problem 48

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

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

lemma $(\forall x.\ P\ a\ x\ |\ (\forall y.\ P\ x\ y))\ \text{--}\text{>}\ (\exists x.\ \forall y.\ P\ x\ y)$
⟨*proof*⟩

Problem 51

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

Problem 52. Almost the same as 51.

lemma $(\exists z\ w.\ \forall x\ y.\ P\ x\ y = (x{=}z\ \&\ y{=}w))\ \text{--}\text{>}$
$(\exists w.\ \forall y.\ \exists z.\ (\forall x.\ P\ x\ y = (x{=}z)) = (y{=}w))$
⟨*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 (∀ *x*. (∃ *y*. *P*(*y*) & *x*=*f*(*y*)) −−> *P*(*x*)) = (∀ *x*. *P*(*x*) −−> *P*(*f*(*x*)))
⟨*proof*⟩

Problem 57

lemma *P* (*f a b*) (*f b c*) & *P* (*f b c*) (*f a c*) &
(∀ *x y z*. *P x y* & *P y z* −−> *P x z*) −−> *P* (*f a b*) (*f a c*)
⟨*proof*⟩

Problem 58 NOT PROVED AUTOMATICALLY

lemma (∀ *x y*. *f*(*x*)=*g*(*y*)) −−> (∀ *x y*. *f*(*f*(*x*))=*f*(*g*(*y*)))
⟨*proof*⟩

Problem 59

lemma (∀ *x*. *P*(*x*) = (~*P*(*f*(*x*)))) −−> (∃ *x*. *P*(*x*) & ~*P*(*f*(*x*)))
⟨*proof*⟩

Problem 60

lemma ∀ *x*. *P x* (*f x*) = (∃ *y*. (∀ *z*. *P z y* −−> *P z* (*f x*)) & *P x y*)
⟨*proof*⟩

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

lemma (∀ *x*. *p a* & (*p x* −−> *p*(*f x*)) −−> *p*(*f*(*f x*))) =
(∀ *x*. (~ *p a* | *p x* | *p*(*f*(*f x*))) &
(~ *p a* | ~ *p*(*f x*) | *p*(*f*(*f x*))))
⟨*proof*⟩

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

lemma (∀ *x*. *F*(*x*) & ~*G*(*x*) −−> (∃ *y*. *H*(*x*,*y*) & *J*(*y*))) &
(∃ *x*. *K*(*x*) & *F*(*x*) & (∀ *y*. *H*(*x*,*y*) −−> *K*(*y*))) &
(∀ *x*. *K*(*x*) −−> ~*G*(*x*)) −−> (∃ *x*. *K*(*x*) & *J*(*x*))
⟨*proof*⟩

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

lemma $(\forall x.\ F(x)\ \&\ \sim G(x) \longrightarrow (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y) \longrightarrow K(y)))\ \&$
$(\forall x.\ K(x) \longrightarrow \sim G(x)) \longrightarrow (\exists x.\ K(x) \longrightarrow \sim G(x))$
⟨*proof*⟩

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

lemma $(\forall x.\ honest(x)\ \&\ industrious(x) \longrightarrow healthy(x))\ \&$
$\sim (\exists x.\ grocer(x)\ \&\ healthy(x))\ \&$
$(\forall x.\ industrious(x)\ \&\ grocer(x) \longrightarrow honest(x))\ \&$
$(\forall x.\ cyclist(x) \longrightarrow industrious(x))\ \&$
$(\forall x.\ \sim healthy(x)\ \&\ cyclist(x) \longrightarrow \sim honest(x))$
$\longrightarrow (\forall x.\ grocer(x) \longrightarrow \sim cyclist(x))$
⟨*proof*⟩

lemma $(\forall x\ y.\ R(x,y) \mid R(y,x))\ \&$
$(\forall x\ y.\ S(x,y)\ \&\ S(y,x) \longrightarrow x{=}y)\ \&$
$(\forall x\ y.\ R(x,y) \longrightarrow S(x,y)) \longrightarrow (\forall x\ y.\ S(x,y) \longrightarrow R(x,y))$
⟨*proof*⟩

17.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y\ \&\ y < z \longrightarrow \sim (z < (x{::}nat))$
⟨*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 $(\forall x\ y\ z.\ R(x,y)\ \&\ R(y,z) \longrightarrow R(x,z))\ \&$
$(\forall x.\ \exists y.\ R(x,y)) \longrightarrow$
$\sim (\forall x.\ P\ x = (\forall y.\ R(x,y) \longrightarrow \sim\ P\ y))$
⟨*proof*⟩

17.2.1 Pelletier's examples

1

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

2

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

3

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

4

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

5

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

6

lemma P $|$ $^{\sim}$ P
⟨*proof*⟩

7

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

14

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

15

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

16

lemma $(P \dashrightarrow Q) \mid (Q \dashrightarrow P)$
⟨*proof*⟩

17

lemma $((P \;\&\; (Q \dashrightarrow R)) \dashrightarrow S) \;=\; ((\sim P \mid Q \mid S) \;\&\; (\sim P \mid \sim R \mid S))$
⟨*proof*⟩

17.2.2 Classical Logic: examples with quantifiers

lemma $(\forall x.\ P\ x \;\&\; Q\ x) = ((\forall x.\ P\ x) \;\&\; (\forall x.\ Q\ x))$
⟨*proof*⟩

lemma $(\exists x.\ P \dashrightarrow Q\ x) \;=\; (P \dashrightarrow (\exists x.\ Q\ x))$
⟨*proof*⟩

lemma $(\exists x.\ P\ x \dashrightarrow Q) = ((\forall x.\ P\ x) \dashrightarrow Q)$
⟨*proof*⟩

lemma $((\forall x.\ P\ x) \mid Q) \;=\; (\forall x.\ P\ x \mid Q)$
⟨*proof*⟩

lemma $(\forall x.\ P\ x \dashrightarrow P(f\ x)) \;\;\&\;\; P\ d \dashrightarrow P(f(f(f\ d)))$
⟨*proof*⟩

Needs double instantiation of EXISTS

lemma $\exists x.\ P\ x \dashrightarrow P\ a \;\&\; P\ b$
⟨*proof*⟩

lemma $\exists z.\ P\ z \dashrightarrow (\forall x.\ P\ x)$
⟨*proof*⟩

From a paper by Claire Quigley

lemma $\exists y.\ ((P\ c \;\&\; Q\ y) \mid (\exists z.\ \sim\ Q\ z)) \mid (\exists x.\ \sim\ P\ x \;\&\; Q\ d)$
⟨*proof*⟩

17.2.3 Hard examples with quantifiers

Problem 18

lemma $\exists y.\ \forall x.\ P\ y \dashrightarrow P\ x$
⟨*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 $(\forall x.\ P = Q\ x)\ \ \text{--->}\ \ (P = (\forall x.\ Q\ x))$
⟨*proof*⟩

Problem 23

lemma $(\forall x.\ P \mid Q\ x)\ =\ (P \mid (\forall x.\ Q\ x))$
⟨*proof*⟩

Problem 24

lemma $\sim(\exists x.\ S\ x\ \&\ Q\ x)\ \&\ (\forall x.\ P\ x\ \text{--->}\ Q\ x \mid R\ x)\ \&$
$\quad (\sim(\exists x.\ P\ x)\ \text{--->}\ (\exists x.\ Q\ x))\ \&\ (\forall x.\ Q\ x \mid R\ x\ \text{--->}\ S\ x)$
$\quad \text{--->}\ (\exists x.\ P\ x\ \&\ R\ x)$
⟨*proof*⟩

Problem 25

lemma $(\exists x.\ P\ x)\ \&$
$\quad (\forall x.\ L\ x\ \text{--->}\ \sim\ (M\ x\ \&\ R\ x))\ \&$
$\quad (\forall x.\ P\ x\ \text{--->}\ (M\ x\ \&\ L\ x))\ \&$
$\quad ((\forall x.\ P\ x\ \text{--->}\ Q\ x) \mid (\exists x.\ P\ x\ \&\ R\ x))$
$\text{--->}\ (\exists x.\ Q\ x\ \&\ P\ x)$
⟨*proof*⟩

Problem 26; has 24 Horn clauses

lemma $((\exists x.\ p\ x) = (\exists x.\ q\ x))\ \&$
$\quad (\forall x.\ \forall y.\ p\ x\ \&\ q\ y\ \text{--->}\ (r\ x = s\ y))$
$\text{--->}\ ((\forall x.\ p\ x\ \text{--->}\ r\ x) = (\forall x.\ q\ x\ \text{--->}\ s\ x))$
⟨*proof*⟩

Problem 27; has 13 Horn clauses

lemma $(\exists x.\ P\ x\ \&\ \sim Q\ x)\ \&$
$\quad (\forall x.\ P\ x\ \text{--->}\ R\ x)\ \&$
$\quad (\forall x.\ M\ x\ \&\ L\ x\ \text{--->}\ P\ x)\ \&$
$\quad ((\exists x.\ R\ x\ \&\ \sim\ Q\ x)\ \text{--->}\ (\forall x.\ L\ x\ \text{--->}\ \sim\ R\ x))$
$\quad \text{--->}\ (\forall x.\ M\ x\ \text{--->}\ \sim L\ x)$
⟨*proof*⟩

Problem 28. AMENDED; has 14 Horn clauses

lemma $(\forall x.\ P\ x \dashrightarrow (\forall x.\ Q\ x)) \ \&$
$((\forall x.\ Q\ x \mid R\ x) \dashrightarrow (\exists x.\ Q\ x\ \&\ S\ x)) \ \&$
$((\exists x.\ S\ x) \dashrightarrow (\forall x.\ L\ x \dashrightarrow M\ x))$
$\dashrightarrow (\forall x.\ P\ x\ \&\ L\ x \dashrightarrow M\ x)$
⟨*proof*⟩

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

lemma $(\exists x.\ F\ x)\ \&\ (\exists y.\ G\ y)$
$\dashrightarrow (\ ((\forall x.\ F\ x \dashrightarrow H\ x)\ \&\ (\forall y.\ G\ y \dashrightarrow J\ y)) \ =$
$(\forall x\ y.\ F\ x\ \&\ G\ y \dashrightarrow H\ x\ \&\ J\ y))$
⟨*proof*⟩

Problem 30

lemma $(\forall x.\ P\ x \mid Q\ x \dashrightarrow \sim R\ x)\ \&\ (\forall x.\ (Q\ x \dashrightarrow \sim S\ x) \dashrightarrow P\ x\ \&\ R\ x)$
$\dashrightarrow (\forall x.\ S\ x)$
⟨*proof*⟩

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

lemma $\sim(\exists x.\ P\ x\ \&\ (Q\ x \mid R\ x)) \ \&$
$(\exists x.\ L\ x\ \&\ P\ x) \ \&$
$(\forall x.\ \sim R\ x \dashrightarrow M\ x)$
$\dashrightarrow (\exists x.\ L\ x\ \&\ M\ x)$
⟨*proof*⟩

Problem 32

lemma $(\forall x.\ P\ x\ \&\ (Q\ x \mid R\ x) \dashrightarrow S\ x)\ \&$
$(\forall x.\ S\ x\ \&\ R\ x \dashrightarrow L\ x)\ \&$
$(\forall x.\ M\ x \dashrightarrow R\ x)$
$\dashrightarrow (\forall x.\ P\ x\ \&\ M\ x \dashrightarrow L\ x)$
⟨*proof*⟩

Problem 33; has 55 Horn clauses

lemma $(\forall x.\ P\ a\ \&\ (P\ x \dashrightarrow P\ b) \dashrightarrow P\ c) \ =$
$(\forall x.\ (\sim P\ a \mid P\ x \mid P\ c)\ \&\ (\sim P\ a \mid \sim P\ b \mid P\ c))$
⟨*proof*⟩

Problem 34: Andrews's challenge has 924 Horn clauses

lemma $((\exists x.\ \forall y.\ p\ x = p\ y) \ = ((\exists x.\ q\ x) = (\forall y.\ p\ y))) \quad =$
$((\exists x.\ \forall y.\ q\ x = q\ y) \ = ((\exists x.\ p\ x) = (\forall y.\ q\ y)))$
⟨*proof*⟩

Problem 35

lemma $\exists x\ y.\ P\ x\ y \dashrightarrow \ (\forall u\ v.\ P\ u\ v)$
⟨*proof*⟩

Problem 36; has 15 Horn clauses

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&\ (\forall x.\ \exists y.\ G\ x\ y)\ \&$
$\quad (\forall x\ y.\ J\ x\ y \mid G\ x\ y \longrightarrow (\forall z.\ J\ y\ z \mid G\ y\ z \longrightarrow H\ x\ z))$
$\quad \longrightarrow (\forall x.\ \exists y.\ H\ x\ y)$
⟨*proof*⟩

Problem 37; has 10 Horn clauses

lemma $(\forall z.\ \exists w.\ \forall x.\ \exists y.$
$\quad (P\ x\ z \longrightarrow P\ y\ w)\ \&\ P\ y\ z\ \&\ (P\ y\ w \longrightarrow (\exists u.\ Q\ u\ w)))\ \&$
$\quad (\forall x\ z.\ {\sim}P\ x\ z \longrightarrow (\exists y.\ Q\ y\ z))\ \&$
$\quad ((\exists x\ y.\ Q\ x\ y) \longrightarrow (\forall x.\ R\ x\ x))$
$\longrightarrow (\forall x.\ \exists y.\ R\ x\ y)$
⟨*proof*⟩

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x.\ p\ a\ \&\ (p\ x \longrightarrow (\exists y.\ p\ y\ \&\ r\ x\ y)) \longrightarrow$
$\quad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$\quad (\forall x.\ ({\sim}p\ a \mid p\ x \mid (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$\quad ({\sim}p\ a \mid {\sim}(\exists y.\ p\ y\ \&\ r\ x\ y) \mid$
$\quad (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
⟨*proof*⟩

Problem 39

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

Problem 40. AMENDED

lemma $(\exists y.\ \forall x.\ F\ x\ y = F\ x\ x)$
$\quad \longrightarrow\ \sim\ (\forall x.\ \exists y.\ \forall z.\ F\ z\ y = ({\sim}F\ z\ x))$
⟨*proof*⟩

Problem 41

lemma $(\forall z.\ (\exists y.\ (\forall x.\ f\ x\ y = (f\ x\ z\ \&\ \sim\ f\ x\ x))))$
$\quad \longrightarrow\ \sim\ (\exists z.\ \forall x.\ f\ x\ z)$
⟨*proof*⟩

Problem 42

lemma $\sim\ (\exists y.\ \forall x.\ p\ x\ y = (\sim\ (\exists z.\ p\ x\ z\ \&\ p\ z\ x)))$
⟨*proof*⟩

Problem 43 NOW PROVED AUTOMATICALLY!!

lemma $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}bool)))$
$\quad \longrightarrow (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}bool)))$
⟨*proof*⟩

Problem 44: 13 Horn clauses; 7-step proof

lemma $(\forall x.\ f\ x \longrightarrow (\exists y.\ g\ y\ \&\ h\ x\ y\ \&\ (\exists y.\ g\ y\ \&\ \sim h\ x\ y)))\ \&$
$(\exists x.\ j\ x\ \&\ (\forall y.\ g\ y \longrightarrow h\ x\ y))$
$\longrightarrow (\exists x.\ j\ x\ \&\ \sim f\ x)$
⟨*proof*⟩

Problem 45; has 27 Horn clauses; 54-step proof

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

Problem 46; has 26 Horn clauses; 21-step proof

lemma $(\forall x.\ f\ x\ \&\ (\forall y.\ f\ y\ \&\ h\ y\ x \longrightarrow g\ y) \longrightarrow g\ x)\ \&$
$((\exists x.\ f\ x\ \&\ \sim g\ x) \longrightarrow$
$(\exists x.\ f\ x\ \&\ \sim g\ x\ \&\ (\forall y.\ f\ y\ \&\ \sim g\ y \longrightarrow j\ x\ y)))\ \&$
$(\forall x\ y.\ f\ x\ \&\ f\ y\ \&\ h\ x\ y \longrightarrow \sim j\ y\ x)$
$\longrightarrow (\forall x.\ f\ x \longrightarrow g\ x)$
⟨*proof*⟩

Problem 47. Schubert's Steamroller. 26 clauses; 63 Horn clauses. 87094 inferences so far. Searching to depth 36

lemma $(\forall x.\ wolf\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ wolf\ x)\ \&$
$(\forall x.\ fox\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ fox\ x)\ \&$
$(\forall x.\ bird\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ bird\ x)\ \&$
$(\forall x.\ caterpillar\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ caterpillar\ x)\ \&$
$(\forall x.\ snail\ x \longrightarrow animal\ x)\ \&\ (\exists x.\ snail\ x)\ \&$
$(\forall x.\ grain\ x \longrightarrow plant\ x)\ \&\ (\exists x.\ grain\ x)\ \&$
$(\forall x.\ animal\ x \longrightarrow$
$((\forall y.\ plant\ y \longrightarrow eats\ x\ y)\ \vee$
$(\forall y.\ animal\ y\ \&\ smaller\text{-}than\ y\ x\ \&$
$(\exists z.\ plant\ z\ \&\ eats\ y\ z) \longrightarrow eats\ x\ y)))\ \&$
$(\forall x\ y.\ bird\ y\ \&\ (snail\ x \vee caterpillar\ x) \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$(\forall x\ y.\ bird\ x\ \&\ fox\ y \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$(\forall x\ y.\ fox\ x\ \&\ wolf\ y \longrightarrow smaller\text{-}than\ x\ y)\ \&$
$(\forall x\ y.\ wolf\ x\ \&\ (fox\ y \vee grain\ y) \longrightarrow \sim eats\ x\ y)\ \&$
$(\forall x\ y.\ bird\ x\ \&\ caterpillar\ y \longrightarrow eats\ x\ y)\ \&$
$(\forall x\ y.\ bird\ x\ \&\ snail\ y \longrightarrow \sim eats\ x\ y)\ \&$
$(\forall x.\ (caterpillar\ x \vee snail\ x) \longrightarrow (\exists y.\ plant\ y\ \&\ eats\ x\ y))$
$\longrightarrow (\exists x\ y.\ animal\ x\ \&\ animal\ y\ \&\ (\exists z.\ grain\ z\ \&\ eats\ y\ z\ \&\ eats\ x\ y))$
⟨*proof*⟩

The Los problem. Circulated by John Harrison

lemma $(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z \longrightarrow P\ x\ z)\ \&$
$(\forall x\ y\ z.\ Q\ x\ y\ \&\ Q\ y\ z \longrightarrow Q\ x\ z)\ \&$
$(\forall x\ y.\ P\ x\ y \longrightarrow P\ y\ x)\ \&$

$(\forall\, x\ y.\ P\ x\ y \mid Q\ x\ y)$
$--> (\forall\, x\ y.\ P\ x\ y) \mid (\forall\, x\ y.\ Q\ x\ y)$
⟨*proof*⟩

A similar example, suggested by Johannes Schumann and credited to Pelletier

lemma $(\forall\, x\ y\ z.\ P\ x\ y --> P\ y\ z --> P\ x\ z) -->$
$(\forall\, x\ y\ z.\ Q\ x\ y --> Q\ y\ z --> Q\ x\ z) -->$
$(\forall\, x\ y.\ Q\ x\ y --> Q\ y\ x) -->\ (\forall\, x\ y.\ P\ x\ y \mid Q\ x\ y) -->$
$(\forall\, x\ y.\ P\ x\ y) \mid (\forall\, x\ y.\ Q\ x\ y)$
⟨*proof*⟩

Problem 50. What has this to do with equality?

lemma $(\forall\, x.\ P\ a\ x \mid (\forall\, y.\ P\ x\ y)) --> (\exists\, x.\ \forall\, y.\ P\ x\ y)$
⟨*proof*⟩

Problem 54: NOT PROVED

lemma $(\forall\, y{::}'a.\ \exists\, z.\ \forall\, x.\ F\ x\ z = (x{=}y)) -->$
$\sim (\exists\, w.\ \forall\, x.\ F\ x\ w = (\forall\, u.\ F\ x\ u --> (\exists\, y.\ F\ y\ u\ \&\ \sim (\exists\, z.\ F\ z\ u\ \&\ F\ z\ y))))$
⟨*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) &
(∀ x y. killed x y --> hates x y & ~richer x y) &
(∀ x. hates agatha x --> ~hates charles x) &
(hates agatha agatha & hates agatha charles) &
(∀ x. lives x & ~richer x agatha --> hates butler x) &
(∀ x. hates agatha x --> hates butler x) &
(∀ x. ~hates x agatha | ~hates x butler | ~hates x charles) -->
(∃ x. killed x agatha)
⟨*proof*⟩

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall\, x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z --> P\ x\ z) \quad --> \quad P\ (f\ a\ b)\ (f\ a\ c)$
⟨*proof*⟩

Problem 58: Challenge found on info-hol

lemma $\forall\, P\ Q\ R\ x.\ \exists\, v\ w.\ \forall\, y\ z.\ P\ x\ \&\ Q\ y --> (P\ v \mid R\ w)\ \&\ (R\ z --> Q\ v)$
⟨*proof*⟩

Problem 59

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

⟨*proof*⟩

Problem 60

lemma $\forall x.\ P\ x\ (f\ x) = (\exists y.\ (\forall z.\ P\ z\ y \longrightarrow P\ z\ (f\ x))\ \&\ P\ x\ y)$
⟨*proof*⟩

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x \longrightarrow p(f\ x)) \longrightarrow p(f(f\ x))) =$
$(\forall x.\ (\sim p\ a \mid p\ x \mid p(f(f\ x)))\ \&$
$(\sim p\ a \mid \sim p(f\ x) \mid p(f(f\ x))))$
⟨*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: $\forall x\ y.\ \ T(i\ x\ y)\ \&\ T\ x \longrightarrow T\ y$
shows *True*
⟨*proof*⟩

Problem 71, as found in TPTP (SYN007+1.005)

lemma $p1 = (p2 = (p3 = (p4 = (p5 = (p1 = (p2 = (p3 = (p4 = p5))))))))$
⟨*proof*⟩

A manual resolution proof of problem 19.

lemma $\exists x.\ \forall y\ z.\ (P(y) \longrightarrow Q(z)) \longrightarrow (P(x) \longrightarrow Q(x))$
⟨*proof*⟩

end

18 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*
constdefs
imp :: *'a ctl* ⇒ *'a ctl* ⇒ *'a ctl* (**infixr** → *75*)
p → *q* ≡ − *p* ∪ *q*

lemma [*intro*!]: *p* ∩ *p* → *q* ⊆ *q* ⟨*proof*⟩
lemma [*intro*!]: *p* ⊆ (*q* → *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*.

consts *model* :: (*'a* × *'a*) *set* ($\mathcal{M}$)

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].

constdefs
EX :: *'a ctl* ⇒ *'a ctl* (EX - [*80*] *90*) EX *p* ≡ {*s*. ∃ *s'*. (*s*, *s'*) ∈ $\mathcal{M}$ ∧ *s'* ∈ *p*}
EF :: *'a ctl* ⇒ *'a ctl* (EF - [*80*] *90*) EF *p* ≡ *lfp* (λ*s*. *p* ∪ EX *s*)
EG :: *'a ctl* ⇒ *'a ctl* (EG - [*80*] *90*) EG *p* ≡ *gfp* (λ*s*. *p* ∩ EX *s*)

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

constdefs
AX :: *'a ctl* ⇒ *'a ctl* (AX - [*80*] *90*) AX *p* ≡ − EX − *p*
AF :: *'a ctl* ⇒ *'a ctl* (AF - [*80*] *90*) AF *p* ≡ − EG − *p*
AG :: *'a ctl* ⇒ *'a ctl* (AG - [*80*] *90*) AG *p* ≡ − EF − *p*

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

19 Basic fixed point properties

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

lemma *lfp-gfp*: *lfp f* = − *gfp* (λ*s* . − (*f* (− *s*)))
⟨*proof*⟩

lemma *lfp-gfp'*: − *lfp f* = *gfp* (λ*s*. − (*f* (− *s*)))
⟨*proof*⟩

lemma *gfp-lfp'*: − *gfp f* = *lfp* (λ*s*. − (*f* (− *s*)))
⟨*proof*⟩

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

lemma *AF-lfp*: AF p = *lfp* ($\lambda s.\ p \cup$ AX s) ⟨*proof*⟩
lemma *AG-gfp*: AG p = *gfp* ($\lambda s.\ p \cap$ AX s) ⟨*proof*⟩

lemma *EF-fp*: EF $p = p \cup$ EX EF p
⟨*proof*⟩

lemma *AF-fp*: AF $p = p \cup$ AX AF p
⟨*proof*⟩

lemma *EG-fp*: EG $p = p \cap$ EX EG p
⟨*proof*⟩

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

lemma *AG-fp*: AG $p = p \cap$ AX AG p
⟨*proof*⟩

This fact may be split up into two inequalities (merely using transitivity of $\subseteq$, which is an instance of the overloaded $\leq$ in Isabelle/HOL).

lemma *AG-fp-1*: AG $p \subseteq p$
⟨*proof*⟩

lemma *AG-fp-2*: AG $p \subseteq$ AX AG p
⟨*proof*⟩

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

lemma *AG-I*: $q \subseteq p \cap$ AX $q \Longrightarrow q \subseteq$ AG p
⟨*proof*⟩

20 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 $(p \rightarrow$ AX $p) =$ AG p
⟨*proof*⟩

21 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 Meson test cases

theory *mesontest2* **imports** *Main* **begin**

end

23 Some examples for Presburger Arithmetic

theory *PresburgerEx* **imports** *Main* **begin**

theorem $(\forall (y{::}int).\ 3\ dvd\ y) ==> \forall (x{::}int).\ b < x\ --> a \leq x$
⟨*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. $Suc(n::nat) < 6 \Longrightarrow 3\ dvd\ z \Longrightarrow 2\ dvd\ (y::int) \Longrightarrow (\exists (x::int).\ 2*x = y)\ \&\ (\exists (k::int).\ 3*k = z)$
⟨*proof*⟩

theorem $\forall (x::nat).\ \exists (y::nat).\ (0::nat) \leq 5 \longrightarrow y = 5 + x$
⟨*proof*⟩

Very slow: about 55 seconds on a 1.8GHz 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 $\forall (x::int)\ y.\ x < y \longrightarrow 2 * x + 1 < 2 * y$
⟨*proof*⟩

theorem $\forall (x::int)\ y.\ 2 * x + 1 \neq 2 * y$
⟨*proof*⟩

theorem $\exists (x::int)\ y.\ 0 < x\ \&\ 0 \leq y\ \&\ 3 * x - 5 * y = 1$
⟨*proof*⟩

theorem ~ $(\exists (x::int)\ (y::int)\ (z::int).\ 4*x + (-6::int)*y = 1)$
⟨*proof*⟩

theorem $\forall (x::int).\ b < x \longrightarrow a \leq x$
⟨*proof*⟩

theorem ~ $(\exists (x::int).\ False)$
⟨*proof*⟩

theorem $\forall (x::int).\ (a::int) < 3 * x \longrightarrow b < 3 * x$
⟨*proof*⟩

theorem $\forall (x::int).\ (2\ dvd\ x) \longrightarrow (\exists (y::int).\ x = 2*y)$
⟨*proof*⟩

theorem $\forall (x::int).\ (2\ dvd\ x) \longrightarrow (\exists (y::int).\ x = 2*y)$
⟨*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 ~ $(\forall (x::int).$
$((2\ dvd\ x) = (\forall (y::int).\ x \neq 2*y+1)\ \mid$

$(\exists\,(q{::}int)\ (u{::}int)\ i.\ 3{*}i + 2{*}q - u < 17)$
$--> 0 < x \mid ((\sim\ 3\ dvd\ x)\ \&(x + 8 = 0))))$
⟨*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*⟩

Very slow: about 80 seconds on a 1.8GHz machine.

theorem $(\exists\,m{::}nat.\ n = 2 * m)\ --> (n + 1)\ div\ 2 = n\ div\ 2$
⟨*proof*⟩

theorem $(\exists\,m{::}int.\ n = 2 * m)\ --> (n + 1)\ div\ 2 = n\ div\ 2$
⟨*proof*⟩

end

24 Quantifier elimination for Presburger arithmetic

theory *Reflected-Presburger*
imports *Main*
begin

datatype *intterm* =
  Cst int
| *Var nat*
| *Neg intterm*
| *Add intterm intterm*
| *Sub intterm intterm*
| *Mult intterm intterm*

consts *I-intterm* :: *int list* ⇒ *intterm* ⇒ *int*
primrec
I-intterm ats (*Cst b*) = *b*
I-intterm ats (*Var n*) = (*ats*!*n*)
I-intterm ats (*Neg it*) = −(*I-intterm ats it*)
I-intterm ats (*Add it1 it2*) = (*I-intterm ats it1*) + (*I-intterm ats it2*)

I-intterm ats (Sub it1 it2) = (I-intterm ats it1) − (I-intterm ats it2)
I-intterm ats (Mult it1 it2) = (I-intterm ats it1) ∗ (I-intterm ats it2)

datatype *QF* =
  Lt intterm intterm
 |*Gt intterm intterm*
 |*Le intterm intterm*
 |*Ge intterm intterm*
 |*Eq intterm intterm*
 |*Divides intterm intterm*
 |*T*
 |*F*
 |*NOT QF*
 |*And QF QF*
 |*Or QF QF*
 |*Imp QF QF*
 |*Equ QF QF*
 |*QAll QF*
 |*QEx QF*

consts *qinterp* :: *int list* ⇒ *QF* ⇒ *bool*
primrec
qinterp ats (Lt it1 it2) = (I-intterm ats it1 < I-intterm ats it2)
qinterp ats (Gt it1 it2) = (I-intterm ats it1 > I-intterm ats it2)
qinterp ats (Le it1 it2) = (I-intterm ats it1 ≤ I-intterm ats it2)
qinterp ats (Ge it1 it2) = (I-intterm ats it1 ≥ I-intterm ats it2)
qinterp ats (Divides it1 it2) = (I-intterm ats it1 dvd I-intterm ats it2)
qinterp ats (Eq it1 it2) = (I-intterm ats it1 = I-intterm ats it2)
qinterp ats T = True
qinterp ats F = False
qinterp ats (NOT p) = (¬(qinterp ats p))
qinterp ats (And p q) = (qinterp ats p ∧ qinterp ats q)
qinterp ats (Or p q) = (qinterp ats p ∨ qinterp ats q)
qinterp ats (Imp p q) = (qinterp ats p ⟶ qinterp ats q)
qinterp ats (Equ p q) = (qinterp ats p = qinterp ats q)
qinterp ats (QAll p) = (∀ x. qinterp (x#ats) p)
qinterp ats (QEx p) = (∃ x. qinterp (x#ats) p)

consts *lift-bin*:: (*'a* ⇒ *'a* ⇒ *'b*) × *'a option* × *'a option* ⇒ *'b option*
recdef *lift-bin measure* (λ(*c*,*a*,*b*). *size a*)
lift-bin (*c*,*Some a*,*Some b*) = *Some* (*c a b*)
lift-bin (*c*,*x*, *y*) = *None*

lemma *lift-bin-Some*:
  assumes *ls*: *lift-bin* (*c*,*x*,*y*) = *Some t*

shows $(\exists\, a.\ x = Some\ a) \wedge (\exists\, b.\ y = Some\ b)$
⟨*proof*⟩

consts *lift-un*:: $('a \Rightarrow 'b) \Rightarrow 'a\ option \Rightarrow 'b\ option$
primrec
lift-un c None = *None*
lift-un c (*Some p*) = *Some* (*c p*)

consts *lift-qe*:: $('a \Rightarrow 'b\ option) \Rightarrow 'a\ option \Rightarrow 'b\ option$
primrec
lift-qe qe None = *None*
lift-qe qe (*Some p*) = *qe p*

consts *qelim* :: $(QF \Rightarrow QF\ option) \times QF \Rightarrow QF\ option$
recdef *qelim measure* $(\lambda(qe,p).\ size\ p)$
qelim (*qe*, (*QAll p*)) = *lift-un NOT* (*lift-qe qe* (*lift-un NOT* (*qelim* (*qe* ,*p*))))
qelim (*qe*, (*QEx p*)) = *lift-qe qe* (*qelim* (*qe*,*p*))
qelim (*qe*, (*And p q*)) = *lift-bin* (*And*, (*qelim* (*qe*, *p*)), (*qelim* (*qe*, *q*)))
qelim (*qe*, (*Or p q*)) = *lift-bin* (*Or*, (*qelim* (*qe*, *p*)), (*qelim* (*qe*, *q*)))
qelim (*qe*, (*Imp p q*)) = *lift-bin* (*Imp*, (*qelim* (*qe*, *p*)), (*qelim* (*qe*, *q*)))
qelim (*qe*, (*Equ p q*)) = *lift-bin* (*Equ*, (*qelim* (*qe*, *p*)), (*qelim* (*qe*, *q*)))
qelim (*qe*,*NOT p*) = *lift-un NOT* (*qelim* (*qe*,*p*))
qelim (*qe*, *p*) = *Some p*

consts *isqfree* :: $QF \Rightarrow bool$
recdef *isqfree measure size*
isqfree (*QAll p*) = *False*
isqfree (*QEx p*) = *False*
isqfree (*And p q*) = (*isqfree p* $\wedge$ *isqfree q*)
isqfree (*Or p q*) = (*isqfree p* $\wedge$ *isqfree q*)
isqfree (*Imp p q*) = (*isqfree p* $\wedge$ *isqfree q*)
isqfree (*Equ p q*) = (*isqfree p* $\wedge$ *isqfree q*)
isqfree (*NOT p*) = *isqfree p*
isqfree p = *True*

lemma *qelim-qfree*:
assumes *qeqf*: $(\bigwedge q\ q'.\ [\![isqfree\ q\ ;\ qe\ q = Some\ q']\!] \Longrightarrow\ isqfree\ q')$
shows *qff*:$\bigwedge p'.\ qelim\ (qe,\ p) = Some\ p' \Longrightarrow isqfree\ p'$
⟨*proof*⟩

lemma *qelim-corr*:
assumes *qecorr*: $(\bigwedge q\ q'\ ats.\ [\![isqfree\ q\ ;\ qe\ q = Some\ q']\!] \Longrightarrow\ (qinterp\ ats\ (QEx\ q)) = (qinterp\ ats\ q'))$
and *qeqf*: $(\bigwedge q\ q'.\ [\![isqfree\ q\ ;\ qe\ q = Some\ q']\!] \Longrightarrow\ isqfree\ q')$
shows *qff*:$\bigwedge p'\ ats.\ qelim\ (qe,\ p) = Some\ p' \Longrightarrow (qinterp\ ats\ p = qinterp\ ats$

p') (**is** $\bigwedge p'$ *ats*. *?Qe p p'* $\Longrightarrow$ (*?F ats p* = *?F ats p'*))
⟨*proof*⟩

consts *lgth* :: *QF* ⇒ *nat*
nnf :: *QF* ⇒ *QF*
primrec
lgth (*Lt it1 it2*) = *1*
lgth (*Gt it1 it2*) = *1*
lgth (*Le it1 it2*) = *1*
lgth (*Ge it1 it2*) = *1*
lgth (*Eq it1 it2*) = *1*
lgth (*Divides it1 it2*) = *1*
lgth T = *1*
lgth F = *1*
lgth (*NOT p*) = *1* + *lgth p*
lgth (*And p q*) = *1* + *lgth p* + *lgth q*
lgth (*Or p q*) = *1* + *lgth p* + *lgth q*
lgth (*Imp p q*) = *1* + *lgth p* + *lgth q*
lgth (*Equ p q*) = *1* + *lgth p* + *lgth q*
lgth (*QAll p*) = *1* + *lgth p*
lgth (*QEx p*) = *1* + *lgth p*

lemma [*simp*] :*0* < *lgth q*
⟨*proof*⟩

recdef *nnf measure* (λ*p*. *lgth p*)
nnf (*Lt it1 it2*) = *Le* (*Sub it1 it2*) (*Cst* (− *1*))
nnf (*Gt it1 it2*) = *Le* (*Sub it2 it1*) (*Cst* (− *1*))
nnf (*Le it1 it2*) = *Le it1 it2*
nnf (*Ge it1 it2*) = *Le it2 it1*
nnf (*Eq it1 it2*) = *Eq it2 it1*
nnf (*Divides d t*) = *Divides d t*
nnf T = *T*
nnf F = *F*
nnf (*And p q*) = *And* (*nnf p*) (*nnf q*)
nnf (*Or p q*) = *Or* (*nnf p*) (*nnf q*)
nnf (*Imp p q*) = *Or* (*nnf* (*NOT p*)) (*nnf q*)
nnf (*Equ p q*) = *Or* (*And* (*nnf p*) (*nnf q*))
(*And* (*nnf* (*NOT p*)) (*nnf* (*NOT q*)))
nnf (*NOT* (*Lt it1 it2*)) = (*Le it2 it1*)
nnf (*NOT* (*Gt it1 it2*)) = (*Le it1 it2*)
nnf (*NOT* (*Le it1 it2*)) = (*Le* (*Sub it2 it1*) (*Cst* (− *1*)))
nnf (*NOT* (*Ge it1 it2*)) = (*Le* (*Sub it1 it2*) (*Cst* (− *1*)))
nnf (*NOT* (*Eq it1 it2*)) = (*NOT* (*Eq it1 it2*))

$nnf\ (NOT\ (Divides\ d\ t)) = (NOT\ (Divides\ d\ t))$
$nnf\ (NOT\ T) = F$
$nnf\ (NOT\ F) = T$
$nnf\ (NOT\ (NOT\ p)) = (nnf\ p)$
$nnf\ (NOT\ (And\ p\ q)) = (Or\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q)))$
$nnf\ (NOT\ (Or\ p\ q)) = (And\ (nnf\ (NOT\ p))\ (nnf\ (NOT\ q)))$
$nnf\ (NOT\ (Imp\ p\ q)) = (And\ (nnf\ p)\ (nnf\ (NOT\ q)))$
$nnf\ (NOT\ (Equ\ p\ q)) = (Or\ (And\ (nnf\ p)\ (nnf\ (NOT\ q)))\ (And\ (nnf\ (NOT\ p))\ (nnf\ q)))$

consts $isnnf :: QF \Rightarrow bool$
recdef $isnnf\ measure\ (\lambda p.\ lgth\ p)$
$isnnf\ (Le\ it1\ it2) = True$
$isnnf\ (Eq\ it1\ it2) = True$
$isnnf\ (Divides\ d\ t) = True$
$isnnf\ T = True$
$isnnf\ F = True$
$isnnf\ (And\ p\ q) = (isnnf\ p \wedge isnnf\ q)$
$isnnf\ (Or\ p\ q) = (isnnf\ p \wedge isnnf\ q)$
$isnnf\ (NOT\ (Divides\ d\ t)) = True$
$isnnf\ (NOT\ (Eq\ it1\ it2)) = True$
$isnnf\ p = False$

lemma *nnf-corr*: $isqfree\ p \Longrightarrow qinterp\ ats\ p = qinterp\ ats\ (nnf\ p)$
⟨*proof*⟩

lemma *nnf-isnnf* : $isqfree\ p \Longrightarrow isnnf\ (nnf\ p)$
⟨*proof*⟩

lemma *nnf-isqfree*: $isnnf\ p \Longrightarrow isqfree\ p$
⟨*proof*⟩

lemma *nnf-qfree*: $isqfree\ p \Longrightarrow isqfree(nnf\ p)$
⟨*proof*⟩

consts $islinintterm :: intterm \Rightarrow bool$
recdef $islinintterm\ measure\ size$
$islinintterm\ (Cst\ i) = True$
$islinintterm\ (Add\ (Mult\ (Cst\ i)\ (Var\ n))\ (Cst\ i')) = (i \neq 0)$
$islinintterm\ (Add\ (Mult\ (Cst\ i)\ (Var\ n))\ (Add\ (Mult\ (Cst\ i')\ (Var\ n'))\ r)) = (\ i \neq 0 \wedge i' \neq 0 \wedge n < n' \wedge islinintterm\ \ (Add\ (Mult\ (Cst\ i')\ (Var\ n'))\ r))$
$islinintterm\ i = False$

lemma *islinintterm-subt*:
assumes *lr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
shows *islinintterm r*
⟨*proof*⟩

lemma *islinintterm-cnz*:
assumes *lr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
shows $i \neq 0$
⟨*proof*⟩

lemma *islininttermc0r*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var n*)) *r*) $\Longrightarrow$ ($c \neq 0$ $\wedge$ *islinintterm r*)
⟨*proof*⟩

consts *islintn* :: (*nat* × *intterm*) ⇒ *bool*
recdef *islintn measure* (λ (*n*,*t*). (*size t*))
islintn (*n0*, *Cst i*) = *True*
islintn (*n0*, *Add* (*Mult* (*Cst i*) (*Var n*)) *r*) = ($i \neq 0 \wedge n0 \leq n \wedge$ *islintn* (*n*+*1*,*r*))
islintn (*n0*, *t*) = *False*

constdefs *islint* :: *intterm* ⇒ *bool*
islint t ≡ *islintn*(*0*,*t*)

lemma *islinintterm-eq-islint*: *islinintterm t* = *islint t*
⟨*proof*⟩

lemma *islintn-mon*:
assumes *lin*: *islintn* (*n*,*t*)
and *mgen*: $m \leq n$
shows *islintn*(*m*,*t*)
⟨*proof*⟩

lemma *islintn-subt*:
assumes *lint*: *islintn*(*n*,*Add* (*Mult* (*Cst i*) (*Var m*)) *r*)
shows *islintn* (*m*+*1*,*r*)
⟨*proof*⟩

lemma *nth-pos*: $0 < n \longrightarrow (x\#xs) ! n = (y\#xs) ! n$
⟨*proof*⟩

lemma *nth-pos2*: $0 < n \Longrightarrow (x\#xs) ! n = xs ! (n - 1)$
⟨*proof*⟩

lemma *intterm-novar0*:
assumes *lin*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
shows *I-intterm* (*x*#*ats*) *r* = *I-intterm* (*y*#*ats*) *r*
⟨*proof*⟩

lemma *linterm-novar0*:
assumes *lin*: *islintn* (*n*,*t*)
and *npos*: *0* < *n*
shows *I-intterm* (*x*#*ats*) *t* = *I-intterm* (*y*#*ats*) *t*
⟨*proof*⟩

lemma *dvd-period*:
assumes *advdd*: (*a*::*int*) *dvd d*
shows (*a dvd* (*x* + *t*)) = (*a dvd* ((*x*+ *c*∗*d*) + *t*))
⟨*proof*⟩

consts *lin-add* :: *intterm* × *intterm* ⇒ *intterm*
recdef *lin-add measure* (λ(*x*,*y*). ((*size x*) + (*size y*)))
lin-add (*Add* (*Mult* (*Cst c1*) (*Var n1*)) (*r1*),*Add* (*Mult* (*Cst c2*) (*Var n2*)) (*r2*))
=
(*if n1*=*n2 then*
(*let c* = *Cst* (*c1* + *c2*)
in (*if c1*+*c2*=*0 then lin-add*(*r1*,*r2*) *else Add* (*Mult c* (*Var n1*)) (*lin-add* (*r1*,*r2*))))
else if n1 ≤ *n2 then* (*Add* (*Mult* (*Cst c1*) (*Var n1*)) (*lin-add* (*r1*,*Add* (*Mult* (*Cst c2*) (*Var n2*)) (*r2*))))
else (*Add* (*Mult* (*Cst c2*) (*Var n2*)) (*lin-add* (*Add* (*Mult* (*Cst c1*) (*Var n1*)) *r1*,*r2*))))
lin-add (*Add* (*Mult* (*Cst c1*) (*Var n1*)) (*r1*),*Cst b*) =
(*Add* (*Mult* (*Cst c1*) (*Var n1*)) (*lin-add* (*r1*, *Cst b*)))
lin-add (*Cst x*,*Add* (*Mult* (*Cst c2*) (*Var n2*)) (*r2*)) =
Add (*Mult* (*Cst c2*) (*Var n2*)) (*lin-add* (*Cst x*,*r2*))
lin-add (*Cst b1*, *Cst b2*) = *Cst* (*b1*+*b2*)

lemma *lin-add-cst-corr*:
assumes *blin* : *islintn*(*n0*,*b*)
shows *I-intterm ats* (*lin-add* (*Cst a*,*b*)) = (*I-intterm ats* (*Add* (*Cst a*) *b*))
⟨*proof*⟩

lemma *lin-add-cst-corr2*:
assumes *blin* : *islintn*(*n0*,*b*)
shows *I-intterm ats* (*lin-add* (*b*,*Cst a*)) = (*I-intterm ats* (*Add b* (*Cst a*)))
⟨*proof*⟩

lemma *lin-add-corrh*: ⋀ *n01 n02*. ⟦ *islintn* (*n01*,*a*) ; *islintn* (*n02*,*b*)⟧
⟹ *I-intterm ats* (*lin-add*(*a*,*b*)) = *I-intterm ats* (*Add a b*)

⟨*proof*⟩

lemma *lin-add-corr*:
assumes *lina*: *islinintterm a*
and *linb*: *islinintterm b*
shows *I-intterm ats* (*lin-add* (*a*,*b*)) = (*I-intterm ats* (*Add a b*))
⟨*proof*⟩

lemma *lin-add-cst-lint*:
assumes *lin*: *islintn* (*n0*,*b*)
shows *islintn* (*n0*, *lin-add* (*Cst i*, *b*))
⟨*proof*⟩

lemma *lin-add-cst-lint2*:
assumes *lin*: *islintn* (*n0*,*b*)
shows *islintn* (*n0*, *lin-add* (*b*,*Cst i*))
⟨*proof*⟩

lemma *lin-add-lint*: ⋀ *n0 n01 n02*. ⟦ *islintn* (*n01*,*a*) ; *islintn* (*n02*,*b*); *n0* ≤ *min n01 n02* ⟧
⟹ *islintn* (*n0*, *lin-add* (*a*,*b*))
⟨*proof*⟩

lemma *lin-add-lin*:
assumes *lina*: *islinintterm a*
and *linb*: *islinintterm b*
shows *islinintterm* (*lin-add* (*a*,*b*))
⟨*proof*⟩

consts *lin-mul* :: *int* × *intterm* ⇒ *intterm*
recdef *lin-mul measure* (λ(*c*,*t*). *size t*)
lin-mul (*c*,*Cst i*) = (*Cst* (*c*∗*i*))
lin-mul (*c*,*Add* (*Mult* (*Cst c'*) (*Var n*)) *r*) =
(*if c* = *0 then* (*Cst 0*) *else*
(*Add* (*Mult* (*Cst* (*c*∗*c'*)) (*Var n*)) (*lin-mul* (*c*,*r*))))

lemma *zmult-zadd-distrib*[*simp*]: (*a*::*int*) ∗ (*b*+*c*) = *a*∗*b* + *a*∗*c*
⟨*proof*⟩

lemma *lin-mul-corr*:
assumes *lint*: *islinintterm t*
shows *I-intterm ats* (*lin-mul* (*c*,*t*)) = *I-intterm ats* (*Mult* (*Cst c*) *t*)
⟨*proof*⟩

lemma *lin-mul-lin*:
assumes *lint*: *islinintterm t*
shows *islinintterm* (*lin-mul*(*c*,*t*))
⟨*proof*⟩

lemma *lin-mul0*:
assumes *lint*: *islinintterm t*
shows *lin-mul*(*0*,*t*) = *Cst 0*
⟨*proof*⟩

lemma *lin-mul-lintn*:
⋀ *m*. *islintn*(*m*,*t*) ⟹ *islintn*(*m*,*lin-mul*(*l*,*t*))
⟨*proof*⟩

constdefs *lin-neg* :: *intterm* ⇒ *intterm*
lin-neg i == *lin-mul* ((−*1*::*int*),*i*)

lemma *lin-neg-corr*:
assumes *lint*: *islinintterm t*
shows *I-intterm ats* (*lin-neg t*) = *I-intterm ats* (*Neg t*)
⟨*proof*⟩

lemma *lin-neg-lin*:
assumes *lint*: *islinintterm t*
shows *islinintterm* (*lin-neg t*)
⟨*proof*⟩

lemma *lin-neg-idemp*:
assumes *lini*: *islinintterm i*
shows *lin-neg* (*lin-neg i*) = *i*
⟨*proof*⟩

lemma *lin-neg-lin-add-distrib*:
assumes *lina* : *islinintterm a*
and *linb* :*islinintterm b*
shows *lin-neg* (*lin-add*(*a*,*b*)) = *lin-add* (*lin-neg a*, *lin-neg b*)
⟨*proof*⟩

consts *linearize* :: *intterm* ⇒ *intterm option*
recdef *linearize measure* (λ*t*. *size t*)
linearize (*Cst b*) = *Some* (*Cst b*)
linearize (*Var n*) = *Some* (*Add* (*Mult* (*Cst 1*) (*Var n*)) (*Cst 0*))
linearize (*Neg i*) = *lift-un lin-neg* (*linearize i*)

linearize (*Add i j*) = *lift-bin*(λ *x*. λ *y*. *lin-add*(*x*,*y*), *linearize i*, *linearize j*)
linearize (*Sub i j*) =
  lift-bin(λ *x*. λ *y*. *lin-add*(*x*,*lin-neg y*), *linearize i*, *linearize j*)
linearize (*Mult i j*) =
  (*case linearize i of*
  None ⇒ *None*
  | *Some li* ⇒ (*case li of*
     Cst b ⇒ (*case linearize j of*
      None ⇒ *None*
     | (*Some lj*) ⇒ *Some* (*lin-mul*(*b*,*lj*)))
  | - ⇒ (*case linearize j of*
      None ⇒ *None*
    | (*Some lj*) ⇒ (*case lj of*
        Cst b ⇒ *Some* (*lin-mul* (*b*,*li*))
      | - ⇒ *None*))))

lemma *linearize-linear1*:
  assumes *lin*: *linearize t* ≠ *None*
  shows *islinintterm* (*the* (*linearize t*))
⟨*proof*⟩

lemma *linearize-linear*: ⋀ *t'*. *linearize t* = *Some t'* ⟹ *islinintterm t'*
⟨*proof*⟩

lemma *linearize-corr1*:
  assumes *lin*: *linearize t* ≠ *None*
  shows *I-intterm ats t* = *I-intterm ats* (*the* (*linearize t*))
⟨*proof*⟩

lemma *linearize-corr*: ⋀ *t'*. *linearize t* = *Some t'* ⟹ *I-intterm ats t* = *I-intterm ats t'*
⟨*proof*⟩

consts *linform* :: *QF* ⇒ *QF option*
primrec
linform (*Le it1 it2*) =
  lift-bin(λ*x*. λ*y*. *Le* (*lin-add*(*x*,*lin-neg y*)) (*Cst 0*),*linearize it1*, *linearize it2*)
linform (*Eq it1 it2*) =
  lift-bin(λ*x*. λ*y*. *Eq* (*lin-add*(*x*,*lin-neg y*)) (*Cst 0*),*linearize it1*, *linearize it2*)
linform (*Divides d t*) =
  (*case linearize d of*
    None ⇒ *None*
   | *Some ld* ⇒ (*case ld of*
         Cst b ⇒
           (*if* (*b*=*0*) *then None*

else
(*case linearize t of*
None ⇒ *None*
| *Some lt* ⇒ *Some* (*Divides ld lt*)))
| - ⇒ *None*))
linform T = *Some T*
linform F = *Some F*
linform (*NOT p*) = *lift-un NOT* (*linform p*)
linform (*And p q*)= *lift-bin*(λ*f*. λ*g*. *And f g*, *linform p*, *linform q*)
linform (*Or p q*) = *lift-bin*(λ*f*. λ*g*. *Or f g*, *linform p*, *linform q*)

consts *islinform* :: *QF* ⇒ *bool*
recdef *islinform measure size*
islinform (*Le it* (*Cst i*)) = (*i*=*0* ∧ *islinintterm it*)
islinform (*Eq it* (*Cst i*)) = (*i*=*0* ∧ *islinintterm it*)
islinform (*Divides* (*Cst d*) *t*) = (*d* ≠ *0* ∧ *islinintterm t*)
islinform T = *True*
islinform F = *True*
islinform (*NOT* (*Divides* (*Cst d*) *t*)) = (*d* ≠ *0* ∧ *islinintterm t*)
islinform (*NOT* (*Eq it* (*Cst i*))) = (*i*=*0* ∧ *islinintterm it*)
islinform (*And p q*)= ((*islinform p*) ∧ (*islinform q*))
islinform (*Or p q*) = ((*islinform p*) ∧ (*islinform q*))
islinform p = *False*

lemma *linform-nnf*:
assumes *nnfp*: *isnnf p*
shows ⋀ *p'*. ⟦*linform p* = *Some p'*⟧ ⟹ *isnnf p'*
⟨*proof*⟩

lemma *linform-corr*: ⋀ *lp*. ⟦ *isnnf p* ; *linform p* = *Some lp* ⟧ ⟹
(*qinterp ats p* = *qinterp ats lp*)
⟨*proof*⟩

lemma *linform-lin*: ⋀ *lp*. ⟦ *isnnf p* ; *linform p* = *Some lp*⟧ ⟹ *islinform lp*
⟨*proof*⟩

lemma *linform-isnnf*: *islinform p* ⟹ *isnnf p*
⟨*proof*⟩

lemma *linform-isqfree*: *islinform p* ⟹ *isqfree p*
⟨*proof*⟩

lemma *linform-qfree*: $\bigwedge p'.\ [\![\ isnnf\ p\ ;\ linform\ p = Some\ p'\ ]\!] \Longrightarrow isqfree\ p'$
⟨*proof*⟩

constdefs *lcm* :: $nat \times nat \Rightarrow nat$
$lcm \equiv (\lambda(m,n).\ m*n\ div\ gcd(m,n))$

constdefs *ilcm* :: $int \Rightarrow int \Rightarrow int$
$ilcm \equiv \lambda i.\lambda j.\ int\ (lcm(nat(abs\ i),nat(abs\ j)))$

lemma *lcm-dvd1*:
assumes *mpos*: $m > 0$
and *npos*: $n>0$
shows $m\ dvd\ (lcm(m,n))$
⟨*proof*⟩

lemma *lcm-dvd2*:
assumes *mpos*: $m > 0$
and *npos*: $n>0$
shows $n\ dvd\ (lcm(m,n))$
⟨*proof*⟩

lemma *ilcm-dvd1*:
assumes *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $a\ dvd\ (ilcm\ a\ b)$
⟨*proof*⟩

lemma *ilcm-dvd2*:
assumes *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $b\ dvd\ (ilcm\ a\ b)$
⟨*proof*⟩

lemma *zdvd-self-abs1*: $(d::int)\ dvd\ (abs\ d)$
⟨*proof*⟩

lemma *zdvd-self-abs2*: $(abs\ (d::int))\ dvd\ d$
⟨*proof*⟩

lemma *lcm-pos*:
assumes *mpos*: $m > 0$
and *npos*: $n>0$
shows $lcm\ (m,n) > 0$

⟨*proof*⟩

lemma *ilcm-pos*:
  assumes *apos*: *0 < a*
  and *bpos*: *0 < b*
  shows *0 < ilcm a b*
⟨*proof*⟩

consts *formlcm* :: *QF ⇒ int*
recdef *formlcm measure size*
formlcm (Le (Add (Mult (Cst c) (Var 0)) r) (Cst i)) = abs c
formlcm (Eq (Add (Mult (Cst c) (Var 0)) r) (Cst i)) = abs c
formlcm (Divides (Cst d) (Add (Mult (Cst c) (Var 0)) r)) = abs c
formlcm (NOT p) = formlcm p
formlcm (And p q)= ilcm (formlcm p) (formlcm q)
formlcm (Or p q) = ilcm (formlcm p) (formlcm q)
formlcm p = 1

consts *divideallc*:: *int × QF ⇒ bool*
recdef *divideallc measure* (λ(*i*,*p*). *size p*)
divideallc (l,Le (Add (Mult (Cst c) (Var 0)) r) (Cst i)) = (c dvd l)
divideallc (l,Eq (Add (Mult (Cst c) (Var 0)) r) (Cst i)) = (c dvd l)
divideallc(l,Divides (Cst d) (Add (Mult (Cst c) (Var 0)) r)) = (c dvd l)
divideallc (l,NOT p) = divideallc(l,p)
divideallc (l,And p q) = (divideallc (l,p) ∧ divideallc (l,q))
divideallc (l,Or p q) = (divideallc (l,p) ∧ divideallc (l,q))
divideallc p = True

lemma *formlcm-pos*:
  assumes *linp*: *islinform p*
  shows *0 < formlcm p*
⟨*proof*⟩

lemma *divideallc-mono*: ⋀ *c*. ⟦ *divideallc*(*c*,*p*) ; *c dvd d*⟧ ⟹ *divideallc* (*d*,*p*)
⟨*proof*⟩

lemma *formlcm-divideallc*:
  assumes *linp*: *islinform p*
  shows *divideallc*(*formlcm p*, *p*)
⟨*proof*⟩

consts *adjustcoeff* :: *int × QF ⇒ QF*
recdef *adjustcoeff measure* (λ(*l*,*p*). *size p*)

```
adjustcoeff (l,(Le (Add (Mult (Cst c) (Var 0)) r) (Cst i))) =
  (if c≤0 then
  Le (Add (Mult (Cst −1) (Var 0)) (lin-mul (− (l div c), r))) (Cst (0::int))
  else
  Le (Add (Mult (Cst 1) (Var 0)) (lin-mul (l div c, r))) (Cst (0::int)))
adjustcoeff (l,(Eq (Add (Mult (Cst c) (Var 0)) r) (Cst i))) =
  (Eq (Add (Mult (Cst 1) (Var 0)) (lin-mul (l div c, r))) (Cst (0::int)))
adjustcoeff (l,Divides (Cst d) (Add (Mult (Cst c) (Var 0)) r)) =
  Divides (Cst ((l div c) ∗ d))
  (Add (Mult (Cst 1) (Var 0)) (lin-mul (l div c, r)))
adjustcoeff (l,NOT (Divides (Cst d) (Add (Mult (Cst c) (Var 0)) r))) = NOT
(Divides (Cst ((l div c) ∗ d))
  (Add (Mult (Cst 1) (Var 0)) (lin-mul (l div c, r))))
adjustcoeff (l,(NOT(Eq (Add (Mult (Cst c) (Var 0)) r) (Cst i)))) =
  (NOT(Eq (Add (Mult (Cst 1) (Var 0)) (lin-mul (l div c, r))) (Cst (0::int))))
adjustcoeff (l,And p q) = And (adjustcoeff (l,p)) (adjustcoeff(l,q))
adjustcoeff (l,Or p q) = Or (adjustcoeff (l,p)) (adjustcoeff(l,q))
adjustcoeff (l,p) = p
```

constdefs *unitycoeff* :: $QF \Rightarrow QF$

```
  unitycoeff p ==
  (let l = formlcm p;
       p′ = adjustcoeff (l,p)
   in (if l=1 then p′ else
      (And (Divides (Cst l) (Add (Mult (Cst 1) (Var 0)) (Cst 0))) p′)))
```

consts *isunified* :: $QF \Rightarrow bool$

recdef *isunified measure size*

```
isunified (Le (Add (Mult (Cst i) (Var 0)) r) (Cst z)) =
  ((abs i) = 1 ∧ (islinform(Le (Add (Mult (Cst i) (Var 0)) r) (Cst z))))
isunified (Eq (Add (Mult (Cst i) (Var 0)) r) (Cst z)) =
  ((abs i) = 1 ∧ (islinform(Le (Add (Mult (Cst i) (Var 0)) r) (Cst z))))
isunified (NOT(Eq (Add (Mult (Cst i) (Var 0)) r) (Cst z))) =
  ((abs i) = 1 ∧ (islinform(Le (Add (Mult (Cst i) (Var 0)) r) (Cst z))))
isunified (Divides (Cst d) (Add (Mult (Cst i) (Var 0)) r)) =
  ((abs i) = 1 ∧ (islinform(Divides (Cst d) (Add (Mult (Cst i) (Var 0)) r))))
isunified (NOT(Divides (Cst d) (Add (Mult (Cst i) (Var 0)) r))) =
  ((abs i) = 1 ∧ (islinform(NOT(Divides (Cst d) (Add (Mult (Cst i) (Var 0))
r)))))
isunified (And p q) = (isunified p ∧ isunified q)
isunified (Or p q) = (isunified p ∧ isunified q)
isunified p = islinform p
```

lemma *unified-islinform*: *isunified* $p \Longrightarrow$ *islinform* p

⟨*proof*⟩

lemma *adjustcoeff-lenpos*:
$0 < n \Longrightarrow adjustcoeff\ (l, Le\ (Add\ (Mult\ (Cst\ i)\ (Var\ n))\ r)\ (Cst\ c)) =$
$Le\ (Add\ (Mult\ (Cst\ i)\ (Var\ n))\ r)\ (Cst\ c)$
⟨*proof*⟩

lemma *adjustcoeff-eqnpos*:
$0 < n \Longrightarrow adjustcoeff\ (l, Eq\ (Add\ (Mult\ (Cst\ i)\ (Var\ n))\ r)\ (Cst\ c)) =$
$Eq\ (Add\ (Mult\ (Cst\ i)\ (Var\ n))\ r)\ (Cst\ c)$
⟨*proof*⟩

lemma *zmult-zle-mono*: $(i::int) \leq j \Longrightarrow 0 \leq k \Longrightarrow k * i \leq k * j$
⟨*proof*⟩

lemma *zmult-zle-mono-eq*:
assumes *kpos*: $0 < k$
shows $((i::int) \leq j) = (k*i \leq k*j)$ (**is** $?P = ?Q$)
⟨*proof*⟩

lemma *adjustcoeff-le-corr*:
assumes *lpos*: $0 < l$
and *ipos*: $0 < (i::int)$
and *dvd*: $i\ dvd\ l$
shows $(i*x + r \leq 0) = (l*x + ((l\ div\ i)*r) \leq 0)$
⟨*proof*⟩

lemma *adjustcoeff-le-corr2*:
assumes *lpos*: $0 < l$
and *ineg*: $(i::int) < 0$
and *dvd*: $i\ dvd\ l$
shows $(i*x + r \leq 0) = ((-l)*x + ((-(l\ div\ i))*r) \leq 0)$
⟨*proof*⟩

lemma *dvd-div-pos*:
assumes *bpos*: $0 < (b::int)$
and *anz*: $a \neq 0$
and *dvd*: $a\ dvd\ b$
shows $(b\ div\ a)*a = b$
⟨*proof*⟩

lemma *adjustcoeff-eq-corr*:
assumes *lpos*: $(0::int) < l$
and *inz*: $i \neq 0$
and *dvd*: $i\ dvd\ l$

shows $(i*x + r = 0) = (l*x + ((l\ div\ i)*r) = 0)$
⟨*proof*⟩

lemma *adjustcoeff-corr*:
assumes *linp*: *islinform p*
and *alldvd*: *divideallc* (*l*,*p*)
and *lpos*: $0 < l$
shows *qinterp* $(a\#ats)\ p$ = *qinterp* $((a*l)\#ats)$ (*adjustcoeff*$(l,\ p)$)
⟨*proof*⟩

lemma *unitycoeff-corr*:
assumes *linp*: *islinform p*
shows *qinterp ats* (*QEx p*) = *qinterp ats* (*QEx* (*unitycoeff p*))
⟨*proof*⟩

lemma *adjustcoeff-unified*:
assumes *linp*: *islinform p*
and *dvdc*: *divideallc*(*l*,*p*)
and *lpos*: $l > 0$
shows *isunified* (*adjustcoeff*$(l,\ p)$)
⟨*proof*⟩

lemma *adjustcoeff-lcm-unified*:
assumes *linp*: *islinform p*
shows *isunified* (*adjustcoeff*(*formlcm p*, *p*))
⟨*proof*⟩

lemma *unitycoeff-unified*:
assumes *linp*: *islinform p*
shows *isunified* (*unitycoeff p*)
⟨*proof*⟩

lemma *unified-isnnf*:
assumes *unifp*: *isunified p*
shows *isnnf p*
⟨*proof*⟩

lemma *unified-isqfree*: *isunified p*⟹ *isqfree p*
⟨*proof*⟩

consts *minusinf* :: *QF* ⇒ *QF*

$plusinf :: QF \Rightarrow QF$
$aset \quad :: QF \Rightarrow intterm\ list$
$bset \quad :: QF \Rightarrow intterm\ list$

recdef *minusinf measure size*
$minusinf\ (Le\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) =$
$\ (if\ c < 0\ then\ F\ else\ T)$
$minusinf\ (Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) = F$
$minusinf\ (NOT(Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z)) = T$
$minusinf\ (And\ p\ q) = And\ (minusinf\ p)\ (minusinf\ q)$
$minusinf\ (Or\ p\ q) = Or\ (minusinf\ p)\ (minusinf\ q)$
$minusinf\ p = p$

recdef *plusinf measure size*
$plusinf\ (Le\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) =$
$\ (if\ c < 0\ then\ T\ else\ F)$
$plusinf\ (Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) = F$
$plusinf\ (NOT\ (Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z)) = T$
$plusinf\ (And\ p\ q) = And\ (plusinf\ p)\ (plusinf\ q)$
$plusinf\ (Or\ p\ q) = Or\ (plusinf\ p)\ (plusinf\ q)$
$plusinf\ p = p$

recdef *bset measure size*
$bset\ (Le\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) =$
$\ (if\ c < 0\ then\ [lin\text{-}add(r,(Cst\ -1)),\ r]$
$\qquad else\ [lin\text{-}add(lin\text{-}neg\ r,(Cst\ -1))])$
$bset\ (Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) =$
$\ (if\ c < 0\ then\ [lin\text{-}add(r,(Cst\ -1))]$
$\qquad else\ [lin\text{-}add(lin\text{-}neg\ r,(Cst\ -1))])$
$bset\ (NOT(Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z)) =$
$\ (if\ c < 0\ then\ [r]$
$\qquad else\ [lin\text{-}neg\ r])$
$bset\ (And\ p\ q) = (bset\ p)\ @\ (bset\ q)$
$bset\ (Or\ p\ q) = (bset\ p)\ @\ (bset\ q)$
$bset\ p = []$

recdef *aset measure size*
$aset\ (Le\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) =$
$\ (if\ c < 0\ then\ [lin\text{-}add\ (r,\ Cst\ 1)]$
$\qquad else\ [lin\text{-}add\ (lin\text{-}neg\ r,\ Cst\ 1),\ lin\text{-}neg\ r])$
$aset\ (Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z) =$
$\ (if\ c < 0\ then\ [lin\text{-}add(r,(Cst\ 1))]$
$\quad else\ [lin\text{-}add(lin\text{-}neg\ r,(Cst\ 1))])$
$aset\ (NOT(Eq\ (Add\ (Mult\ (Cst\ c)\ (Var\ 0))\ r)\ z)) =$
$\ (if\ c < 0\ then\ [r]$
$\quad else\ [lin\text{-}neg\ r])$
$aset\ (And\ p\ q) = (aset\ p)\ @\ (aset\ q)$
$aset\ (Or\ p\ q) = (aset\ p)\ @\ (aset\ q)$
$aset\ p = []$

consts *divlcm* :: *QF* ⇒ *int*
recdef *divlcm measure size*
divlcm (*Divides* (*Cst d*) (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*)) = (*abs d*)
divlcm (*NOT p*) = *divlcm p*
divlcm (*And p q*)= *ilcm* (*divlcm p*) (*divlcm q*)
divlcm (*Or p q*) = *ilcm* (*divlcm p*) (*divlcm q*)
divlcm p = *1*

consts *alldivide* :: *int* × *QF* ⇒ *bool*
recdef *alldivide measure* (%(*d*,*p*). *size p*)
alldivide (*d*,(*Divides* (*Cst d'*) (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*))) =
(*d' dvd d*)
alldivide (*d*,(*NOT p*)) = *alldivide* (*d*,*p*)
alldivide (*d*,(*And p q*)) = (*alldivide* (*d*,*p*) ∧ *alldivide* (*d*,*q*))
alldivide (*d*,(*Or p q*)) = ((*alldivide* (*d*,*p*)) ∧ (*alldivide* (*d*,*q*)))
alldivide (*d*,*p*) = *True*

lemma *alldivide-mono*: ⋀ *d'*. ⟦ *alldivide* (*d*,*p*) ; *d dvd d'*⟧ ⟹ *alldivide* (*d'*,*p*)
⟨*proof*⟩

lemma *zdvd-eq-zdvd-abs*: (*d*::*int*) *dvd d'* = (*d dvd* (*abs d'*))
⟨*proof*⟩

lemma *zdvd-refl-abs*: (*d*::*int*) *dvd* (*abs d*)
⟨*proof*⟩

lemma *divlcm-pos*:
assumes
linp: *islinform p*
shows *0* < *divlcm p*
⟨*proof*⟩

lemma *nz-le*: (*x*::*int*) > *0* ⟹ *x* ≠ *0* ⟨*proof*⟩

lemma *divlcm-corr*:
assumes
linp: *islinform p*
shows *alldivide* (*divlcm p*,*p*)
⟨*proof*⟩

lemma *minusinf-eq*:
assumes *unifp*: *isunified p*
shows $\exists\ z.\ \forall\ x.\ x < z \longrightarrow (qinterp\ (x\#ats)\ p = qinterp\ (x\#ats)\ (minusinf\ p))$
⟨*proof*⟩

lemma *minusinf-repeats*:
assumes *alldvd*: *alldivide* (d,p)
and *unity*: *isunified p*
shows $qinterp\ (x\#ats)\ (minusinf\ p) = qinterp\ ((x + c*d)\#ats)\ (minusinf\ p)$
⟨*proof*⟩

lemma *minusinf-repeats2*:
assumes *alldvd*: *alldivide* (d,p)
and *unity*: *isunified p*
shows $\forall\ x\ k.\ (qinterp\ (x\#ats)\ (minusinf\ p) = qinterp\ ((x - k*d)\#ats)\ (minusinf\ p))$
(**is** $\forall\ x\ k.\ ?P\ x = ?P\ (x - k*d)$)
⟨*proof*⟩

lemma *minusinf-lemma*:
assumes *unifp*: *isunified p*
and *exminf*: $\exists\ j \in \{1\ ..d\}.\ qinterp\ (j\#ats)\ (minusinf\ p)$ (**is** $\exists\ j \in \{1\ ..\ d\}.\ ?P1\ j$)
shows $\exists\ x.\ qinterp\ (x\#ats)\ p$ (**is** $\exists\ x.\ ?P\ x$)
⟨*proof*⟩

lemma *minusinf-disj*:
assumes *unifp*: *isunified p*
shows $(\exists\ x.\ qinterp\ (x\#ats)\ (minusinf\ p)) =$
$(\exists\ j \in \{\ 1..\ divlcm\ p\}.\ qinterp\ (j\#ats)\ (minusinf\ p))$
(**is** $(\exists\ x.\ ?P\ x) = (\exists\ j \in \{\ 1..\ ?d\}.\ ?P\ j)$)
⟨*proof*⟩

lemma *minusinf-qfree*:
assumes *linp* : *islinform p*
shows *isqfree* (*minusinf p*)
⟨*proof*⟩

lemma *bset-lin*:
assumes *unifp*: *isunified p*
shows $\forall\ b \in set\ (bset\ p).\ islinintterm\ b$

⟨*proof*⟩

lemma *bset-disj-repeat*:
assumes *unifp*: *isunified p*
and *alldvd*: *alldivide* (*d*,*p*)
and *dpos*: *0* < *d*
and *nob*: (*qinterp* (*x*#*ats*) *q*) ∧ ¬(∃*j*∈ {*1* .. *d*}. ∃ *b* ∈ *set* (*bset p*). (*qinterp* (((*I-intterm* (*a*#*ats*) *b*) + *j*)#*ats*) *q*)) ∧(*qinterp* (*x*#*ats*) *p*)
(**is** *?Q x* ∧ ¬(∃ *j*∈ {*1*.. *d*}. ∃ *b*∈ *?B*. *?Q* (*?I a b* + *j*)) ∧ *?P x*)
shows *?P* (*x* −*d*)
⟨*proof*⟩

lemma *bset-disj-repeat2*:
assumes *unifp*: *isunified p*

shows ∀ *x*. ¬(∃*j*∈ {*1* .. (*divlcm p*)}. ∃ *b* ∈ *set* (*bset p*).
(*qinterp* (((*I-intterm* (*a*#*ats*) *b*) + *j*)#*ats*) *p*))
⟶ (*qinterp* (*x*#*ats*) *p*) ⟶ (*qinterp* ((*x* − (*divlcm p*))#*ats*) *p*)
(**is** ∀ *x*. ¬(∃ *j*∈ {*1* .. *?d*}. ∃ *b*∈ *?B*. *?P* (*?I a b* + *j*)) ⟶ *?P x* ⟶ *?P* (*x* − *?d*))
⟨*proof*⟩

lemma *cooper-mi-eq*:
assumes *unifp* : *isunified p*
shows (∃ *x*. *qinterp* (*x*#*ats*) *p*) =
((∃ *j* ∈ {*1* .. (*divlcm p*)}. *qinterp* (*j*#*ats*) (*minusinf p*)) ∨
(∃ *j* ∈ {*1* .. (*divlcm p*)}. ∃ *b* ∈ *set* (*bset p*).
qinterp (((*I-intterm* (*a*#*ats*) *b*) + *j*)#*ats*) *p*))
(**is** (∃ *x*. *?P x*) = ((∃ *j*∈ {*1* .. *?d*}. *?MP j*) ∨ (∃ *j* ∈ *?D*. ∃ *b*∈ *?B*. *?P* (*?I a b* + *j*))))
⟨*proof*⟩

consts *mirror*:: *QF* ⇒ *QF*
recdef *mirror measure size*
mirror (*Le* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*) =
(*Le* (*Add* (*Mult* (*Cst* (− *c*)) (*Var 0*)) *r*) *z*)
mirror (*Eq* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*) =
(*Eq* (*Add* (*Mult* (*Cst* (− *c*)) (*Var 0*)) *r*) *z*)
mirror (*Divides* (*Cst d*) (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*)) =
(*Divides* (*Cst d*) (*Add* (*Mult* (*Cst* (− *c*)) (*Var 0*)) *r*))
mirror (*NOT*(*Divides* (*Cst d*) (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*))) =
(*NOT*(*Divides* (*Cst d*) (*Add* (*Mult* (*Cst* (− *c*)) (*Var 0*)) *r*)))
mirror (*NOT*(*Eq* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*)) =
(*NOT*(*Eq* (*Add* (*Mult* (*Cst* (− *c*)) (*Var 0*)) *r*) *z*))
mirror (*And p q*) = *And* (*mirror p*) (*mirror q*)

mirror (*Or p q*) = *Or* (*mirror p*) (*mirror q*)
mirror p = *p*

lemma[*simp*]: (*abs* (*i*::*int*) = *1*) = (*i* =*1* ∨ *i* = −*1*) ⟨*proof*⟩
lemma *mirror-unified*:
assumes *unif*: *isunified p*
shows *isunified* (*mirror p*)
⟨*proof*⟩

lemma *plusinf-eq-minusinf-mirror*:
assumes *unifp*: *isunified p*
shows (*qinterp* (*x*#*ats*) (*plusinf p*)) = (*qinterp* ((− *x*)#*ats*) (*minusinf* (*mirror p*)))
⟨*proof*⟩

lemma *aset-eq-bset-mirror*:
assumes *unifp*: *isunified p*
shows *set* (*aset p*) = *set* (*map lin-neg* (*bset* (*mirror p*)))
⟨*proof*⟩

lemma *aset-eq-bset-mirror2*:
assumes *unifp*: *isunified p*
shows *aset p* = *map lin-neg* (*bset* (*mirror p*))
⟨*proof*⟩

lemma *divlcm-mirror-eq*:
assumes *unifp*: *isunified p*
shows *divlcm p* = *divlcm* (*mirror p*)
⟨*proof*⟩

lemma *mirror-interp*:
assumes *unifp*: *isunified p*
shows (*qinterp* (*x*#*ats*) *p*) = (*qinterp* ((− *x*)#*ats*) (*mirror p*)) (**is** *?P x* = *?MP* (−*x*))
⟨*proof*⟩

lemma *mirror-interp2*:
assumes *unifp*: *islinform p*
shows (*qinterp* (*x*#*ats*) *p*) = (*qinterp* ((− *x*)#*ats*) (*mirror p*)) (**is** *?P x* = *?MP* (−*x*))
⟨*proof*⟩

lemma *mirror-ex*:
assumes *unifp*: *isunified p*
shows $(\exists\ x.\ (qinterp\ (x\#ats)\ p)) = (\exists\ y.\ (qinterp\ (y\#ats)\ (mirror\ p)))$
(**is** $(\exists\ x.\ ?P\ x) = (\exists\ y.\ ?MP\ y)$)
⟨*proof*⟩

lemma *mirror-ex2*:
assumes *unifp*: *isunified p*
shows *qinterp ats* $(QEx\ p) = qinterp\ ats\ (QEx\ (mirror\ p))$
⟨*proof*⟩

lemma *cooper-pi-eq*:
assumes *unifp* : *isunified p*
shows $(\exists\ x.\ qinterp\ (x\#ats)\ p) =$
$((\exists\ j \in \{1\ ..\ (divlcm\ p)\}.\ qinterp\ (-j\#ats)\ (plusinf\ p)) \vee$
$(\exists\ j \in \{1\ ..\ (divlcm\ p)\}.\ \exists\ b \in set\ (aset\ p).$
$qinterp\ (((I\text{-}intterm\ (a\#ats)\ b) - j)\#ats)\ p))$
(**is** $(\exists\ x.\ ?P\ x) = ((\exists\ j \in \{1\ ..\ ?d\}.\ ?PP\ (-j)) \vee (\exists\ j \in ?D.\ \exists\ b \in ?A.\ ?P\ (?I\ a\ b - j))))$
⟨*proof*⟩

consts *subst-it*:: *intterm* ⇒ *intterm* ⇒ *intterm*
primrec
subst-it i $(Cst\ b) = Cst\ b$
subst-it i $(Var\ n) = (if\ n = 0\ then\ i\ else\ Var\ n)$
subst-it i $(Neg\ it) = Neg\ (subst\text{-}it\ i\ it)$
subst-it i $(Add\ it1\ it2) = Add\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$
subst-it i $(Sub\ it1\ it2) = Sub\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$
subst-it i $(Mult\ it1\ it2) = Mult\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$

lemma *subst-it-corr*:
$I\text{-}intterm\ (a\#ats)\ (subst\text{-}it\ i\ t) = I\text{-}intterm\ ((I\text{-}intterm\ (a\#ats)\ i)\#ats)\ t$
⟨*proof*⟩

consts *subst-p*:: *intterm* ⇒ *QF* ⇒ *QF*
primrec
subst-p i $(Le\ it1\ it2) = Le\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$
subst-p i $(Lt\ it1\ it2) = Lt\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$
subst-p i $(Ge\ it1\ it2) = Ge\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$
subst-p i $(Gt\ it1\ it2) = Gt\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$
subst-p i $(Eq\ it1\ it2) = Eq\ (subst\text{-}it\ i\ it1)\ (subst\text{-}it\ i\ it2)$

subst-p i (Divides d t) = Divides (subst-it i d) (subst-it i t)
subst-p i T = T
subst-p i F = F
subst-p i (And p q) = And (subst-p i p) (subst-p i q)
subst-p i (Or p q) = Or (subst-p i p) (subst-p i q)
subst-p i (Imp p q) = Imp (subst-p i p) (subst-p i q)
subst-p i (Equ p q) = Equ (subst-p i p) (subst-p i q)
subst-p i (NOT p) = (NOT (subst-p i p))

lemma *subst-p-corr*:
assumes *qf*: *isqfree p*
shows *qinterp (a # ats) (subst-p i p) = qinterp ((I-intterm (a#ats) i)#ats) p*
⟨*proof*⟩

consts *novar0I*:: *intterm ⇒ bool*
primrec
novar0I (Cst i) = True
novar0I (Var n) = (n > 0)
novar0I (Neg a) = (novar0I a)
novar0I (Add a b) = (novar0I a ∧ novar0I b)
novar0I (Sub a b) = (novar0I a ∧ novar0I b)
novar0I (Mult a b) = (novar0I a ∧ novar0I b)

consts *novar0*:: *QF ⇒ bool*
recdef *novar0 measure size*
novar0 (Lt a b) = (novar0I a ∧ novar0I b)
novar0 (Gt a b) = (novar0I a ∧ novar0I b)
novar0 (Le a b) = (novar0I a ∧ novar0I b)
novar0 (Ge a b) = (novar0I a ∧ novar0I b)
novar0 (Eq a b) = (novar0I a ∧ novar0I b)
novar0 (Divides a b) = (novar0I a ∧ novar0I b)
novar0 T = True
novar0 F = True
novar0 (NOT p) = novar0 p
novar0 (And p q) = (novar0 p ∧ novar0 q)
novar0 (Or p q) = (novar0 p ∧ novar0 q)
novar0 (Imp p q) = (novar0 p ∧ novar0 q)
novar0 (Equ p q) = (novar0 p ∧ novar0 q)
novar0 p = False

lemma *I-intterm-novar0*:
assumes *nov0*: *novar0I x*
shows *I-intterm (a#ats) x = I-intterm (b#ats) x*
⟨*proof*⟩

lemma *subst-p-novar0-corr*:
assumes *qfp*: *isqfree p*
  and *nov0*: *novar0I i*
  shows *qinterp* (*a*#*ats*) (*subst-p i p*) = *qinterp* (*I-intterm* (*b*#*ats*) *i*#*ats*) *p*
⟨*proof*⟩

lemma *lin-novar0*:
  assumes *linx*: *islinintterm x*
  and *nov0*: *novar0I x*
  shows ∃ *n* > *0*. *islintn*(*n*,*x*)
⟨*proof*⟩

lemma *lintnpos-novar0*:
 assumes *npos*: *n* > *0*
  and *linx*: *islintn*(*n*,*x*)
  shows *novar0I x*
⟨*proof*⟩

lemma *lin-add-novar0*:
  assumes *nov0a*: *novar0I a*
  and *nov0b* : *novar0I b*
  and *lina* : *islinintterm a*
  and *linb*: *islinintterm b*
  shows *novar0I* (*lin-add* (*a*,*b*))
⟨*proof*⟩

lemma *lin-mul-novar0*:
  assumes *linx*: *islinintterm x*
  and *nov0*: *novar0I x*
  shows *novar0I* (*lin-mul*(*i*,*x*))
  ⟨*proof*⟩

lemma *lin-neg-novar0*:
  assumes *linx*: *islinintterm x*
  and *nov0*: *novar0I x*
  shows *novar0I* (*lin-neg x*)
⟨*proof*⟩

lemma *intterm-subt-novar0*:
  assumes *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var n*)) *r*)
  shows *novar0I r*
⟨*proof*⟩

consts *decrvarsI*:: *intterm* ⇒ *intterm*
primrec
decrvarsI (*Cst i*) = (*Cst i*)
decrvarsI (*Var n*) = (*Var* (*n* − *1*))
decrvarsI (*Neg a*) = (*Neg* (*decrvarsI a*))
decrvarsI (*Add a b*) = (*Add* (*decrvarsI a*) (*decrvarsI b*))
decrvarsI (*Sub a b*) = (*Sub* (*decrvarsI a*) (*decrvarsI b*))
decrvarsI (*Mult a b*) = (*Mult* (*decrvarsI a*) (*decrvarsI b*))

lemma *intterm-decrvarsI*:
assumes *nov0*: *novar0I t*
shows *I-intterm* (*a*#*ats*) *t* = *I-intterm ats* (*decrvarsI t*)
⟨*proof*⟩

consts *decrvars*:: *QF* ⇒ *QF*
primrec
decrvars (*Lt a b*) = (*Lt* (*decrvarsI a*) (*decrvarsI b*))
decrvars (*Gt a b*) = (*Gt* (*decrvarsI a*) (*decrvarsI b*))
decrvars (*Le a b*) = (*Le* (*decrvarsI a*) (*decrvarsI b*))
decrvars (*Ge a b*) = (*Ge* (*decrvarsI a*) (*decrvarsI b*))
decrvars (*Eq a b*) = (*Eq* (*decrvarsI a*) (*decrvarsI b*))
decrvars (*Divides a b*) = (*Divides* (*decrvarsI a*) (*decrvarsI b*))
decrvars T = *T*
decrvars F = *F*
decrvars (*NOT p*) = (*NOT* (*decrvars p*))
decrvars (*And p q*) = (*And* (*decrvars p*) (*decrvars q*))
decrvars (*Or p q*) = (*Or* (*decrvars p*) (*decrvars q*))
decrvars (*Imp p q*) = (*Imp* (*decrvars p*) (*decrvars q*))
decrvars (*Equ p q*) = (*Equ* (*decrvars p*) (*decrvars q*))

lemma *decrvars-qfree*: *isqfree p* ⟹ *isqfree* (*decrvars p*)
⟨*proof*⟩

lemma *novar0-qfree*: *novar0 p* ⟹ *isqfree p*
⟨*proof*⟩

lemma *qinterp-novar0*:
assumes *nov0*: *novar0 p*
shows *qinterp* (*a*#*ats*) *p* = *qinterp ats* (*decrvars p*)
⟨*proof*⟩

lemma *bset-novar0*:
assumes *unifp*: *isunified p*
shows ∀ *b*∈ *set* (*bset p*). *novar0I b*
⟨*proof*⟩

lemma *subst-it-novar0*:
assumes *nov0x*: *novar0I x*
shows *novar0I* (*subst-it x t*)
⟨*proof*⟩

lemma *subst-p-novar0*:
assumes *nov0x*:*novar0I x*
and *qfp*: *isqfree p*
shows *novar0* (*subst-p x p*)
⟨*proof*⟩

lemma *linearize-novar0*:
assumes *nov0t*: *novar0I t*
shows ⋀ *t'*. *linearize t* = *Some t'* ⟹ *novar0I t'*
⟨*proof*⟩

consts *psimpl* :: *QF* ⇒ *QF*
recdef *psimpl measure size*
psimpl (*Le l r*) =
(*case* (*linearize* (*Sub l r*)) *of*
None ⇒ *Le l r*
| *Some x* ⇒ (*case x of*
Cst i ⇒ (*if i* ≤ *0 then T else F*)
| - ⇒ (*Le x* (*Cst 0*))))
psimpl (*Eq l r*) =
(*case* (*linearize* (*Sub l r*)) *of*
None ⇒ *Eq l r*
| *Some x* ⇒ (*case x of*
Cst i ⇒ (*if i* = *0 then T else F*)
| - ⇒ (*Eq x* (*Cst 0*))))

psimpl (*Divides* (*Cst d*) *t*) =
(*case* (*linearize t*) *of*
None ⇒ (*Divides* (*Cst d*) *t*)
| *Some c* ⇒ (*case c of*
Cst i ⇒ (*if d dvd i then T else F*)
| - ⇒ (*Divides* (*Cst d*) *c*)))

psimpl (*And p q*) =
(*let p'*= *psimpl p*
in (*case p' of*
F ⇒ *F*
|*T* ⇒ *psimpl q*
| - ⇒ *let q'* = *psimpl q*
in (*case q' of*

```
                F ⇒ F
              | T ⇒ p'
              | - ⇒ (And p' q'))))

psimpl (Or p q) =
  (let p'= psimpl p
  in (case p' of
        T ⇒ T
      | F ⇒ psimpl q
      | - ⇒ let q' = psimpl q
            in (case q' of
                  T ⇒ T
                | F ⇒ p'
                | - ⇒ (Or p' q'))))

psimpl (Imp p q) =
  (let p'= psimpl p
  in (case p' of
       F ⇒ T
      |T ⇒ psimpl q
      | NOT p1 ⇒ let q' = psimpl q
            in (case q' of
                  F ⇒ p1
                | T ⇒ T
                | - ⇒ (Or p1 q'))
      | - ⇒ let q' = psimpl q
            in (case q' of
                  F ⇒ NOT p'
                | T ⇒ T
                | - ⇒ (Imp p' q'))))

psimpl (Equ p q) =
  (let p'= psimpl p ; q' = psimpl q
  in (case p' of
        T ⇒ q'
      | F ⇒ (case q' of
                T ⇒ F
              | F ⇒ T
              | NOT q1 ⇒ q1
              | - ⇒ NOT q')
      | NOT p1 ⇒  (case q' of
                T ⇒ p'
              | F ⇒ p1
              | NOT q1 ⇒ (Equ p1 q1)
              | - ⇒ (Equ p' q'))
      | - ⇒ (case q' of
                T ⇒ p'
              | F ⇒ NOT p'
              | - ⇒ (Equ p' q'))))
```

```
psimpl (NOT p) =
  (let p' = psimpl p
  in ( case p' of
       F ⇒ T
     | T ⇒ F
     | NOT p1 ⇒ p1
     | - ⇒ (NOT p')))
psimpl p = p
```

lemma *psimpl-corr*: *qinterp ats p = qinterp ats (psimpl p)*
⟨*proof*⟩

lemma *psimpl-novar0*:
  assumes *nov0p*: *novar0 p*
  shows *novar0 (psimpl p)*
  ⟨*proof*⟩

```
consts explode-disj :: (intterm list × QF) ⇒ QF
recdef explode-disj measure (λ(is,p). length is)
explode-disj ([],p) = F
explode-disj (i#is,p) =
  (let pi = psimpl (subst-p i p)
   in ( case pi of
        T ⇒ T
      | F ⇒ explode-disj (is,p)
      | - ⇒ (let r = explode-disj (is,p)
              in (case r of
                   T ⇒ T
                 | F ⇒ pi
                 | - ⇒ Or pi r))))
```

lemma *explode-disj-disj*:
  assumes *qfp*: *isqfree p*
  shows (*qinterp (x#xs) (explode-disj(i#is,p))) =*
  (*qinterp (x#xs) (subst-p i p) ∨ (qinterp (x#xs) (explode-disj(is,p))))*
  ⟨*proof*⟩

lemma *explode-disj-corr*:
  assumes *qfp*: *isqfree p*
  shows (∃ *x ∈ set xs. qinterp (a#ats) (subst-p x p)) =*
  (*qinterp (a#ats) (explode-disj(xs,p))*) (**is** (∃ *x ∈ set xs. ?P x*) = (*?DP a xs*))
  ⟨*proof*⟩

lemma *explode-disj-novar0*:
assumes *nov0xs*: $\forall x \in set\ xs.\ novar0I\ x$
and *qfp*: *isqfree p*
shows *novar0* (*explode-disj* (*xs*,*p*))
⟨*proof*⟩

lemma *eval-Or-cases*:
qinterp (*a*#*ats*) (*case f of*
$T \Rightarrow T$
| $F \Rightarrow g$
| - $\Rightarrow$ (*case g of*
$T \Rightarrow T$
| $F \Rightarrow f$
| - $\Rightarrow$ *Or f g*)) = (*qinterp* (*a*#*ats*) *f* $\vee$ *qinterp* (*a*#*ats*) *g*)
⟨*proof*⟩

lemma *or-case-novar0*:
assumes *fnTF*: $f \neq T \wedge f \neq F$
and *gnTF*: $g \neq T \wedge g \neq F$
and *f0*: *novar0 f*
and *g0*: *novar0 g*
shows *novar0*
(*case f of* $T \Rightarrow T$ | $F \Rightarrow g$
| - $\Rightarrow$ (*case g of* $T \Rightarrow T$ | $F \Rightarrow f$ | - $\Rightarrow$ *Or f g*))
⟨*proof*⟩

constdefs *list-insert* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
list-insert x xs $\equiv$ (*if x mem xs then xs else x*#*xs*)

lemma *list-insert-set*: *set* (*list-insert x xs*) = *set* (*x*#*xs*)
⟨*proof*⟩

consts *list-union* :: $('a\ list \times 'a\ list) \Rightarrow 'a\ list$

recdef *list-union measure* (λ(*xs*,*ys*). *length xs*)
list-union ([], *ys*) = *ys*
list-union (*xs*, []) = *xs*
list-union (*x*#*xs*,*ys*) = *list-insert x* (*list-union* (*xs*,*ys*))

lemma *list-union-set*: *set* (*list-union*(*xs*,*ys*)) = *set* (*xs*@*ys*)
⟨*proof*⟩

consts *list-set* ::$'a\ list \Rightarrow 'a\ list$
primrec

```
  list-set [] = []
  list-set (x#xs) = list-insert x (list-set xs)

lemma list-set-set: set xs = set (list-set xs)
⟨proof⟩

consts iupto :: int × int ⇒ int list
recdef iupto measure (λ (i,j). nat (j − i +1))
iupto(i,j) = (if j<i then [] else (i#(iupto(i+1,j))))


lemma iupto-set: set (iupto(i,j)) = {i .. j}
⟨proof⟩

consts all-sums :: int × intterm list ⇒ intterm list
recdef all-sums measure (λ(i,is). length is)
all-sums (j,[]) = []
all-sums (j,i#is) = (map (λx. lin-add (i,(Cst x))) (iupto(1,j))@(all-sums (j,is)))

lemma all-sums-novar0:
  assumes nov0xs: ∀ x∈ set xs. novar0I x
  and linxs: ∀ x∈ set xs. islinintterm x
  shows ∀ x∈ set (all-sums (d,xs)). novar0I x
  ⟨proof⟩


lemma all-sums-ex:
  (∃ j∈ {1..d}. ∃ b∈ (set xs). P (lin-add(b,Cst j))) =
  (∃ x∈ set (all-sums (d,xs)). P x)
⟨proof⟩




consts explode-minf :: (QF × intterm list) ⇒ QF
recdef explode-minf measure size
explode-minf (q,B) =
  (let d = divlcm q;
       pm = minusinf q;
        dj1 = explode-disj ((map Cst (iupto (1, d))),pm)
   in (case dj1 of
         T ⇒ T
       | F ⇒ explode-disj (all-sums (d,B),q)
        | - ⇒ (let dj2 = explode-disj (all-sums (d,B),q)
              in ( case dj2 of
                    T ⇒ T
                  | F ⇒ dj1
                  | - ⇒ Or dj1 dj2))))
```

lemma *explode-minf-novar0*:
assumes *unifp* : *isunified p*
and *bst*: *set* (*bset p*) = *set B*
shows *novar0* (*explode-minf* (*p*,*B*))
⟨*proof*⟩

lemma *explode-minf-corr*:
assumes *unifp* : *isunified p*
and *bst*: *set* (*bset p*) = *set B*
shows (∃ *x* . *qinterp* (*x*#*ats*) *p*) = (*qinterp* (*a*#*ats*) (*explode-minf* (*p*,*B*)))
(**is** (∃ *x*. *?P x*) = (*?EXP a p*))
⟨*proof*⟩

lemma *explode-minf-corr2*:
assumes *unifp* : *isunified p*
and *bst*: *set* (*bset p*) = *set B*
shows (*qinterp ats* (*QEx p*)) = (*qinterp ats* (*decrvars*(*explode-minf* (*p*,*B*))))
(**is** *?P* = (*?Qe p*))
⟨*proof*⟩

constdefs *unify*:: *QF* ⇒ (*QF* × *intterm list*)
unify p ≡
(*let q* = *unitycoeff p*;
B = *list-set*(*bset q*);
A = *list-set* (*aset q*)
in
if (*length B* ≤ *length A*)
then (*q*,*B*)
else (*mirror q*, *map lin-neg A*))

lemma *unify-ex*:
assumes *linp*: *islinform p*
shows *qinterp ats* (*QEx p*) = *qinterp ats* (*QEx* (*fst* (*unify p*)))
⟨*proof*⟩

lemma *unify-unified*:
assumes *linp*: *islinform p*
shows *isunified* (*fst* (*unify p*))
⟨*proof*⟩

lemma *unify-qfree*:
assumes *linp*: *islinform p*
shows *isqfree* (*fst*(*unify p*))
⟨*proof*⟩

lemma *unify-bst*:
assumes *linp*: *islinform p*
and *unif*: *unify p* = (*q*,*B*)
shows *set B* = *set* (*bset q*)
⟨*proof*⟩

lemma *explode-minf-unify-novar0*:
assumes *linp*: *islinform p*
shows *novar0* (*explode-minf* (*unify p*))
⟨*proof*⟩

lemma *explode-minf-unify-corr2*:
assumes *linp*: *islinform p*
shows *qinterp ats* (*QEx p*) = *qinterp ats* (*decrvars*(*explode-minf*(*unify p*)))
⟨*proof*⟩

constdefs *cooper*:: *QF* ⇒ *QF option*
cooper p ≡ *lift-un* (λ*q*. *decrvars*(*explode-minf* (*unify q*))) (*linform* (*nnf p*))

lemma *cooper-qfree*: (⋀ *q q'*. ⟦*isqfree q* ; *cooper q* = *Some q'*⟧ ⟹ *isqfree q'*)
⟨*proof*⟩

lemma *cooper-corr*: (⋀ *q q' ats*. ⟦*isqfree q* ; *cooper q* = *Some q'*⟧ ⟹ (*qinterp ats* (*QEx q*)) = (*qinterp ats q'*)) (**is** ⋀ *q q' ats*. ⟦ - ; - ⟧ ⟹ (*?P ats* (*QEx q*) = *?P ats q'*))
⟨*proof*⟩

constdefs *pa*:: *QF* ⇒ *QF option*
pa p ≡ *lift-un psimpl* (*qelim*(*cooper*, *p*))

lemma *psimpl-qfree*: *isqfree p* ⟹ *isqfree* (*psimpl p*)
⟨*proof*⟩

theorem *pa-qfree*: ⋀ *p'*. *pa p* = *Some p'* ⟹ *isqfree p'*
⟨*proof*⟩

theorem *pa-corr*:

$\bigwedge p'.\ pa\ p = Some\ p' \Longrightarrow (qinterp\ ats\ p = qinterp\ ats\ p')$
⟨*proof*⟩

lemma [*code*]: *linearize* (*Mult i j*) =
(*case linearize i of*
None ⇒ *None*
| *Some li* ⇒ (*case li of*
Cst b ⇒ (*case linearize j of*
None ⇒ *None*
| (*Some lj*) ⇒ *Some* (*lin-mul*(*b*,*lj*)))
| - ⇒ (*case linearize j of*
None ⇒ *None*
| (*Some lj*) ⇒ (*case lj of*
Cst b ⇒ *Some* (*lin-mul* (*b*,*li*))
| - ⇒ *None*))))
⟨*proof*⟩

lemma [*code*]: *psimpl* (*And p q*) =
(*let p'*= *psimpl p*
in (*case p' of*
F ⇒ *F*
|*T* ⇒ *psimpl q*
| - ⇒ *let q'* = *psimpl q*
in (*case q' of*
F ⇒ *F*
| *T* ⇒ *p'*
| - ⇒ (*And p' q'*))))

⟨*proof*⟩

lemma [*code*]: *psimpl* (*Or p q*) =
(*let p'*= *psimpl p*
in (*case p' of*
T ⇒ *T*
| *F* ⇒ *psimpl q*
| - ⇒ *let q'* = *psimpl q*
in (*case q' of*
T ⇒ *T*
| *F* ⇒ *p'*
| - ⇒ (*Or p' q'*))))

⟨*proof*⟩

lemma [*code*]: *psimpl* (*Imp p q*) =
(*let p'*= *psimpl p*
in (*case p' of*
F ⇒ *T*
|*T* ⇒ *psimpl q*
| *NOT p1* ⇒ *let q'* = *psimpl q*

```
        in (case q' of
              F ⇒ p1
            | T ⇒ T
            | - ⇒ (Or p1 q'))
  | - ⇒ let q' = psimpl q
        in (case q' of
              F ⇒ NOT p'
            | T ⇒ T
            | - ⇒ (Imp p' q'))))
```
⟨*proof*⟩

declare *zdvd-iff-zmod-eq-0* [*code*]

end

25 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*
  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*

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
  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]

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*⟩

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

end

26 The accessible part of a relation

theory *Accessible-Part*
imports *Main*
begin

26.1 Inductive definition

Inductive definition of the accessible part $acc\ r$ of a relation; see also [**?**].

consts
  $acc :: ('a \times 'a)\ set \Rightarrow 'a\ set$
inductive $acc\ r$
  intros
    $accI$: $(!!y.\ (y, x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$

syntax
  $termi :: ('a \times 'a)\ set \Rightarrow 'a\ set$
translations
  $termi\ r == acc\ (r^{-1})$

26.2 Induction rules

theorem *acc-induct*:
  $a \in acc\ r \Longrightarrow$
    $(!!x.\ x \in acc\ r \Longrightarrow \forall y.\ (y, x) \in r \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
⟨*proof*⟩

theorems *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

theorem *acc-downward*: $b \in acc\ r \Longrightarrow (a, b) \in r \Longrightarrow a \in acc\ r$
  ⟨*proof*⟩

lemma *acc-downwards-aux*: $(b, a) \in r^* \Longrightarrow a \in acc\ r \longrightarrow b \in acc\ r$
  ⟨*proof*⟩

theorem *acc-downwards*: $a \in acc\ r \Longrightarrow (b, a) \in r^* \Longrightarrow b \in acc\ r$
  ⟨*proof*⟩

theorem *acc-wfI*: $\forall x.\ x \in acc\ r \Longrightarrow wf\ r$
  ⟨*proof*⟩

theorem *acc-wfD*: $wf\ r \Longrightarrow x \in acc\ r$
  ⟨*proof*⟩

theorem *wf-acc-iff*: $wf\ r = (\forall x.\ x \in acc\ r)$
  ⟨*proof*⟩

end

27 Multisets

theory *Multiset*
imports *Accessible-Part*
begin

27.1 The type of multisets

typedef *'a multiset = {f::'a => nat. finite {x . 0 < f x}}*
⟨*proof*⟩

lemmas *multiset-typedef* [*simp*] =
Abs-multiset-inverse Rep-multiset-inverse Rep-multiset
and [*simp*] = *Rep-multiset-inject* [*symmetric*]

constdefs
Mempty :: *'a multiset* ({#})
{#} == Abs-multiset (λa. 0)

single :: *'a => 'a multiset* ({#-#})
{#a#} == Abs-multiset (λb. if b = a then 1 else 0)

count :: *'a multiset => 'a => nat*
count == Rep-multiset

MCollect :: *'a multiset => ('a => bool) => 'a multiset*
MCollect M P == Abs-multiset (λx. if P x then Rep-multiset M x else 0)

syntax
-Melem :: *'a => 'a multiset => bool* ((-/ :# -) [*50*, *51*] *50*)
-MCollect :: *pttrn => 'a multiset => bool => 'a multiset* ((*1*{# - : -./ -#}))
translations
a :# M == 0 < count M a
{#x:M. P#} == MCollect M (λx. P)

constdefs
set-of :: *'a multiset => 'a set*
set-of M == {x. x :# M}

instance *multiset* :: (*type*) {*plus*, *minus*, *zero*} ⟨*proof*⟩

defs (**overloaded**)
union-def: *M + N == Abs-multiset (λa. Rep-multiset M a + Rep-multiset N a)*
diff-def: *M − N == Abs-multiset (λa. Rep-multiset M a − Rep-multiset N a)*
Zero-multiset-def [*simp*]: *0 == {#}*
size-def: *size M == setsum (count M) (set-of M)*

constdefs
multiset-inter :: *'a multiset ⇒ 'a multiset ⇒ 'a multiset* (**infixl** *#∩ 70*)
multiset-inter A B ≡ A − (A − B)

Preservation of the representing set *multiset*.

lemma *const0-in-multiset* [*simp*]: $(\lambda a.\ 0) \in multiset$
⟨*proof*⟩

lemma *only1-in-multiset* [*simp*]: $(\lambda b.\ if\ b = a\ then\ 1\ else\ 0) \in multiset$
⟨*proof*⟩

lemma *union-preserves-multiset* [*simp*]:
$M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
⟨*proof*⟩

lemma *diff-preserves-multiset* [*simp*]:
$M \in multiset \Longrightarrow (\lambda a.\ M\ a - N\ a) \in multiset$
⟨*proof*⟩

27.2 Algebraic properties of multisets

27.2.1 Union

lemma *union-empty* [*simp*]: $M + \{\#\} = M \wedge \{\#\} + M = M$
⟨*proof*⟩

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
⟨*proof*⟩

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
⟨*proof*⟩

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
⟨*proof*⟩

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

instance *multiset* :: (*type*) *comm-monoid-add*
⟨*proof*⟩

27.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
⟨*proof*⟩

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
⟨*proof*⟩

27.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}\ a = 0$
⟨*proof*⟩

lemma *count-single* [*simp*]: *count* $\{\#b\#\}\ a = (if\ b = a\ then\ 1\ else\ 0)$

⟨*proof*⟩

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*
⟨*proof*⟩

lemma *count-diff* [*simp*]: *count* (*M* − *N*) *a* = *count M a* − *count N a*
⟨*proof*⟩

27.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* {#} = {}
⟨*proof*⟩

lemma *set-of-single* [*simp*]: *set-of* {#*b*#} = {*b*}
⟨*proof*⟩

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
⟨*proof*⟩

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
⟨*proof*⟩

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
⟨*proof*⟩

27.2.5 Size

lemma *size-empty* [*simp*]: *size* {#} = *0*
⟨*proof*⟩

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
⟨*proof*⟩

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
⟨*proof*⟩

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
⟨*proof*⟩

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
⟨*proof*⟩

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
⟨*proof*⟩

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ==> ∃ *a*. *a* :# *M*
⟨*proof*⟩

27.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: $(M = N) = (\forall a.\ count\ M\ a = count\ N\ a)$
⟨*proof*⟩

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
⟨*proof*⟩

lemma *single-eq-single* [*simp*]: $(\{\#a\#\} = \{\#b\#\}) = (a = b)$
⟨*proof*⟩

lemma *union-eq-empty* [*iff*]: $(M + N = \{\#\}) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *empty-eq-union* [*iff*]: $(\{\#\} = M + N) = (M = \{\#\} \wedge N = \{\#\})$
⟨*proof*⟩

lemma *union-right-cancel* [*simp*]: $(M + K = N + K) = (M = (N::'a\ multiset))$
⟨*proof*⟩

lemma *union-left-cancel* [*simp*]: $(K + M = K + N) = (M = (N::'a\ multiset))$
⟨*proof*⟩

lemma *union-is-single*:
$(M + N = \{\#a\#\}) = (M = \{\#a\#\} \wedge N=\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\})$
⟨*proof*⟩

lemma *single-is-union*:
$(\{\#a\#\} = M + N) = (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
⟨*proof*⟩

lemma *add-eq-conv-diff*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\})$
⟨*proof*⟩

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

27.2.7 Intersection

lemma *multiset-inter-count*:
$count\ (A\ \#\cap\ B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
⟨*proof*⟩

lemma *multiset-inter-commute*: $A\ \#\cap\ B = B\ \#\cap\ A$
⟨*proof*⟩

lemma *multiset-inter-assoc*: $A\ \#\cap\ (B\ \#\cap\ C) = A\ \#\cap\ B\ \#\cap\ C$

⟨*proof*⟩

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
⟨*proof*⟩

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *multiset-union-diff-commute*: $B \#\cap C = \{\#\} \Longrightarrow A + B - C = A - C + B$
⟨*proof*⟩

27.3 Induction over multisets

lemma *setsum-decr*:
finite F ==> (0::nat) < f a ==>
setsum (f (a := f a - 1)) F = (if a∈F then setsum f F - 1 else setsum f F)
⟨*proof*⟩

lemma *rep-multiset-induct-aux*:
assumes *P (λa. (0::nat))*
and *!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))*
shows *∀f. f ∈ multiset --> setsum f {x. 0 < f x} = n --> P f*
⟨*proof*⟩

theorem *rep-multiset-induct*:
f ∈ multiset ==> P (λa. 0) ==>
(!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
⟨*proof*⟩

theorem *multiset-induct* [*induct type*: *multiset*]:
assumes *prem1*: *P {#}*
and *prem2*: *!!M x. P M ==> P (M + {#x#})*
shows *P M*
⟨*proof*⟩

lemma *MCollect-preserves-multiset*:
M ∈ multiset ==> (λx. if P x then M x else 0) ∈ multiset
⟨*proof*⟩

lemma *count-MCollect* [*simp*]:
count {# x:M. P x #} a = (if P a then count M a else 0)
⟨*proof*⟩

lemma *set-of-MCollect* [*simp*]: *set-of {# x:M. P x #} = set-of M ∩ {x. P x}*
⟨*proof*⟩

lemma *multiset-partition*: $M = \{\#\ x{:}M.\ P\ x\ \#\} + \{\#\ x{:}M.\ \neg\ P\ x\ \#\}$
⟨*proof*⟩

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
⟨*proof*⟩

declare *multiset-typedef* [*simp del*]

27.4 Multiset orderings

27.4.1 Well-foundedness

constdefs
mult1 :: $('a \times 'a)\ set \Rightarrow ('a\ multiset \times 'a\ multiset)\ set$
mult1 $r ==$
$\{(N, M).\ \exists a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$
$(\forall b.\ b :\# K \longrightarrow (b, a) \in r)\}$

mult :: $('a \times 'a)\ set \Rightarrow ('a\ multiset \times 'a\ multiset)\ set$
mult $r == (mult1\ r)^+$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
⟨*proof*⟩

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r \Longrightarrow$
$(\exists M.\ (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists K.\ (\forall b.\ b :\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
(**concl is** *?case1* $(mult1\ r) \vee$ *?case2*)
⟨*proof*⟩

lemma *all-accessible*: $wf\ r \Longrightarrow \forall M.\ M \in acc\ (mult1\ r)$
⟨*proof*⟩

theorem *wf-mult1*: $wf\ r \Longrightarrow wf\ (mult1\ r)$
⟨*proof*⟩

theorem *wf-mult*: $wf\ r \Longrightarrow wf\ (mult\ r)$
⟨*proof*⟩

27.4.2 Closure-free presentation

lemma *diff-union-single-conv*: $a :\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
⟨*proof*⟩

One direction.

lemma *mult-implies-one-step*:
trans $r \Longrightarrow (M, N) \in mult\ r \Longrightarrow$
$\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$

$(\forall k \in \textit{set-of } K.\ \exists j \in \textit{set-of } J.\ (k, j) \in r)$
⟨*proof*⟩

lemma *elem-imp-eq-diff-union*: $a :\# M \Longrightarrow M = M - \{\#a\#\} + \{\#a\#\}$
⟨*proof*⟩

lemma *size-eq-Suc-imp-eq-union*: $\textit{size } M = \textit{Suc } n \Longrightarrow \exists a\ N.\ M = N + \{\#a\#\}$
⟨*proof*⟩

lemma *one-step-implies-mult-aux*:
$\textit{trans } r \Longrightarrow$
$\forall I\ J\ K.\ (\textit{size } J = n \wedge J \neq \{\#\} \wedge (\forall k \in \textit{set-of } K.\ \exists j \in \textit{set-of } J.\ (k, j) \in r))$
$\longrightarrow (I + K, I + J) \in \textit{mult } r$
⟨*proof*⟩

lemma *one-step-implies-mult*:
$\textit{trans } r \Longrightarrow J \neq \{\#\} \Longrightarrow \forall k \in \textit{set-of } K.\ \exists j \in \textit{set-of } J.\ (k, j) \in r$
$\Longrightarrow (I + K, I + J) \in \textit{mult } r$
⟨*proof*⟩

27.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* ⟨*proof*⟩

defs (**overloaded**)
less-multiset-def: $M' < M == (M', M) \in \textit{mult } \{(x', x).\ x' < x\}$
le-multiset-def: $M' <= M == M' = M \vee M' < (M::'a\ \textit{multiset})$

lemma *trans-base-order*: $\textit{trans } \{(x', x).\ x' < (x::'a::\textit{order})\}$
⟨*proof*⟩

Irreflexivity.

lemma *mult-irrefl-aux*:
$\textit{finite } A \Longrightarrow (\forall x \in A.\ \exists y \in A.\ x < (y::'a::\textit{order})) \longrightarrow A = \{\}$
⟨*proof*⟩

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::\textit{order multiset})$
⟨*proof*⟩

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::\textit{order multiset}) \Longrightarrow R$
⟨*proof*⟩

Transitivity.

theorem *mult-less-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::\textit{order multiset})$
⟨*proof*⟩

Asymmetry.

theorem *mult-less-not-sym*: $M < N \Longrightarrow \neg\ N < (M::'a::\textit{order multiset})$

⟨*proof*⟩

theorem *mult-less-asym*:
$M < N \Longrightarrow (\neg P \Longrightarrow N < (M::'a::order\ multiset)) \Longrightarrow P$
⟨*proof*⟩

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
⟨*proof*⟩

Anti-symmetry.

theorem *mult-le-antisym*:
$M <= N \Longrightarrow N <= M \Longrightarrow M = (N::'a::order\ multiset)$
⟨*proof*⟩

Transitivity.

theorem *mult-le-trans*:
$K <= M \Longrightarrow M <= N \Longrightarrow K <= (N::'a::order\ multiset)$
⟨*proof*⟩

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
⟨*proof*⟩

Partial order.

instance *multiset* :: (*order*) *order*
⟨*proof*⟩

27.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r \Longrightarrow trans\ r \Longrightarrow (C + B, C + D) \in mult1\ r$
⟨*proof*⟩

lemma *union-less-mono2*: $B < D \Longrightarrow C + B < C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-less-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-less-mono*:
$A < C \Longrightarrow B < D \Longrightarrow A + B < C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-le-mono*:
$A <= C \Longrightarrow B <= D \Longrightarrow A + B <= C + (D::'a::order\ multiset)$
⟨*proof*⟩

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
⟨*proof*⟩

27.5 Link with lists

consts
multiset-of :: *'a list* ⇒ *'a multiset*
primrec
multiset-of [] = {#}
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
⟨*proof*⟩

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of x*) = (*x* = [])
⟨*proof*⟩

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
⟨*proof*⟩

lemma *mem-set-multiset-eq*: *x* ∈ *set xs* = (*x* :# *multiset-of xs*)
⟨*proof*⟩

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

lemma *surj-multiset-of*: *surj multiset-of*
⟨*proof*⟩

lemma *set-count-greater-0*: *set x* = {*a*. *0* < *count* (*multiset-of x*) *a*}
⟨*proof*⟩

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* ∈ *set x then 1 else 0*))
⟨*proof*⟩

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of ys* ⟹ *set xs* = *set ys*
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-eq-distinct*:
⟦*distinct x*; *distinct y*⟧
⟹ (*set x* = *set y*) = (*multiset-of x* = *multiset-of y*)
⟨*proof*⟩

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
⟨*proof*⟩

lemma *multiset-of-compl-union*[*simp*]:
multiset-of [*x*∈*xs*. *P x*] + *multiset-of* [*x*∈*xs*. ¬*P x*] = *multiset-of xs*
⟨*proof*⟩

lemma *count-filter*:
count (*multiset-of xs*) *x* = *length* [*y* ∈ *xs*. *y* = *x*]
⟨*proof*⟩

27.6 Pointwise ordering induced by count

consts
mset-le :: ['*a multiset*, '*a multiset*] ⇒ *bool*

syntax
-*mset-le* :: '*a multiset* ⇒ '*a multiset* ⇒ *bool* (- ≤# - [*50*,*51*] *50*)
translations
x ≤# *y* == *mset-le x y*

defs
mset-le-def: *xs* ≤# *ys* == (∀ *a*. *count xs a* ≤ *count ys a*)

lemma *mset-le-refl*[*simp*]: *xs* ≤# *xs*
⟨*proof*⟩

lemma *mset-le-trans*: ⟦ *xs* ≤# *ys*; *ys* ≤# *zs* ⟧ ⟹ *xs* ≤# *zs*
⟨*proof*⟩

lemma *mset-le-antisym*: ⟦ *xs*≤# *ys*; *ys* ≤# *xs*⟧ ⟹ *xs* = *ys*
⟨*proof*⟩

lemma *mset-le-exists-conv*:
(*xs* ≤# *ys*) = (∃ *zs*. *ys* = *xs* + *zs*)
⟨*proof*⟩

lemma *mset-le-mono-add-right-cancel*[*simp*]: (*xs* + *zs* ≤# *ys* + *zs*) = (*xs* ≤# *ys*)
⟨*proof*⟩

lemma *mset-le-mono-add-left-cancel*[*simp*]: (*zs* + *xs* ≤# *zs* + *ys*) = (*xs* ≤# *ys*)
⟨*proof*⟩

lemma *mset-le-mono-add*: ⟦ *xs* ≤# *ys*; *vs* ≤# *ws* ⟧ ⟹ *xs* + *vs* ≤# *ys* + *ws*
⟨*proof*⟩

lemma *mset-le-add-left*[*simp*]: *xs* ≤# *xs* + *ys*
⟨*proof*⟩

lemma *mset-le-add-right*[*simp*]: *ys* ≤# *xs* + *ys*
⟨*proof*⟩

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups x*) ≤# *multiset-of x*
⟨*proof*⟩

end

28 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*)

constdefs
total :: ('*a* ⇒ '*a* ⇒ *bool*) => *bool*
total r == (∀ *x y*. *r x y* | *r y x*)

transf :: ('*a* ⇒ '*a* ⇒ *bool*) => *bool*
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

29 Insertion Sort

theory *InSort*
imports *Sorting*
begin

consts
ins :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
insort :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ list$

primrec
ins le x [] = [*x*]
ins le x (*y*#*ys*) = (*if le x y then* (*x*#*y*#*ys*) *else y*#(*ins le x ys*))

primrec
insort le [] = []
insort le (*x*#*xs*) = *ins le x* (*insort le xs*)

lemma *multiset-ins*[*simp*]:
$\bigwedge y.$ *multiset-of* (*ins le x xs*) = *multiset-of* (*x*#*xs*)
⟨*proof*⟩

theorem *insort-permutes*[*simp*]:
$\bigwedge x.$ *multiset-of* (*insort le xs*) = *multiset-of xs*
⟨*proof*⟩

lemma *set-ins* [*simp*]: *set*(*ins le x xs*) = *insert x* (*set xs*)
⟨*proof*⟩

lemma *sorted-ins*[*simp*]:
⟦ *total le*; *transf le* ⟧ ⟹ *sorted le* (*ins le x xs*) = *sorted le xs*
⟨*proof*⟩

theorem *sorted-insort*:
[| *total*(*le*); *transf*(*le*) |] ==> *sorted le* (*insort le xs*)
⟨*proof*⟩

end

30 Quicksort

theory *Qsort*

imports *Sorting*
begin

30.1 Version 1: higher-order

consts *qsort* :: $('a \Rightarrow 'a \Rightarrow bool) * 'a\ list \Rightarrow 'a\ list$

recdef *qsort measure* (*size o snd*)
qsort(*le*, []) = []
qsort(*le*, *x*#*xs*) = *qsort*(*le*, [*y*:*xs* . ~ *le x y*]) @ [*x*] @
qsort(*le*, [*y*:*xs* . *le x y*])
(**hints** *recdef-simp*: *length-filter-le*[*THEN le-less-trans*])

lemma *qsort-permutes* [*simp*]:
multiset-of (*qsort*(*le*,*xs*)) = *multiset-of xs*
⟨*proof*⟩

lemma *set-qsort* [*simp*]: *set* (*qsort*(*le*,*xs*)) = *set xs*
⟨*proof*⟩

lemma *sorted-qsort*:
total(*le*) ==> *transf*(*le*) ==> *sorted le* (*qsort*(*le*,*xs*))
⟨*proof*⟩

30.2 Version 2:type classes

consts *quickSort* :: ('*a*::*linorder*) *list* => '*a list*

recdef *quickSort measure size*
quickSort [] = []
quickSort (*x*#*l*) = *quickSort* [*y*:*l*. ~ *x*≤*y*] @ [*x*] @ *quickSort* [*y*:*l*. *x*≤*y*]
(**hints** *recdef-simp*: *length-filter-le*[*THEN le-less-trans*])

lemma *quickSort-permutes*[*simp*]:
multiset-of (*quickSort xs*) = *multiset-of xs*
⟨*proof*⟩

lemma *set-quickSort*[*simp*]: *set* (*quickSort xs*) = *set xs*
⟨*proof*⟩

theorem *sorted-quickSort*: *sorted* (*op* ≤) (*quickSort xs*)
⟨*proof*⟩

end

31 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* ∪ *set ys*
⟨*proof*⟩

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

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

theorem *sorted-msort*: *sorted* (*op* ≤) (*msort xs*)
⟨*proof*⟩

theorem *multiset-of-msort*: *multiset-of* (*msort xs*) = *multiset-of xs*
⟨*proof*⟩

end

32 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*]: $\forall n.\ k{=}f(n) \longrightarrow n <= f(n)$
⟨*proof*⟩

lemma *lemma1*: $n <= f(n)$
⟨*proof*⟩

lemma *lemma2*: $f(n) < f(Suc(n))$
⟨*proof*⟩

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

lemma *f-id*: $f(n) = n$
⟨*proof*⟩

end

33 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.

constdefs *sq* :: *'a*::*times* => *'a*
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*:
!!*x1*::*'a*::*comm-ring*.
(*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 74s on a 2.5GHz Apple G5.

lemma !!$p1$::$'a$::$comm\text{-}ring$.
$(sq\ p1 + sq\ q1 + sq\ r1 + sq\ s1 + sq\ t1 + sq\ u1 + sq\ v1 + sq\ w1) *$
$(sq\ p2 + sq\ q2 + sq\ r2 + sq\ s2 + sq\ t2 + sq\ u2 + sq\ v2 + sq\ w2)$
$= sq\ (p1*p2 - q1*q2 - r1*r2 - s1*s2 - t1*t2 - u1*u2 - v1*v2 - w1*w2)$
$+$
$sq\ (p1*q2 + q1*p2 + r1*s2 - s1*r2 + t1*u2 - u1*t2 - v1*w2 + w1*v2)$
$+$
$sq\ (p1*r2 - q1*s2 + r1*p2 + s1*q2 + t1*v2 + u1*w2 - v1*t2 - w1*u2)$
$+$
$sq\ (p1*s2 + q1*r2 - r1*q2 + s1*p2 + t1*w2 - u1*v2 + v1*u2 - w1*t2)$
$+$
$sq\ (p1*t2 - q1*u2 - r1*v2 - s1*w2 + t1*p2 + u1*q2 + v1*r2 + w1*s2)$
$+$
$sq\ (p1*u2 + q1*t2 - r1*w2 + s1*v2 - t1*q2 + u1*p2 - v1*s2 + w1*r2)$
$+$
$sq\ (p1*v2 + q1*w2 + r1*t2 - s1*u2 - t1*r2 + u1*s2 + v1*p2 - w1*q2)$
$+$
$sq\ (p1*w2 - q1*v2 + r1*u2 + s1*t2 - t1*s2 - u1*r2 + v1*q2 + w1*p2)$
⟨*proof*⟩

end

34 Proving equalities in commutative rings

theory *Commutative-Ring*
imports *Main*
uses (*comm-ring.ML*)
begin

Syntax of multivariate polynomials (pol) and polynomial expressions.

datatype $'a$ *pol* =
  Pc $'a$
| *Pinj nat* $'a$ *pol*
| *PX* $'a$ *pol nat* $'a$ *pol*

datatype $'a$ *polex* =
  Pol $'a$ *pol*
| *Add* $'a$ *polex* $'a$ *polex*
| *Sub* $'a$ *polex* $'a$ *polex*
| *Mul* $'a$ *polex* $'a$ *polex*
| *Pow* $'a$ *polex nat*
| *Neg* $'a$ *polex*

Interpretation functions for the shadow syntax.

consts
  Ipol :: $'a$::{*comm-ring*,*recpower*} *list* ⇒ $'a$ *pol* ⇒ $'a$

Ipolex :: *'a*::{*comm-ring*,*recpower*} *list* ⇒ *'a polex* ⇒ *'a*

primrec
Ipol l (*Pc c*) = *c*
Ipol l (*Pinj i P*) = *Ipol* (*drop i l*) *P*
Ipol l (*PX P x Q*) = *Ipol l P* ∗ (*hd l*) ^*x* + *Ipol* (*drop 1 l*) *Q*

primrec
Ipolex l (*Pol P*) = *Ipol l P*
Ipolex l (*Add P Q*) = *Ipolex l P* + *Ipolex l Q*
Ipolex l (*Sub P Q*) = *Ipolex l P* − *Ipolex l Q*
Ipolex l (*Mul P Q*) = *Ipolex l P* ∗ *Ipolex l Q*
Ipolex l (*Pow p n*) = *Ipolex l p* ^ *n*
Ipolex l (*Neg P*) = − *Ipolex l P*

Create polynomial normalized polynomials given normalized inputs.

constdefs
mkPinj :: *nat* ⇒ *'a pol* ⇒ *'a pol*
mkPinj x P ≡ (*case P of*
Pc c ⇒ *Pc c* |
Pinj y P ⇒ *Pinj* (*x* + *y*) *P* |
PX p1 y p2 ⇒ *Pinj x P*)

constdefs
mkPX :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *nat* ⇒ *'a pol* ⇒ *'a pol*
mkPX P i Q == (*case P of*
Pc c ⇒ (*if* (*c* = *0*) *then* (*mkPinj 1 Q*) *else* (*PX P i Q*)) |
Pinj j R ⇒ *PX P i Q* |
PX P2 i2 Q2 ⇒ (*if* (*Q2* = (*Pc 0*)) *then* (*PX P2* (*i*+*i2*) *Q*) *else* (*PX P i Q*))
)

Defining the basic ring operations on normalized polynomials

consts
add :: *'a*::{*comm-ring*,*recpower*} *pol* × *'a pol* ⇒ *'a pol*
mul :: *'a*::{*comm-ring*,*recpower*} *pol* × *'a pol* ⇒ *'a pol*
neg :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
sqr :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol*
pow :: *'a*::{*comm-ring*,*recpower*} *pol* × *nat* ⇒ *'a pol*

Addition

recdef *add measure* (λ(*x*, *y*). *size x* + *size y*)
add (*Pc a*, *Pc b*) = *Pc* (*a* + *b*)
add (*Pc c*, *Pinj i P*) = *Pinj i* (*add* (*P*, *Pc c*))
add (*Pinj i P*, *Pc c*) = *Pinj i* (*add* (*P*, *Pc c*))
add (*Pc c*, *PX P i Q*) = *PX P i* (*add* (*Q*, *Pc c*))
add (*PX P i Q*, *Pc c*) = *PX P i* (*add* (*Q*, *Pc c*))
add (*Pinj x P*, *Pinj y Q*) =
(*if x*=*y then mkPinj x* (*add* (*P*, *Q*))
else (*if x*>*y then mkPinj y* (*add* (*Pinj* (*x*−*y*) *P*, *Q*))

else mkPinj x (add (Pinj (y−x) Q, P))))
add (Pinj x P, PX Q y R) =
(if x=0 then add(P, PX Q y R)
else (if x=1 then PX Q y (add (R, P))
else PX Q y (add (R, Pinj (x − 1) P))))
add (PX P x R, Pinj y Q) =
(if y=0 then add(PX P x R, Q)
else (if y=1 then PX P x (add (R, Q))
else PX P x (add (R, Pinj (y − 1) Q))))
add (PX P1 x P2, PX Q1 y Q2) =
(if x=y then mkPX (add (P1, Q1)) x (add (P2, Q2))
else (if x>y then mkPX (add (PX P1 (x−y) (Pc 0), Q1)) y (add (P2,Q2))
else mkPX (add (PX Q1 (y−x) (Pc 0), P1)) x (add (P2,Q2))))

Multiplication

recdef *mul measure* (λ(*x*, *y*). *size x* + *size y*)
mul (Pc a, Pc b) = Pc (a∗b)
mul (Pc c, Pinj i P) = (if c=0 then Pc 0 else mkPinj i (mul (P, Pc c)))
mul (Pinj i P, Pc c) = (if c=0 then Pc 0 else mkPinj i (mul (P, Pc c)))
mul (Pc c, PX P i Q) =
(if c=0 then Pc 0 else mkPX (mul (P, Pc c)) i (mul (Q, Pc c)))
mul (PX P i Q, Pc c) =
(if c=0 then Pc 0 else mkPX (mul (P, Pc c)) i (mul (Q, Pc c)))
mul (Pinj x P, Pinj y Q) =
(if x=y then mkPinj x (mul (P, Q))
else (if x>y then mkPinj y (mul (Pinj (x−y) P, Q))
else mkPinj x (mul (Pinj (y−x) Q, P))))
mul (Pinj x P, PX Q y R) =
(if x=0 then mul(P, PX Q y R)
else (if x=1 then mkPX (mul (Pinj x P, Q)) y (mul (R, P))
else mkPX (mul (Pinj x P, Q)) y (mul (R, Pinj (x − 1) P))))
mul (PX P x R, Pinj y Q) =
(if y=0 then mul(PX P x R, Q)
else (if y=1 then mkPX (mul (Pinj y Q, P)) x (mul (R, Q))
else mkPX (mul (Pinj y Q, P)) x (mul (R, Pinj (y − 1) Q))))
mul (PX P1 x P2, PX Q1 y Q2) =
add (mkPX (mul (P1, Q1)) (x+y) (mul (P2, Q2)),
add (mkPX (mul (P1, mkPinj 1 Q2)) x (Pc 0), mkPX (mul (Q1, mkPinj 1 P2)) y (Pc 0)))
(**hints** *simp add*: *mkPinj-def split*: *pol.split*)

Negation

primrec
neg (Pc c) = Pc (−c)
neg (Pinj i P) = Pinj i (neg P)
neg (PX P x Q) = PX (neg P) x (neg Q)

Substraction

constdefs

sub :: *'a*::{*comm-ring*,*recpower*} *pol* ⇒ *'a pol* ⇒ *'a pol*
sub p q ≡ *add* (*p*, *neg q*)

Square for Fast Exponentation

primrec
sqr (*Pc c*) = *Pc* (*c* ∗ *c*)
sqr (*Pinj i P*) = *mkPinj i* (*sqr P*)
sqr (*PX A x B*) = *add* (*mkPX* (*sqr A*) (*x* + *x*) (*sqr B*),
mkPX (*mul* (*mul* (*Pc* (*1* + *1*)), *A*), *mkPinj 1 B*)) *x* (*Pc 0*))

Fast Exponentation

lemma *pow-wf*:*odd n* ⟹ (*n*::*nat*) *div 2* < *n* ⟨*proof*⟩
recdef *pow measure* (λ(*x*, *y*). *y*)
pow (*p*, *0*) = *Pc 1*
pow (*p*, *n*) = (*if even n then* (*pow* (*sqr p*, *n div 2*)) *else mul* (*p*, *pow* (*sqr p*, *n div 2*)))
(**hints** *simp add*: *pow-wf*)

lemma *pow-if*:
pow (*p*,*n*) =
(*if n* = *0 then Pc 1 else if even n then pow* (*sqr p*, *n div 2*)
else mul (*p*, *pow* (*sqr p*, *n div 2*)))
⟨*proof*⟩

Normalization of polynomial expressions

consts *norm* :: *'a*::{*comm-ring*,*recpower*} *polex* ⇒ *'a pol*
primrec
norm (*Pol P*) = *P*
norm (*Add P Q*) = *add* (*norm P*, *norm Q*)
norm (*Sub p q*) = *sub* (*norm p*) (*norm q*)
norm (*Mul P Q*) = *mul* (*norm P*, *norm Q*)
norm (*Pow p n*) = *pow* (*norm p*, *n*)
norm (*Neg P*) = *neg* (*norm P*)

mkPinj preserve semantics

lemma *mkPinj-ci*: *Ipol l* (*mkPinj a B*) = *Ipol l* (*Pinj a B*)
⟨*proof*⟩

mkPX preserves semantics

lemma *mkPX-ci*: *Ipol l* (*mkPX A b C*) = *Ipol l* (*PX A b C*)
⟨*proof*⟩

Correctness theorems for the implemented operations

Negation

lemma *neg-ci*: ⋀*l*. *Ipol l* (*neg P*) = −(*Ipol l P*)
⟨*proof*⟩

Addition

lemma *add-ci*: $\bigwedge l.\ Ipol\ l\ (add\ (P,\ Q)) = Ipol\ l\ P + Ipol\ l\ Q$
⟨*proof*⟩

Multiplication

lemma *mul-ci*: $\bigwedge l.\ Ipol\ l\ (mul\ (P,\ Q)) = Ipol\ l\ P * Ipol\ l\ Q$
⟨*proof*⟩

Substraction

lemma *sub-ci*: $Ipol\ l\ (sub\ p\ q) = Ipol\ l\ p - Ipol\ l\ q$
⟨*proof*⟩

Square

lemma *sqr-ci*:$\bigwedge ls.\ Ipol\ ls\ (sqr\ p) = Ipol\ ls\ p * Ipol\ ls\ p$
⟨*proof*⟩

Power

lemma *even-pow*:$even\ n \Longrightarrow pow\ (p,\ n) = pow\ (sqr\ p,\ n\ div\ 2)$ ⟨*proof*⟩

lemma *pow-ci*: $\bigwedge p.\ Ipol\ ls\ (pow\ (p,\ n)) = (Ipol\ ls\ p)\ \hat{}\ n$
⟨*proof*⟩

Normalization preserves semantics

lemma *norm-ci*:$Ipolex\ l\ Pe = Ipol\ l\ (norm\ Pe)$
⟨*proof*⟩

Reflection lemma: Key to the (incomplete) decision procedure

lemma *norm-eq*:
assumes *eq*: $norm\ P1\ = norm\ P2$
shows $Ipolex\ l\ P1 = Ipolex\ l\ P2$
⟨*proof*⟩

Code generation

⟨*ML*⟩

end

35 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 $((x::int) - y)\hat{}3 = x\hat{}3 + 3*x*y\hat{}2 + (-3)*y*x\hat{}2 - y\hat{}3$
⟨*proof*⟩

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

lemma $((a::int) + b + c)\hat{}2 = a\hat{}2 + b\hat{}2 + c\hat{}2 + 2*a*b + 2*b*c + 2*a*c$
⟨*proof*⟩

lemma $((a::int) - b - c)\hat{}2 = a\hat{}2 + b\hat{}2 + c\hat{}2 - 2*a*b + 2*b*c - 2*a*c$
⟨*proof*⟩

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

lemma $(a::int)\hat{}2 - b\hat{}2 = (a - b) * (a + b)$
⟨*proof*⟩

lemma $(a::int)\hat{}3 - b\hat{}3 = (a - b) * (a\hat{}2 + a*b + b\hat{}2)$
⟨*proof*⟩

lemma $(a::int)\hat{}3 + b\hat{}3 = (a + b) * (a\hat{}2 - a*b + b\hat{}2)$
⟨*proof*⟩

lemma $(a::int)\hat{}4 - b\hat{}4 = (a - b) * (a + b)*(a\hat{}2 + b\hat{}2)$
⟨*proof*⟩

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

end

36 Proof of the relative completeness of method comm-ring

theory *Commutative-Ring-Complete*
imports *Commutative-Ring*
begin

consts *isnorm* :: (*'a*::{*comm-ring*,*recpower*}) *pol* ⇒ *bool*
recdef *isnorm measure size*
isnorm (*Pc c*) = *True*
isnorm (*Pinj i* (*Pc c*)) = *False*
isnorm (*Pinj i* (*Pinj j Q*)) = *False*
isnorm (*Pinj 0 P*) = *False*
isnorm (*Pinj i* (*PX Q1 j Q2*)) = *isnorm* (*PX Q1 j Q2*)
isnorm (*PX P 0 Q*) = *False*
isnorm (*PX* (*Pc c*) *i Q*) = (*c* ≠ *0* & *isnorm Q*)
isnorm (*PX* (*PX P1 j* (*Pc c*)) *i Q*) = (*c*≠*0* ∧ *isnorm*(*PX P1 j* (*Pc c*))∧*isnorm Q*)
isnorm (*PX P i Q*) = (*isnorm P* ∧ *isnorm Q*)

lemma *norm-Pinj-0-False*:*isnorm* (*Pinj 0 P*) = *False*
⟨*proof*⟩

lemma *norm-PX-0-False*:*isnorm* (*PX* (*Pc 0*) *i Q*) = *False*
⟨*proof*⟩

lemma *norm-Pinj*:*isnorm* (*Pinj i Q*) ⟹ *isnorm Q*
⟨*proof*⟩

lemma *norm-PX2*:*isnorm* (*PX P i Q*) ⟹ *isnorm Q*
⟨*proof*⟩

lemma *norm-PX1*:*isnorm* (*PX P i Q*) ⟹ *isnorm P*
⟨*proof*⟩

lemma *mkPinj-cn*:⟦*y*~=*0*; *isnorm Q*⟧ ⟹ *isnorm* (*mkPinj y Q*)
⟨*proof*⟩

lemma *norm-PXtrans*:
assumes *A*:*isnorm* (*PX P x Q*) **and** *isnorm Q2*
shows *isnorm* (*PX P x Q2*)
⟨*proof*⟩

lemma *norm-PXtrans2*: **assumes** *A*:*isnorm* (*PX P x Q*) **and** *isnorm Q2* **shows** *isnorm* (*PX P* (*Suc* (*n*+*x*)) *Q2*)
⟨*proof*⟩

lemma *mkPX-cn*:
assumes *x* ≠ *0* **and** *isnorm P* **and** *isnorm Q*
shows *isnorm* (*mkPX P x Q*)
⟨*proof*⟩

lemma *add-cn*:$[\![isnorm\ P;\ (isnorm\ Q)]\!] \Longrightarrow isnorm\ (add\ (P,\ Q))$
⟨*proof*⟩

lemma *mul-cn* :$[\![isnorm\ P;\ (isnorm\ Q)]\!] \Longrightarrow isnorm\ (mul\ (P,\ Q))$
⟨*proof*⟩

lemma *neg-cn*: $isnorm\ P \Longrightarrow isnorm\ (neg\ P)$
⟨*proof*⟩

lemma *sub-cn*:$[\![isnorm\ p;\ isnorm\ q]\!] \Longrightarrow isnorm\ (sub\ p\ q)$
⟨*proof*⟩

lemma *sqr-cn*:$isnorm\ P \Longrightarrow isnorm\ (sqr\ P)$
⟨*proof*⟩

lemma *pow-cn*:!! $P.\ [\![isnorm\ P]\!] \Longrightarrow isnorm\ (pow\ (P,\ n))$
⟨*proof*⟩

end

37 Set Theory examples: Cantor's Theorem, Schröder-Berstein 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 $(X = Y \cup Z) =$
$(Y \subseteq X \wedge Z \subseteq X \wedge (\forall V.\ Y \subseteq V \wedge Z \subseteq V \longrightarrow X \subseteq V))$
⟨*proof*⟩

lemma $(X = Y \cap Z) =$
$(X \subseteq Y \wedge X \subseteq Z \wedge (\forall V.\ V \subseteq Y \wedge V \subseteq Z \longrightarrow V \subseteq X))$
⟨*proof*⟩

Trivial example of term synthesis: apparently hard for some provers!

lemma $a \neq b \Longrightarrow a \in ?X \wedge b \notin ?X$
⟨*proof*⟩

37.1 Examples for the *blast* paper

lemma $(\bigcup x \in C.\ f\ x \cup g\ x) = \bigcup (f\ `\ C)\ \cup\ \bigcup (g\ `\ C)$
— Union-image, called *Un-Union-image* in Main HOL
⟨*proof*⟩

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
⟨*proof*⟩

Both of the singleton examples can be proved very quickly by *blast del*: *UNIV-I* but not by *blast* alone. For some reason, *UNIV-I* greatly increases the search space.

lemma *singleton-example-1*:
$\bigwedge S::'a\ set\ set.\ \forall x \in S.\ \forall y \in S.\ x \subseteq y \Longrightarrow \exists z.\ S \subseteq \{z\}$
⟨*proof*⟩

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*⟩

37.2 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*⟩

37.3 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*:
$inj\ (f :: 'a \Rightarrow 'b) \Longrightarrow inj\ (g :: 'b \Rightarrow 'a)$
$\Longrightarrow \exists h{::}\ 'a \Rightarrow 'b.\ inj\ h \wedge surj\ h$
⟨*proof*⟩

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.
⟨*proof*⟩

lemma $D \in F \Longrightarrow \exists G.\ \forall A \in G.\ \exists B \in F.\ A \subseteq B$
— Example 2.
⟨*proof*⟩

lemma $P\ a \Longrightarrow \exists A.\ (\forall x \in A.\ P\ x) \wedge (\exists y.\ y \in A)$
— Example 3.
⟨*proof*⟩

lemma $a < b \wedge b < (c{::}int) \Longrightarrow \exists A.\ a \notin A \wedge b \in A \wedge c \notin A$
— Example 4.
⟨*proof*⟩

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 5, page 298.
⟨*proof*⟩

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 6.
⟨*proof*⟩

lemma $\exists A.\ a \notin A$
— Example 7.

⟨*proof*⟩

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

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 :: $[Const,\ Const] \Rightarrow Const$

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

t-const :: $TyConst \Rightarrow Ty$
t-fun :: $[Ty,\ Ty] \Rightarrow Ty$ (- –> - [51,51] 1000)

v-const :: $Const \Rightarrow Val$
v-clos :: $Clos \Rightarrow Val$

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

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

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

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

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

isof :: $[Const,\ Ty] \Rightarrow bool$ (- *isof* - [36,36] 50)
isof-env :: $[ValEnv, TyEnv] \Rightarrow bool$ (- *isofenv* -)

hasty-fun :: $(Val * Ty)\ set \Rightarrow (Val * Ty)\ set$
hasty-rel :: $(Val * Ty)\ set$
hasty :: $[Val,\ Ty] \Rightarrow bool$ (- *hasty* - [36,36] 50)
hasty-env :: $[ValEnv, TyEnv] \Rightarrow bool$ (- *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*⟩

end

38 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*

constdefs
  monotone :: [*'a* => *'a*, *'a set*, (*'a* ∗*'a*)*set*] => *bool*
  monotone f A r == ∀ *x*∈*A*. ∀ *y*∈*A*. (*x*, *y*): *r* −−> ((*f x*), (*f y*)) : *r*

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

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

lub :: [$'a$ set, $'a$ $potype$] => $'a$
lub S po == $least$ (%x. $\forall$ y∈S. (y,x): $order$ po) po

glb :: [$'a$ set, $'a$ $potype$] => $'a$
glb S po == $greatest$ (%x. $\forall$ y∈S. (x,y): $order$ po) po

$isLub$ :: [$'a$ set, $'a$ $potype$, $'a$] => $bool$
$isLub$ S po == %L. (L: $pset$ po & ($\forall$ y∈S. (y,L): $order$ po) &
($\forall$ z∈$pset$ po. ($\forall$ y∈S. (y,z): $order$ po) −−> (L,z): $order$ po))

$isGlb$ :: [$'a$ set, $'a$ $potype$, $'a$] => $bool$
$isGlb$ S po == %G. (G: $pset$ po & ($\forall$ y∈S. (G,y): $order$ po) &
($\forall$ z ∈ $pset$ po. ($\forall$ y∈S. (z,y): $order$ po) −−> (z,G): $order$ po))

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

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

constdefs
Bot :: $'a$ $potype$ => $'a$
Bot po == $least$ (%x. $True$) po

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

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

$CompleteLattice$ :: ($'a$ $potype$) set
$CompleteLattice$ == {cl. cl: $PartialOrder$ &
($\forall$ S. S <= $pset$ cl −−> ($\exists$ L. $isLub$ S cl L)) &
($\forall$ S. S <= $pset$ cl −−> ($\exists$ G. $isGlb$ S cl G))}

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

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

constdefs
$sublattice$ :: ($'a$ $potype$ * $'a$ set)set
$sublattice$ ==

SIGMA cl: *CompleteLattice*.
{*S*. *S* <= *pset cl* &
(| *pset* = *S*, *order* = *induced S* (*order cl*) |): *CompleteLattice* }

syntax
@*SL* :: ['*a set*, '*a potype*] => *bool* (- <<= - [*51*,*50*]*50*)

translations
S <<= *cl* == *S* : *sublattice* `` {*cl*}

constdefs
dual :: '*a potype* => '*a potype*
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*|)

38.1 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*: [| *refl A r*; $x \in A$|] ==> $(x, x) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *antisymE*: [| *antisym r*; $(a, b) \in r$; $(b, a) \in r$ |] ==> $a = b$
⟨*proof*⟩

lemma (**in** *PO*) *transE*: [| *trans r*; $(a, b) \in r$; $(b, c) \in r$|] ==> $(a,c) \in r$
⟨*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*:
$S <= A$ ==> (| *pset* = S, *order* = *induced S r* |) ∈ *PartialOrder*
⟨*proof*⟩

lemma (**in** *PO*) *indE*: [| $(x, y) \in$ *induced S r*; $S <= A$ |] ==> $(x, y) \in r$
⟨*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*: $S <= A$ ==> ∃ *L*. *isLub S cl L*
⟨*proof*⟩

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

lemma *isLub-lub*: (∃ *L*. *isLub S cl L*) = *isLub S cl* (*lub S cl*)
⟨*proof*⟩

lemma *isGlb-glb*: (∃ *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* ∈ *PartialOrder*
⟨*proof*⟩

lemma *Rdual*:
$\forall S.\ (S <= A \dashrightarrow (\exists L.\ isLub\ S\ (|\ pset = A,\ order = r|)\ L))$
$\Longrightarrow \forall S.\ (S <= A \dashrightarrow (\exists G.\ isGlb\ S\ (|\ pset = A,\ order = r|)\ G))$
⟨*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*: $CompleteLattice <= PartialOrder$
⟨*proof*⟩

lemmas *CL-imp-PO* = *CL-subset-PO* [*THEN subsetD*]

declare *CL-imp-PO* [*THEN Tarski.PO-imp-refl*, *simp*]
declare *CL-imp-PO* [*THEN Tarski.PO-imp-sym*, *simp*]
declare *CL-imp-PO* [*THEN Tarski.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*:
$[|\ po \in PartialOrder;\ (\forall S.\ S <= pset\ po \dashrightarrow (\exists L.\ isLub\ S\ po\ L));$
$(\forall S.\ S <= pset\ po \dashrightarrow (\exists G.\ isGlb\ S\ po\ G))|]$
$\Longrightarrow po \in CompleteLattice$
⟨*proof*⟩

lemma (**in** *CL*) *CL-dualCL*: $dual\ cl \in CompleteLattice$
⟨*proof*⟩

lemma (**in** *PO*) *dualA-iff*: $pset\ (dual\ cl) = pset\ cl$
⟨*proof*⟩

lemma (**in** *PO*) *dualr-iff*: $((x,\ y) \in (order(dual\ cl))) = ((y,\ x) \in order\ cl)$
⟨*proof*⟩

lemma (**in** *PO*) *monotone-dual*:
$monotone\ f\ (pset\ cl)\ (order\ cl)$
$\Longrightarrow monotone\ f\ (pset\ (dual\ cl))\ (order(dual\ cl))$
⟨*proof*⟩

lemma (**in** *PO*) *interval-dual*:
[| $x \in A$; $y \in A$|] ==> *interval* $r\ x\ y$ = *interval* (*order*(*dual cl*)) $y\ x$
⟨*proof*⟩

lemma (**in** *PO*) *interval-not-empty*:
[| *trans* r; *interval* $r\ a\ b \neq \{\}$ |] ==> $(a, b) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *interval-imp-mem*: $x \in$ *interval* $r\ a\ b$ ==> $(a, x) \in r$
⟨*proof*⟩

lemma (**in** *PO*) *left-in-interval*:
[| $a \in A$; $b \in A$; *interval* $r\ a\ b \neq \{\}$ |] ==> $a \in$ *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* $r\ a\ b$
⟨*proof*⟩

38.2 sublattice

lemma (**in** *PO*) *sublattice-imp-CL*:
S <<= *cl* ==> (| *pset* = S, *order* = *induced* $S\ r$ |) ∈ *CompleteLattice*
⟨*proof*⟩

lemma (**in** *CL*) *sublatticeI*:
[| S <= A; (| *pset* = S, *order* = *induced* $S\ r$ |) ∈ *CompleteLattice* |]
==> S <<= *cl*
⟨*proof*⟩

38.3 lub

lemma (**in** *CL*) *lub-unique*: [| S <= A; *isLub* S *cl* x; *isLub* S *cl* L|] ==> $x = L$
⟨*proof*⟩

lemma (**in** *CL*) *lub-upper*: [|S <= A; $x \in S$|] ==> $(x,$ *lub* S *cl*$) \in r$
⟨*proof*⟩

lemma (**in** *CL*) *lub-least*:
[| S <= A; $L \in A$; ∀ $x \in S$. $(x,L) \in r$ |] ==> (*lub* S *cl*, $L) \in r$
⟨*proof*⟩

lemma (**in** *CL*) *lub-in-lattice*: S <= A ==> *lub* S *cl* ∈ A
⟨*proof*⟩

lemma (**in** *CL*) *lubI*:
[| S <= A; $L \in A$; ∀ $x \in S$. $(x,L) \in r$;
∀ $z \in A$. (∀ $y \in S$. $(y,z) \in r$) −−> $(L,z) \in r$ |] ==> L = *lub* S *cl*
⟨*proof*⟩

lemma (**in** *CL*) *lubIa*: [| *S* <= *A*; *isLub S cl L* |] ==> *L* = *lub S cl*
⟨*proof*⟩

lemma (**in** *CL*) *isLub-in-lattice*: *isLub S cl L* ==> *L* ∈ *A*
⟨*proof*⟩

lemma (**in** *CL*) *isLub-upper*: [|*isLub S cl L*; *y* ∈ *S*|] ==> (*y*, *L*) ∈ *r*
⟨*proof*⟩

lemma (**in** *CL*) *isLub-least*:
[| *isLub S cl L*; *z* ∈ *A*; ∀ *y* ∈ *S*. (*y*, *z*) ∈ *r*|] ==> (*L*, *z*) ∈ *r*
⟨*proof*⟩

lemma (**in** *CL*) *isLubI*:
[| *L* ∈ *A*; ∀ *y* ∈ *S*. (*y*, *L*) ∈ *r*;
(∀ *z* ∈ *A*. (∀ *y* ∈ *S*. (*y*, *z*):*r*) −−> (*L*, *z*) ∈ *r*)|] ==> *isLub S cl L*
⟨*proof*⟩

38.4 glb

lemma (**in** *CL*) *glb-in-lattice*: *S* <= *A* ==> *glb S cl* ∈ *A*
⟨*proof*⟩

lemma (**in** *CL*) *glb-lower*: [|*S* <= *A*; *x* ∈ *S*|] ==> (*glb S cl*, *x*) ∈ *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* ∈ *A* −> *A*
⟨*proof*⟩

lemma (**in** *CLF*) *monotone-f*: *monotone f A r*
⟨*proof*⟩

lemma (**in** *CLF*) *CLF-dual*: (*cl*,*f*) ∈ *CLF* ==> (*dual cl*, *f*) ∈ *CLF*
⟨*proof*⟩

38.5 fixed points

lemma *fix-subset*: *fix f A* <= *A*
⟨*proof*⟩

lemma *fix-imp-eq*: $x \in \mathit{fix}\ f\ A \Longrightarrow f\ x = x$
⟨*proof*⟩

lemma *fixf-subset*:
$[\!|\ A <= B;\ x \in \mathit{fix}\ (\%y\colon A.\ f\ y)\ A\ |\!] \Longrightarrow x \in \mathit{fix}\ f\ B$
⟨*proof*⟩

38.6 lemmas for Tarski, lub

lemma (**in** *CLF*) *lubH-le-flubH*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} \Longrightarrow (\mathit{lub}\ H\ cl, f\ (\mathit{lub}\ H\ cl)) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *flubH-le-lubH*:
$[\!|\ H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}\ |\!] \Longrightarrow (f\ (\mathit{lub}\ H\ cl), \mathit{lub}\ H\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *lubH-is-fixp*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} \Longrightarrow \mathit{lub}\ H\ cl \in \mathit{fix}\ f\ A$
⟨*proof*⟩

lemma (**in** *CLF*) *fix-in-H*:
$[\!|\ H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\};\ \ x \in P\ |\!] \Longrightarrow x \in H$
⟨*proof*⟩

lemma (**in** *CLF*) *fixf-le-lubH*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\} \Longrightarrow \forall x \in \mathit{fix}\ f\ A.\ (x, \mathit{lub}\ H\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *lubH-least-fixf*:
$H = \{x.\ (x, f\ x) \in r\ \&\ x \in A\}$
$\Longrightarrow \forall L.\ (\forall y \in \mathit{fix}\ f\ A.\ (y,L) \in r) \longrightarrow (\mathit{lub}\ H\ cl, L) \in r$
⟨*proof*⟩

38.7 Tarski fixpoint theorem 1, first part

lemma (**in** *CLF*) *T-thm-1-lub*: $\mathit{lub}\ P\ cl = \mathit{lub}\ \{x.\ (x, f\ x) \in r\ \&\ x \in A\}\ cl$
⟨*proof*⟩

lemma (**in** *CLF*) *glbH-is-fixp*: $H = \{x.\ (f\ x, x) \in r\ \&\ x \in A\} \Longrightarrow \mathit{glb}\ H\ cl \in P$
— Tarski for glb
⟨*proof*⟩

lemma (**in** *CLF*) *T-thm-1-glb*: $\mathit{glb}\ P\ cl = \mathit{glb}\ \{x.\ (f\ x, x) \in r\ \&\ x \in A\}\ cl$
⟨*proof*⟩

38.8 interval

lemma (**in** *CLF*) *rel-imp-elem*: $(x, y) \in r \Longrightarrow x \in A$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-subset*: [| $a \in A$; $b \in A$ |] ==> *interval* $r\ a\ b <= A$
⟨*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*:
[| $S <=$ *interval* $r\ a\ b$; $x \in S$ |] ==> $(a, x) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *interval-lemma2*:
[| $S <=$ *interval* $r\ a\ b$; $x \in S$ |] ==> $(x, b) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *a-less-lub*:
[| $S <= A$; $S \neq \{\}$;
$\forall x \in S.\ (a,x) \in r$; $\forall y \in S.\ (y, L) \in r$ |] ==> $(a,L) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *glb-less-b*:
[| $S <= A$; $S \neq \{\}$;
$\forall x \in S.\ (x,b) \in r$; $\forall y \in S.\ (G, y) \in r$ |] ==> $(G,b) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *S-intv-cl*:
[| $a \in A$; $b \in A$; $S <=$ *interval* $r\ a\ b$ |]==> $S <= A$
⟨*proof*⟩

lemma (**in** *CLF*) *L-in-interval*:
[| $a \in A$; $b \in A$; $S <=$ *interval* $r\ a\ b$;
$S \neq \{\}$; *isLub* $S\ cl\ L$; *interval* $r\ a\ b \neq \{\}$ |] ==> $L \in$ *interval* $r\ a\ b$
⟨*proof*⟩

lemma (**in** *CLF*) *G-in-interval*:
[| $a \in A$; $b \in A$; *interval* $r\ a\ b \neq \{\}$; $S <=$ *interval* $r\ a\ b$; *isGlb* $S\ cl\ G$;
$S \neq \{\}$ |] ==> $G \in$ *interval* $r\ a\ b$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma (**in** *CLF*) *intv-CL-lub*:
[| $a \in A$; $b \in A$; *interval* $r\ a\ b \neq \{\}$ |]
==> $\forall S.\ S <=$ *interval* $r\ a\ b$ −−>

$(\exists L.\ isLub\ S\ (|\ pset = interval\ r\ a\ b,$
$order = induced\ (interval\ r\ a\ b)\ r\ |)\ \ L)$
⟨*proof*⟩

lemmas (**in** *CLF*) *intv-CL-glb* = *intv-CL-lub* [*THEN Rdual*]

lemma (**in** *CLF*) *interval-is-sublattice*:
$[|\ a \in A;\ b \in A;\ interval\ r\ a\ b \neq \{\}\ |]$
$==>\ interval\ r\ a\ b <<=\ cl$
⟨*proof*⟩

lemmas (**in** *CLF*) *interv-is-compl-latt* =
interval-is-sublattice [*THEN sublattice-imp-CL*]

38.9 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*: $Bot\ cl \in A$
⟨*proof*⟩

lemma (**in** *CLF*) *Top-in-lattice*: $Top\ cl \in A$
⟨*proof*⟩

lemma (**in** *CLF*) *Top-prop*: $x \in A ==> (x,\ Top\ cl) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-prop*: $x \in A ==> (Bot\ cl,\ x) \in r$
⟨*proof*⟩

lemma (**in** *CLF*) *Top-intv-not-empty*: $x \in A\ \ ==>\ interval\ r\ x\ (Top\ cl) \neq \{\}$
⟨*proof*⟩

lemma (**in** *CLF*) *Bot-intv-not-empty*: $x \in A ==>\ interval\ r\ (Bot\ cl)\ x \neq \{\}$
⟨*proof*⟩

38.10 fixed points form a partial order

lemma (**in** *CLF*) *fixf-po*: $(|\ pset = P,\ order = induced\ P\ r|) \in PartialOrder$
⟨*proof*⟩

lemma (**in** *Tarski*) *Y-subset-A*: $Y <= A$
⟨*proof*⟩

lemma (**in** *Tarski*) *lubY-in-A*: $lub\ Y\ cl \in A$
⟨*proof*⟩

lemma (**in** *Tarski*) *lubY-le-flubY*: $(lub\ Y\ cl, f\ (lub\ Y\ cl)) \in r$
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-subset*: $intY1 <= A$
⟨*proof*⟩

lemmas (**in** *Tarski*) *intY1-elem* = *intY1-subset* [*THEN subsetD*]

lemma (**in** *Tarski*) *intY1-f-closed*: $x \in intY1 \Longrightarrow f\ x \in intY1$
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-func*: $(\%x{:}\ intY1.\ f\ x) \in intY1 \to intY1$
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-mono*:
$monotone\ (\%x{:}\ intY1.\ f\ x)\ intY1\ (induced\ intY1\ r)$
⟨*proof*⟩

lemma (**in** *Tarski*) *intY1-is-cl*:
$(|\ pset = intY1,\ order = induced\ intY1\ r\ |) \in CompleteLattice$
⟨*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*: $[|\ z \in P; \forall\, y{\in}Y.\ (y,\ z) \in induced\ P\ r\ |]$
$==> ((\%x{:}\ intY1.\ f\ x)\ z,\ z) \in induced\ intY1\ r$
⟨*proof*⟩

lemma (**in** *Tarski*) *tarski-full-lemma*:
$\exists\, L.\ isLub\ Y\ (|\ pset = P,\ order = induced\ P\ r\ |)\ L$
⟨*proof*⟩

lemma *CompleteLatticeI-simp*:
$[|\ (|\ pset = A,\ order = r\ |) \in PartialOrder;$
$\forall\, S.\ S <= A \dashrightarrow (\exists\, L.\ isLub\ S\ (|\ pset = A,\ order = r\ |)\ \ L)\ |]$
$==> (|\ pset = A,\ order = r\ |) \in CompleteLattice$
⟨*proof*⟩

theorem (**in** *CLF*) *Tarski-full*:
$(|\ pset = P,\ order = induced\ P\ r|) \in CompleteLattice$
⟨*proof*⟩

end

39 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

40 Examples for the 'refute' command

theory *Refute-Examples* **imports** *Main*

begin

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
⟨*proof*⟩

41 Examples and Test Cases

41.1 Propositional logic

lemma *True*
refute
⟨*proof*⟩

lemma *False*
refute

⟨*proof*⟩

lemma P
refute
⟨*proof*⟩

lemma $\sim P$
refute
⟨*proof*⟩

lemma $P \,\&\, Q$
refute
⟨*proof*⟩

lemma $P \mid Q$
refute
⟨*proof*⟩

lemma $P \longrightarrow Q$
refute
⟨*proof*⟩

lemma $(P{::}bool) = Q$
refute
⟨*proof*⟩

lemma $(P \mid Q) \longrightarrow (P \,\&\, Q)$
refute
⟨*proof*⟩

41.2 Predicate logic

lemma $P\ x\ y\ z$
refute
⟨*proof*⟩

lemma $P\ x\ y \longrightarrow P\ y\ x$
refute
⟨*proof*⟩

lemma $P\ (f\ (f\ x)) \longrightarrow P\ x \longrightarrow P\ (f\ x)$
refute
⟨*proof*⟩

41.3 Equality

lemma $P = True$
refute
⟨*proof*⟩

lemma $P = False$
  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 *distinct* $[a,b]$
  refute
  ⟨*proof*⟩
  refute
⟨*proof*⟩

41.4 First-Order Logic

lemma $\exists\, x.\ P\ x$
  refute
⟨*proof*⟩

lemma $\forall\, x.\ P\ x$
  refute
⟨*proof*⟩

lemma $EX!\ x.\ P\ x$
  refute
⟨*proof*⟩

lemma *Ex P*
  refute
⟨*proof*⟩

lemma *All P*
  refute
⟨*proof*⟩

lemma *Ex1 P*

refute
⟨*proof*⟩

lemma $(\exists\, x.\ P\ x) \longrightarrow (\forall\, x.\ P\ x)$
refute
⟨*proof*⟩

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (\exists\, y.\ \forall\, x.\ P\ x\ y)$
refute
⟨*proof*⟩

lemma $(\exists\, x.\ P\ x) \longrightarrow (EX!\ x.\ P\ x)$
refute
⟨*proof*⟩

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
⟨*proof*⟩

"A type has at most 5 elements."

lemma $a{=}b \mid a{=}c \mid a{=}d \mid a{=}e \mid a{=}f \mid b{=}c \mid b{=}d \mid b{=}e \mid b{=}f \mid c{=}d \mid c{=}e \mid c{=}f \mid d{=}e \mid d{=}f \mid e{=}f$
refute
⟨*proof*⟩

lemma $\forall\, a\ b\ c\ d\ e\ f.\ a{=}b \mid a{=}c \mid a{=}d \mid a{=}e \mid a{=}f \mid b{=}c \mid b{=}d \mid b{=}e \mid b{=}f \mid c{=}d \mid c{=}e \mid c{=}f \mid d{=}e \mid d{=}f \mid e{=}f$
refute — quantification causes an expansion of the formula; the previous version with free variables is refuted much faster
⟨*proof*⟩

"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
⟨*proof*⟩

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*⟩

41.5 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 *P All*
  refute
⟨*proof*⟩

lemma *P Ex*
  refute
⟨*proof*⟩

lemma *P Ex1*
  refute
⟨*proof*⟩

"The transitive closure 'T' of an arbitrary relation 'P' is non-empty."

constdefs
trans :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
trans $P == (ALL\ x\ y\ z.\ P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z)$
subset :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
subset $P\ Q == (ALL\ x\ y.\ P\ x\ y \longrightarrow Q\ x\ y)$
trans-closure :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
trans-closure $P\ Q ==$ (*subset* $Q\ P$) & (*trans* P) & ($ALL\ R.$ *subset* $Q\ R \longrightarrow$ *trans* $R \longrightarrow$ *subset* $P\ R$)

lemma *trans-closure* $T\ P \longrightarrow (\exists\, x\ y.\ T\ x\ y)$
refute
⟨*proof*⟩

"The union of transitive closures is equal to the transitive closure of unions."

lemma $(\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow T\ x\ y) \longrightarrow$ *trans* $T \longrightarrow (\forall\, Q.\ (\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow Q\ x\ y) \longrightarrow$ *trans* $Q \longrightarrow$ *subset* $T\ Q)$
$\longrightarrow$ *trans-closure* $TP\ P$
$\longrightarrow$ *trans-closure* $TR\ R$
$\longrightarrow (T\ x\ y = (TP\ x\ y \mid TR\ x\ y))$
refute
⟨*proof*⟩

"Every surjective function is invertible."

lemma $(\forall\, y.\ \exists\, x.\ y = f\ x) \longrightarrow (\exists\, g.\ \forall\, x.\ g\ (f\ x) = x)$
refute
⟨*proof*⟩

"Every invertible function is surjective."

lemma $(\exists\, g.\ \forall\, x.\ g\ (f\ x) = x) \longrightarrow (\forall\, y.\ \exists\, x.\ y = f\ x)$
refute
⟨*proof*⟩

Every point is a fixed point of some function.

lemma $\exists\, f.\ f\ x = x$
refute [*maxsize=4*]
⟨*proof*⟩

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*⟩

41.6 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*⟩

41.7 Schematic variables

lemma *?P*
refute
⟨*proof*⟩

lemma $x = ?y$
refute
⟨*proof*⟩

41.8 Abstractions

lemma $(\lambda x.\ x) = (\lambda x.\ y)$
refute
⟨*proof*⟩

lemma $(\lambda f.\ f\ x) = (\lambda f.\ True)$
refute
⟨*proof*⟩

lemma $(\lambda x.\ x) = (\lambda y.\ y)$
refute
⟨*proof*⟩

41.9 Sets

lemma $P\ (A::'a\ set)$
refute
⟨*proof*⟩

lemma P $(A{::}'a\ set\ set)$
refute
⟨*proof*⟩

lemma $\{x.\ P\ x\} = \{y.\ P\ y\}$
refute
⟨*proof*⟩

lemma $x : \{x.\ P\ x\}$
refute
⟨*proof*⟩

lemma $P\ op$:
refute
⟨*proof*⟩

lemma P $(op{:}\ x)$
refute
⟨*proof*⟩

lemma $P\ Collect$
refute
⟨*proof*⟩

lemma $A\ Un\ B = A\ Int\ B$
refute
⟨*proof*⟩

lemma $(A\ Int\ B)\ Un\ C = (A\ Un\ C)\ Int\ B$
refute
⟨*proof*⟩

lemma $Ball\ A\ P \longrightarrow Bex\ A\ P$
refute
⟨*proof*⟩

41.10 arbitrary

lemma *arbitrary*
refute
⟨*proof*⟩

lemma P *arbitrary*
refute
⟨*proof*⟩

lemma *arbitrary* x
refute

⟨*proof*⟩

lemma *arbitrary arbitrary*
refute
⟨*proof*⟩

41.11 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* ⟶ *P* (*The P*)
refute
⟨*proof*⟩

41.12 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* ⟶ *P* (*Eps P*)
refute [*maxsize*=*3*]
⟨*proof*⟩

41.13 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 $'a\ T\text{-}bij = \{(f::'a \Rightarrow 'a).\ \forall y.\ \exists!x.\ f\ x = y\}$
⟨*proof*⟩

lemma $P\ (f::(myTdecl\ myTdef)\ T\text{-}bij)$
refute
⟨*proof*⟩

41.14 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.

⟨*ML*⟩

41.14.1 unit

lemma $P\ (x::unit)$
refute
⟨*proof*⟩

lemma $\forall x::unit.\ P\ x$
refute
⟨*proof*⟩

lemma $P\ ()$
refute
⟨*proof*⟩

lemma $P\ (unit\text{-}rec\ u\ x)$
refute
⟨*proof*⟩

lemma $P\ (case\ x\ of\ () \Rightarrow u)$
refute
⟨*proof*⟩

41.14.2 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 $P\ (option\text{-}rec\ n\ s\ x)$
refute
⟨*proof*⟩

lemma $P\ (case\ x\ of\ None \Rightarrow n \mid Some\ u \Rightarrow s\ u)$
refute
⟨*proof*⟩

41.14.3 *

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 *P* (*prod-rec p x*)
refute
⟨*proof*⟩

lemma *P* (*case x of Pair a b ⇒ p a b*)
refute
⟨*proof*⟩

41.14.4 +

lemma *P* (*x*::*'a*+*'b*)
refute
⟨*proof*⟩

lemma ∀ *x*::*'a*+*'b*. *P x*
refute
⟨*proof*⟩

lemma *P* (*Inl x*)
refute
⟨*proof*⟩

lemma *P* (*Inr x*)
refute
⟨*proof*⟩

lemma *P Inl*
refute
⟨*proof*⟩

lemma *P* (*sum-rec l r x*)
refute
⟨*proof*⟩

lemma *P* (*case x of Inl a ⇒ l a | Inr b ⇒ r b*)
refute
⟨*proof*⟩

41.14.5 Non-recursive datatypes

datatype *T1* = *A* | *B*

lemma *P* (*x*::*T1*)
refute
⟨*proof*⟩

lemma ∀ *x*::*T1*. *P x*
refute

⟨*proof*⟩

lemma *P A*
refute
⟨*proof*⟩

lemma *P* (*T1-rec a b x*)
refute
⟨*proof*⟩

lemma *P* (*case x of A* ⇒ *a* | *B* ⇒ *b*)
refute
⟨*proof*⟩

datatype *'a T2* = *C T1* | *D 'a*

lemma *P* (*x*::*'a T2*)
refute
⟨*proof*⟩

lemma ∀ *x*::*'a T2*. *P x*
refute
⟨*proof*⟩

lemma *P D*
refute
⟨*proof*⟩

lemma *P* (*T2-rec c d x*)
refute
⟨*proof*⟩

lemma *P* (*case x of C u* ⇒ *c u* | *D v* ⇒ *d v*)
refute
⟨*proof*⟩

datatype (*'a*,*'b*) *T3* = *E 'a* ⇒ *'b*

lemma *P* (*x*::(*'a*,*'b*) *T3*)
refute
⟨*proof*⟩

lemma ∀ *x*::(*'a*,*'b*) *T3*. *P x*
refute
⟨*proof*⟩

lemma *P E*
refute
⟨*proof*⟩

lemma *P* (*T3-rec e x*)
refute
⟨*proof*⟩

lemma *P* (*case x of E f ⇒ e f*)
refute
⟨*proof*⟩

41.14.6 Recursive datatypes

nat

lemma *P* (*x*::*nat*)
refute
⟨*proof*⟩

lemma ∀ *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 *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 *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 *'a BinTree* = *Leaf 'a* | *Node 'a BinTree 'a BinTree*

lemma *P* (*x*::*'a BinTree*)
refute
⟨*proof*⟩

lemma ∀ *x*::*'a BinTree*. *P x*
refute
⟨*proof*⟩

lemma *P* (*Node* (*Leaf x*) (*Leaf y*))
refute
⟨*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*⟩

41.14.7 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 *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 *P* (*aexp-bexp-rec-2 number ite equal x*)
refute
⟨*proof*⟩

lemma *P* (*case x of Equal a1 a2* ⇒ *equal a1 a2*)
refute
⟨*proof*⟩

41.14.8 Other datatype examples

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 *P* (*Trie-rec-1 a b c x*)
refute

⟨*proof*⟩

lemma *P* (*Trie-rec-2 a b c 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* (λ*n*. *Leaf*))
refute
⟨*proof*⟩

lemma *P* (*InfTree-rec leaf node x*)
refute
⟨*proof*⟩

datatype *'a lambda* = *Var 'a* | *App 'a lambda 'a lambda* | *Lam 'a* ⇒ *'a lambda*

lemma *P* (*x*::*'a lambda*)
refute
⟨*proof*⟩

lemma ∀ *x*::*'a lambda*. *P x*
refute
⟨*proof*⟩

lemma *P* (*Lam* (λ*a*. *Var a*))
refute
⟨*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* ⇒ *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 $P\ (E\ (C\ (\lambda a.\ True)))$
refute
⟨*proof*⟩

lemma $P\ (U\text{-}rec\text{-}1\ e\ f\ g\ h\ i\ x)$
refute
⟨*proof*⟩

lemma $P\ (U\text{-}rec\text{-}2\ e\ f\ g\ h\ i\ x)$
refute
⟨*proof*⟩

lemma $P\ (U\text{-}rec\text{-}3\ e\ f\ g\ h\ i\ x)$
refute
⟨*proof*⟩

41.15 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*⟩

41.16 Inductively defined sets

consts
$arbitrarySet :: 'a\ set$
inductive *arbitrarySet*
intros
$arbitrary : arbitrarySet$

lemma $x : arbitrarySet$
refute
⟨*proof*⟩

consts
  evenCard :: *'a set set*
inductive *evenCard*
intros
  $\{\}$: *evenCard*
  $[\![\ S : evenCard;\ x \notin S;\ y \notin S;\ x \neq y\ ]\!] \Longrightarrow S \cup \{x,\ y\} : evenCard$

lemma S : *evenCard*
  refute
⟨*proof*⟩

consts
  even :: *nat set*
  odd :: *nat set*
inductive *even odd*
intros
  0 : *even*
  n : *even* $\Longrightarrow$ *Suc* n : *odd*
  n : *odd* $\Longrightarrow$ *Suc* n : *even*

lemma n : *odd*
    — unfortunately, this little example already takes too long
⟨*proof*⟩

41.17 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 $a \mathbin{@} [] = b \mathbin{@} []$
refute
⟨*proof*⟩

lemma $a \mathbin{@} b = b \mathbin{@} a$
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*⟩

41.18 Axiomatic type classes and overloading

A type class without axioms:

axclass *classA*

lemma $P\ (x{::}'a{::}classA)$
refute
⟨*proof*⟩

The axiom of this type class does not contain any type variables, but is internally converted into one that does:

axclass *classB*
classB-ax: $P \mid\sim P$

lemma $P\ (x{::}'a{::}classB)$
refute
⟨*proof*⟩

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
⟨*proof*⟩

lemma $\exists\, x\ y.\ (x::'a::classC) \neq y$
  refute — no countermodel exists
⟨*proof*⟩

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
⟨*proof*⟩

A type class with multiple superclasses:

axclass *classE* < *classC*, *classD*

lemma $P\ (x::'a::classE)$
  refute
⟨*proof*⟩

lemma $P\ (x::'a::\{classB,\ classE\})$
  refute
⟨*proof*⟩

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* :: $'a \Rightarrow 'a$

defs (**overloaded**)
  inverse-bool: *inverse* (*b*::*bool*) == ~ *b*
  inverse-set : *inverse* (*S*::'*a set*) == −*S*
  inverse-pair: *inverse p* == (*inverse* (*fst p*), *inverse* (*snd p*))

lemma *inverse b*
  refute
⟨*proof*⟩

lemma *P* (*inverse* (*S*::'*a set*))
  refute
⟨*proof*⟩

lemma *P* (*inverse* (*p*::'*a*×'*b*))
  refute
⟨*proof*⟩

end

42 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.

42.1 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 ⇒ 'a ⇒ 'a list ⇒ 'a list*
primrec
replace a b [] = []
replace a b (x#xs) = (if (x=a) then (b#(replace a b xs))
else (x#(replace a b xs)))

lemma *occurs a xs = occurs b (replace a b xs)*
quickcheck
— Wrong. Precondition needed.
⟨*proof*⟩

lemma *occurs b xs = 0 ∨ a=b ⟶ occurs a xs = occurs b (replace a b xs)*
quickcheck
⟨*proof*⟩

42.2 Trees

datatype *'a tree = Twig | Leaf 'a | Branch 'a tree 'a tree*

consts
leaves :: 'a tree ⇒ 'a list
primrec
leaves Twig = []
leaves (Leaf a) = [a]
leaves (Branch l r) = (leaves l) @ (leaves r)

consts
plant :: 'a list ⇒ 'a tree
primrec
plant [] = Twig
plant (x#xs) = Branch (Leaf x) (plant xs)

consts
mirror :: 'a tree ⇒ '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

43 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* ≠ [] ==> *bv-to-nat bv* = *bitval* (*hd bv*) * *2* ^ (*length bv* − *1*) + *bv-to-nat* (*tl bv*)
⟨*proof*⟩

constdefs
half-adder :: [*bit*,*bit*] => *bit list*
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*⟩

constdefs
full-adder :: [*bit*,*bit*,*bit*] => *bit list*
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*⟩

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 [rule-format]:
    ∀ bs. length as = length bs -->
          length (carry-chain-adder as bs c) = Suc (length bs)
     (is ?P as)
⟨proof⟩

lemma cca-correct [rule-format]:
    ∀ bs. length as = length bs -->
          bv-to-nat (carry-chain-adder as bs c) =
          bv-to-nat as + bv-to-nat bs + bitval c
     (is ?P as)
⟨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.