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 :: $o \Rightarrow prop$ (- *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 $-$
assume $x = y$
thus $y = x$ **by** (*rule subst*) (*rule refl*)
qed

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

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

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

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

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

1.1.3 Derived connectives

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 (*unfold false-def*)
assume $\forall A.\ A$
thus A **..**
qed

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

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

theorem *notE* [*elim*]: $\neg\ A \Longrightarrow A \Longrightarrow B$
proof (*unfold not-def*)
assume $A \longrightarrow \bot$
also assume A
finally have $\bot$ **..**
thus B **..**
qed

lemma *notE'*: $A \Longrightarrow \neg\ A \Longrightarrow B$
by (*rule notE*)

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

theorem *conjI* [*intro*]: $A \Longrightarrow B \Longrightarrow A \wedge B$
proof (*unfold conj-def*)
assume A **and** B
show $\forall C.\ (A \longrightarrow B \longrightarrow C) \longrightarrow C$
proof
fix C **show** $(A \longrightarrow B \longrightarrow C) \longrightarrow C$
proof

assume $A \longrightarrow B \longrightarrow C$
also have A **.**
also have B **.**
finally show C **.**
qed
qed
qed

theorem *conjE* [*elim*]: $A \wedge B \Longrightarrow (A \Longrightarrow B \Longrightarrow C) \Longrightarrow C$
proof (*unfold conj-def*)
assume *c*: $\forall C. (A \longrightarrow B \longrightarrow C) \longrightarrow C$
assume $A \Longrightarrow B \Longrightarrow C$
moreover {
from *c* **have** $(A \longrightarrow B \longrightarrow A) \longrightarrow A$ **..**
also have $A \longrightarrow B \longrightarrow A$
proof
assume A
thus $B \longrightarrow A$ **..**
qed
finally have A **.**
} moreover {
from *c* **have** $(A \longrightarrow B \longrightarrow B) \longrightarrow B$ **..**
also have $A \longrightarrow B \longrightarrow B$
proof
show $B \longrightarrow B$ **..**
qed
finally have B **.**
} ultimately show C **.**
qed

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

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

  proof
    fix C **show** $(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$
    proof
      show $(B \longrightarrow C) \longrightarrow C$
      proof
        assume $B \longrightarrow C$
        also have B **.**
        finally show C **.**
      qed
    qed
  qed
qed

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

theorem *exI* [*intro*]: $P\ a \Longrightarrow \exists\, x.\ P\ x$
proof (*unfold Ex-def*)
  assume $P\ a$
  show $\forall C.\ (\forall x.\ P\ x \longrightarrow C) \longrightarrow C$
  proof
    fix C **show** $(\forall x.\ P\ x \longrightarrow C) \longrightarrow C$
    proof
      assume $\forall x.\ P\ x \longrightarrow C$
      hence $P\ a \longrightarrow C$ **..**
      also have $P\ a$ **.**
      finally show C **.**
    qed
  qed
qed

theorem *exE* [*elim*]: $\exists\, x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow C) \Longrightarrow C$
proof (*unfold Ex-def*)
  assume *c*: $\forall C.\ (\forall x.\ P\ x \longrightarrow C) \longrightarrow C$
  assume *r*: $\bigwedge x.\ P\ x \Longrightarrow C$
  from *c* **have** $(\forall x.\ P\ x \longrightarrow C) \longrightarrow C$ **..**

also have $\forall x.\ P\ x \longrightarrow C$
  proof
    fix x **show** $P\ x \longrightarrow C$
    proof
      assume $P\ x$
      thus C **by** $(rule\ r)$
    qed
  qed
  finally show C **.**
qed

1.2 Classical logic

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

theorem (**in** *classical*)
  Peirce's-Law: $((A \longrightarrow B) \longrightarrow A) \longrightarrow A$
proof
  assume a: $(A \longrightarrow B) \longrightarrow A$
  show A
  proof (*rule classical*)
    assume $\neg\ A$
    have $A \longrightarrow B$
    proof
      assume A
      thus B **by** (*rule contradiction*)
    qed
    with a **show** A **..**
  qed
qed

theorem (**in** *classical*)
  double-negation: $\neg\ \neg\ A \Longrightarrow A$
proof −
  assume $\neg\ \neg\ A$
  show A
  proof (*rule classical*)
    assume $\neg\ A$
    thus *?thesis* **by** (*rule contradiction*)
  qed
qed

theorem (**in** *classical*)
  tertium-non-datur: $A \vee \neg\ A$
proof (*rule double-negation*)
  show $\neg\ \neg\ (A \vee \neg\ A)$
  proof
    assume $\neg\ (A \vee \neg\ A)$

```
    have ¬ A
    proof
      assume A hence A ∨ ¬ A ..
      thus ⊥ by (rule contradiction)
    qed
    hence A ∨ ¬ A ..
    thus ⊥ by (rule contradiction)
  qed
qed

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

lemma (in classical) (¬ A ⟹ A) ⟹ A
proof −
  assume r: ¬ A ⟹ A
  show A
  proof (rule classical-cases)
    assume A thus A .
  next
    assume ¬ A thus A by (rule r)
  qed
qed

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*
by *simp*

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

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

Not handled automatically: too complicated.

consts *variant* :: *nat* ∗ *nat list* => *nat*
recdef (**permissive**) *variant measure* (λ(*n*,*ns*). *size* (*filter* (λ*y*. *n* ≤ *y*) *ns*))
  variant (*x*, *L*) = (*if x mem L then variant* (*Suc x*, *L*) *else x*)

consts *gcd* :: *nat* ∗ *nat* => *nat*
recdef *gcd* *measure* (λ(*x*, *y*). *x* + *y*)
  gcd (*0*, *y*) = *y*
  gcd (*Suc x*, *0*) = *Suc x*
  gcd (*Suc x*, *Suc y*) =
    (*if y* ≤ *x then gcd* (*x* − *y*, *Suc y*) *else gcd* (*Suc x*, *y* − *x*))

The silly *g* function: example of nested recursion. Not handled automatically. In fact, *g* is the zero constant function.

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

lemma *g-terminates*: *g x* < *Suc x*
  apply (*induct x rule*: *g.induct*)
   apply (*auto simp add*: *g.simps*)
  done

lemma *g-zero*: $g\ x = 0$
  apply (*induct x rule*: *g.induct*)
   apply (*simp-all add*: *g.simps g-terminates*)
  done

consts *Div* :: *nat* ∗ *nat* => *nat* ∗ *nat*
recdef *Div* *measure fst*
  Div (*0*, *x*) = (*0*, *0*)
  Div (*Suc x*, *y*) =
    (*let* (*q*, *r*) = *Div* (*x*, *y*)
    in if y ≤ *Suc r then* (*Suc q*, *0*) *else* (*q*, *Suc r*))

Not handled automatically. Should be the predecessor function, but there is an unnecessary "looping" recursive call in *k 1*.

consts *k* :: *nat* => *nat*

recdef (**permissive**) *k* *less-than*
  k 0 = *0*
  k (*Suc n*) =
   (*let x* = *k 1*
    in if False then k (*Suc 1*) *else n*)

consts *part* :: (′*a* => *bool*) ∗ ′*a list* ∗ ′*a list* ∗ ′*a list* => ′*a list* ∗ ′*a list*
recdef *part* *measure* (λ(*P*, *l*, *l1*, *l2*). *size l*)
  part (*P*, [], *l1*, *l2*) = (*l1*, *l2*)
  part (*P*, *h* # *rst*, *l1*, *l2*) =
    (*if P h then part* (*P*, *rst*, *h* # *l1*, *l2*)
    else part (*P*, *rst*, *l1*, *h* # *l2*))

consts *fqsort* :: (′*a* => ′*a* => *bool*) ∗ ′*a list* => ′*a list*
recdef (**permissive**) *fqsort* *measure* (*size o snd*)
  fqsort (*ord*, []) = []
  fqsort (*ord*, *x* # *rst*) =
  (*let* (*less*, *more*) = *part* ((λ*y*. *ord y x*), *rst*, ([], []))
  in fqsort (*ord*, *less*) @ [*x*] @ *fqsort* (*ord*, *more*))

Silly example which demonstrates the occasional need for additional congruence rules (here: *map-cong*). If the congruence rule is removed, an unprovable termination condition is generated! Termination not proved automatically. TFL requires λ*x*. *mapf x* instead of *mapf*.

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

recdef-tc *mapf-tc*: *mapf*

apply (*rule allI*)
apply (*case-tac n = 0*)
apply *simp-all*
done

Removing the termination condition from the generated thms:

lemma *mapf* (*Suc n*) = *concat* (*map mapf* (*replicate n n*))
apply (*simp add*: *mapf.simps mapf-tc*)
done

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

end

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 (*induct-tac x rule*: *wf-induct* [*OF WF*])
fix *x*
assume *IHYP*: $\forall y.$ (*y*,*x*) $\in$ *r* --> *S y* (*wfrec r F y*)

then have $\forall y.\ (y,x) \in r \dashrightarrow S\ y\ (cut\ (wfrec\ r\ F)\ r\ x\ y)$ **by** (*simp add*: *tfl-cut-apply*)
with *INV* **have** $S\ x\ (F\ (cut\ (wfrec\ r\ F)\ r\ x)\ x)$ **by** (*unfold indinv-def*, *blast*)
thus $S\ x\ (wfrec\ r\ F\ x)$ **using** *WF* **by** (*simp add*: *wfrec*)
qed

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

theorem *ind-fixpoint-on-lemma*:
assumes *WF*: *wf r* **and**
INV: $\forall f.\ \forall x \in D.\ (\forall y \in D.\ (y,x) \in r \dashrightarrow S\ y\ (wfrec\ r\ F\ y)\ \&\ f\ y = wfrec\ r\ F\ y)$
$\dashrightarrow S\ x\ (wfrec\ r\ F\ x)\ \&\ F\ f\ x = wfrec\ r\ F\ x$ **and**
D: $x \in D$
shows $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x\ \&\ S\ x\ (wfrec\ r\ F\ x)$
proof (*rule indinv-on-wfrec* [*OF WF* - *D*, *of* % *a b*. $F\ (wfrec\ r\ F)\ a = b\ \&\ wfrec\ r\ F\ a = b\ \&\ S\ a\ b$ *F*, *simplified*])
show *indinv-on r D* (%*a b*. $F\ (wfrec\ r\ F)\ a = b\ \&\ wfrec\ r\ F\ a = b\ \&\ S\ a\ b$) *F*
proof (*unfold indinv-on-def*, *clarify*)
fix *f x*
assume *A1*: $\forall y \in D.\ (y,\ x) \in r \dashrightarrow F\ (wfrec\ r\ F)\ y = f\ y\ \&\ wfrec\ r\ F\ y = f\ y\ \&\ S\ y\ (f\ y)$
assume *D'*: $x \in D$
from *A1 INV* [*THEN spec*, *of f*, *THEN bspec*, *OF D'*]
have $S\ x\ (wfrec\ r\ F\ x)$ **and**
$F\ f\ x = wfrec\ r\ F\ x$ **by** *auto*
moreover
from *A1* **have** $\forall y \in D.\ (y,\ x) \in r \dashrightarrow S\ y\ (wfrec\ r\ F\ y)$ **by** *auto*
with *D' INV* [*THEN spec*, *of wfrec r F*, *simplified*]
have $F\ (wfrec\ r\ F)\ x = wfrec\ r\ F\ x$ **by** *blast*
ultimately show $F\ (wfrec\ r\ F)\ x = F\ f\ x\ \&\ wfrec\ r\ F\ x = F\ f\ x\ \&\ S\ x\ (F\ f\ x)$ **by** *auto*
qed
qed

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

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

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

end

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* => *nat*

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

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

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

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

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

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

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

thm *g′.simps*

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

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

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

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

thm *g′.simps* [*simplified*]

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

consts *g-even* :: *nat => nat*
recdef (**permissive**) *g-even less-than*
  g-even (*Suc* (*Suc 0*)) = *3*
  g-even n = *g-even* (*g-even* (*n* − *2*) − *1*)

We can prove a conditional version of the unsolved termination condition for *g-even* by proving a stronger inductive invariant.

lemma *g-even-indinv*: ∃ *k*. *n* = *Suc* (*Suc* (*2*∗*k*)) ==> *g-even n* = *3*
apply (*rule-tac D*={*n*. ∃ *k*. *n* = *Suc* (*Suc* (*2*∗*k*))} **and** *x=n* **in** *tfl-indinv-on-wfrec* [*OF g-even-def*])
apply (*auto simp add*: *indinv-on-def split*: *nat.split*)
by (*case-tac ka*, *auto*)

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

theorem *g-even-n*: ∃ *k*. *n* = *2*∗*k* + *4* ==> *g-even n* = *g-even* (*g-even* (*n* − *2*) − *1*)
apply (*subgoal-tac* (∃ *k*. *n* − *2* = *2*∗*k* + *2*) & (∃ *k*. *n* = *2*∗*k* + *2*))
by (*auto simp add*: *g-even-indinv*, *arith*)

McCarthy's ninety-one function. This function requires a non-standard measure to prove termination.

consts *ninety-one* :: *nat => nat*
recdef (**permissive**) *ninety-one measure* (%*n*. *101* − *n*)
  ninety-one x = (*if 100* < *x*
    then x − *10*
    else (*ninety-one* (*ninety-one* (*x*+*11*))))

To discharge the termination condition, we will prove a strengthened inductive invariant: S x y == x ¡ y + 11

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

Proving the termination condition using the strengthened inductive invariant.

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

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

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

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* (λ*f*. *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′*) (λ*y 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*
apply (*simp add*: *SC-def*)
done

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

lemma *PROJ-0* [*simp*]: *PROJ 0* (*x* # *l*) = *x*

apply (*simp add*: *PROJ-def*)
done

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

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

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

PROPERTY A 4

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

PROPERTY A 5-, the single-step lemma

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

PROPERTY A 5, monotonicity for <

lemma *ack-less-mono2*: *j* < *k* ==> *ack* (*i*, *j*) < *ack* (*i*, *k*)
apply (*induct i k rule*: *ack.induct*)
apply *simp-all*
apply (*blast elim*!: *less-SucE intro*: *less-trans*)
done

PROPERTY A 5', monotonicity for ≤

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

PROPERTY A 6

lemma *ack2-le-ack1* [*iff*]: *ack* (*i*, *Suc j*) ≤ *ack* (*Suc i*, *j*)
apply (*induct j*)
apply *simp-all*
apply (*blast intro*: *ack-le-mono2 less-ack2* [*THEN Suc-leI*] *le-trans*)
done

PROPERTY A 7-, the single-step lemma

lemma *ack-less-ack-Suc1* [*iff*]: $ack\ (i,\ j) < ack\ (Suc\ i,\ j)$
  apply (*blast intro*: *ack-less-mono2 less-le-trans*)
  done

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

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

PROPERTY A 8

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

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

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

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

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

lemma *ack-less-mono1*: $i < j \Longrightarrow ack\ (i,\ k) < ack\ (j,\ k)$
  apply (*drule less-imp-Suc-add*)
  apply (*blast intro*!: *ack-less-mono1-aux*)
  done

PROPERTY A 7', monotonicity for ≤

lemma *ack-le-mono1*: $i \le j \Longrightarrow ack\ (i,\ k) \le ack\ (j,\ k)$
  apply (*simp add*: *order-le-less*)
  apply (*blast intro*: *ack-less-mono1*)
  done

PROPERTY A 10

lemma *ack-nest-bound*: *ack*(*i1*, *ack* (*i2*, *j*)) < *ack* (*2* + (*i1* + *i2*), *j*)
apply (*simp add*: *numerals*)
apply (*rule ack2-le-ack1* [*THEN* [*2*] *less-le-trans*])
apply *simp*
apply (*rule le-add1* [*THEN ack-le-mono1*, *THEN le-less-trans*])
apply (*rule ack-less-mono1* [*THEN ack-less-mono2*])
apply (*simp add*: *le-imp-less-Suc le-add2*)
done

PROPERTY A 11

lemma *ack-add-bound*: *ack* (*i1*, *j*) + *ack* (*i2*, *j*) < *ack* (*4* + (*i1* + *i2*), *j*)
apply (*rule-tac j* = *ack* (*Suc* (*Suc 0*), *ack* (*i1* + *i2*, *j*)) **in** *less-trans*)
prefer *2*
apply (*rule ack-nest-bound* [*THEN less-le-trans*])
apply (*simp add*: *Suc3-eq-add-3*)
apply *simp*
apply (*cut-tac* $i = i1$ **and** $m1 = i2$ **and** $k = j$ **in** *le-add1* [*THEN ack-le-mono1*])
apply (*cut-tac* $i = i2$ **and** $m1 = i1$ **and** $k = j$ **in** *le-add2* [*THEN ack-le-mono1*])
apply *auto*
done

PROPERTY A 12. Article uses existential quantifier but the ALF proof used $k + 4$. Quantified version must be nested $\exists\, k'.\ \forall\, i\ j.$ ...

lemma *ack-add-bound2*: $i <$ *ack* (*k*, *j*) ==> $i + j <$ *ack* (*4* + *k*, *j*)
apply (*rule-tac j* = *ack* (*k*, *j*) + *ack* (*0*, *j*) **in** *less-trans*)
prefer *2*
apply (*rule ack-add-bound* [*THEN less-le-trans*])
apply *simp*
apply (*rule add-less-mono less-ack2* | *assumption*)+
done

Inductive definition of the *PR* functions

MAIN RESULT

lemma *SC-case*: *SC l* < *ack* (*1*, *list-add l*)
apply (*unfold SC-def*)
apply (*induct l*)
apply (*simp-all add*: *le-add1 le-imp-less-Suc*)
done

lemma *CONST-case*: *CONST k l* < *ack* (*k*, *list-add l*)
apply *simp*
done

lemma *PROJ-case* [*rule-format*]: $\forall\, i.$ *PROJ i l* < *ack* (*0*, *list-add l*)
apply (*simp add*: *PROJ-def*)
apply (*induct l*)
apply *simp-all*
apply (*rule allI*)

apply (*case-tac i*)
apply (*simp* (*no-asm-simp*) *add*: *le-add1 le-imp-less-Suc*)
apply (*simp* (*no-asm-simp*))
apply (*blast intro*: *less-le-trans intro*!: *le-add2*)
done

COMP case

lemma *COMP-map-aux*: *fs* ∈ *lists* (*PRIMREC* ∩ {*f*. ∃ *kf*. ∀ *l*. *f l* < *ack* (*kf*, *list-add l*)})
==> ∃ *k*. ∀ *l*. *list-add* (*map* (λ*f*. *f l*) *fs*) < *ack* (*k*, *list-add l*)
apply (*erule lists.induct*)
apply (*rule-tac x* = *0* **in** *exI*)
apply *simp*
apply *safe*
apply *simp*
apply (*rule exI*)
apply (*blast intro*: *add-less-mono ack-add-bound less-trans*)
done

lemma *COMP-case*:
∀ *l*. *g l* < *ack* (*kg*, *list-add l*) ==>
fs ∈ *lists*(*PRIMREC Int* {*f*. ∃ *kf*. ∀ *l*. *f l* < *ack*(*kf*, *list-add l*)})
==> ∃ *k*. ∀ *l*. *COMP g fs l* < *ack*(*k*, *list-add l*)
apply (*unfold COMP-def*)
apply (*frule Int-lower1* [*THEN lists-mono*, *THEN subsetD*])
— Now, if meson tolerated map, we could finish with (*drule COMP-map-aux*, *meson ack-less-mono2 ack-nest-bound less-trans*)
apply (*erule COMP-map-aux* [*THEN exE*])
apply (*rule exI*)
apply (*rule allI*)
apply (*drule spec*)+
apply (*erule less-trans*)
apply (*blast intro*: *ack-less-mono2 ack-nest-bound less-trans*)
done

PREC case

lemma *PREC-case-aux*:
∀ *l*. *f l* + *list-add l* < *ack* (*kf*, *list-add l*) ==>
∀ *l*. *g l* + *list-add l* < *ack* (*kg*, *list-add l*) ==>
PREC f g l + *list-add l* < *ack* (*Suc* (*kf* + *kg*), *list-add l*)
apply (*unfold PREC-def*)
apply (*case-tac l*)
apply *simp-all*
apply (*blast intro*: *less-trans*)
apply (*erule ssubst*) — get rid of the needless assumption
apply (*induct-tac a*)
apply *simp-all*

base case

apply (*blast intro*: *le-add1* [*THEN le-imp-less-Suc*, *THEN ack-less-mono1*] *less-trans*)

induction step

apply (*rule Suc-leI* [*THEN le-less-trans*])
apply (*rule le-refl* [*THEN add-le-mono*, *THEN le-less-trans*])
prefer *2*
apply (*erule spec*)
apply (*simp add*: *le-add2*)

final part of the simplification

apply *simp*
apply (*rule le-add2* [*THEN ack-le-mono1*, *THEN le-less-trans*])
apply (*erule ack-less-mono2*)
done

lemma *PREC-case*:
$\forall l.\ f\ l < ack\ (kf,\ list\text{-}add\ l) ==>$
$\forall l.\ g\ l < ack\ (kg,\ list\text{-}add\ l) ==>$
$\exists k.\ \forall l.\ PREC\ f\ g\ l < ack\ (k,\ list\text{-}add\ l)$
apply (*rule exI*)
apply (*rule allI*)
apply (*rule le-less-trans* [*OF le-add1 PREC-case-aux*])
apply (*blast intro*: *ack-add-bound2*)+
done

lemma *ack-bounds-PRIMREC*: $f \in PRIMREC ==> \exists k.\ \forall l.\ f\ l < ack\ (k,\ list\text{-}add\ l)$
apply (*erule PRIMREC.induct*)
apply (*blast intro*: *SC-case CONST-case PROJ-case COMP-case PREC-case*)+
done

lemma *ack-not-PRIMREC*: $(\lambda l.\ case\ l\ of\ [] => 0\ |\ x\ \#\ l' => ack\ (x,\ x)) \notin PRIMREC$
apply (*rule notI*)
apply (*erule ack-bounds-PRIMREC* [*THEN exE*])
apply (*rule less-irrefl*)
apply (*drule-tac* $x = [x]$ **in** *spec*)
apply *simp*
done

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

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

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

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

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

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

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

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

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

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

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

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

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

theorem (**in** *abelian-group-context*)
inv-prod': $(x \cdot y)^{-1} = x^{-1} \cdot y^{-1}$
proof −
have $(x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$ **by** (*rule inv-prod*)
also have ... $= x^{-1} \cdot y^{-1}$ **by** (*rule commute*)
finally show *?thesis* **.**
qed

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

Some more properties of inversion in general group theory follow.

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

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

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

We see that this representation of structures as local contexts is rather light-weight 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:

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

The corresponding introduction rule is as follows:

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

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 :: $'a \Rightarrow\ 'a \Rightarrow\ 'a$ (**infixl** $\cdot_\imath$ *70*)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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
fix $p\ q\ r :: 'a \times 'b$
have $prod\ S\ (prod\ S\ (fst\ p)\ (fst\ q))\ (fst\ r) =$
$prod\ S\ (fst\ p)\ (prod\ S\ (fst\ q)\ (fst\ r))$
by (*rule semigroup.assoc* [*OF S*])

moreover have *prod T* (*prod T* (*snd p*) (*snd q*)) (*snd r*) =
prod T (*snd p*) (*prod T* (*snd q*) (*snd r*))
by (*rule semigroup.assoc* [*OF T*])
ultimately
show *prod* (*semigroup-product S T*) (*prod* (*semigroup-product S T*) *p q*) *r* =
prod (*semigroup-product S T*) *p* (*prod* (*semigroup-product S T*) *q r*)
by (*simp add*: *semigroup-product-def*)
qed

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

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

lemma
includes *semigroup S* + *semigroup T*
fixes *U* (**structure**)
defines $U \equiv$ *semigroup-product S T*
shows *semigroup U*
proof
fix $p\ q\ r :: {}'a \times {}'b$
have $(fst\ p \cdot_1 fst\ q) \cdot_1 fst\ r = fst\ p \cdot_1 (fst\ q \cdot_1 fst\ r)$
by (*rule S.assoc*)
moreover have $(snd\ p \cdot_2 snd\ q) \cdot_2 snd\ r = snd\ p \cdot_2 (snd\ q \cdot_2 snd\ r)$
by (*rule T.assoc*)
ultimately show $(p \cdot_3 q) \cdot_3 r = p \cdot_3 (q \cdot_3 r)$
by (*simp add*: *U-def semigroup-product-def semigroup.defs*)
qed

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

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

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

proof −
let *?G′* = *semigroup.truncate G* **and** *?H′* = *semigroup.truncate H*
have *prod* (*semigroup-product ?G′ ?H′*) = *prod I*
by (*simp add*: *I-def group-product-def group.defs*
semigroup-product-def semigroup.defs)
moreover
have *semigroup ?G′* **and** *semigroup ?H′*
using *prems* **by** (*simp-all add*: *semigroup-def semigroup.defs*)
then have *semigroup* (*semigroup-product ?G′ ?H′*) **..**
ultimately show *?thesis* **by** (*simp add*: *I-def semigroup-def*)
qed
show *group-axioms I*
proof
fix p :: $'a \times 'b$
have $(fst\ p)^{-1}{}_1 \cdot_1 fst\ p = \mathbf{1}_1$
by (*rule G.left-inv*)
moreover have $(snd\ p)^{-1}{}_2 \cdot_2 snd\ p = \mathbf{1}_2$
by (*rule H.left-inv*)
ultimately show $p^{-1}{}_3 \cdot_3 p = \mathbf{1}_3$
by (*simp add*: *I-def group-product-def group.defs*
semigroup-product-def semigroup.defs)
have $\mathbf{1}_1 \cdot_1 fst\ p = fst\ p$ **by** (*rule G.left-one*)
moreover have $\mathbf{1}_2 \cdot_2 snd\ p = snd\ p$ **by** (*rule H.left-one*)
ultimately show $\mathbf{1}_3 \cdot_3 p = p$
by (*simp add*: *I-def group-product-def group.defs*
semigroup-product-def semigroup.defs)
qed
qed

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

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* |)
by (*simp only*: *point.make-def*)

lemma *xpos* (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) = *m*
by *simp*

lemma (| *xpos* = *m*, *ypos* = *n*, ... = *p* |) (| *xpos*:= *0* |) = (| *xpos* = *0*, *ypos* = *n*, ... = *p* |)
by *simp*

Equality of records.

lemma *n* = *n'* ==> *p* = *p'* ==> (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |)
— introduction of concrete record equality
by *simp*

lemma (| *xpos* = *n*, *ypos* = *p* |) = (| *xpos* = *n'*, *ypos* = *p'* |) ==> *n* = *n'*
  — elimination of concrete record equality
  by *simp*

lemma *r* (| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
  — introduction of abstract record equality
  by *simp*

lemma *r* (| *xpos* := *n* |) = *r* (| *xpos* := *n'* |) ==> *n* = *n'*
  — elimination of abstract record equality (manual proof)
proof −
  assume *r* (| *xpos* := *n* |) = *r* (| *xpos* := *n'* |) (**is** *?lhs* = *?rhs*)
  hence *xpos ?lhs* = *xpos ?rhs* **by** *simp*
  thus *?thesis* **by** *simp*
qed

Surjective pairing

lemma *r* = (| *xpos* = *xpos r*, *ypos* = *ypos r* |)
  by *simp*

lemma *r* = (| *xpos* = *xpos r*, *ypos* = *ypos r*, ... = *point.more r* |)
  by *simp*

Representation of records by cases or (degenerate) induction.

lemma *r*(| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
proof (*cases r*)
  fix *xpos ypos more*
  assume *r* = (| *xpos* = *xpos*, *ypos* = *ypos*, ... = *more* |)
  thus *?thesis* **by** *simp*
qed

lemma *r* (| *xpos* := *n* |) (| *ypos* := *m* |) = *r* (| *ypos* := *m* |) (| *xpos* := *n* |)
proof (*induct r*)
  fix *xpos ypos more*
  show (| *xpos* = *xpos*, *ypos* = *ypos*, ... = *more* |) (| *xpos* := *n*, *ypos* := *m* |) =
      (| *xpos* = *xpos*, *ypos* = *ypos*, ... = *more* |) (| *ypos* := *m*, *xpos* := *n* |)
    by *simp*
qed

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
proof (*cases r*)
  fix *xpos ypos more*
  assume *r* = (|*xpos* = *xpos*, *ypos* = *ypos*, . . . = *more*|)
  thus *?thesis* **by** *simp*
qed

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)

proof (*cases r*)
  case *fields*
  thus *?thesis* **by** *simp*
qed

lemma *r* (| *xpos* := *n* |) (| *xpos* := *m* |) = *r* (| *xpos* := *m* |)
  by (*cases r*) *simp*

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*))
  by (*simp add*: *getX-def setX-def incX-def*)

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 == $\forall$ *x y z*.
    times M (*times M x y*) *z* = *times M x* (*times M y z*) $\wedge$
    times M (*one M*) *x* = *x* $\wedge$ *times M x* (*one M*) = *x*

  group :: (| *times* :: *'a* => *'a* => *'a*, *one* :: *'a*, *inv* :: *'a* => *'a*, ... :: *'b* |) => *bool*
  group G == *monoid G* $\wedge$ ($\forall$ *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* := $\lambda x\ y.$ *times M y x* |)

end

9 String examples

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

lemma *hd ''ABCD''* = *CHR ''A''*
by *simp*

lemma *hd ''ABCD''* $\neq$ *CHR ''B''*
by *simp*

lemma *''ABCD''* $\neq$ *''ABCX''*
by *simp*

lemma *''ABCD''* = *''ABCD''*
by *simp*

lemma *''ABCDEFGHIJKLMNOPQRSTUVWXYZ''* $\neq$
''ABCDEFGHIJKLMNOPQRSTUVWXY''
by *simp*

lemma *set ''Foobar''* = {*CHR ''F''*, *CHR ''a''*, *CHR ''b''*, *CHR ''o''*, *CHR ''r''*}
by (*simp add*: *insert-commute*)

lemma *set ''Foobar''* = *?X*
by (*simp add*: *insert-commute*)

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*)
apply *simp* **oops**

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

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

apply *simp* **oops**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.2 Arithmetic Method Tests

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

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

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

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

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

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

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

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

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

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

by *arith*

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

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

10.3 The Integers

Addition

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

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

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

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

Negation

lemma − (*65745*::*int*) = −*65745*
by *simp*

lemma − (−*54321*::*int*) = *54321*
by *simp*

Multiplication

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

lemma (−*84*::*int*) ∗ *51* = −*4284*
by *simp*

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

lemma (*1359*::*int*) ∗ −*2468* = −*3354012*
by *simp*

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

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

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

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

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

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

No integer overflow!

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

Quotient and Remainder

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

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

A negative divisor

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

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

A negative dividend[1]

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

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

A negative dividend *and* divisor

lemma $(-10::int)\ div\ -3 = 3$

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

by *simp*

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

A few bigger examples

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

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

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

Division by shifting

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

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

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

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

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

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

Powers

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

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

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

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

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

10.4 The Natural Numbers

Successor

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

Addition

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

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

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

Subtraction

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

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

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

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

Multiplication

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

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

Quotient and Remainder

lemma (*10*::*nat*) *div 3* = *3*
by *simp*

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

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

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

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

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

Powers

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

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

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

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

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

Testing the cancellation of complementary terms

lemma $y + (x + -x) = (0::int) + y$
  by *simp*

lemma $y + (-x + (- y + x)) = (0::int)$
  by *simp*

lemma $-x + (y + (- y + x)) = (0::int)$
  by *simp*

lemma $x + (x + (- x + (- x + (- y + - z)))) = (0::int) - y - z$
  by *simp*

lemma $x + x - x - x - y - z = (0::int) - y - z$
  by *simp*

lemma $x + y + z - (x + z) = y - (0::int)$
  by *simp*

lemma $x + (y + (y + (y + (-x + -x)))) = (0::int) + y - x + y + y$
  by *simp*

lemma $x + (y + (y + (y + (-y + -x)))) = y + (0::int) + y$
  by *simp*

lemma $x + y - x + z - x - y - z + x < (1::int)$
  by *simp*

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

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

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

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

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′*: $A \vee \neg A$

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

11.4 Proof term of script

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

end

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

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

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

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

end
```

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)

parse-translation ⟪
  let
    fun antiquote-tr i (Const (-antiquote, -) $ (t as Const (-antiquote, -) $ -)) =
          skip-antiquote-tr i t
      | antiquote-tr i (Const (-antiquote, -) $ t) =
          antiquote-tr i t $ Bound i
```

```
    | antiquote-tr i (t $ u) = antiquote-tr i t $ antiquote-tr i u
    | antiquote-tr i (Abs (x, T, t)) = Abs (x, T, antiquote-tr (i + 1) t)
    | antiquote-tr - a = a
  and skip-antiquote-tr i ((c as Const (-antiquote, -)) $ t) =
        c $ skip-antiquote-tr i t
    | skip-antiquote-tr i t = antiquote-tr i t;

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

basic examples

term «*a* + *b* + *c*»
term «*a* + *b* + *c* + ´*x* + ´*y* + *1*»
term «´(*f w*) + ´*x*»
term «*f* ´*x* ´*y z*»

advanced examples

term ««´´*x* + ´*y*»»
term ««´´*x* + ´*y*» *o* ´*f*»
term «´(*f o* ´*g*)»
term ««´´(*f o* ´*g*)»»

end

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*

typed-print-translation ⟪
let
fun split-tr′ - T1
(*Abs* (*x*, *xT*, *Const* (*split*, *T2*) $ *Abs* (*y*, *yT*, *Abs* (-, *Type* (*unit*, []), *b*)))
:: *ts*) =
if Term.loose-bvar1 (*b*, *0*) *then raise Match*
else Term.list-comb
(*Const* (*split*, *T1*) $ *Abs* (*x*, *xT*, *Const* (*split*, *T2*) $
Abs (*y*, *yT*, *Syntax.const -K* $ *Term.incr-boundvars* ~*1 b*)), *ts*)
| *split-tr′* - - - = *raise Match*;
in [(*split*, *split-tr′*)] *end*
⟫

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

The sum of the first n odd squares.

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

The sum of the first n odd cubes

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

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

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

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

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

Sum of fourth powers: three versions.

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

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

lemma *int-sum-of-fourth-powers*:
 $30 * int\ (\sum i{=}0..{<}m.\ i * i * i * i) =$
  $int\ m * (int\ m - 1) * (int(2 * m) - 1) *$
  $(int(3 * m * m) - int(3 * m) - 1)$
 by (*induct m*) (*simp-all add*: *int-mult*)

lemma *of-nat-sum-of-fourth-powers*:
 $30 * of\text{-}nat\ (\sum i{=}0..{<}m.\ i * i * i * i) =$
  $of\text{-}nat\ m * (of\text{-}nat\ m - 1) * (of\text{-}nat\ (2 * m) - 1) *$
  $(of\text{-}nat\ (3 * m * m) - of\text{-}nat\ (3 * m) - (1{::}int))$
 by (*induct m*) *simp-all*

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

lemma *sum-of-2-powers*: $(\sum i{=}0..{<}n.\ 2\hat{\ }i) = 2\hat{\ }n - (1{::}nat)$
 by (*induct n*) (*auto split*: *nat-diff-split*)

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

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

end

16 Higher-Order Logic: Intuitionistic predicate calculus problems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma ~~((~P-->Q) = (~Q --> P))

by *iprover*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
lemma (ALL x. P = Q(x))  -->  (P = (ALL x. Q(x)))
  by iprover


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


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


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


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


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


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


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

by *iprover*

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

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

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

lemma ${\sim}\ (EX\ x.\ ALL\ y.\ F\ y\ x = ({\sim}F\ y\ y))$
by *iprover*

lemma $(EX\ y.\ ALL\ x.\ F\ x\ y = F\ x\ x)\ {-}{-}{>}$
${\sim}(ALL\ x.\ EX\ y.\ ALL\ z.\ F\ z\ y = ({\sim}\ F\ z\ x))$
by *iprover*

lemma $(ALL\ x.\ f(x)\ {-}{-}{>}$
$(EX\ y.\ g(y)\ \&\ h\ x\ y\ \&\ (EX\ y.\ g(y)\ \&\ {\sim}\ h\ x\ y)))\ \ \&$
$(EX\ x.\ j(x)\ \&\ (ALL\ y.\ g(y)\ {-}{-}{>}\ h\ x\ y))$
${-}{-}{>}\ (EX\ x.\ j(x)\ \&\ {\sim}f(x))$
by *iprover*

lemma $(a{=}b\ |\ c{=}d)\ \&\ (a{=}c\ |\ b{=}d)\ {-}{-}{>}\ a{=}d\ |\ b{=}c$
by *iprover*

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

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

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

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

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

end

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

If and only if

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

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

Sample problems from F. J. Pelletier, Seventy-Five Problems for Testing Automatic Theorem Provers, J. Automated Reasoning 2 (1986), 191-216. Errata, JAR 4 (1988), 236-236.

The hardest problems – judging by experience with several theorem provers, including matrix ones – are 34 and 43.

17.1.1 Pelletier's examples

1

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

2

lemma $({\sim}\;{\sim}\;P) = \;P$
by *blast*

3

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

4

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

5

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

6

lemma $P \mid {\sim}\; P$
by *blast*

7

lemma $P \mid {\sim}\;{\sim}\;{\sim}\; P$
by *blast*

8. Peirce's law

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

9

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

10

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

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

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

12. "Dijkstra's law"

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

13. Distributive law

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

14

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

15

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

16

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

17

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

17.1.2 Classical Logic: examples with quantifiers

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

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

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

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

From Wishnu Prasetya

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

17.1.3 Problems requiring quantifier duplication

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

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

by *blast*

Needs multiple instantiation of the quantifier.

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

Needs double instantiation of the quantifier

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

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

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

17.1.4 Hard examples with quantifiers

Problem 18

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

Problem 19

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

Problem 20

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

Problem 21

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

Problem 22

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

Problem 23

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

Problem 24

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

$\quad --> (\exists x.\ P(x) \& R(x))$
by *blast*

Problem 25

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

Problem 26

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

Problem 27

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

Problem 28. AMENDED

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

Problem 29. Essentially the same as Principia Mathematica *11.71

lemma $(\exists x.\ F(x))\ \&\ (\exists y.\ G(y))$
$\quad --> (\ ((\forall x.\ F(x) --> H(x))\ \&\ (\forall y.\ G(y) --> J(y)))\ =$
$\qquad (\forall x\ y.\ F(x)\ \&\ G(y)\ -->\ H(x)\ \&\ J(y)))$
by *blast*

Problem 30

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

Problem 31

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

$--> (\exists\, x.\ L(x)\ \&\ M(x))$
by *blast*

Problem 32

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

Problem 33

lemma $(\forall\, x.\ P(a)\ \&\ (P(x) --> P(b)) --> P(c))\ =$
$(\forall\, x.\ ({\sim}P(a)\ |\ P(x)\ |\ P(c))\ \&\ ({\sim}P(a)\ |\ {\sim}P(b)\ |\ P(c)))$
by *blast*

Problem 34 AMENDED (TWICE!!)

Andrews's challenge

lemma $((\exists\, x.\ \forall\, y.\ p(x) = p(y))\ =$
$((\exists\, x.\ q(x)) = (\forall\, y.\ p(y))))\ \ =$
$((\exists\, x.\ \forall\, y.\ q(x) = q(y))\ =$
$((\exists\, x.\ p(x)) = (\forall\, y.\ q(y))))$
by *blast*

Problem 35

lemma $\exists\, x\ y.\ P\ x\ y\ -->\ (\forall\, u\ v.\ P\ u\ v)$
by *blast*

Problem 36

lemma $(\forall\, x.\ \exists\, y.\ J\ x\ y)\ \&$
$(\forall\, x.\ \exists\, y.\ G\ x\ y)\ \&$
$(\forall\, x\ y.\ J\ x\ y\ |\ G\ x\ y\ -->$
$(\forall\, z.\ J\ y\ z\ |\ G\ y\ z\ --> H\ x\ z))$
$--> (\forall\, x.\ \exists\, y.\ H\ x\ y)$
by *blast*

Problem 37

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

Problem 38

lemma $(\forall\, x.\ p(a)\ \&\ (p(x)\ --> (\exists\, y.\ p(y)\ \&\ r\ x\ y))\ -->$
$(\exists\, z.\ \exists\, w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$(\forall\, x.\ ({\sim}p(a)\ |\ p(x)\ |\ (\exists\, z.\ \exists\, w.\ p(z)\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$

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

Problem 39

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

Problem 40. AMENDED

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

Problem 41

lemma $(\forall z.\ \exists y.\ \forall x.\ f\ x\ y = (f\ x\ z\ \&\ \sim\ f\ x\ x))$
$\quad --> \sim\ (\exists z.\ \forall x.\ f\ x\ z)$
by *blast*

Problem 42

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

Problem 43!!

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

Problem 44

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

Problem 45

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

17.1.5 Problems (mainly) involving equality or functions

Problem 48

lemma $(a{=}b \mid c{=}d)\ \&\ (a{=}c \mid b{=}d)\ -->\ a{=}d \mid b{=}c$

by *blast*

Problem 49 NOT PROVED AUTOMATICALLY. Hard because it involves substitution for Vars the type constraint ensures that x,y,z have the same type as a,b,u.

lemma $(\exists\, x\ y{::}'a.\ \forall\, z.\ z{=}x \mid z{=}y)\ \&\ P(a)\ \&\ P(b)\ \&\ ({\sim}\, a{=}b)$
$\quad --> (\forall\, u{::}'a.\ P(u))$
apply *safe*
apply (*rule-tac* $x = a$ **in** *allE*, *assumption*)
apply (*rule-tac* $x = b$ **in** *allE*, *assumption*, *fast*) — blast's treatment of equality can't do it
done

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

lemma $(\forall\, x.\ P\ a\ x \mid (\forall\, y.\ P\ x\ y)) --> (\exists\, x.\ \forall\, y.\ P\ x\ y)$
by *blast*

Problem 51

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

Problem 52. Almost the same as 51.

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

Problem 55

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

lemma *lives*(*agatha*) & *lives*(*butler*) & *lives*(*charles*) &
(*killed agatha agatha* | *killed butler agatha* | *killed charles agatha*) &
($\forall$ *x y*. *killed x y* --> *hates x y* & ~*richer x y*) &
($\forall$ *x*. *hates agatha x* --> ~*hates charles x*) &
(*hates agatha agatha* & *hates agatha charles*) &
($\forall$ *x*. *lives*(*x*) & ~*richer x agatha* --> *hates butler x*) &
($\forall$ *x*. *hates agatha x* --> *hates butler x*) &
($\forall$ *x*. ~*hates x agatha* | ~*hates x butler* | ~*hates x charles*) -->
killed ?who agatha
by *fast*

Problem 56

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

Problem 57

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

$(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z\ \text{-->}\ P\ x\ z)\quad \text{-->}\quad P\ (f\ a\ b)\ (f\ a\ c)$
by *blast*

Problem 58 NOT PROVED AUTOMATICALLY

lemma $(\forall x\ y.\ f(x){=}g(y))\ \text{-->}\ (\forall x\ y.\ f(f(x)){=}f(g(y)))$
by (*fast intro*: *arg-cong* [*of* **concl**: *f*])

Problem 59

lemma $(\forall x.\ P(x) = (\sim P(f(x))))\ \text{-->}\ (\exists x.\ P(x)\ \&\ \sim P(f(x)))$
by *blast*

Problem 60

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

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

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ \text{-->}\ p(f\ x))\ \text{-->}\ p(f(f\ x)))\ =$
$(\forall x.\ (\sim\ p\ a\ |\ p\ x\ |\ p(f(f\ x)))\ \&$
$(\sim\ p\ a\ |\ \sim\ p(f\ x)\ |\ p(f(f\ x))))$
by *blast*

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

lemma $(\forall x.\ F(x)\ \&\ \sim G(x)\ \text{-->}\ (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ \text{-->}\ K(y)))\ \&$
$(\forall x.\ K(x)\ \text{-->}\ \sim G(x))\ \ \text{-->}\ \ (\exists x.\ K(x)\ \&\ J(x))$
by *fast*

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

lemma $(\forall x.\ F(x)\ \&\ \sim G(x)\ \text{-->}\ (\exists y.\ H(x,y)\ \&\ J(y)))\ \&$
$(\exists x.\ K(x)\ \&\ F(x)\ \&\ (\forall y.\ H(x,y)\ \text{-->}\ K(y)))\ \&$
$(\forall x.\ K(x)\ \text{-->}\ \sim G(x))\quad \text{-->}\quad (\exists x.\ K(x)\ \text{-->}\ \sim G(x))$
by *fast*

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

lemma $(\forall x.\ \mathit{honest}(x)\ \&\ \mathit{industrious}(x)\ \text{-->}\ \mathit{healthy}(x))\ \&$
$\sim\ (\exists x.\ \mathit{grocer}(x)\ \&\ \mathit{healthy}(x))\ \&$
$(\forall x.\ \mathit{industrious}(x)\ \&\ \mathit{grocer}(x)\ \text{-->}\ \mathit{honest}(x))\ \&$
$(\forall x.\ \mathit{cyclist}(x)\ \text{-->}\ \mathit{industrious}(x))\ \&$
$(\forall x.\ \sim\mathit{healthy}(x)\ \&\ \mathit{cyclist}(x)\ \text{-->}\ \sim\mathit{honest}(x))$
$\text{-->}\ (\forall x.\ \mathit{grocer}(x)\ \text{-->}\ \sim\mathit{cyclist}(x))$
by *blast*

lemma $(\forall x\ y.\ R(x,y)\ |\ R(y,x))\ \&$
$(\forall x\ y.\ S(x,y)\ \&\ S(y,x)\ \text{-->}\ x{=}y)\ \&$
$(\forall x\ y.\ R(x,y)\ \text{-->}\ S(x,y))\quad \text{-->}\quad (\forall x\ y.\ S(x,y)\ \text{-->}\ R(x,y))$
by *blast*

17.2 Model Elimination Prover

Trying out meson with arguments

lemma $x < y$ & $y < z$ --> ~ (z < (x::*nat*))
by (*meson order-less-irrefl order-less-trans*)

The "small example" from Bezem, Hendriks and de Nivelle, Automatic Proof Construction in Type Theory Using Resolution, JAR 29: 3-4 (2002), pages 253-275

lemma ($\forall\, x\ y\ z.\ R(x,y)$ & $R(y,z)$ --> $R(x,z)$) &
($\forall\, x.\ \exists\, y.\ R(x,y)$) -->
~ ($\forall\, x.\ P\ x$ = ($\forall\, y.\ R(x,y)$ --> ~ $P\ y$))
by (*tactic*⟨⟨*safe-best-meson-tac 1*⟩⟩)
— In contrast, *meson* is SLOW: 7.6s on griffon

17.2.1 Pelletier's examples

1

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

2

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

3

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

4

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

5

lemma ((P|Q)-->(P|R)) --> (P|(Q-->R))
by *blast*

6

lemma P | ~ P
by *blast*

7

lemma P | ~ ~ ~ P
by *blast*

8. Peirce's law

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

9

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

10

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

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

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

12. "Dijkstra's law"

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

13. Distributive law

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

14

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

15

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

16

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

17

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

17.2.2 Classical Logic: examples with quantifiers

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

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

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

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

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

Needs double instantiation of EXISTS

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

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

From a paper by Claire Quigley

lemma $\exists y.\ ((P\ c\ \&\ Q\ y) \mid (\exists z.\ {\sim}\ Q\ z)) \mid (\exists x.\ {\sim}\ P\ x\ \&\ Q\ d)$
by *fast*

17.2.3 Hard examples with quantifiers

Problem 18

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

Problem 19

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

Problem 20

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

Problem 21

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

Problem 22

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

Problem 23

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

Problem 24

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

Problem 25

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

Problem 26; has 24 Horn clauses

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

Problem 27; has 13 Horn clauses

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

Problem 28. AMENDED; has 14 Horn clauses

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

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

lemma $(\exists x.\ F\ x)\ \&\ (\exists y.\ G\ y)$
${-}{-}{>}\ (\ ((\forall x.\ F\ x\ {-}{-}{>}\ H\ x)\ \&\ (\forall y.\ G\ y\ {-}{-}{>}\ J\ y))\ =$
$(\forall x\ y.\ F\ x\ \&\ G\ y\ {-}{-}{>}\ H\ x\ \&\ J\ y))$
by *blast*

Problem 30

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

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

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

Problem 32

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

Problem 33; has 55 Horn clauses

lemma $(\forall x.\ P\ a\ \&\ (P\ x\ {-}{-}{>}\ P\ b){-}{-}{>}P\ c)\ =$
$(\forall x.\ (\sim P\ a\ |\ P\ x\ |\ P\ c)\ \&\ (\sim P\ a\ |\ \sim P\ b\ |\ P\ c))$
by *blast*

Problem 34: Andrews's challenge has 924 Horn clauses

lemma $((\exists x.\ \forall y.\ p\ x = p\ y)\ = ((\exists x.\ q\ x) = (\forall y.\ p\ y)))\quad =$
$((\exists x.\ \forall y.\ q\ x = q\ y)\ = ((\exists x.\ p\ x) = (\forall y.\ q\ y)))$
by *blast*

Problem 35

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

Problem 36; has 15 Horn clauses

lemma $(\forall x.\ \exists y.\ J\ x\ y)\ \&\ (\forall x.\ \exists y.\ G\ x\ y)\ \&$
$(\forall x\ y.\ J\ x\ y\ |\ G\ x\ y\ {-}{-}{>}\ (\forall z.\ J\ y\ z\ |\ G\ y\ z\ {-}{-}{>}\ H\ x\ z))$
${-}{-}{>}\ (\forall x.\ \exists y.\ H\ x\ y)$
by *blast*

Problem 37; has 10 Horn clauses

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

Problem 38

Quite hard: 422 Horn clauses!!

lemma $(\forall x.\ p\ a\ \&\ (p\ x\ {-}{-}{>}\ (\exists y.\ p\ y\ \&\ r\ x\ y))\ {-}{-}{>}$
$(\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ =$
$(\forall x.\ (\sim p\ a\ |\ p\ x\ |\ (\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z))\ \&$
$(\sim p\ a\ |\ \sim(\exists y.\ p\ y\ \&\ r\ x\ y)\ |$

$(\exists z.\ \exists w.\ p\ z\ \&\ r\ x\ w\ \&\ r\ w\ z)))$
by *blast*

Problem 39

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

Problem 40. AMENDED

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

Problem 41

lemma $(\forall z.\ (\exists y.\ (\forall x.\ f\ x\ y = (f\ x\ z\ \&\ \sim\ f\ x\ x))))$
$\quad --> \sim\ (\exists z.\ \forall x.\ f\ x\ z)$
by *blast*

Problem 42

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

Problem 43 NOW PROVED AUTOMATICALLY!!

lemma $(\forall x.\ \forall y.\ q\ x\ y = (\forall z.\ p\ z\ x = (p\ z\ y{::}bool)))$
$\quad --> (\forall x.\ (\forall y.\ q\ x\ y = (q\ y\ x{::}bool)))$
by *blast*

Problem 44: 13 Horn clauses; 7-step proof

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

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

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

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

lemma $(\forall x.\ f\ x\ \&\ (\forall y.\ f\ y\ \&\ h\ y\ x\ --> g\ y)\ --> g\ x)\ \&$
$\quad ((\exists x.\ f\ x\ \&\ \sim g\ x)\ -->$
$\quad (\exists x.\ f\ x\ \&\ \sim g\ x\ \&\ (\forall y.\ f\ y\ \&\ \sim g\ y\ --> j\ x\ y)))\ \&$
$\quad (\forall x\ y.\ f\ x\ \&\ f\ y\ \&\ h\ x\ y\ --> \sim j\ y\ x)$
$\quad --> (\forall x.\ f\ x\ --> g\ x)$

by *blast*

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

lemma $(\forall x.\ \mathit{wolf}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{wolf}\ x)\ \&$
$(\forall x.\ \mathit{fox}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{fox}\ x)\ \&$
$(\forall x.\ \mathit{bird}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{bird}\ x)\ \&$
$(\forall x.\ \mathit{caterpillar}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{caterpillar}\ x)\ \&$
$(\forall x.\ \mathit{snail}\ x \longrightarrow \mathit{animal}\ x)\ \&\ (\exists x.\ \mathit{snail}\ x)\ \&$
$(\forall x.\ \mathit{grain}\ x \longrightarrow \mathit{plant}\ x)\ \&\ (\exists x.\ \mathit{grain}\ x)\ \&$
$(\forall x.\ \mathit{animal}\ x \longrightarrow$
$((\forall y.\ \mathit{plant}\ y \longrightarrow \mathit{eats}\ x\ y)\ \vee$
$(\forall y.\ \mathit{animal}\ y\ \&\ \mathit{smaller\text{-}than}\ y\ x\ \&$
$(\exists z.\ \mathit{plant}\ z\ \&\ \mathit{eats}\ y\ z) \longrightarrow \mathit{eats}\ x\ y)))\ \&$
$(\forall x\ y.\ \mathit{bird}\ y\ \&\ (\mathit{snail}\ x \vee \mathit{caterpillar}\ x) \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{fox}\ y \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{fox}\ x\ \&\ \mathit{wolf}\ y \longrightarrow \mathit{smaller\text{-}than}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{wolf}\ x\ \&\ (\mathit{fox}\ y \vee \mathit{grain}\ y) \longrightarrow {\sim}\mathit{eats}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{caterpillar}\ y \longrightarrow \mathit{eats}\ x\ y)\ \&$
$(\forall x\ y.\ \mathit{bird}\ x\ \&\ \mathit{snail}\ y \longrightarrow {\sim}\mathit{eats}\ x\ y)\ \&$
$(\forall x.\ (\mathit{caterpillar}\ x \vee \mathit{snail}\ x) \longrightarrow (\exists y.\ \mathit{plant}\ y\ \&\ \mathit{eats}\ x\ y))$
$\longrightarrow (\exists x\ y.\ \mathit{animal}\ x\ \&\ \mathit{animal}\ y\ \&\ (\exists z.\ \mathit{grain}\ z\ \&\ \mathit{eats}\ y\ z\ \&\ \mathit{eats}\ x\ y))$
by (*tactic*⟨⟨*safe-best-meson-tac 1*⟩⟩)

— Nearly twice as fast as *meson*, which performs iterative deepening rather than best-first search

The Los problem. Circulated by John Harrison

lemma $(\forall x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z \dashrightarrow P\ x\ z)\ \&$
$(\forall x\ y\ z.\ Q\ x\ y\ \&\ Q\ y\ z \dashrightarrow Q\ x\ z)\ \&$
$(\forall x\ y.\ P\ x\ y \dashrightarrow P\ y\ x)\ \&$
$(\forall x\ y.\ P\ x\ y\ |\ Q\ x\ y)$
$\dashrightarrow (\forall x\ y.\ P\ x\ y)\ |\ (\forall x\ y.\ Q\ x\ y)$
by *meson*

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

lemma $(\forall x\ y\ z.\ P\ x\ y \dashrightarrow P\ y\ z \dashrightarrow P\ x\ z) \dashrightarrow$
$(\forall x\ y\ z.\ Q\ x\ y \dashrightarrow Q\ y\ z \dashrightarrow Q\ x\ z) \dashrightarrow$
$(\forall x\ y.\ Q\ x\ y \dashrightarrow Q\ y\ x) \dashrightarrow (\forall x\ y.\ P\ x\ y\ |\ Q\ x\ y) \dashrightarrow$
$(\forall x\ y.\ P\ x\ y)\ |\ (\forall x\ y.\ Q\ x\ y)$
by *meson*

Problem 50. What has this to do with equality?

lemma $(\forall x.\ P\ a\ x\ |\ (\forall y.\ P\ x\ y)) \dashrightarrow (\exists x.\ \forall y.\ P\ x\ y)$
by *blast*

Problem 54: NOT PROVED

lemma $(\forall y{::}'a.\ \exists z.\ \forall x.\ F\ x\ z = (x{=}y)) \dashrightarrow$

$\sim (\exists\, w.\ \forall\, x.\ F\ x\ w = (\forall\, u.\ F\ x\ u \ {-}{-}{>}\ (\exists\, y.\ F\ y\ u\ \&\ \sim (\exists\, z.\ F\ z\ u\ \&\ F\ z\ y))))$
oops

Problem 55

Non-equational version, from Manthey and Bry, CADE-9 (Springer, 1988). *meson* cannot report who killed Agatha.

lemma *lives agatha* & *lives butler* & *lives charles* &
(*killed agatha agatha* | *killed butler agatha* | *killed charles agatha*) &
($\forall$ *x y*. *killed x y* --> *hates x y* & ~*richer x y*) &
($\forall$ *x*. *hates agatha x* --> ~*hates charles x*) &
(*hates agatha agatha* & *hates agatha charles*) &
($\forall$ *x*. *lives x* & ~*richer x agatha* --> *hates butler x*) &
($\forall$ *x*. *hates agatha x* --> *hates butler x*) &
($\forall$ *x*. ~*hates x agatha* | ~*hates x butler* | ~*hates x charles*) -->
($\exists$ *x*. *killed x agatha*)
by *meson*

Problem 57

lemma $P\ (f\ a\ b)\ (f\ b\ c)\ \&\ P\ (f\ b\ c)\ (f\ a\ c)\ \&$
$(\forall\, x\ y\ z.\ P\ x\ y\ \&\ P\ y\ z \ {-}{-}{>}\ P\ x\ z) \quad {-}{-}{>} \quad P\ (f\ a\ b)\ (f\ a\ c)$
by *blast*

Problem 58: Challenge found on info-hol

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

Problem 59

lemma $(\forall\, x.\ P\ x = ({\sim}P(f\ x))) \ {-}{-}{>}\ (\exists\, x.\ P\ x\ \&\ {\sim}P(f\ x))$
by *blast*

Problem 60

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

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

lemma $(\forall\, x.\ p\ a\ \&\ (p\ x \ {-}{-}{>}\ p(f\ x)) \ {-}{-}{>}\ p(f(f\ x))) \ =$
$(\forall\, x.\ ({\sim}\ p\ a\ |\ p\ x\ |\ p(f(f\ x)))\ \&$
$({\sim}\ p\ a\ |\ {\sim}\ p(f\ x)\ |\ p(f(f\ x))))$
by *blast*

* Charles Morgan's problems *

lemma
assumes a: $\forall\, x\ y.\ \ T(i\ x(i\ y\ x))$
and b: $\forall\, x\ y\ z.\ T(i\ (i\ x\ (i\ y\ z))\ (i\ (i\ x\ y)\ (i\ x\ z)))$
and c: $\forall\, x\ y.\ \ \ T(i\ (i\ (n\ x)\ (n\ y))\ (i\ y\ x))$
and c': $\forall\, x\ y.\ \ \ T(i\ (i\ y\ x)\ (i\ (n\ x)\ (n\ y)))$
and d: $\forall\, x\ y.\ \ \ T(i\ x\ y)\ \&\ T\ x \ {-}{-}{>}\ T\ y$

shows *True*
proof –
  from *a b d* **have** $\forall x.\ T(i\ x\ x)$ **by** *blast*
  from *a b c d* **have** $\forall x.\ T(i\ x\ (n(n\ x)))$ — Problem 66
    by *meson*
      — SLOW: 18s on griffon. 208346 inferences, depth 23
  from *a b c d* **have** $\forall x.\ T(i\ (n(n\ x))\ x)$ — Problem 67
    by *meson*
      — 4.9s on griffon. 51061 inferences, depth 21
  from *a b c' d* **have** $\forall x.\ T(i\ x\ (n(n\ x)))$
      — Problem 68: not proved. Listed as satisfiable in TPTP (LCL078-1)
oops

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

lemma $p1 = (p2 = (p3 = (p4 = (p5 = (p1 = (p2 = (p3 = (p4 = p5))))))))$
by *blast*

A manual resolution proof of problem 19.

lemma $\exists x.\ \forall y\ z.\ (P(y){-}{-}{>}Q(z))\ {-}{-}{>}\ (P(x){-}{-}{>}Q(x))$
proof (*rule ccontr*, *skolemize*, *make-clauses*)
  fix *f g*
  assume P: $\bigwedge U.\ \neg\ P\ U \Longrightarrow False$
    and Q: $\bigwedge U.\ Q\ U \Longrightarrow False$
    and PQ: $\bigwedge U.\ [\![P\ (f\ U);\ \neg\ Q\ (g\ U)]\!] \Longrightarrow False$
  have *cl4*: $\bigwedge U.\ \neg\ Q\ (g\ U) \Longrightarrow False$
    by (*rule P* [*binary 0 PQ 0*])
  show *False*
    by (*rule Q* [*binary 0 cl4 0*])
qed

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

lemma [*intro*!]: $p \cap p \rightarrow q \subseteq q$ **by** (*unfold imp-def*) *auto*
lemma [*intro*!]: $p \subseteq (q \rightarrow p)$ **by** (*unfold imp-def*) *rule*

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

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

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
  $EX :: 'a\ ctl \Rightarrow 'a\ ctl$ (EX - [*80*] *90*) EX $p \equiv \{s.\ \exists s'.\ (s, s') \in \mathcal{M} \wedge s' \in p\}$
  $EF :: 'a\ ctl \Rightarrow 'a\ ctl$ (EF - [*80*] *90*) EF $p \equiv lfp\ (\lambda s.\ p \cup \text{EX}\ s)$
  $EG :: 'a\ ctl \Rightarrow 'a\ ctl$ (EG - [*80*] *90*) EG $p \equiv gfp\ (\lambda s.\ p \cap \text{EX}\ s)$

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

constdefs
  $AX :: 'a\ ctl \Rightarrow 'a\ ctl$ (AX - [*80*] *90*) AX $p \equiv -$ EX $- p$
  $AF :: 'a\ ctl \Rightarrow 'a\ ctl$ (AF - [*80*] *90*) AF $p \equiv -$ EG $- p$
  $AG :: 'a\ ctl \Rightarrow 'a\ ctl$ (AG - [*80*] *90*) AG $p \equiv -$ 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*: $lfp\ f = - gfp\ (\lambda s\ .\ - (f\ (- s)))$
proof
  show $lfp\ f \subseteq - gfp\ (\lambda s.\ - f\ (- s))$
  proof
    fix x **assume** l: $x \in lfp\ f$
    show $x \in - gfp\ (\lambda s.\ - f\ (- s))$
    proof
      assume $x \in gfp\ (\lambda s.\ - f\ (- s))$
      then obtain u **where** $x \in u$ **and** $u \subseteq - f\ (- u)$ **by** (*unfold gfp-def*) *auto*

then have $f\ (-\ u) \subseteq -\ u$ **by** *auto*
then have $lfp\ f \subseteq -\ u$ **by** (*rule lfp-lowerbound*)
from l **and** *this* **have** $x \notin u$ **by** *auto*
then show *False* **by** *contradiction*
qed
qed
show $-\ gfp\ (\lambda s.\ -\ f\ (-\ s)) \subseteq lfp\ f$
proof (*rule lfp-greatest*)
fix u **assume** $f\ u \subseteq u$
then have $-\ u \subseteq -\ f\ u$ **by** *auto*
then have $-\ u \subseteq -\ f\ (-\ (-\ u))$ **by** *simp*
then have $-\ u \subseteq gfp\ (\lambda s.\ -\ f\ (-\ s))$ **by** (*rule gfp-upperbound*)
then show $-\ gfp\ (\lambda s.\ -\ f\ (-\ s)) \subseteq u$ **by** *auto*
qed
qed

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

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

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

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

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

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

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

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

lemma *AG-fp*: AG $p = p \cap \mathrm{AX\ AG}\ p$
proof –

have *mono* ($\lambda s.\ p \cap \mathrm{AX}\ s$) **by** *rule* (*auto simp add*: *AX-def EX-def*)
then show *?thesis* **by** (*simp only*: *AG-gfp*) (*rule gfp-unfold*)
qed

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

lemma *AG-fp-1*: $\mathrm{AG}\ p \subseteq p$
proof –
note *AG-fp* **also have** $p \cap \mathrm{AX}\ \mathrm{AG}\ p \subseteq p$ **by** *auto*
finally show *?thesis* **.**
qed

lemma *AG-fp-2*: $\mathrm{AG}\ p \subseteq \mathrm{AX}\ \mathrm{AG}\ p$
proof –
note *AG-fp* **also have** $p \cap \mathrm{AX}\ \mathrm{AG}\ p \subseteq \mathrm{AX}\ \mathrm{AG}\ p$ **by** *auto*
finally show *?thesis* **.**
qed

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

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

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*: $\mathrm{AX}\ (p \cap q) = \mathrm{AX}\ p \cap \mathrm{AX}\ q$ **by** *auto*
lemma *AX-mono*: $p \subseteq q \Longrightarrow \mathrm{AX}\ p \subseteq \mathrm{AX}\ q$ **by** *auto*
lemma *AG-mono*: $p \subseteq q \Longrightarrow \mathrm{AG}\ p \subseteq \mathrm{AG}\ q$
by (*simp only*: *AG-gfp*, *rule gfp-mono*) *auto*

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

lemma *AG-AX*: $\mathrm{AG}\ p \subseteq \mathrm{AX}\ p$
proof –
have $\mathrm{AG}\ p \subseteq \mathrm{AX}\ \mathrm{AG}\ p$ **by** (*rule AG-fp-2*)
also have $\mathrm{AG}\ p \subseteq p$ **by** (*rule AG-fp-1*) **moreover note** *AX-mono*
finally show *?thesis* **.**
qed

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

```
lemma AG-AG: AG AG p = AG p
proof
  show AG AG p ⊆ AG p by (rule AG-fp-1)
next
  show AG p ⊆ AG AG p
  proof (rule AG-I)
    have AG p ⊆ AG p ..
    moreover have AG p ⊆ AX AG p by (rule AG-fp-2)
    ultimately show AG p ⊆ AG p ∩ AX AG p ..
  qed
qed
```

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

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

```
    show AG p ⊆ AG (p → AX p)
    proof -
      have AG p = AG AG p by (simp only: AG-AG)
      also have AG p ⊆ AX p by (rule AG-AX) moreover note AG-mono
      also have AX p ⊆ (p → AX p) .. moreover note AG-mono
      finally show ?thesis .
    qed
  qed
qed
```

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

end
```

22 Meson test cases

```
theory mesontest2 imports Main begin

end
```

23 Some examples for Presburger Arithmetic

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

theorem $(\forall (y::int).\ 3\ dvd\ y) \Longrightarrow \forall (x::int).\ b < x \longrightarrow a \leq x$
  by *presburger*

theorem !! $(y::int)\ (z::int)\ (n::int).\ 3\ dvd\ z \Longrightarrow 2\ dvd\ (y::int) \Longrightarrow$
  $(\exists (x::int).\ \ 2{*}x = \ y)\ \&\ (\exists (k::int).\ 3{*}k = z)$
  by *presburger*

theorem !! $(y::int)\ (z::int)\ n.\ Suc(n::nat) < 6 \Longrightarrow\ \ 3\ dvd\ z \Longrightarrow$
  $2\ dvd\ (y::int) \Longrightarrow (\exists (x::int).\ \ 2{*}x = \ y)\ \&\ (\exists (k::int).\ 3{*}k = z)$
  by *presburger*

theorem $\forall (x::nat).\ \exists (y::nat).\ (0::nat) \leq 5 \longrightarrow y = 5 + x$
  by *presburger*

Very slow: about 55 seconds on a 1.8GHz machine.

theorem $\forall (x::nat).\ \exists (y::nat).\ y = 5 + x \mid x\ div\ 6 + 1{=}\ 2$
  by *presburger*

theorem $\exists (x::int).\ 0 < x$
  by *presburger*

theorem $\forall (x::int)\ y.\ x < y \longrightarrow 2 * x + 1 < 2 * y$
  by *presburger*

theorem $\forall (x::int)\ y.\ 2 * x + 1 \neq 2 * y$
  by *presburger*

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

theorem ~ $(\exists (x::int)\ (y::int)\ (z::int).\ 4{*}x + (-6::int){*}y = 1)$
  by *presburger*

theorem $\forall (x::int).\ b < x \longrightarrow a \leq x$
  apply (*presburger* (*no-quantify*))
  oops

theorem ~ $(\exists (x::int).\ False)$
  by *presburger*

theorem $\forall (x::int).\ (a::int) < 3 * x \longrightarrow b < 3 * x$
  apply (*presburger* (*no-quantify*))
  oops

theorem $\forall (x::int).\ (2\ dvd\ x) \longrightarrow (\exists (y::int).\ x = 2{*}y)$

by *presburger*

theorem $\forall (x::int).\ (2\ dvd\ x) \ {-}{-}{>}\ (\exists (y::int).\ x = 2{*}y)$
by *presburger*

theorem $\forall (x::int).\ (2\ dvd\ x) = (\exists (y::int).\ x = 2{*}y)$
by *presburger*

theorem $\forall (x::int).\ ((2\ dvd\ x) = (\forall (y::int).\ x \neq 2{*}y + 1))$
by *presburger*

theorem $\sim (\forall (x::int).$
$((2\ dvd\ x) = (\forall (y::int).\ x \neq 2{*}y{+}1) \mid$
$(\exists (q::int)\ (u::int)\ i.\ 3{*}i + 2{*}q - u < 17)$
${-}{-}{>}\ 0 < x \mid ((\sim 3\ dvd\ x)\ \&(x + 8 = 0))))$
by *presburger*

theorem $\sim (\forall (i::int).\ 4 \leq i\ {-}{-}{>}\ (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
by *presburger*

theorem $\forall (i::int).\ 8 \leq i\ {-}{-}{>}\ (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
by *presburger*

theorem $\exists (j::int).\ \forall i.\ j \leq i\ {-}{-}{>}\ (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i)$
by *presburger*

theorem $\sim (\forall j\ (i::int).\ j \leq i\ {-}{-}{>}\ (\exists x\ y.\ 0 \leq x\ \&\ 0 \leq y\ \&\ 3 * x + 5 * y = i))$
by *presburger*

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

theorem $(\exists m::nat.\ n = 2 * m)\ {-}{-}{>}\ (n + 1)\ div\ 2 = n\ div\ 2$
by *presburger*

theorem $(\exists m::int.\ n = 2 * m)\ {-}{-}{>}\ (n + 1)\ div\ 2 = n\ div\ 2$
by *presburger*

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* $\Rightarrow$ *intterm* $\Rightarrow$ *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* $\Rightarrow$ *QF* $\Rightarrow$ *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* $\leq$ *I-intterm ats it2*)
qinterp ats (*Ge it1 it2*) = (*I-intterm ats it1* $\geq$ *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*) = ($\neg$(*qinterp ats p*))
qinterp ats (*And p q*) = (*qinterp ats p* $\wedge$ *qinterp ats q*)
qinterp ats (*Or p q*) = (*qinterp ats p* $\vee$ *qinterp ats q*)
qinterp ats (*Imp p q*) = (*qinterp ats p* $\longrightarrow$ *qinterp ats q*)

qinterp ats (*Equ p q*) = (*qinterp ats p* = *qinterp ats q*)
qinterp ats (*QAll p*) = ($\forall$ *x*. *qinterp* (*x#ats*) *p*)
qinterp ats (*QEx p*) = ($\exists$ *x*. *qinterp* (*x#ats*) *p*)

consts *lift-bin*:: ($'a \Rightarrow 'a \Rightarrow 'b$) $\times$ $'a$ *option* $\times$ $'a$ *option* $\Rightarrow$ $'b$ *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*)
using *ls*
by (*cases x*, *auto*) (*cases y*, *auto*)+

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* (λ(*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'
  using *qeqf*
proof (*induct* p)
  case (*Lt* a b)
  have *qelim* (*qe*, *Lt* a b) = *Some* (*Lt* a b) **by** *simp*
  moreover have *qelim* (*qe*,*Lt* a b) = *Some* p' **.**
  ultimately have p' = *Lt* a b **by** *simp*
  moreover have *isqfree* (*Lt* a b) **by** *simp*
  ultimately
  show *?case* **by** *simp*
next
  case (*Gt* a b)
  have *qelim* (*qe*, *Gt* a b) = *Some* (*Gt* a b) **by** *simp*
  moreover have *qelim* (*qe*,*Gt* a b) = *Some* p' **.**
  ultimately have p' = *Gt* a b **by** *simp*
  moreover have *isqfree* (*Gt* a b) **by** *simp*
  ultimately
  show *?case* **by** *simp*
next
  case (*Le* a b)
  have *qelim* (*qe*, *Le* a b) = *Some* (*Le* a b) **by** *simp*
  moreover have *qelim* (*qe*,*Le* a b) = *Some* p' **.**
  ultimately have p' = *Le* a b **by** *simp*
  moreover have *isqfree* (*Le* a b) **by** *simp*
  ultimately
  show *?case* **by** *simp*
next
  case (*Ge* a b)
  have *qelim* (*qe*, *Ge* a b) = *Some* (*Ge* a b) **by** *simp*
  moreover have *qelim* (*qe*,*Ge* a b) = *Some* p' **.**
  ultimately have p' = *Ge* a b **by** *simp*
  moreover have *isqfree* (*Ge* a b) **by** *simp*
  ultimately
  show *?case* **by** *simp*
next
  case (*Eq* a b)
  have *qelim* (*qe*, *Eq* a b) = *Some* (*Eq* a b) **by** *simp*
  moreover have *qelim* (*qe*,*Eq* a b) = *Some* p' **.**
  ultimately have p' = *Eq* a b **by** *simp*
  moreover have *isqfree* (*Eq* a b) **by** *simp*
  ultimately
  show *?case* **by** *simp*
next
  case (*Divides* a b)

have *qelim* (*qe*, *Divides a b*) = *Some* (*Divides a b*) **by** *simp*
moreover have *qelim* (*qe*,*Divides a b*) = *Some p′* **.**
ultimately have *p′* = *Divides a b* **by** *simp*
moreover have *isqfree* (*Divides a b*) **by** *simp*
ultimately
show *?case* **by** *simp*
next
case *T*
have *qelim*(*qe*,*T*) = *Some T* **by** *simp*
moreover have *qelim*(*qe*,*T*) = *Some p′* **.**
ultimately have *p′* = *T* **by** *simp*
moreover have *isqfree T* **by** *simp*
ultimately show *?case* **by** *simp*
next
case *F*
have *qelim*(*qe*,*F*) = *Some F* **by** *simp*
moreover have *qelim*(*qe*,*F*) = *Some p′* **.**
ultimately have *p′* = *F* **by** *simp*
moreover have *isqfree F* **by** *simp*
ultimately show *?case* **by** *simp*
next
case (*NOT p*)
from *NOT.prems* **have** ∃ *p1*. *qelim*(*qe*,*p*) = *Some p1*
by (*cases qelim*(*qe*,*p*)) *simp-all*
then obtain *p1* **where** *p1-def*: *qelim*(*qe*,*p*) = *Some p1* **by** *blast*
from *NOT.prems* **have** $\bigwedge$*q q′*. ⟦*isqfree q*; *qe q* = *Some q′*⟧ ⟹ *isqfree q′*
by *blast*
with *NOT.hyps p1-def* **have** *p1qf*: *isqfree p1* **by** *blast*
then have *p′* = *NOT p1* **using** *NOT.prems p1-def*
by (*cases qelim*(*qe*,*NOT p*)) *simp-all*
then show *?case* **using** *p1qf* **by** *simp*
next
case (*And p q*)
from *And.prems* **have** *p1q1*: (∃ *p1*. *qelim*(*qe*,*p*) = *Some p1*) ∧
(∃ *q1*. *qelim*(*qe*,*q*) = *Some q1*) **using** *lift-bin-Some*[**where** *c*=*And*] **by** *simp*
from *p1q1* **obtain** *p1* **and** *q1*
where *p1-def*: *qelim*(*qe*,*p*) = *Some p1*
and *q1-def*: *qelim*(*qe*,*q*) = *Some q1* **by** *blast*
from *prems* **have** *qf1*:*isqfree p1*
using *p1-def* **by** *blast*
from *prems* **have** *qf2*:*isqfree q1*
using *q1-def* **by** *blast*
from *And.prems* **have** *qelim*(*qe*,*And p q*) = *Some p′* **by** *blast*
then have *p′* = *And p1 q1* **using** *p1-def q1-def* **by** *simp*
then
show *?case* **using** *qf1 qf2* **by** *simp*
next
case (*Or p q*)
from *Or.prems* **have** *p1q1*: (∃ *p1*. *qelim*(*qe*,*p*) = *Some p1*) ∧

$(\exists\, q1.\ qelim(qe,q) = Some\ q1)$ **using** *lift-bin-Some*[**where** $c{=}Or$] **by** *simp*
from *p1q1* **obtain** *p1* **and** *q1*
where *p1-def*: $qelim(qe,p) = Some\ p1$
and *q1-def*: $qelim(qe,q) = Some\ q1$ **by** *blast*
from *prems* **have** *qf1*:*isqfree p1*
using *p1-def* **by** *blast*
from *prems* **have** *qf2*:*isqfree q1*
using *q1-def* **by** *blast*
from *Or.prems* **have** $qelim(qe,Or\ p\ q) = Some\ p'$ **by** *blast*
then have $p' = Or\ p1\ q1$ **using** *p1-def q1-def* **by** *simp*
then
show *?case* **using** *qf1 qf2* **by** *simp*
next
case (*Imp p q*)
from *Imp.prems* **have** *p1q1*: $(\exists\, p1.\ qelim(qe,p) = Some\ p1) \wedge$
$(\exists\, q1.\ qelim(qe,q) = Some\ q1)$ **using** *lift-bin-Some*[**where** $c{=}Imp$] **by** *simp*
from *p1q1* **obtain** *p1* **and** *q1*
where *p1-def*: $qelim(qe,p) = Some\ p1$
and *q1-def*: $qelim(qe,q) = Some\ q1$ **by** *blast*
from *prems* **have** *qf1*:*isqfree p1*
using *p1-def* **by** *blast*
from *prems* **have** *qf2*:*isqfree q1*
using *q1-def* **by** *blast*
from *Imp.prems* **have** $qelim(qe,Imp\ p\ q) = Some\ p'$ **by** *blast*
then have $p' = Imp\ p1\ q1$ **using** *p1-def q1-def* **by** *simp*
then
show *?case* **using** *qf1 qf2* **by** *simp*
next
case (*Equ p q*)
from *Equ.prems* **have** *p1q1*: $(\exists\, p1.\ qelim(qe,p) = Some\ p1) \wedge$
$(\exists\, q1.\ qelim(qe,q) = Some\ q1)$ **using** *lift-bin-Some*[**where** $c{=}Equ$] **by** *simp*
from *p1q1* **obtain** *p1* **and** *q1*
where *p1-def*: $qelim(qe,p) = Some\ p1$
and *q1-def*: $qelim(qe,q) = Some\ q1$ **by** *blast*
from *prems* **have** *qf1*:*isqfree p1*
using *p1-def* **by** *blast*
from *prems* **have** *qf2*:*isqfree q1*
using *q1-def* **by** *blast*
from *Equ.prems* **have** $qelim(qe,Equ\ p\ q) = Some\ p'$ **by** *blast*
then have $p' = Equ\ p1\ q1$ **using** *p1-def q1-def* **by** *simp*
then
show *?case* **using** *qf1 qf2* **by** *simp*
next
case (*QEx p*)
from *QEx.prems* **have** $\exists\ p1.\ qelim(qe,p) = Some\ p1$
by (*cases* $qelim(qe,p)$) *simp-all*
then obtain *p1* **where** *p1-def*: $qelim(qe,p) = Some\ p1$ **by** *blast*
from *QEx.prems* **have** $\bigwedge q\ q'.\ [\![isqfree\ q;\ qe\ q = Some\ q']\!] \Longrightarrow isqfree\ q'$
by *blast*

with *QEx.hyps p1-def* **have** *p1qf*: *isqfree p1* **by** *blast*
from *QEx.prems* **have** *qe p1 = Some p′* **using** *p1-def* **by** *simp*
with *QEx.prems* **show** *?case* **using** *p1qf*
by *simp*
next
case (*QAll p*)
from *QAll.prems*
have ∃ *p1*. *lift-qe qe* (*lift-un NOT* (*qelim* (*qe* ,*p*))) = *Some p1*
by (*cases lift-qe qe* (*lift-un NOT* (*qelim* (*qe* ,*p*)))) *simp-all*
then obtain *p1* **where**
p1-def:*lift-qe qe* (*lift-un NOT* (*qelim* (*qe* ,*p*))) = *Some p1* **by** *blast*
then have ∃ *p2*. *lift-un NOT* (*qelim* (*qe* ,*p*)) = *Some p2*
by (*cases qelim* (*qe* ,*p*)) *simp-all*
then obtain *p2*
where *p2-def*:*lift-un NOT* (*qelim* (*qe* ,*p*)) = *Some p2* **by** *blast*
then have ∃ *p3*. *qelim*(*qe*,*p*) = *Some p3* **by** (*cases qelim*(*qe*,*p*)) *simp-all*
then obtain *p3* **where** *p3-def*: *qelim*(*qe*,*p*) = *Some p3* **by** *blast*
with *prems* **have** *qf3*: *isqfree p3* **by** *blast*
have *p2-def2*: *p2* = *NOT p3* **using** *p2-def p3-def* **by** *simp*
then have *qf2*: *isqfree p2* **using** *qf3* **by** *simp*
have *p1-edf2*: *qe p2* = *Some p1* **using** *p1-def p2-def* **by** *simp*
with *QAll.prems* **have** *qf1*: *isqfree p1* **using** *qf2* **by** *blast*
from *QAll.prems* **have** *p′* = *NOT p1* **using** *p1-def* **by** *simp*
with *qf1* **show** *?case* **by** *simp*
qed

lemma *qelim-corr*:
assumes *qecorr*: (⋀ *q q′ ats*. ⟦*isqfree q* ; *qe q* = *Some q′*⟧ ⟹ (*qinterp ats* (*QEx q*)) = (*qinterp ats q′*))
and *qeqf*: (⋀ *q q′*. ⟦*isqfree q* ; *qe q* = *Some q′*⟧ ⟹ *isqfree q′*)
shows *qff*:⋀ *p′ ats*. *qelim* (*qe*, *p*) = *Some p′* ⟹ (*qinterp ats p* = *qinterp ats p′*) (**is** ⋀ *p′ ats*. *?Qe p p′* ⟹ (*?F ats p* = *?F ats p′*))
using *qeqf qecorr*
proof (*induct p*)
case (*NOT f*)
from *NOT.prems* **have** ∃*f′*. *?Qe f f′* **by** (*cases qelim*(*qe*,*f*)) *simp-all*
then obtain *f′* **where** *df′*: *?Qe f f′* **by** *blast*
with *prems* **have** *feqf′*: *?F ats f* = *?F ats f′* **by** *blast*
from *NOT.prems df′* **have** *p′* = *NOT f′* **by** *simp*
with *feqf′* **show** *?case* **by** *simp*

next
case (*And f g*)
from *And.prems* **have** *f1g1*: (∃*f1*. *qelim*(*qe*,*f*) = *Some f1*) ∧
(∃*g1*. *qelim*(*qe*,*g*) = *Some g1*) **using** *lift-bin-Some*[**where** *c*=*And*] **by** *simp*
from *f1g1* **obtain** *f1* **and** *g1*
where *f1-def*: *qelim*(*qe*, *f*) = *Some f1*
and *g1-def*: *qelim*(*qe*,*g*) = *Some g1* **by** *blast*

```
  from prems f1-def have feqf1: ?F ats f = ?F ats f1 by blast
  from prems g1-def have geqg1: ?F ats g = ?F ats g1 by blast
  from And.prems f1-def g1-def have p' = And f1 g1 by simp
  with feqf1 geqg1 show ?case by simp

next
  case (Or f g)
  from Or.prems have f1g1: (∃f1. qelim(qe,f) = Some f1) ∧
    (∃g1. qelim(qe,g) = Some g1) using lift-bin-Some[where c=Or] by simp
  from f1g1 obtain f1 and g1
    where f1-def: qelim(qe, f) = Some f1
    and g1-def: qelim(qe,g) = Some g1 by blast
  from prems f1-def have feqf1: ?F ats f = ?F ats  f1 by blast
  from prems g1-def have geqg1: ?F ats g = ?F ats g1 by blast
  from Or.prems f1-def g1-def have p' = Or f1 g1 by simp
  with feqf1 geqg1 show ?case by simp
next
  case (Imp f g)
  from Imp.prems have f1g1: (∃f1. qelim(qe,f) = Some f1) ∧
    (∃g1. qelim(qe,g) = Some g1) using lift-bin-Some[where c=Imp] by simp
  from f1g1 obtain f1 and g1
    where f1-def: qelim(qe, f) = Some f1
    and g1-def: qelim(qe,g) = Some g1 by blast
  from prems f1-def have feqf1: ?F ats f = ?F ats f1 by blast
  from prems g1-def have geqg1: ?F ats g = ?F ats g1 by blast
  from Imp.prems f1-def g1-def have p' = Imp f1 g1 by simp
  with feqf1 geqg1 show ?case by simp
next
  case (Equ f g)
  from Equ.prems have f1g1: (∃f1. qelim(qe,f) = Some f1) ∧
    (∃g1. qelim(qe,g) = Some g1) using lift-bin-Some[where c=Equ] by simp
  from f1g1 obtain f1 and g1
    where f1-def: qelim(qe, f) = Some f1
    and g1-def: qelim(qe,g) = Some g1 by blast
  from prems f1-def have feqf1: ?F ats f = ?F ats f1 by blast
  from prems g1-def have geqg1: ?F ats g = ?F ats g1 by blast
  from Equ.prems f1-def g1-def have p' = Equ f1 g1 by simp
  with feqf1 geqg1 show ?case by simp
next
  case (QEx f)
    from QEx.prems have ∃ f1. ?Qe f f1
    by  (cases qelim(qe,f)) simp-all
  then obtain f1 where f1-def: qelim(qe,f) = Some f1 by blast
  from prems have qf1:isqfree f1 using qelim-qfree by blast
  from prems have feqf1: ∀ ats. qinterp ats f = qinterp ats f1
    using f1-def qf1 by blast
  then  have ?F ats (QEx f) = ?F ats (QEx f1)
    by simp
  from prems have qelim (qe,QEx f) = Some p' by blast
```

```
  then  have ∃ f'. qe f1 = Some f' using f1-def by simp
  then obtain f' where fdef': qe f1 = Some f' by blast
  with prems have exf1: ?F ats (QEx f1) = ?F ats f' using qf1 by blast
  have fp: ?Qe (QEx f) f' using f1-def fdef' by simp
  from prems have ?Qe (QEx f) p' by blast
  then have p' = f' using fp by simp
  then show ?case using feqf1 exf1 by simp
next
  case (QAll f)
  from QAll.prems
  have ∃ f0. lift-un NOT (lift-qe qe (lift-un NOT (qelim (qe ,f)))) =
    Some f0
    by (cases lift-un NOT (lift-qe qe (lift-un NOT (qelim (qe ,f)))))
      simp-all
  then obtain f0
    where f0-def: lift-un NOT (lift-qe qe (lift-un NOT (qelim (qe ,f)))) =
    Some f0 by blast
  then have ∃ f1. lift-qe qe (lift-un NOT (qelim (qe ,f))) = Some f1
    by (cases lift-qe qe (lift-un NOT (qelim (qe ,f)))) simp-all
  then obtain f1 where
    f1-def:lift-qe qe (lift-un NOT (qelim (qe ,f))) = Some f1 by blast
  then have ∃ f2. lift-un NOT (qelim (qe ,f)) = Some f2
    by (cases qelim (qe ,f)) simp-all
  then obtain f2
    where f2-def:lift-un NOT (qelim (qe ,f)) = Some f2 by blast
  then have ∃ f3. qelim(qe,f) = Some f3 by (cases qelim(qe,f)) simp-all
  then obtain f3 where f3-def: qelim(qe,f) = Some f3 by blast
  from prems have qf3:isqfree f3 using qelim-qfree by blast
  from prems have feqf3: ∀ ats. qinterp ats f = qinterp ats f3
    using f3-def qf3 by blast
  have f23:f2 = NOT f3 using f2-def f3-def by simp
  then have feqf2: ∀ ats. qinterp ats f = qinterp ats (NOT f2)
    using feqf3 by simp
  have qf2: isqfree f2 using f23 qf3 by simp
  have qe f2 = Some f1 using f1-def f2-def f23 by simp
  with prems have exf2eqf1: ?F ats (QEx f2) = ?F ats f1 using qf2 by blast
  have f0 = NOT f1 using f0-def f1-def by simp
  then have f0eqf1: ?F ats f0 = ?F ats (NOT f1) by simp
  from prems have qelim (qe, QAll f) = Some p' by blast
  then have f0eqp': p' = f0 using f0-def by simp
  have ?F ats (QAll f) = (∀x. ?F (x#ats) f) by simp
  also have ... = (¬ (∃ x. ?F (x#ats) (NOT f))) by simp
  also have ... = (¬ (∃ x. ?F (x#ats) (NOT (NOT f2)))) using feqf2
    by auto
  also have ... = (¬ (∃ x. ?F (x#ats) f2)) by simp
  also have ... = (¬ (?F ats f1)) using exf2eqf1 by simp
  finally show ?case using f0eqp' f0eqf1 by simp
qed simp-all
```

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*
apply (*induct-tac q*)
apply(*auto*)
done

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* ⇒ *bool*
recdef *isnnf measure* (*λp. 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* ∧ *isnnf q*)
isnnf (*Or p q*) = (*isnnf p* ∧ *isnnf q*)
isnnf (*NOT* (*Divides d t*)) = *True*
isnnf (*NOT* (*Eq it1 it2*)) = *True*
isnnf p = *False*

lemma *nnf-corr*: *isqfree p* ⟹ *qinterp ats p* = *qinterp ats* (*nnf p*)
by (*induct p rule*: *nnf.induct*,*simp-all*)
(*arith*, *arith*, *arith*, *arith*, *arith*, *arith*, *arith*, *arith*, *arith*, *blast*)

lemma *nnf-isnnf* : *isqfree p* ⟹ *isnnf* (*nnf p*)
by (*induct p rule*: *nnf.induct*, *auto*)

lemma *nnf-isqfree*: *isnnf p* ⟹ *isqfree p*
by (*induct p rule*: *isnnf.induct*) *auto*

lemma *nnf-qfree*: *isqfree p* ⟹ *isqfree*(*nnf p*)
using *nnf-isqfree nnf-isnnf* **by** *simp*

consts *islinintterm* :: *intterm* ⇒ *bool*
recdef *islinintterm measure size*
islinintterm (*Cst i*) = *True*
islinintterm (*Add* (*Mult* (*Cst i*) (*Var n*)) (*Cst i'*)) = (*i* ≠ *0*)
islinintterm (*Add* (*Mult* (*Cst i*) (*Var n*)) (*Add* (*Mult* (*Cst i'*) (*Var n'*)) *r*)) = (*i* ≠ *0* ∧ *i'* ≠ *0* ∧ *n* < *n'* ∧ *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*
using *lr*
by (*induct* *r* *rule*: *islinintterm.induct*) *auto*

lemma *islinintterm-cnz*:
  assumes *lr*: *islinintterm* (*Add* (*Mult* (*Cst* *i*) (*Var* *n*)) *r*)
  shows $i \neq 0$
using *lr*
by (*induct* *r* *rule*: *islinintterm.induct*) *auto*

lemma *islinintermc0r*: *islinintterm* (*Add* (*Mult* (*Cst* *c*) (*Var* *n*)) *r*) $\Longrightarrow$ ($c \neq 0$ $\wedge$ *islinintterm* *r*)
by (*induct* *r* *rule*: *islinintterm.induct*, *simp-all*)

consts *islintn* :: (*nat* $\times$ *intterm*) $\Rightarrow$ *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* $\Rightarrow$ *bool*
  islint *t* $\equiv$ *islintn*(*0*,*t*)

lemma *islinintterm-eq-islint*: *islinintterm* *t* = *islint* *t*
  using *islint-def*
by (*induct* *t* *rule*: *islinintterm.induct*) *auto*

lemma *islintn-mon*:
  assumes *lin*: *islintn* (*n*,*t*)
  and *mgen*: $m \leq n$
  shows *islintn*(*m*,*t*)
  using *lin* *mgen*
by (*induct* *t* *rule*: *islintn.induct*) *auto*

lemma *islintn-subt*:
  assumes *lint*: *islintn*(*n*,*Add* (*Mult* (*Cst* *i*) (*Var* *m*)) *r*)
  shows *islintn* (*m*+*1*,*r*)
using *lint*
by *auto*

lemma *nth-pos*: $0 < n \longrightarrow (x\#xs) \mathbin{!} n = (y\#xs) \mathbin{!} n$
using *Nat.gr0-conv-Suc*
by *clarsimp*

lemma *nth-pos2*: $0 < n \Longrightarrow (x\#xs) \mathbin{!} n = xs \mathbin{!} (n - 1)$
using *Nat.gr0-conv-Suc*
by *clarsimp*

lemma *intterm-novar0*:
  assumes *lin*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
  shows *I-intterm* (*x#ats*) *r* = *I-intterm* (*y#ats*) *r*
using *lin*
by (*induct r rule*: *islinintterm.induct*) (*simp-all add*: *nth-pos2*)

lemma *linterm-novar0*:
  assumes *lin*: *islintn* (*n*,*t*)
  and *npos*: $0 < n$
  shows *I-intterm* (*x#ats*) *t* = *I-intterm* (*y#ats*) *t*
using *lin npos*
by (*induct n t rule*: *islintn.induct*) (*simp-all add*: *nth-pos2*)

lemma *dvd-period*:
  assumes *advdd*: (*a*::*int*) *dvd d*
  shows (*a dvd* (*x* + *t*)) = (*a dvd* ((*x*+ *c*∗*d*) + *t*))
using *advdd*
proof−
  from *advdd* **have** ∀ *x*.∀ *k*. (((*a*::*int*) *dvd* (*x* + *t*)) = (*a dvd*
 (*x*+*k*∗*d* + *t*))) **by** (*rule dvd-modd-pinf*)
  then show *?thesis* **by** *simp*
qed

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*))
using *blin*
by (*induct n0 b rule*: *islintn.induct*) *auto*

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*)))
using *blin*
by (*induct n0 b rule*: *islintn.induct*) *auto*

lemma *lin-add-corrh*: $\bigwedge$ *n01 n02*. $[\![$ *islintn* (*n01*,*a*) ; *islintn* (*n02*,*b*)$]\!]$
  $\Longrightarrow$ *I-intterm ats* (*lin-add*(*a*,*b*)) = *I-intterm ats* (*Add a b*)
proof(*induct a b rule*: *lin-add.induct*)
  case (*58 i n r j m s*)
  have $(n = m \wedge i+j = 0) \vee (n = m \wedge i+j \neq 0) \vee n < m \vee m < n$ **by** *arith*
  moreover
  {**assume** $n=m \wedge i+j=0$ **hence** *?case* **using** *prems* **by** (*auto simp add*: *sym*[*OF zadd-zmult-distrib*]) }
  moreover
  {**assume** $n=m \wedge i+j \neq 0$ **hence** *?case* **using** *prems* **by** (*auto simp add*: *Let-def zadd-zmult-distrib*)}
  moreover
  {**assume** $n < m$ **hence** *?case* **using** *prems* **by** *auto* }
  moreover
  {**assume** $n > m$ **hence** *?case* **using** *prems* **by** *auto* }
  ultimately show *?case* **by** *blast*
qed (*auto simp add*: *lin-add-cst-corr lin-add-cst-corr2 Let-def*)

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*))
using *lina linb islinintterm-eq-islint islint-def lin-add-corrh*
by *blast*

lemma *lin-add-cst-lint*:
  assumes *lin*: *islintn* (*n0*,*b*)
  shows *islintn* (*n0*, *lin-add* (*Cst i*, *b*))
using *lin*
by (*induct n0 b rule*: *islintn.induct*) *auto*

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

using *lin*
by (*induct n0 b rule*: *islintn.induct*) *auto*

lemma *lin-add-lint*: $\bigwedge$ *n0 n01 n02*. ⟦ *islintn* (*n01*,*a*) ; *islintn* (*n02*,*b*); *n0* $\leq$ *min n01 n02* ⟧
$\Longrightarrow$ *islintn* (*n0*, *lin-add* (*a*,*b*))
proof (*induct a b rule*: *lin-add.induct*)
case (*58 i n r j m s*)
have ($n = m \wedge i + j = 0$) $\vee$ ($n = m \wedge i+j \neq 0$) $\vee$ $n < m \vee m < n$ **by** *arith*
moreover
{ assume $n = m$
and $i+j = 0$
hence *?case* **using** *58 islintn-mon*[**where** *m* = *n01* **and** *n* = *Suc m*]
islintn-mon[**where** *m* = *n02* **and** *n* = *Suc m*] **by** *auto* **}**
moreover
{ assume $n = m$
and $i+j \neq 0$
hence *?case* **using** *58 islintn-mon*[**where** *m* = *n01* **and** *n* = *Suc m*]
islintn-mon[**where** *m* = *n02* **and** *n* = *Suc m*] **by** (*auto simp add*: *Let-def*) **}**
moreover
{ assume $n < m$ **hence** *?case* **using** *58* **by** *force* **}**
moreover
{ assume $m < n$
hence *?case* **using** *58*
apply (*auto simp add*: *Let-def*)
apply (*erule allE*[**where** *x* = *Suc m*])
by (*erule allE*[**where** *x* = *Suc m*]) *simp* **}**
ultimately show *?case* **by** *blast*
qed(*simp-all add*: *Let-def lin-add-cst-lint lin-add-cst-lint2*)

lemma *lin-add-lin*:
assumes *lina*: *islinintterm a*
and *linb*: *islinintterm b*
shows *islinintterm* (*lin-add* (*a*,*b*))
using *islinintterm-eq-islint islint-def lin-add-lint lina linb* **by** *auto*

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−
have *a*∗(*b*+*c*) = (*b*+*c*)∗*a* **by** *simp*
moreover have (*b*+*c*)∗*a* = *b*∗*a* + *c*∗*a* **by** (*simp add*: *zadd-zmult-distrib*)

ultimately show *?thesis* **by** *simp*
qed

lemma *lin-mul-corr*:
  assumes *lint*: *islinintterm* *t*
  shows *I-intterm ats* (*lin-mul* (*c*,*t*)) = *I-intterm ats* (*Mult* (*Cst c*) *t*)
using *lint*
proof (*induct c t rule*: *lin-mul.induct*)
  case (*21 c c′ n r*)
  have *islinintterm* (*Add* (*Mult* (*Cst c′*) (*Var n*)) *r*) **.**
  then have *islinintterm r*
    by (*rule islinintterm-subt*[*of c′ n r*])
  then show *?case* **using** *21.hyps 21.prems* **by** *simp*
qed(*auto*)

lemma *lin-mul-lin*:
  assumes *lint*: *islinintterm t*
  shows *islinintterm* (*lin-mul*(*c*,*t*))
using *lint*
by (*induct t rule*: *islinintterm.induct*) *auto*

lemma *lin-mul0*:
  assumes *lint*: *islinintterm t*
  shows *lin-mul*(*0*,*t*) = *Cst 0*
  using *lint*
  by (*induct t rule*: *islinintterm.induct*) *auto*

lemma *lin-mul-lintn*:
  $\bigwedge$ *m*. *islintn*(*m*,*t*) $\Longrightarrow$ *islintn*(*m*,*lin-mul*(*l*,*t*))
  by (*induct l t rule*: *lin-mul.induct*) *simp-all*

constdefs *lin-neg* :: *intterm* $\Rightarrow$ *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*)
  using *lint lin-mul-corr*
  by (*simp add*: *lin-neg-def lin-mul-corr*)

lemma *lin-neg-lin*:
  assumes *lint*: *islinintterm* *t*
  shows *islinintterm* (*lin-neg t*)
using *lint*

by (*simp add*: *lin-mul-lin lin-neg-def*)

lemma *lin-neg-idemp*:
  assumes *lini*: *islinintterm i*
  shows *lin-neg* (*lin-neg i*) = *i*
using *lini*
by (*induct i rule*: *islinintterm.induct*) (*auto simp add*: *lin-neg-def*)

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*)
using *lina linb*
proof (*induct a b rule*: *lin-add.induct*)
  case (*58 c1 n1 r1 c2 n2 r2*)
  from *prems* **have** *lincnr1*:*islinintterm* (*Add* (*Mult* (*Cst c1*) (*Var n1*)) *r1*) **by** *simp*
  have *linr1*: *islinintterm r1* **by** (*rule islinintterm-subt*[*OF lincnr1*])
  from *prems* **have** *lincnr2*: *islinintterm* (*Add* (*Mult* (*Cst c2*) (*Var n2*)) *r2*) **by** *simp*
  have *linr2*: *islinintterm r2* **by** (*rule islinintterm-subt*[*OF lincnr2*])
  have $n1 = n2 \lor n1 < n2 \lor n1 > n2$ **by** *arith*
  show *?case* **using** *prems linr1 linr2* **by** (*simp-all add*: *lin-neg-def Let-def*)
next
  case (*59 c n r b*)
  from *prems* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var n*)) *r*) **by** *simp*
  have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
  show *?case* **using** *prems linr* **by** (*simp add*: *lin-neg-def Let-def*)
next
  case (*60 b c n r*)
  from *prems* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var n*)) *r*) **by** *simp*
  have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
  show *?case* **using** *prems linr* **by** (*simp add*: *lin-neg-def Let-def*)
qed (*simp-all add*: *lin-neg-def*)

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))
using lin
proof (induct t rule: linearize.induct)
  case (1 b) show ?case by simp
next
  case (2 n) show ?case by simp
next
  case (3 i) show ?case
    proof−
    have (linearize i = None) ∨ (∃li. linearize i = Some li) by auto
    moreover
    { assume linearize i = None with prems have ?thesis by auto}
    moreover
    { assume lini: ∃li. linearize i = Some li
      from lini obtain li where  linearize i = Some li by blast
      have linli: islinintterm li by (simp!)
      moreover have linearize (Neg i) = Some (lin-neg li) using prems by simp
     moreover from linli have islinintterm(lin-neg li) by (simp add: lin-neg-lin)
      ultimately have ?thesis by simp
    }
    ultimately show ?thesis by blast
  qed
next
  case (4 i j) show ?case
    proof−
     have (linearize i = None) ∨ ((∃ li. linearize i = Some li) ∧ linearize j =
None) ∨ ((∃ li. linearize i = Some li) ∧ (∃ lj. linearize j = Some lj)) by auto
    moreover
    {
      assume nlini: linearize i = None
      from nlini have linearize (Add i j) = None
       by (simp add: Let-def measure-def inv-image-def) then have ?thesis using
prems by auto}
    moreover
    { assume nlinj: linearize j = None
        and lini: ∃ li. linearize i = Some li
      from nlinj lini have linearize (Add i j) = None
```

by (*simp add*: *Let-def measure-def inv-image-def*, *auto*) **with** *prems* **have** *?thesis* **by** *auto*}
moreover
{ **assume** *lini*: $\exists$ *li*. *linearize i* = *Some li*
and *linj*: $\exists$ *lj*. *linearize j* = *Some lj*
from *lini* **obtain** *li* **where** *linearize i* = *Some li* **by** *blast*
have *linli*: *islinintterm li* **by** (*simp*!)
from *linj* **obtain** *lj* **where** *linearize j* = *Some lj* **by** *blast*
have *linlj*: *islinintterm lj* **by** (*simp*!)
moreover from *lini linj* **have** *linearize* (*Add i j*) = *Some* (*lin-add* (*li*,*lj*))
by (*simp add*: *measure-def inv-image-def*, *auto*!)
moreover from *linli linlj* **have** *islinintterm*(*lin-add* (*li*,*lj*)) **by** (*simp add*: *lin-add-lin*)
ultimately have *?thesis* **by** *simp* }
ultimately show *?thesis* **by** *blast*
qed
next
case (*5 i j*)**show** *?case*
proof−
have (*linearize i* = *None*) $\vee$ (($\exists$ *li*. *linearize i* = *Some li*) $\wedge$ *linearize j* = *None*) $\vee$ (($\exists$ *li*. *linearize i* = *Some li*) $\wedge$ ($\exists$ *lj*. *linearize j* = *Some lj*)) **by** *auto*
moreover
{
assume *nlini*: *linearize i* = *None*
from *nlini* **have** *linearize* (*Sub i j*) = *None* **by** (*simp add*: *Let-def measure-def inv-image-def*) **then have** *?thesis* **by** (*auto*!)
}
moreover
{
assume *lini*: $\exists$ *li*. *linearize i* = *Some li*
and *nlinj*: *linearize j* = *None*
from *nlinj lini* **have** *linearize* (*Sub i j*) = *None*
by (*simp add*: *Let-def measure-def inv-image-def*, *auto*) **then have** *?thesis* **by** (*auto*!)
}
moreover
{
assume *lini*: $\exists$ *li*. *linearize i* = *Some li*
and *linj*: $\exists$ *lj*. *linearize j* = *Some lj*
from *lini* **obtain** *li* **where** *linearize i* = *Some li* **by** *blast*
have *linli*: *islinintterm li* **by** (*simp*!)
from *linj* **obtain** *lj* **where** *linearize j* = *Some lj* **by** *blast*
have *linlj*: *islinintterm lj* **by** (*simp*!)
moreover from *lini linj* **have** *linearize* (*Sub i j*) = *Some* (*lin-add* (*li*,*lin-neg lj*))
by (*simp add*: *measure-def inv-image-def*, *auto*!)
moreover from *linli linlj* **have** *islinintterm*(*lin-add* (*li*,*lin-neg lj*)) **by** (*simp add*: *lin-add-lin lin-neg-lin*)
ultimately have *?thesis* **by** *simp*

}
ultimately show *?thesis* **by** *blast*
qed
next
case (*6 i j*)**show** *?case*
proof−
have *cses*: (*linearize i* = *None*) ∨
((∃ *li*. *linearize i* = *Some li*) ∧ *linearize j* = *None*) ∨
((∃ *li*. *linearize i* = *Some li*) ∧ (∃ *bj*. *linearize j* = *Some* (*Cst bj*)))
∨ ((∃ *bi*. *linearize i* = *Some* (*Cst bi*)) ∧ (∃ *lj*. *linearize j* = *Some lj*))
∨ ((∃ *li*. *linearize i* = *Some li* ∧ ¬ (∃ *bi*. *li* = *Cst bi*)) ∧ (∃ *lj*. *linearize j* = *Some lj* ∧ ¬ (∃ *bj*. *lj* = *Cst bj*))) **by** *auto*
moreover
{
assume *nlini*: *linearize i* = *None*
from *nlini* **have** *linearize* (*Mult i j*) = *None*
by (*simp add*: *Let-def measure-def inv-image-def*)
with *prems* **have** *?thesis* **by** *auto* }
moreover
{ **assume** *lini*: ∃ *li*. *linearize i* = *Some li*
and *nlinj*: *linearize j* = *None*
from *lini* **obtain** *li* **where** *linearize i* = *Some li* **by** *blast*
moreover from *nlinj lini* **have** *linearize* (*Mult i j*) = *None*
using *prems*
by (*cases li*) (*auto simp add*: *Let-def measure-def inv-image-def*)
with *prems* **have** *?thesis* **by** *auto*}
moreover
{ **assume** *lini*: ∃ *li*. *linearize i* = *Some li*
and *linj*: ∃ *bj*. *linearize j* = *Some* (*Cst bj*)
from *lini* **obtain** *li* **where** *li-def*: *linearize i* = *Some li* **by** *blast*
from *prems* **have** *linli*: *islinintterm li* **by** *simp*
moreover
from *linj* **obtain** *bj* **where** *bj-def*: *linearize j* = *Some* (*Cst bj*) **by** *blast*
have *linlj*: *islinintterm* (*Cst bj*) **by** *simp*
moreover from *lini linj prems*
have *linearize* (*Mult i j*) = *Some* (*lin-mul* (*bj*,*li*))
by (*cases li*) (*auto simp add*: *measure-def inv-image-def*)
moreover from *linli linlj* **have** *islinintterm*(*lin-mul* (*bj*,*li*)) **by** (*simp add*: *lin-mul-lin*)
ultimately have *?thesis* **by** *simp* }
moreover
{ **assume** *lini*: ∃ *bi*. *linearize i* = *Some* (*Cst bi*)
and *linj*: ∃ *lj*. *linearize j* = *Some lj*
from *lini* **obtain** *bi* **where** *linearize i* = *Some* (*Cst bi*) **by** *blast*
from *prems* **have** *linli*: *islinintterm* (*Cst bi*) **by** *simp*
moreover
from *linj* **obtain** *lj* **where** *linearize j* = *Some lj* **by** *blast*
from *prems* **have** *linlj*: *islinintterm lj* **by** *simp*
moreover from *lini linj prems* **have** *linearize* (*Mult i j*) = *Some* (*lin-mul*

$(bi,lj))$
by (*simp add*: *measure-def inv-image-def*)
moreover from *linli linlj* **have** *islinintterm*(*lin-mul* (*bi*,*lj*)) **by** (*simp add*: *lin-mul-lin*)
ultimately have *?thesis* **by** *simp* **}**
moreover
{ assume *linc*: $\exists$ *li*. *linearize i* = *Some li* $\wedge \neg$ ($\exists$ *bi*. *li* = *Cst bi*)
and *ljnc*: $\exists$ *lj*. *linearize j* = *Some lj* $\wedge \neg$ ($\exists$ *bj*. *lj* = *Cst bj*)
from *linc* **obtain** *li* **where** *linearize i* = *Some li* $\wedge \neg$ ($\exists$ *bi*. *li* = *Cst bi*) **by** *blast*
moreover
from *ljnc* **obtain** *lj* **where** *linearize j* = *Some lj* $\wedge \neg$ ($\exists$ *bj*. *lj* = *Cst bj*) **by** *blast*
ultimately have *linearize* (*Mult i j*) = *None*
by (*cases li*, *auto simp add*: *measure-def inv-image-def*) (*cases lj*, *auto*)+
with *prems* **have** *?thesis* **by** *simp* **}**
ultimately show *?thesis* **by** *blast*
qed
qed

lemma *linearize-linear*: $\bigwedge$ t'. *linearize t* = *Some* $t' \Longrightarrow$ *islinintterm* t'
proof−
fix t'
assume *lint*: *linearize t* = *Some* t'
from *lint* **have** *lt*: *linearize t* $\neq$ *None* **by** *auto*
then have *islinintterm* (*the* (*linearize t*)) **by** (*rule-tac linearize-linear1*[*OF lt*])
with *lint* **show** *islinintterm* t' **by** *simp*
qed

lemma *linearize-corr1*:
assumes *lin*: *linearize t* $\neq$ *None*
shows *I-intterm ats t* = *I-intterm ats* (*the* (*linearize t*))
using *lin*
proof (*induct t rule*: *linearize.induct*)
case (*3 i*) **show** *?case*
proof−
have (*linearize i* = *None*) $\vee$ ($\exists$*li*. *linearize i* = *Some li*) **by** *auto*
moreover
{
assume *linearize i* = *None*
have *?thesis* **using** *prems* **by** *simp*
}
moreover
{
assume *lini*: $\exists$*li*. *linearize i* = *Some li*
from *lini* **have** *lini2*: *linearize i* $\neq$ *None* **by** *simp*
from *lini* **obtain** *li* **where** *linearize i* = *Some li* **by** *blast*
from *lini2 lini* **have** *islinintterm* (*the* (*linearize i*))

by (*simp add*: *linearize-linear1*[*OF lini2*])
then have *linli*: *islinintterm li* using *prems* by *simp*
have *ieqli*: *I-intterm ats i* = *I-intterm ats li* using *prems* by *simp*
moreover have *linearize* (*Neg i*) = *Some* (*lin-neg li*) using *prems* by *simp*
moreover from *ieqli linli* have *I-intterm ats* (*Neg i*) = *I-intterm ats* (*lin-neg li*) by (*simp add*: *lin-neg-corr*[*OF linli*])
ultimately have *?thesis* using *prems* by (*simp add*: *lin-neg-corr*)
}
ultimately show *?thesis* by *blast*
qed
next
case (*4 i j*) show *?case*
proof−
have (*linearize i* = *None*) ∨ ((∃ *li*. *linearize i* = *Some li*) ∧ *linearize j* = *None*) ∨ ((∃ *li*. *linearize i* = *Some li*) ∧ (∃ *lj*. *linearize j* = *Some lj*)) by *auto*
moreover
{
assume *nlini*: *linearize i* = *None*
from *nlini* have *linearize* (*Add i j*) = *None* by (*simp add*: *Let-def measure-def inv-image-def*) then have *?thesis* using *prems* by *auto*
}
moreover
{
assume *nlinj*: *linearize j* = *None*
and *lini*: ∃ *li*. *linearize i* = *Some li*
from *nlinj lini* have *linearize* (*Add i j*) = *None*
by (*simp add*: *Let-def measure-def inv-image-def*, *auto*)
then have *?thesis* using *prems* by *auto*
}
moreover
{
assume *lini*: ∃ *li*. *linearize i* = *Some li*
and *linj*: ∃ *lj*. *linearize j* = *Some lj*
from *lini* have *lini2*: *linearize i* ≠ *None* by *simp*
from *linj* have *linj2*: *linearize j* ≠ *None* by *simp*
from *lini* obtain *li* where *linearize i* = *Some li* by *blast*
from *lini2* have *islinintterm* (*the* (*linearize i*)) by (*simp add*: *linearize-linear1*)
then have *linli*: *islinintterm li* using *prems* by *simp*
from *linj* obtain *lj* where *linearize j* = *Some lj* by *blast*
from *linj2* have *islinintterm* (*the* (*linearize j*)) by (*simp add*: *linearize-linear1*)
then have *linlj*: *islinintterm lj* using *prems* by *simp*
moreover from *lini linj* have *linearize* (*Add i j*) = *Some* (*lin-add* (*li*,*lj*))
using *prems* by (*simp add*: *measure-def inv-image-def*)
moreover from *linli linlj* have *I-intterm ats* (*lin-add* (*li*,*lj*)) = *I-intterm ats* (*Add li lj*) by (*simp add*: *lin-add-corr*)
ultimately have *?thesis* using *prems* by *simp*
}
ultimately show *?thesis* by *blast*
qed

next
case (*5 i j*)**show** *?case*
proof−
have (*linearize i* = *None*) ∨ ((∃ *li*. *linearize i* = *Some li*) ∧ *linearize j* = *None*) ∨ ((∃ *li*. *linearize i* = *Some li*) ∧ (∃ *lj*. *linearize j* = *Some lj*)) **by** *auto*
moreover
{
assume *nlini*: *linearize i* = *None*
from *nlini* **have** *linearize* (*Sub i j*) = *None* **by** (*simp add*: *Let-def measure-def inv-image-def*) **then have** *?thesis* **using** *prems* **by** *auto*
}
moreover
{
assume *lini*: ∃ *li*. *linearize i* = *Some li*
and *nlinj*: *linearize j* = *None*
from *nlinj lini* **have** *linearize* (*Sub i j*) = *None*
by (*simp add*: *Let-def measure-def inv-image-def*, *auto*) **with** *prems* **have** *?thesis* **by** *auto*
}
moreover
{
assume *lini*: ∃ *li*. *linearize i* = *Some li*
and *linj*: ∃ *lj*. *linearize j* = *Some lj*
from *lini* **have** *lini2*: *linearize i* ≠ *None* **by** *simp*
from *linj* **have** *linj2*: *linearize j* ≠ *None* **by** *simp*
from *lini* **obtain** *li* **where** *linearize i* = *Some li* **by** *blast*
from *lini2* **have** *islinintterm* (*the* (*linearize i*)) **by** (*simp add*: *linearize-linear1*)
with *prems* **have** *linli*: *islinintterm li* **by** *simp*
from *linj* **obtain** *lj* **where** *linearize j* = *Some lj* **by** *blast*
from *linj2* **have** *islinintterm* (*the* (*linearize j*)) **by** (*simp add*: *linearize-linear1*)
with *prems* **have** *linlj*: *islinintterm lj* **by** *simp*
moreover from *prems* **have** *linearize* (*Sub i j*) = *Some* (*lin-add* (*li*,*lin-neg lj*))
by (*simp add*: *measure-def inv-image-def*)
moreover from *linlj* **have** *linnlj*:*islinintterm* (*lin-neg lj*) **by** (*simp add*: *lin-neg-lin*)
moreover from *linli linnlj* **have** *I-intterm ats* (*lin-add* (*li*,*lin-neg lj*)) = *I-intterm ats* (*Add li* (*lin-neg lj*)) **by** (*simp only*: *lin-add-corr*[*OF linli linnlj*])
moreover from *linli linlj linnlj* **have** *I-intterm ats* (*Add li* (*lin-neg lj*)) = *I-intterm ats* (*Sub li lj*)
by (*simp add*: *lin-neg-corr*)
ultimately have *?thesis* **using** *prems* **by** *simp*
}
ultimately show *?thesis* **by** *blast*
qed
next
case (*6 i j*)**show** *?case*
proof−
have *cses*: (*linearize i* = *None*) ∨

$((\exists\ li.\ linearize\ i = Some\ li) \wedge linearize\ j = None) \vee$
$((\exists\ li.\ linearize\ i = Some\ li) \wedge (\exists\ bj.\ linearize\ j = Some\ (Cst\ bj)))$
$\vee\ ((\exists\ bi.\ linearize\ i = Some\ (Cst\ bi)) \wedge (\exists\ lj.\ linearize\ j = Some\ lj))$
$\vee\ ((\exists\ li.\ linearize\ i = Some\ li \wedge \neg\ (\exists\ bi.\ li = Cst\ bi)) \wedge (\exists\ lj.\ linearize\ j = Some\ lj \wedge \neg\ (\exists\ bj.\ lj = Cst\ bj)))$ **by** *auto*

moreover
{
assume *nlini*: *linearize i = None*
from *nlini* **have** *linearize (Mult i j) = None* **by** (*simp add*: *Let-def measure-def inv-image-def*) **with** *prems* **have** *?thesis* **by** *auto*
}
moreover
{
assume *lini*: $\exists$ *li. linearize i = Some li*
and *nlinj*: *linearize j = None*

from *lini* **obtain** *li* **where** *linearize i = Some li* **by** *blast*
moreover from *prems* **have** *linearize (Mult i j) = None*
by (*cases li*) (*simp-all add*: *Let-def measure-def inv-image-def*)
with *prems* **have** *?thesis* **by** *auto*
}
moreover
{
assume *lini*: $\exists$ *li. linearize i = Some li*
and *linj*: $\exists$ *bj. linearize j = Some (Cst bj)*
from *lini* **have** *lini2*: *linearize i* $\neq$ *None* **by** *simp*
from *linj* **have** *linj2*: *linearize j* $\neq$ *None* **by** *auto*
from *lini* **obtain** *li* **where** *linearize i = Some li* **by** *blast*
from *lini2* **have** *islinintterm (the (linearize i))* **by** (*simp add*: *linearize-linear1*)
with *prems* **have** *linli*: *islinintterm li* **by** *simp*
moreover
from *linj* **obtain** *bj* **where** *linearize j = Some (Cst bj)* **by** *blast*
have *linlj*: *islinintterm (Cst bj)* **by** *simp*
moreover from *prems* **have** *linearize (Mult i j) = Some (lin-mul (bj,li))*
by (*cases li*) (*auto simp add*: *measure-def inv-image-def*)
then have *lm1*: *I-intterm ats (the(linearize (Mult i j))) = I-intterm ats (lin-mul (bj,li))* **by** *simp*
moreover from *linli linlj* **have** *I-intterm ats (lin-mul(bj,li)) = I-intterm ats (Mult li (Cst bj))* **by** (*simp add*: *lin-mul-corr*)
with *prems*
have *I-intterm ats (lin-mul(bj,li)) = I-intterm ats (Mult li (the (linearize j)))*
by *auto*
moreover have *I-intterm ats (Mult li (the (linearize j))) = I-intterm ats (Mult i (the (linearize j)))* **using** *prems* **by** *simp*
moreover have *I-intterm ats i = I-intterm ats (the (linearize i))*
using *lini2 lini 6.hyps* **by** *simp*
moreover have *I-intterm ats j = I-intterm ats (the (linearize j))*
using *prems* **by** (*cases li*) (*auto simp add*: *measure-def inv-image-def*)

ultimately have *?thesis* **by** *auto* **}**
moreover
{ assume *lini*: $\exists$ *bi*. *linearize i* = *Some* (*Cst bi*)
and *linj*: $\exists$ *lj*. *linearize j* = *Some lj*
from *lini* **have** *lini2* : *linearize i* $\neq$ *None* **by** *auto*
from *linj* **have** *linj2* : *linearize j* $\neq$ *None* **by** *auto*
from *lini* **obtain** *bi* **where** *linearize i* = *Some* (*Cst bi*) **by** *blast*
have *linli*: *islinintterm* (*Cst bi*) **using** *prems* **by** *simp*
moreover
from *linj* **obtain** *lj* **where** *linearize j* = *Some lj* **by** *blast*
from *linj2* **have** *islinintterm* (*the* (*linearize j*)) **by** (*simp add*: *linearize-linear1*)

then have *linlj*: *islinintterm lj* **by** (*simp*!)
moreover from *linli lini linj* **have** *linearize* (*Mult i j*) = *Some* (*lin-mul* (*bi*,*lj*)) **apply** (*simp add*: *measure-def inv-image-def*)
apply *auto* **by** (*case-tac li*::*intterm*,*auto*!)
then have *lm1*: *I-intterm ats* (*the*(*linearize* (*Mult i j*))) = *I-intterm ats* (*lin-mul* (*bi*,*lj*)) **by** *simp*
moreover from *linli linlj* **have** *I-intterm ats* (*lin-mul*(*bi*,*lj*)) = *I-intterm ats* (*Mult* (*Cst bi*) *lj*) **by** (*simp add*: *lin-mul-corr*)
then have *I-intterm ats* (*lin-mul*(*bi*,*lj*)) = *I-intterm ats* (*Mult* (*the* (*linearize i*)) *lj*) **by** (*auto*!)
moreover have *I-intterm ats* (*Mult* (*the* (*linearize i*)) *lj*) = *I-intterm ats* (*Mult* (*the* (*linearize i*)) *j*) **using** *lini lini2* **by** (*simp*!)
moreover have *I-intterm ats i* = *I-intterm ats* (*the* (*linearize i*))
using *lini2 lini 6.hyps* **by** *simp*
moreover have *I-intterm ats j* = *I-intterm ats* (*the* (*linearize j*))
using *linj linj2 lini lini2 linli linlj 6.hyps* **by** (*auto*!)

ultimately have *?thesis* **by** *auto* **}**
moreover
{ assume *linc*: $\exists$ *li*. *linearize i* = *Some li* $\wedge$ $\neg$ ($\exists$ *bi*. *li* = *Cst bi*)
and *ljnc*: $\exists$ *lj*. *linearize j* = *Some lj* $\wedge$ $\neg$ ($\exists$ *bj*. *lj* = *Cst bj*)
from *linc* **obtain** *li* **where** $\exists$ *li*. *linearize i* = *Some li* $\wedge$ $\neg$ ($\exists$ *bi*. *li* = *Cst bi*) **by** *blast*
moreover
from *ljnc* **obtain** *lj* **where** $\exists$ *lj*. *linearize j* = *Some lj* $\wedge$ $\neg$ ($\exists$ *bj*. *lj* = *Cst bj*) **by** *blast*
ultimately have *linearize* (*Mult i j*) = *None*
apply (*simp add*: *measure-def inv-image-def*)
apply (*case-tac linearize i*, *auto*)
apply (*case-tac a*)
apply (*auto*!)
by (*case-tac lj*,*auto*)+
then have *?thesis* **by** (*simp*!) **}**
ultimately show *?thesis* **by** *blast*
qed
qed *simp-all*

lemma *linearize-corr*: $\bigwedge t'$. *linearize t = Some t'* $\Longrightarrow$ *I-intterm ats t = I-intterm ats t'*
proof−
  fix t'
  assume *lint*: *linearize t = Some t'*
  show *I-intterm ats t = I-intterm ats t'*
  proof−
    from *lint* **have** *lt*: *linearize t* $\neq$ *None* **by** *simp*
    then have *I-intterm ats t = I-intterm ats (the (linearize t))*
      by *(rule-tac linearize-corr1[OF lt])*
    with *lint* **show** *?thesis* **by** *simp*
  qed
qed

consts *linform* :: *QF* $\Rightarrow$ *QF option*
primrec
linform (Le it1 it2) =
  lift-bin(λx. λy. *Le (lin-add(x,lin-neg y)) (Cst 0),linearize it1, linearize it2)*
linform (Eq it1 it2) =
  lift-bin(λx. λy. *Eq (lin-add(x,lin-neg y)) (Cst 0),linearize it1, linearize it2)*
linform (Divides d t) =
  (case linearize d of
    None $\Rightarrow$ *None*
   | *Some ld* $\Rightarrow$ *(case ld of*
          Cst b $\Rightarrow$
               (if (b=0) then None
               else
                (case linearize t of
                 None $\Rightarrow$ *None*
               | *Some lt* $\Rightarrow$ *Some (Divides ld lt)))*
        | - $\Rightarrow$ *None))*
linform T = Some T
linform F = Some F
linform (NOT p) = lift-un NOT (linform p)
linform (And p q)= lift-bin(λf. λg. *And f g, linform p, linform q)*
linform (Or p q) = lift-bin(λf. λg. *Or f g, linform p, linform q)*

consts *islinform* :: *QF* $\Rightarrow$ *bool*
recdef *islinform measure size*
islinform (Le it (Cst i)) = (i=0 $\wedge$ *islinintterm it)*
islinform (Eq it (Cst i)) = (i=0 $\wedge$ *islinintterm it)*
islinform (Divides (Cst d) t) = (d $\neq$ *0* $\wedge$ *islinintterm t)*
islinform T = True
islinform F = True
islinform (NOT (Divides (Cst d) t)) = (d $\neq$ *0* $\wedge$ *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 $\bigwedge$ *p'*. ⟦*linform p* = *Some p'*⟧ ⟹ *isnnf p'*
using *nnfp*
proof (*induct p rule*: *isnnf.induct*, *simp-all*)
  case (*goal1 a b p'*)
  show *?case*
    using *prems*
    by (*cases linearize a*, *auto*) (*cases linearize b*, *auto*)
next
  case (*goal2 a b p'*)
  show *?case*
    using *prems*
    by (*cases linearize a*, *auto*) (*cases linearize b*, *auto*)
next
  case (*goal3 d t p'*)
  show *?case*
    using *prems*
    apply (*cases linearize d*, *auto*)
    apply (*case-tac a*,*auto*)
    apply (*case-tac int=0*,*auto*)
    by (*cases linearize t*,*auto*)
next
  case (*goal4 f g p'*) **show** *?case*
    using *prems*
    by (*cases linform f*, *auto*) (*cases linform g*, *auto*)
next
  case (*goal5 f g p'*) **show** *?case*
    using *prems*
    by (*cases linform f*, *auto*) (*cases linform g*, *auto*)
next
  case (*goal6 d t p'*) **show** *?case*
    using *prems*
    apply (*cases linearize d*, *auto*)
    apply (*case-tac a*, *auto*)
    apply (*case-tac int = 0*,*auto*)
    by (*cases linearize t*, *auto*)
next
  case (*goal7 a b p'*)
  show *?case*
    using *prems*
    by (*cases linearize a*, *auto*) (*cases linearize b*, *auto*)

qed

lemma *linform-corr*: $\bigwedge$ *lp*. $[\![$ *isnnf p* ; *linform p* = *Some lp* $]\!] \Longrightarrow$
(*qinterp ats p* = *qinterp ats lp*)
proof (*induct p rule*: *linform.induct*)
case (*Le x y*)
show *?case*
using *Le.prems*
proof−
have ($\exists$ *lx ly*. *linearize x* = *Some lx* $\wedge$ *linearize y* = *Some ly*) $\vee$
(*linearize x* = *None*) $\vee$ (*linearize y* = *None*)**by** *auto*
moreover
{
assume *linxy*: $\exists$ *lx ly*. *linearize x* = *Some lx* $\wedge$ *linearize y* = *Some ly*
from *linxy* **obtain** *lx ly*
where *lxly*:*linearize x* = *Some lx* $\wedge$ *linearize y* = *Some ly* **by** *blast*
then
have *lxeqx*: *I-intterm ats x* = *I-intterm ats lx*
by (*simp add*: *linearize-corr*)
from *lxly* **have** *lxlin*: *islinintterm lx*
by (*auto simp add*: *linearize-linear*)
from *lxly* **have** *lyeqy*: *I-intterm ats y* = *I-intterm ats ly*
by (*simp add*: *linearize-corr*)
from *lxly* **have** *lylin*: *islinintterm ly*
by (*auto simp add*: *linearize-linear*)
from *prems*
have *lpeqle*: *lp* = (*Le* (*lin-add*(*lx*,*lin-neg ly*)) (*Cst 0*))
by *auto*
moreover
have *lin1*: *islinintterm* (*Cst 1*) **by** *simp*
then
have *?thesis*
using *lxlin lylin lin1 lin-add-lin lin-neg-lin prems lxly lpeqle*
by (*simp add*: *lin-add-corr lin-neg-corr lxeqx lyeqy*)

}

moreover
{
assume *linearize x* = *None*
have *?thesis* **using** *prems* **by** *simp*
}

moreover
{
assume *linearize y* = *None*
then have *?thesis* **using** *prems*

```
      by (case-tac linearize x, auto)
    }
    ultimately show ?thesis by blast
  qed

next
  case (Eq x y)
  show ?case
    using Eq.prems
  proof-
    have (∃ lx ly. linearize x = Some lx ∧ linearize y = Some ly) ∨
      (linearize x = None) ∨ (linearize y = None)by auto
    moreover
    {
      assume linxy: ∃ lx ly. linearize x = Some lx ∧ linearize y = Some ly
      from linxy obtain lx ly
        where lxly:linearize x = Some lx ∧ linearize y = Some ly by blast
      then
      have lxeqx: I-intterm ats x = I-intterm ats lx
        by (simp add: linearize-corr)
      from lxly have lxlin: islinintterm lx
        by (auto simp add: linearize-linear)
      from lxly have lyeqy: I-intterm ats y = I-intterm ats ly
        by (simp add: linearize-corr)
      from lxly have lylin: islinintterm ly
        by (auto simp add: linearize-linear)
      from prems
      have lpeqle: lp =  (Eq (lin-add(lx,lin-neg ly)) (Cst 0))
        by auto
      moreover
      have lin1: islinintterm (Cst 1) by simp
      then
      have ?thesis
        using lxlin lylin lin1 lin-add-lin lin-neg-lin prems lxly lpeqle
        by (simp add: lin-add-corr lin-neg-corr lxeqx lyeqy)

    }

    moreover
    {
      assume linearize x = None
      have ?thesis using prems by simp
    }

    moreover
    {
      assume linearize y = None
      then have ?thesis using prems
        by (case-tac linearize x, auto)
```

```
    }
    ultimately show ?thesis by blast
  qed

next
  case (Divides d t)
  show ?case
    using Divides.prems
    apply (case-tac linearize d,auto)
    apply (case-tac a, auto)
    apply (case-tac int = 0, auto)
    apply (case-tac linearize t, auto)
    apply (simp add: linearize-corr)
    apply (case-tac a, auto)
    apply (case-tac int = 0, auto)
    by (case-tac linearize t, auto simp add: linearize-corr)
next
  case (NOT f) show ?case
    using prems
  proof−
    have (∃ lf. linform f = Some lf) ∨ (linform f = None) by auto
    moreover
    {
      assume linf: ∃ lf. linform f = Some lf
      from prems have isnnf (NOT f) by simp
      then have fnnf: isnnf f by (cases f) auto
      from linf obtain lf where lf: linform f = Some lf by blast
      then have lp = NOT lf using prems by auto
      with NOT.prems NOT.hyps lf fnnf
      have ?case by simp
    }
    moreover
    {
      assume linform f = None
      then
      have linform (NOT f) = None by simp
      then
      have ?thesis  using NOT.prems by simp
    }
    ultimately show ?thesis by blast
  qed
next
  case (Or f g)
  show ?case using Or.hyps
  proof −
    have ((∃ lf. linform f = Some lf ) ∧ (∃ lg. linform g = Some lg)) ∨
      (linform f = None) ∨ (linform g = None) by auto
    moreover
    {
```

```
      assume linf: ∃ lf. linform f = Some lf
        and ling: ∃ lg. linform g = Some lg
      from linf obtain lf where lf: linform f = Some lf by blast
      from ling obtain lg where lg: linform g = Some lg by blast
      from lf lg have linform (Or f g) = Some (Or lf lg) by simp
      then have lp = Or lf lg using lf lg prems by simp
      with lf lg prems have ?thesis by simp
    }
    moreover
    {
      assume linform f = None
      then have ?thesis using Or.prems by auto
    }
    moreover
    {
      assume linform g = None
      then have ?thesis using Or.prems by (case-tac linform f, auto)

    }
    ultimately show ?thesis by blast
  qed
next
  case (And f g)
  show ?case using And.hyps
  proof -
    have ((∃ lf. linform f = Some lf ) ∧ (∃ lg. linform g = Some lg)) ∨
      (linform f = None) ∨ (linform g = None) by auto
    moreover
    {
      assume linf: ∃ lf. linform f = Some lf
        and ling: ∃ lg. linform g = Some lg
      from linf obtain lf where lf: linform f = Some lf by blast
      from ling obtain lg where lg: linform g = Some lg by blast
      from lf lg have linform (And f g) = Some (And lf lg) by simp
      then have lp = And lf lg using lf lg prems by simp
      with lf lg prems have ?thesis by simp
    }
    moreover
    {
      assume linform f = None
      then have ?thesis using And.prems by auto
    }
    moreover
    {
      assume linform g = None
      then have ?thesis using And.prems by (case-tac linform f, auto)

    }
    ultimately show ?thesis by blast
```

qed

qed *simp-all*

lemma *linform-lin*: $\bigwedge$ *lp*. $[\![$ *isnnf p* ; *linform p* = *Some lp*$]\!] \Longrightarrow$ *islinform lp*
proof (*induct p rule*: *linform.induct*)
  case (*Le x y*)
 have ((∃ *lx*. *linearize x* = *Some lx*) ∧ (∃ *ly*. *linearize y* = *Some ly*)) ∨
  (*linearize x* = *None*) ∨ (*linearize y* = *None*) **by** *clarsimp*
 moreover
 {
  assume *linx*: ∃ *lx*. *linearize x* = *Some lx*
   and *liny*: ∃ *ly*. *linearize y* = *Some ly*
  from *linx* **obtain** *lx* **where** *lx*: *linearize x* = *Some lx* **by** *blast*
  from *liny* **obtain** *ly* **where** *ly*: *linearize y* = *Some ly* **by** *blast*
  from *lx* **have** *lxlin*: *islinintterm lx* **by** (*simp add*: *linearize-linear*)
  from *ly* **have** *lylin*: *islinintterm ly* **by** (*simp add*: *linearize-linear*)
  have *lin1*:*islinintterm* (*Cst 1*) **by** *simp*
  have *lin0*: *islinintterm* (*Cst 0*) **by** *simp*
  from *prems* **have** *lp* = *Le* (*lin-add*(*lx*,*lin-neg ly*)) (*Cst 0*)
   by *auto*
  with *lin0 lin1 lxlin lylin prems*
  have *?case* **by** (*simp add*: *lin-add-lin lin-neg-lin*)

 }

 moreover
 {
  assume *linearize x* = *None*
  then have *?case* **using** *prems* **by** *simp*
 }
 moreover
 {
  assume *linearize y* = *None*
  then have *?case* **using** *prems* **by** (*case-tac linearize x*,*simp-all*)
 }
 ultimately show *?case* **by** *blast*
next
  case (*Eq x y*)
 have ((∃ *lx*. *linearize x* = *Some lx*) ∧ (∃ *ly*. *linearize y* = *Some ly*)) ∨
  (*linearize x* = *None*) ∨ (*linearize y* = *None*) **by** *clarsimp*
 moreover
 {
  assume *linx*: ∃ *lx*. *linearize x* = *Some lx*
   and *liny*: ∃ *ly*. *linearize y* = *Some ly*
  from *linx* **obtain** *lx* **where** *lx*: *linearize x* = *Some lx* **by** *blast*
  from *liny* **obtain** *ly* **where** *ly*: *linearize y* = *Some ly* **by** *blast*

from *lx* **have** *lxlin*: *islinintterm lx* **by** (*simp add*: *linearize-linear*)
from *ly* **have** *lylin*: *islinintterm ly* **by** (*simp add*: *linearize-linear*)
have *lin1*:*islinintterm* (*Cst 1*) **by** *simp*
have *lin0*: *islinintterm* (*Cst 0*) **by** *simp*
from *prems* **have** *lp* = *Eq* (*lin-add*(*lx*,*lin-neg ly*)) (*Cst 0*)
by *auto*
with *lin0 lin1 lxlin lylin prems*
have *?case* **by** (*simp add*: *lin-add-lin lin-neg-lin*)

}

moreover
{
assume *linearize x* = *None*
then have *?case* **using** *prems* **by** *simp*
}
moreover
{
assume *linearize y* = *None*
then have *?case* **using** *prems* **by** (*case-tac linearize x*,*simp-all*)
}
ultimately show *?case* **by** *blast*
next
case (*Divides d t*)
show *?case*
using *prems*
apply (*case-tac linearize d*, *auto*)
apply (*case-tac a*, *auto*)
apply (*case-tac int* = *0*, *auto*)

by (*case-tac linearize t*,*auto simp add*: *linearize-linear*)
next
case (*Or f g*)
show *?case* **using** *Or.hyps*
proof −
have ((∃ *lf*. *linform f* = *Some lf*) ∧ (∃ *lg*. *linform g* = *Some lg*)) ∨
(*linform f* = *None*) ∨ (*linform g* = *None*) **by** *auto*
moreover
{
assume *linf*: ∃ *lf*. *linform f* = *Some lf*
and *ling*: ∃ *lg*. *linform g* = *Some lg*
from *linf* **obtain** *lf* **where** *lf*: *linform f* = *Some lf* **by** *blast*
from *ling* **obtain** *lg* **where** *lg*: *linform g* = *Some lg* **by** *blast*
from *lf lg* **have** *linform* (*Or f g*) = *Some* (*Or lf lg*) **by** *simp*
then have *lp* = *Or lf lg* **using** *lf lg prems* **by** *simp*
with *lf lg prems* **have** *?thesis* **by** *simp*
}
moreover
{

```
      assume linform f = None
      then have ?thesis using Or.prems  by auto
    }
    moreover
    {
      assume linform g = None
      then have ?thesis using Or.prems  by (case-tac linform f, auto)

    }
    ultimately show ?thesis by blast
  qed
next
  case (And f g)
  show ?case using And.hyps
  proof −
    have ((∃ lf. linform f = Some lf ) ∧ (∃ lg. linform g = Some lg)) ∨
      (linform f = None) ∨ (linform g = None) by auto
    moreover
    {
      assume linf: ∃ lf. linform f = Some lf
        and ling: ∃ lg. linform g = Some lg
      from linf obtain lf where lf: linform f = Some lf by blast
      from ling obtain lg where lg: linform g = Some lg by blast
      from lf lg have linform (And f g) = Some (And lf lg) by simp
      then have lp = And lf lg using lf lg prems  by simp
      with lf lg prems have ?thesis by simp
    }
    moreover
    {
      assume linform f = None
      then have ?thesis using And.prems  by auto
    }
    moreover
    {
      assume linform g = None
      then have ?thesis using And.prems  by (case-tac linform f, auto)

    }
    ultimately show ?thesis by blast
  qed
next
  case (NOT f) show ?case
    using prems
  proof−
    have (∃ lf. linform f = Some lf) ∨ (linform f = None) by auto
    moreover
    {
      assume linf: ∃ lf. linform f = Some lf
      from prems have isnnf (NOT f) by simp
```

then have *fnnf*: *isnnf f* **by** (*cases f*) *auto*
from *linf* **obtain** *lf* **where** *lf*: *linform f* = *Some lf* **by** *blast*
then have *lp* = *NOT lf* **using** *prems* **by** *auto*
with *NOT.prems NOT.hyps lf fnnf*
have *?thesis*
using *fnnf*
apply (*cases f*, *auto*)
prefer *2*
apply (*case-tac linearize intterm1*,*auto*)
apply (*case-tac a*, *auto*)
apply (*case-tac int* = *0*, *auto*)
apply (*case-tac linearize intterm2*)
apply (*auto simp add*: *linearize-linear*)
apply (*case-tac linearize intterm1*,*auto*)
by (*case-tac linearize intterm2*)
(*auto simp add*: *linearize-linear lin-add-lin lin-neg-lin*)
}
moreover
{
assume *linform f* = *None*
then
have *linform* (*NOT f*) = *None* **by** *simp*
then
have *?thesis* **using** *NOT.prems* **by** *simp*
}
ultimately show *?thesis* **by** *blast*
qed
qed (*simp-all*)

lemma *linform-isnnf*: *islinform p* $\Longrightarrow$ *isnnf p*
by (*induct p rule*: *islinform.induct*) *auto*

lemma *linform-isqfree*: *islinform p* $\Longrightarrow$ *isqfree p*
using *linform-isnnf nnf-isqfree* **by** *simp*

lemma *linform-qfree*: $\bigwedge p'$. $[\![$ *isnnf p* ; *linform p* = *Some p'* $]\!]$ $\Longrightarrow$ *isqfree p'*
using *linform-isqfree linform-lin*
by *simp*

constdefs *lcm* :: *nat* × *nat* ⇒ *nat*
lcm ≡ (λ(*m*,*n*). *m*∗*n div gcd*(*m*,*n*))

constdefs *ilcm* :: *int* ⇒ *int* ⇒ *int*
ilcm ≡ λ*i*.λ*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−
  have *gcd*(*m*,*n*) *dvd n* **by** *simp*
  then obtain *k* **where** *n* = *gcd*(*m*,*n*) ∗ *k* **using** *dvd-def* **by** *auto*
  then have *m*∗*n div gcd*(*m*,*n*) = *m*∗(*gcd*(*m*,*n*)∗*k*) *div gcd*(*m*,*n*) **by** (*simp add*: *mult-ac*)
  also have ... = *m*∗*k* **using** *mpos npos gcd-zero* **by** *simp*
  finally show *?thesis* **by** (*simp add*: *lcm-def*)
qed

lemma *lcm-dvd2*:
  assumes *mpos*: *m >0*
  and *npos*: *n>0*
  shows *n dvd* (*lcm*(*m*,*n*))
proof−
  have *gcd*(*m*,*n*) *dvd m* **by** *simp*
  then obtain *k* **where** *m* = *gcd*(*m*,*n*) ∗ *k* **using** *dvd-def* **by** *auto*
  then have *m*∗*n div gcd*(*m*,*n*) = (*gcd*(*m*,*n*)∗*k*)∗*n div gcd*(*m*,*n*) **by** (*simp add*: *mult-ac*)
  also have ... = *n*∗*k* **using** *mpos npos gcd-zero* **by** *simp*
  finally show *?thesis* **by** (*simp add*: *lcm-def*)
qed

lemma *ilcm-dvd1*:
assumes *anz*: *a* ≠ *0*
  and *bnz*: *b* ≠ *0*
  shows *a dvd* (*ilcm a b*)
proof−
  let *?na* = *nat* (*abs a*)
  let *?nb* = *nat* (*abs b*)
  have *nap*: *?na >0* **using** *anz* **by** *simp*
  have *nbp*: *?nb >0* **using** *bnz* **by** *simp*
  from *nap nbp* **have** *?na dvd lcm*(*?na*,*?nb*) **using** *lcm-dvd1* **by** *simp*
  thus *?thesis* **by** (*simp add*: *ilcm-def dvd-int-iff*)
qed

lemma *ilcm-dvd2*:
assumes *anz*: *a* ≠ *0*
  and *bnz*: *b* ≠ *0*
  shows *b dvd* (*ilcm a b*)
proof−
  let *?na* = *nat* (*abs a*)
  let *?nb* = *nat* (*abs b*)
  have *nap*: *?na >0* **using** *anz* **by** *simp*
  have *nbp*: *?nb >0* **using** *bnz* **by** *simp*

```
  from nap nbp have ?nb dvd lcm(?na,?nb) using lcm-dvd2 by simp
  thus ?thesis by (simp add: ilcm-def dvd-int-iff)
qed

lemma zdvd-self-abs1: (d::int) dvd (abs d)
by (case-tac d <0, simp-all)

lemma zdvd-self-abs2: (abs (d::int)) dvd d
by (case-tac d<0, simp-all)



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

proof(rule ccontr, simp add: lcm-def gcd-zero)
assume h:m*n div gcd(m,n) = 0
from mpos npos have gcd (m,n) ≠ 0 using gcd-zero by simp
hence gcdp: gcd(m,n) > 0 by simp
with h
have m*n < gcd(m,n)
  by (cases m * n < gcd (m, n)) (auto simp add: div-if[OF gcdp, where m=m*n])
moreover
have gcd(m,n) dvd m by simp
 with mpos dvd-imp-le have t1:gcd(m,n) ≤ m by simp
 with npos have t1:gcd(m,n)*n ≤ m*n by simp
 have gcd(m,n) ≤ gcd(m,n)*n using npos by simp
 with t1 have gcd(m,n) ≤ m*n by arith
ultimately show False by simp
qed

lemma ilcm-pos:
  assumes apos:  0 < a
  and bpos: 0 < b
  shows 0 < ilcm  a b
proof−
  let ?na = nat (abs a)
  let ?nb = nat (abs b)
  have nap: ?na >0 using apos by simp
  have nbp: ?nb >0 using bpos by simp
  have 0 < lcm (?na,?nb) by (rule lcm-pos[OF nap nbp])
  thus ?thesis by (simp add: ilcm-def)
qed


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*
using *linp*
proof (*induct p rule*: *formlcm.induct*, *simp-all add*: *ilcm-pos*)
  case (*goal1 c r i*)
  have *i*=*0* ∨ *i* ≠ *0* **by** *simp*
  moreover
  {
    assume *i* ≠ *0* **then have** *?case* **using** *prems* **by** *simp*
  }
  moreover
  {
    assume *iz*: *i* = *0*
    then have *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
    then have *c*≠*0*
      using *prems*
      by (*simp add*: *islininttermc0r*[**where** *c*=*c* **and** *n*=*0* **and** *r*=*r*])
    then have *?case* **by** *simp*
  }
  ultimately
  show *?case* **by** *blast*
next
  case (*goal2 c r i*)
  have *i*=*0* ∨ *i* ≠ *0* **by** *simp*
  moreover
  {
    assume *i* ≠ *0* **then have** *?case* **using** *prems* **by** *simp*
  }

moreover
{
assume *iz*: $i = 0$
then have *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
then have $c \neq 0$
using *prems*
by (*simp add*: *islininttermc0r*[**where** $c=c$ **and** $n=0$ **and** $r=r$])
then have *?case* **by** *simp*
}
ultimately
show *?case* **by** *blast*

next
case (*goal3 d c r*)
show *?case* **using** *prems* **by** (*simp add*: *islininttermc0r*[**where** $c=c$ **and** $n=0$ **and** $r=r$])
next
case (*goal4 f*)
show *?case* **using** *prems*
by (*cases f*,*auto*) (*case-tac intterm2*, *auto*,*case-tac intterm1*, *auto*)
qed

lemma *divideallc-mono*: $\bigwedge$ *c*. ⟦ *divideallc*(*c*,*p*) ; *c dvd d*⟧ $\Longrightarrow$ *divideallc* (*d*,*p*)
proof (*induct d p rule*: *divideallc.induct*, *simp-all*)
case (*goal1 l a b*) **show** *?case* **by** (*rule zdvd-trans* [**where** $m=a$ **and** $n=b$ **and** $k=l$])
next
case (*goal2 l a b*) **show** *?case* **by** (*rule zdvd-trans* [**where** $m=a$ **and** $n=b$ **and** $k=l$])
next
case (*goal3 l a b*) **show** *?case* **by** (*rule zdvd-trans* [**where** $m=a$ **and** $n=b$ **and** $k=l$])
next
case (*goal4 l f g k*)
have *divideallc* (*l*,*g*) **using** *prems* **by** *clarsimp*
moreover have *divideallc* (*l*,*f*) **using** *prems* **by** *clarsimp*
ultimately
show *?case* **by** *simp*
next
case (*goal5 l f g k*)
have *divideallc* (*l*,*g*) **using** *prems* **by** *clarsimp*
moreover have *divideallc* (*l*,*f*) **using** *prems* **by** *clarsimp*
ultimately
show *?case* **by** *simp*

qed

```
lemma formlcm-divideallc:
  assumes linp: islinform p
  shows divideallc(formlcm p, p)
using linp
proof (induct p rule: formlcm.induct, simp-all add: zdvd-self-abs1)
  case (goal1 f)
  show ?case using prems
    by (cases f,auto) (case-tac intterm2, auto, case-tac intterm1,auto)
next
  case (goal2 f g)
  have formlcm f >0 using formlcm-pos prems by simp
    hence formlcm f ≠ 0 by simp
  moreover have formlcm g > 0 using formlcm-pos prems by simp
  hence formlcm g ≠ 0 by simp
  ultimately
  show ?case using prems formlcm-pos
     by (simp add: ilcm-dvd1 ilcm-dvd2
       divideallc-mono[where c=formlcm f and d=ilcm (formlcm f) (formlcm g)]

      divideallc-mono[where c=formlcm g and d=ilcm (formlcm f) (formlcm g)])
next
  case (goal3 f g)
  have formlcm f >0 using formlcm-pos prems by simp
    hence formlcm f ≠ 0 by simp
  moreover have formlcm g > 0 using formlcm-pos prems by simp
  hence formlcm g ≠ 0 by simp
  ultimately
  show ?case using prems
    by (simp add: ilcm-dvd1 ilcm-dvd2
      divideallc-mono[where c=formlcm f and d=ilcm (formlcm f) (formlcm g)]
      divideallc-mono[where c=formlcm g and d=ilcm (formlcm f) (formlcm g)])
qed


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* ⇒ *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* ⇒ *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* ⟹ *islinform p*
by (*induct p rule*: *isunified.induct*) *auto*

lemma *adjustcoeff-lenpos*:
 0 < *n* ⟹ *adjustcoeff* (*l*, *Le* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*) (*Cst c*)) =
  Le (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*) (*Cst c*)
by (*cases n*, *auto*)

lemma *adjustcoeff-eqnpos*:
 0 < *n* ⟹ *adjustcoeff* (*l*, *Eq* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*) (*Cst c*)) =
  Eq (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*) (*Cst c*)
by (*cases n*, *auto*)

lemma *zmult-zle-mono*: $(i{::}int) \le j \Longrightarrow 0 \le k \Longrightarrow k * i \le k * j$
  apply (*erule order-le-less* [*THEN iffD1*, *THEN disjE*, *of 0::int*])
  apply (*erule order-le-less* [*THEN iffD1*, *THEN disjE*])
  apply (*rule order-less-imp-le*)
  apply (*rule zmult-zless-mono2*)
  apply *simp-all*
  done

lemma *zmult-zle-mono-eq*:
  assumes *kpos*: $0 < k$
  shows $((i{::}int) \le j) = (k{*}i \le k{*}j)$ (**is** $?P = ?Q$)
proof
  assume *P*: $?P$
  from *kpos* **have** *kge0*: $0 \le k$ **by** *simp*
  show $?Q$
    by (*rule zmult-zle-mono*[*OF P kge0*])
next
  assume $?Q$
  then have $k{*}i - k{*}j \le 0$ **by** *simp*
  then have *le1*: $k{*}(i{-}j) \le k{*}0$
    by (*simp add*: *zdiff-zmult-distrib2*)
  have $i - j \le 0$
    by (*rule mult-left-le-imp-le*[*OF le1 kpos*])
  then
  show $?P$ **by** *simp*
qed

lemma *adjustcoeff-le-corr*:
  assumes *lpos*: $0 < l$
  and *ipos*: $0 < (i{::}int)$
  and *dvd*: *i dvd l*
  shows $(i{*}x + r \le 0) = (l{*}x + ((l\ div\ i){*}r) \le 0)$
proof−
  from *lpos ipos* **have** *ilel*: $i \le l$ **by** (*simp add*: *zdvd-imp-le* [*OF dvd lpos*])
  from *ipos* **have** *inz*: $i \ne 0$ **by** *simp*
  have $i\ div\ i \le l\ div\ i$
    by (*simp add*: *zdiv-mono1*[*OF ilel ipos*])
  then have *ldivipos*:$0 < l\ div\ i$
    by (*simp add*: *zdiv-self*[*OF inz*])

  from *dvd* **have** $\exists\, i'.\ i{*}i' = l$ **by** (*auto simp add*: *dvd-def*)
  then obtain i' **where** *ii'eql*: $i{*}i' = l$ **by** *blast*
  have $(i * x + r \le 0) = (l\ div\ i * (i * x + r) \le l\ div\ i * 0)$
    by (*rule zmult-zle-mono-eq*[*OF ldivipos*, **where** $i{=}i{*}x + r$ **and** $j{=}0$])
  also
  have $(l\ div\ i * (i * x + r) \le l\ div\ i * 0) = ((l\ div\ i * i) * x + ((l\ div\ i){*}r) \le 0)$
    by (*simp add*: *mult-ac*)

also have $((l\ div\ i * i) * x + ((l\ div\ i)*r) \leq 0) = (l*x + ((l\ div\ i)*r) \leq 0)$
using *sym*[*OF ii'eql*] *inz*
by (*simp add*: *zmult-ac*)
finally
show *?thesis*
by *simp*
qed

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−
from *dvd* **have** *midvdl*: $-i$ *dvd l* **by** *simp*
from *ineg* **have** *mipos*: $0 < -i$ **by** *simp*
from *lpos ineg* **have** *milel*: $-i \leq l$ **by** (*simp add*: *zdvd-imp-le* [*OF midvdl lpos*])
from *ineg* **have** *inz*: $i \neq 0$ **by** *simp*
have $l\ div\ i \leq -i\ div\ i$
by (*simp add*: *zdiv-mono1-neg*[*OF milel ineg*])
then have $l\ div\ i \leq -1$
apply (*simp add*: *zdiv-zminus1-eq-if*[*OF inz*, **where** $a=i$])
by (*simp add*: *zdiv-self*[*OF inz*])
then have *ldivineg*: $l\ div\ i < 0$ **by** *simp*
then have *mldivipos*: $0 < -(l\ div\ i)$ **by** *simp*

from *dvd* **have** $\exists\, i'.\ i*i' = l$ **by** (*auto simp add*: *dvd-def*)
then obtain i' **where** *ii'eql*: $i*i' = l$ **by** *blast*
have $(i * x + r \leq 0) = (-(l\ div\ i) * (i * x + r) \leq -(l\ div\ i) * 0)$
by (*rule zmult-zle-mono-eq*[*OF mldivipos*, **where** $i=i*x + r$ **and** $j=0$])
also
have $(-(l\ div\ i) * (i * x + r) \leq -(l\ div\ i) * 0) = (-((l\ div\ i) * i) * x \leq (l\ div\ i)*r)$
by (*simp add*: *mult-ac*)
also have $(-((l\ div\ i) * i) * x \leq (l\ div\ i)*r) = (-(l*x) \leq (l\ div\ i)*r)$
using *sym*[*OF ii'eql*] *inz*
by (*simp add*: *zmult-ac*)
finally
show *?thesis*
by *simp*
qed

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−

```
  from anz have 0 < a ∨ a < 0 by arith
  moreover
  {
    assume apos: 0 < a
    from bpos apos have aleb: a≤b by (simp add: zdvd-imp-le [OF dvd bpos])
    have a div a≤ b div a
      by (simp add: zdiv-mono1[OF aleb apos])
    then have bdivapos:0 < b div a
      by (simp add: zdiv-self[OF anz])

    from dvd have ∃ a'. a*a' = b by (auto simp add: dvd-def)
    then obtain a' where aa'eqb: a*a' = b by blast
    then have ?thesis  using anz sym[OF aa'eqb] by simp

  }
  moreover
  {
    assume aneg: a < 0
    from dvd have midvdb: −a dvd b by simp
    from aneg have mapos: 0 < −a by simp
     from bpos aneg have maleb: −a≤b by (simp add: zdvd-imp-le [OF midvdb
bpos])
    from aneg have anz: a ≠ 0 by simp
    have b div a≤ −a div a
      by (simp add: zdiv-mono1-neg[OF maleb aneg])
    then have b div a ≤ −1
      apply (simp add: zdiv-zminus1-eq-if[OF anz, where a=a])
      by (simp add: zdiv-self[OF anz])
    then have bdivaneg: b div a < 0 by simp
    then have mbdivapos: 0 < − (b div a) by simp

    from dvd have ∃ a'. a*a' = b by (auto simp add: dvd-def)
    then obtain a' where aa'eqb: a*a' = b by blast
    then have ?thesis using anz sym[OF aa'eqb] by (simp)
  }
  ultimately show ?thesis by blast
qed

lemma adjustcoeff-eq-corr:
  assumes lpos: (0::int) < l
  and inz: i ≠ 0
  and dvd: i dvd l
  shows (i*x + r = 0) = (l*x + ((l div i)*r) = 0)
proof−
  have ldvdii: (l div i)*i = l by (rule dvd-div-pos[OF lpos inz dvd])
  have ldivinz: l div i ≠ 0 using inz ldvdii lpos by auto
  have (i*x + r = 0) = ((l div i)*(i*x + r) = (l div i)*0)
    using ldivinz by arith
  also have ... = (((l div i)*i)*x + (l div i)*r = 0)
```

```
    by (simp add: zmult-ac)
  finally show ?thesis using ldvdii by simp
qed
```

```
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))
using linp alldvd
proof (induct p rule: islinform.induct,simp-all)
  case (goal1 t c)
  from prems have cz: c=0 by simp
    then have ?case
      using prems
    proof(induct t rule: islinintterm.induct)
      case (2 i n i′) show ?case using prems
        proof−
          from prems have i≠0 by simp
          then
          have (n=0 ∧ i < 0) ∨ (n=0 ∧ i > 0) ∨ n≠0 by arith
          moreover
          {
            assume n≠0 then have ?thesis
              by (simp add: nth-pos2 adjustcoeff-lenpos)
          }
          moreover
          {
            assume nz: n=0
              and ipos: 0 < i
            from prems nz have idvdl: i dvd l by simp
            have (i∗a + i′ ≤ 0) = (l∗a+ ((l div i)∗i′) ≤ 0)
              by (rule adjustcoeff-le-corr[OF lpos ipos idvdl])
            then
            have ?thesis using prems by (simp add: mult-ac)
          }
          moreover
          {
            assume nz: n=0
              and ineg: i < 0
            from prems nz have idvdl: i dvd l by simp
            have (i∗a+i′ ≤ 0) = (−l∗a + (−(l div i) ∗ i′) ≤ 0)
              by (rule adjustcoeff-le-corr2[OF lpos ineg idvdl])
            then
            have ?thesis using prems
              by (simp add: zmult-ac)
```

```
      }
      ultimately show ?thesis by blast
    qed
  next
    case (3 i n i' n' r) show ?case  using prems
    proof-
      from prems
      have lininrp: islinintterm (Add (Mult (Cst i') (Var n')) r)
        by simp
      then
      have islint (Add (Mult (Cst i') (Var n')) (r))
        by (simp add: islinintterm-eq-islint)
      then have linr: islintn(Suc n',r)
     by (simp add: islinintterm-subt[OF lininrp] islinintterm-eq-islint islint-def)
      from lininrp have linr2: islinintterm r
        by (simp add: islinintterm-subt[OF lininrp])
      from prems have n < n' by simp
      then have nppos: 0 < n' by simp
      from prems have i≠0 by simp
      then
      have (n=0 ∧ i < 0) ∨ (n=0 ∧ i > 0) ∨ n≠0 by arith
      moreover
      {
        assume nnz: n≠0
          from linr have ?thesis using nppos nnz intterm-novar0[OF lininrp]
prems
            apply (simp add: adjustcoeff-lenpos linterm-novar0[OF linr, where
x=a and y=a*l])
          by (simp add: nth-pos2)

      }
      moreover
      {
        assume nz: n=0
          and ipos: 0 < i
        from prems nz have idvdl: i dvd l by simp
        have (i * a + (i' * (a # ats) ! n' + I-intterm (a # ats) r) ≤ 0) =
          (l * a + l div i * (i' * (a # ats) ! n' + I-intterm (a # ats) r) ≤ 0)
          by (rule adjustcoeff-le-corr[OF lpos ipos idvdl])
        then
        have ?thesis using prems linr linr2
          by (simp add: mult-ac nth-pos2 lin-mul-corr
            linterm-novar0[OF linr, where x=a and y=a*l])
      }
      moreover
      {
        assume nz: n=0
          and ineg: i < 0
        from prems nz have idvdl: i dvd l by simp
```

```
          have (i * a + (i' * (a # ats) ! n' + I-intterm (a # ats) r) ≤ 0) =
            (− l * a + − (l div i) * (i' * (a # ats) ! n' + I-intterm (a # ats) r)
≤ 0)
              by (rule adjustcoeff-le-corr2[OF lpos ineg idvdl, where x=a and
r=(i'* (a#ats) ! n' + I-intterm (a#ats) r )])
          then
          have ?thesis using prems linr linr2
            by (simp add: zmult-ac nth-pos2 lin-mul-corr
              linterm-novar0[OF linr, where x=a and y=a*l] )
        }
        ultimately show ?thesis by blast
      qed
    qed simp-all
    then show ?case by simp

next
  case (goal2 t c)
  from prems have cz: c=0 by simp
    then have ?case
      using prems
    proof(induct t rule: islinintterm.induct)
      case (2 i n i') show ?case using prems
        proof−
          from prems have inz: i≠0 by simp
          then
          have n=0 ∨ n≠0 by arith
          moreover
          {
            assume n≠0 then have ?thesis
              by (simp add: nth-pos2 adjustcoeff-eqnpos)
          }
          moreover
          {
            assume nz: n=0
            from prems nz have idvdl: i dvd l by simp
            have (i*a + i' = 0) = (l*a+ ((l div i)*i') = 0)
              by (rule adjustcoeff-eq-corr[OF lpos inz idvdl])
            then
            have ?thesis using prems by (simp add: mult-ac)
          }
          ultimately show ?thesis by blast
        qed
      next
        case (3 i n i' n' r) show ?case  using prems
        proof−
          from prems
          have lininrp: islinintterm (Add (Mult (Cst i') (Var n')) r)
            by simp
          then
```

```
      have islint (Add (Mult (Cst i') (Var n')) (r))
        by (simp add: islinintterm-eq-islint)
      then have linr: islintn(Suc n',r)
      by (simp add: islinintterm-subt[OF lininrp] islinintterm-eq-islint islint-def)
      from lininrp have linr2: islinintterm r
        by (simp add: islinintterm-subt[OF lininrp])
      from prems have n < n' by simp
      then have nppos: 0 < n' by simp
      from prems have i≠0 by simp
      then
      have n=0 ∨ n≠0 by arith
      moreover
      {
        assume nnz: n≠0
          from linr have ?thesis using nppos nnz intterm-novar0[OF lininrp]
prems
           apply (simp add: adjustcoeff-eqnpos linterm-novar0[OF linr, where
x=a and y=a*l])
          by (simp add: nth-pos2)

      }
      moreover
      {
        assume nz: n=0
        from prems have inz: i ≠ 0 by auto
        from prems nz have idvdl: i dvd l by simp
        have (i * a + (i' * (a # ats) ! n' + I-intterm (a # ats) r) = 0) =
          (l * a + l div i * (i' * (a # ats) ! n' + I-intterm (a # ats) r) = 0)
          by (rule adjustcoeff-eq-corr[OF lpos inz idvdl])
        then
        have ?thesis using prems linr linr2
          by (simp add: mult-ac nth-pos2 lin-mul-corr
            linterm-novar0[OF linr, where x=a and y=a*l])
      }
      ultimately show ?thesis by blast
    qed
  qed simp-all
  then show ?case by simp

next
 case (goal3 d t) show ?case
  using prems
  proof (induct t rule: islinintterm.induct)
   case (2 i n i')
   have n=0 ∨ (∃ m. (n = Suc m)) by arith
   moreover
   {
    assume ∃ m. n = Suc m
    then have ?case using prems  by auto
```

```
  }
  moreover
  {
    assume nz: n=0
    from prems have inz: i≠0 by simp
    from prems have idvdl: i dvd l by simp
    have ldiviieql: l div i * i = l by (rule dvd-div-pos[OF lpos inz idvdl])
    with lpos have ldivinz: 0 ≠ l div i by auto

    then have ?case using prems
      apply simp
      apply (simp add:
        ac-dvd-eq[OF ldivinz, where m=d and c=i and n=a and t=i']
        ldiviieql)
      by (simp add: zmult-commute)
  }
  ultimately show ?case by blast

next
  case (3 i n i' n' r)
  from prems
  have lininrp: islinintterm (Add (Mult (Cst i') (Var n')) r)
    by simp
  then
  have islint (Add (Mult (Cst i') (Var n')) (r))
    by (simp add: islinintterm-eq-islint)
  then have linr: islintn(Suc n',r)
    by (simp add: islinintterm-subt[OF lininrp] islinintterm-eq-islint islint-def)
  from lininrp have linr2: islinintterm r
    by (simp add: islinintterm-subt[OF lininrp])
  from prems have n < n' by simp
  then have nppos: 0 < n' by simp
  from prems have inz: i≠0 by simp

  have n=0 ∨ (∃ m. (n = Suc m)) by arith
  moreover
  {
    assume ∃ m. n = Suc m
    then have npos: 0 < n by arith
    have ?case using nppos intterm-novar0[OF lininrp] prems
      apply (auto simp add: linterm-novar0[OF linr, where x=a and y=a*l])
      by (simp-all add: nth-pos2)
  }
  moreover
  {
    assume nz: n=0
    from prems have idvdl: i dvd l by simp
    have ldiviieql: l div i * i = l by (rule dvd-div-pos[OF lpos inz idvdl])
    with lpos have ldivinz: 0 ≠ l div i by auto
```

```
        then have ?case using prems linr2 linr
          apply (simp add: nth-pos2 lin-mul-corr linterm-novar0)

          apply (simp add: ac-dvd-eq[OF ldivinz, where m=d and c=i and n=a
and t=(i' * ats ! (n' − Suc 0) + I-intterm (a # ats) r)] ldiviieql)
         by (simp add: zmult-ac linterm-novar0[OF linr, where x=a and y=a*l])
      }
      ultimately show ?case by blast

    qed simp-all
next
  case (goal4 d t) show ?case
    using prems
    proof (induct t rule: islinintterm.induct)
      case (2 i n i')
      have n=0 ∨ (∃ m. (n = Suc m)) by arith
      moreover
      {
        assume ∃ m. n = Suc m
        then have ?case using prems  by auto
      }
      moreover
      {
        assume nz: n=0
        from prems have inz: i≠0 by simp
        from prems have idvdl: i dvd l by simp
        have ldiviieql: l div i * i = l by (rule dvd-div-pos[OF lpos inz idvdl])
        with lpos have ldivinz: 0 ≠ l div i by auto

        then have ?case using prems
          apply simp
          apply (simp add:
            ac-dvd-eq[OF ldivinz, where m=d and c=i and n=a and t=i']
            ldiviieql)
          by (simp add: zmult-commute)
      }
      ultimately show ?case by blast

    next
      case (3 i n i' n' r)
      from prems
      have lininrp: islinintterm (Add (Mult (Cst i') (Var n')) r)
        by simp
      then
      have islint (Add (Mult (Cst i') (Var n')) (r))
        by (simp add: islinintterm-eq-islint)
      then have linr: islintn(Suc n',r)
        by (simp add: islinintterm-subt[OF lininrp] islinintterm-eq-islint islint-def)
```

```
      from lininrp have linr2: islinintterm r
        by (simp add: islinintterm-subt[OF lininrp])
      from prems have n < n' by simp
      then have nppos: 0 < n' by simp
      from prems have inz: i≠0 by simp

      have n=0 ∨ (∃ m. (n = Suc m)) by arith
      moreover
      {
        assume ∃ m. n = Suc m
        then have npos: 0 < n by arith
        have ?case using nppos intterm-novar0[OF lininrp] prems
          apply (auto simp add: linterm-novar0[OF linr, where x=a and y=a*l])
          by (simp-all add: nth-pos2)
      }
      moreover
      {
        assume nz: n=0
        from prems have idvdl: i dvd l by simp
        have ldiviieql: l div i * i = l by (rule dvd-div-pos[OF lpos inz idvdl])
        with lpos have ldivinz: 0 ≠ l div i by auto

        then have ?case using prems linr2 linr
          apply (simp add: nth-pos2 lin-mul-corr linterm-novar0)

          apply (simp add: ac-dvd-eq[OF ldivinz, where m=d and c=i and n=a
and t=(i' * ats ! (n' − Suc 0) + I-intterm (a # ats) r)] ldiviieql)
         by (simp add: zmult-ac linterm-novar0[OF linr, where x=a and y=a*l])
      }
      ultimately show ?case by blast

    qed simp-all
next
    case (goal5 t c)
  from prems have cz: c=0 by simp
    then have ?case
      using prems
    proof(induct t rule: islinintterm.induct)
      case (2 i n i') show ?case using prems
        proof−
          from prems have inz: i≠0 by simp
          then
          have n=0 ∨ n≠0 by arith
          moreover
          {
            assume n≠0 then have ?thesis
              using prems
              by (cases n, simp-all)
          }
```

```
    moreover
    {
      assume nz: n=0
      from prems nz have idvdl: i dvd l by simp
      have (i*a + i' = 0) = (l*a+ ((l div i)*i') = 0)
        by (rule adjustcoeff-eq-corr[OF lpos inz idvdl])
      then
      have ?thesis using prems by (simp add: mult-ac)
    }
    ultimately show ?thesis by blast
  qed
next
  case (3 i n i' n' r) show ?case  using prems
  proof-
    from prems
    have lininrp: islinintterm (Add (Mult (Cst i') (Var n')) r)
      by simp
    then
    have islint (Add (Mult (Cst i') (Var n')) (r))
      by (simp add: islinintterm-eq-islint)
    then have linr: islintn(Suc n',r)
   by (simp add: islinintterm-subt[OF lininrp] islinintterm-eq-islint islint-def)
    from lininrp have linr2: islinintterm r
      by (simp add: islinintterm-subt[OF lininrp])
    from prems have n < n' by simp
    then have nppos: 0 < n' by simp
    from prems have i≠0 by simp
    then
    have n=0 ∨ n≠0 by arith
    moreover
    {
      assume nnz: n≠0
    then have ?thesis using prems linr nppos nnz intterm-novar0[OF lininrp]
        by (cases n, simp-all)
      (simp add: nth-pos2 linterm-novar0[OF linr, where x=a and y=a*l])
    }
    moreover
    {
      assume nz: n=0
      from prems have inz: i ≠ 0 by auto
      from prems nz have idvdl: i dvd l by simp
      have (i * a + (i' * (a # ats) ! n' + I-intterm (a # ats) r) = 0) =
        (l * a + l div i * (i' * (a # ats) ! n' + I-intterm (a # ats) r) = 0)
        by (rule adjustcoeff-eq-corr[OF lpos inz idvdl])
      then
      have ?thesis using prems linr linr2
        by (simp add: mult-ac nth-pos2 lin-mul-corr
          linterm-novar0[OF linr, where x=a and y=a*l])
    }
```

ultimately show *?thesis* **by** *blast*
qed
qed *simp-all*
then show *?case* **by** *simp*

qed

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

have *lpos*: *0* < *formlcm p* **by** (*rule formlcm-pos*[*OF linp*])
have *dvd* : *divideallc* (*formlcm p*, *p*) **by** (*rule formlcm-divideallc*[*OF linp*])
show *?thesis* **using** *prems lpos dvd*
proof (*simp add*: *unitycoeff-def Let-def*,*case-tac formlcm p* = *1*,
simp-all add: *adjustcoeff-corr*)
show ($\exists$ *x*. *qinterp* (*x* ∗ *formlcm p* # *ats*) (*adjustcoeff* (*formlcm p*, *p*))) =
($\exists$ *x*. *formlcm p dvd x* $\wedge$
qinterp (*x* # *ats*) (*adjustcoeff* (*formlcm p*, *p*)))
(**is** ($\exists$ *x*. *?P*(*x*∗ (*formlcm p*))) = ($\exists$ *x*. *formlcm p dvd x* $\wedge$ *?P x*))
proof−
have ($\exists$ *x*. *?P*(*x*∗ (*formlcm p*))) = ($\exists$ *x*. *?P*((*formlcm p*)∗*x*))
by (*simp add*: *mult-commute*)
also have ($\exists$ *x*. *?P*((*formlcm p*)∗*x*)) = ($\exists$ *x*. (*formlcm p dvd x*) $\wedge$ *?P x*)
by (*simp add*: *unity-coeff-ex*[**where** *P*=*?P*])
finally show *?thesis* **by** *simp*
qed
qed
qed

lemma *adjustcoeff-unified*:
assumes *linp*: *islinform p*
and *dvdc*: *divideallc*(*l*,*p*)
and *lpos*: *l* > *0*
shows *isunified* (*adjustcoeff*(*l*, *p*))
using *linp dvdc lpos*
proof(*induct l p rule*: *adjustcoeff.induct*,*simp-all add*: *lin-mul-lintn islinintterm-eq-islint islint-def*)
case (*goal1 l d c r*)
from *prems* **have** *c* >*0* $\vee$ *c* < *0* **by** *auto*
moreover {
assume *cpos*: *c* > *0*
from *prems* **have** *lp*: *l* > *0* **by** *simp*
from *prems* **have** *cdvdl*: *c dvd l* **by** *simp*
have *clel*: *c* ≤ *l* **by** (*rule zdvd-imp-le*[*OF cdvdl lp*])
have *c div c* ≤ *l div c* **by** (*rule zdiv-mono1*[*OF clel cpos*])

then have *?case* **using** *cpos* **by** (*simp add*: *zdiv-self*)
}
moreover {
assume *cneg*: $c < 0$

have *mcpos*: $-c > 0$ **by** *simp*
then have *mcnz*: $-c \neq 0$ **by** *simp*
from *prems* **have** *mcdvdl*: $-c$ *dvd* l
by *simp*
then have *l1*:l *mod* $-c = 0$ **by** (*simp add*: *zdvd-iff-zmod-eq-0*)
from *prems* **have** *lp*: $l > 0$ **by** *simp*
have *mclel*: $-c \leq l$ **by** (*rule zdvd-imp-le*[*OF mcdvdl lp*])
have l *div* $c = (-l$ *div* $-c)$ **by** *simp*
also have $\ldots = -$ (l *div* $-c$) **using** *l1*
by (*simp only*: *zdiv-zminus1-eq-if*[*OF mcnz*, **where** *a*=*l*]) *simp*
finally have *diveq*: l *div* $c = -$ (l *div* $-c$) **by** *simp*

have $-c$ *div* $-c \leq l$ *div* $-c$ **by** (*rule zdiv-mono1*[*OF mclel mcpos*])
then have $0 < l$ *div* $-c$ **using** *cneg*
by (*simp add*: *zdiv-self*)
then have *?case* **using** *diveq* **by** *simp*
}
ultimately show *?case* **by** *blast*
next
case (*goal2 l p*) **from** *prems* **have** $c > 0 \vee c < 0$ **by** *auto*
moreover {
assume *cpos*: $c > 0$
from *prems* **have** *lp*: $l > 0$ **by** *simp*
from *prems* **have** *cdvdl*: c *dvd* l **by** *simp*
have *clel*: $c \leq l$ **by** (*rule zdvd-imp-le*[*OF cdvdl lp*])
have c *div* $c \leq l$ *div* c **by** (*rule zdiv-mono1*[*OF clel cpos*])
then have *?case* **using** *cpos* **by** (*simp add*: *zdiv-self*)
}
moreover {
assume *cneg*: $c < 0$

have *mcpos*: $-c > 0$ **by** *simp*
then have *mcnz*: $-c \neq 0$ **by** *simp*
from *prems* **have** *mcdvdl*: $-c$ *dvd* l
by *simp*
then have *l1*:l *mod* $-c = 0$ **by** (*simp add*: *zdvd-iff-zmod-eq-0*)
from *prems* **have** *lp*: $l > 0$ **by** *simp*
have *mclel*: $-c \leq l$ **by** (*rule zdvd-imp-le*[*OF mcdvdl lp*])
have l *div* $c = (-l$ *div* $-c)$ **by** *simp*
also have $\ldots = -$ (l *div* $-c$) **using** *l1*
by (*simp only*: *zdiv-zminus1-eq-if*[*OF mcnz*, **where** *a*=*l*]) *simp*
finally have *diveq*: l *div* $c = -$ (l *div* $-c$) **by** *simp*

have $-c$ *div* $-c \leq l$ *div* $-c$ **by** (*rule zdiv-mono1*[*OF mclel mcpos*])

then have $0 < l$ *div* $-c$ **using** *cneg*
by (*simp add*: *zdiv-self*)
then have *?case* **using** *diveq* **by** *simp*
}
ultimately show *?case* **by** *blast*
qed

lemma *adjustcoeff-lcm-unified*:
assumes *linp*: *islinform p*
shows *isunified* (*adjustcoeff*(*formlcm p*, *p*))
using *linp adjustcoeff-unified formlcm-pos formlcm-divideallc*
by *simp*

lemma *unitycoeff-unified*:
assumes *linp*: *islinform p*
shows *isunified* (*unitycoeff p*)
using *linp formlcm-pos*[*OF linp*]
proof (*auto simp add*: *unitycoeff-def Let-def adjustcoeff-lcm-unified*)
assume *f1*: *formlcm p* = *1*
have *isunified* (*adjustcoeff*(*formlcm p*, *p*))
by (*rule adjustcoeff-lcm-unified*[*OF linp*])
with *f1*
show *isunified* (*adjustcoeff*(*1*,*p*)) **by** *simp*
qed

lemma *unified-isnnf*:
assumes *unifp*: *isunified p*
shows *isnnf p*
using *unified-islinform*[*OF unifp*] *linform-isnnf*
by *simp*

lemma *unified-isqfree*: *isunified p* $\Longrightarrow$ *isqfree p*
using *unified-islinform linform-isqfree*
by *auto*

consts *minusinf* :: *QF* $\Rightarrow$ *QF*
plusinf :: *QF* $\Rightarrow$ *QF*
aset :: *QF* $\Rightarrow$ *intterm list*
bset :: *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-add*(*r*,(*Cst* −*1*)), *r*]
else [*lin-add*(*lin-neg r*,(*Cst* −*1*))])
bset (*Eq* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*) =
(*if c* < *0 then* [*lin-add*(*r*,(*Cst* −*1*))]
else [*lin-add*(*lin-neg r*,(*Cst* −*1*))])
bset (*NOT*(*Eq* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*)) =
(*if c* < *0 then* [*r*]
else [*lin-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-add* (*r*, *Cst 1*)]
else [*lin-add* (*lin-neg r*, *Cst 1*), *lin-neg r*])
aset (*Eq* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*) =
(*if c* < *0 then* [*lin-add*(*r*,(*Cst 1*))]
else [*lin-add*(*lin-neg r*,(*Cst 1*))])
aset (*NOT*(*Eq* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) *z*)) =
(*if c* < *0 then* [*r*]
else [*lin-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(*induct d p rule*: *alldivide.induct*, *simp-all add*: *ilcm-dvd1 ilcm-dvd2*)
  fix *d1 d2 d3*
  assume *th1*:*d2 dvd* (*d1*::*int*)
    and *th2*: *d1 dvd d3*
  show *d2 dvd d3* **by** (*rule zdvd-trans*[*OF th1 th2*])
qed

lemma *zdvd-eq-zdvd-abs*: (*d*::*int*) *dvd d′* = (*d dvd* (*abs d′*))
proof−
  have *d′* < *0* ∨ *d′* ≥ *0* **by** *arith*
  moreover
  {
    assume *dn′*: *d′* < *0*
    then have *abs d′* = − *d′* **by** *simp*
    then
    have *?thesis* **by** (*simp*)
  }
  moreover
  {
    assume *dp′*: *d′* ≥ *0*
    then have *abs d′* = *d′* **by** *simp*
    then have *?thesis* **by** *simp*
  }
    ultimately show *?thesis* **by** *blast*
qed

lemma *zdvd-refl-abs*: (*d*::*int*) *dvd* (*abs d*)
proof−
  have *d dvd d* **by** *simp*
  then show *?thesis* **by** (*simp add*: *iffD1* [*OF zdvd-eq-zdvd-abs* [**where** *d* = *d*
and *d′*=*d*]])
qed

lemma *divlcm-pos*:
  assumes

linp: *islinform p*
shows *0* < *divlcm p*
using *linp*
proof (*induct p rule*: *divlcm.induct*,*simp-all add*: *ilcm-pos*)
case (*goal1 f*) **show** *?case*
using *prems*
by (*cases f*, *auto*) (*case-tac intterm1*, *auto*)
qed

lemma *nz-le*: (*x*::*int*) > *0* $\Longrightarrow$ *x* $\neq$ *0* **by** *auto*

lemma *divlcm-corr*:
assumes
linp: *islinform p*
shows *alldivide* (*divlcm p*,*p*)
using *linp divlcm-pos*
proof (*induct p rule*: *divlcm.induct*,*simp-all add*: *zdvd-refl-abs*,*clarsimp simp add*: *Nat.gr0-conv-Suc*)
case (*goal1 f*)
have *islinform f* **using** *prems*
by (*cases f*, *auto*) (*case-tac intterm2*, *auto*,*case-tac intterm1*, *auto*)
then have *alldivide* (*divlcm f*, *f*) **using** *prems* **by** *simp*
moreover have *divlcm* (*NOT f*) = *divlcm f* **by** *simp*
moreover have *alldivide* (*x*,*f*) = *alldivide* (*x*,*NOT f*) **by** *simp*
ultimately show *?case* **by** *simp*
next
case (*goal2 f g*)
have *dvd1*: (*divlcm f*) *dvd* (*ilcm* (*divlcm f*) (*divlcm g*))
using *prems* **by**(*simp add*: *ilcm-dvd1 nz-le*)
have *dvd2*: (*divlcm g*) *dvd* (*ilcm* (*divlcm f*) (*divlcm g*))
using *prems* **by** (*simp add*: *ilcm-dvd2 nz-le*)
from *dvd1 prems*
have *alldivide* (*ilcm* (*divlcm f*) (*divlcm g*), *f*)
by (*simp add*: *alldivide-mono*[**where** *d*= *divlcm f* **and** *p*=*f* **and** *d'* =*ilcm* (*divlcm f*) (*divlcm g*)])
moreover **from** *dvd2 prems*
have *alldivide* (*ilcm* (*divlcm f*) (*divlcm g*), *g*)
by (*simp add*: *alldivide-mono*[**where** *d*= *divlcm g* **and** *p*=*g* **and** *d'* =*ilcm* (*divlcm f*) (*divlcm g*)])
ultimately show *?case* **by** *simp*
next
case (*goal3 f g*)
have *dvd1*: (*divlcm f*) *dvd* (*ilcm* (*divlcm f*) (*divlcm g*))
using *prems* **by** (*simp add*: *nz-le ilcm-dvd1*)
have *dvd2*: (*divlcm g*) *dvd* (*ilcm* (*divlcm f*) (*divlcm g*))
using *prems* **by** (*simp add*: *nz-le ilcm-dvd2*)
from *dvd1 prems*
have *alldivide* (*ilcm* (*divlcm f*) (*divlcm g*), *f*)
by (*simp add*: *alldivide-mono*[**where** *d*= *divlcm f* **and** *p*=*f* **and** *d'* =*ilcm*

```
(divlcm f) (divlcm g)])
  moreover   from dvd2 prems
   have alldivide (ilcm (divlcm f) (divlcm g), g)
     by (simp add: alldivide-mono[where d= divlcm g and p=g and d' =ilcm
(divlcm f) (divlcm g)])
  ultimately show ?case by simp
qed
```

```
lemma minusinf-eq:
  assumes unifp: isunified p
  shows ∃ z. ∀ x. x < z ⟶ (qinterp (x#ats) p = qinterp (x#ats) (minusinf p))
using unifp unified-islinform[OF unifp]
proof (induct p rule: minusinf.induct)
  case (1 c r z)
  have c <0 ∨ 0 ≤ c by arith
  moreover
  {
    assume cneg:  c < 0
    from prems have z0: z= Cst 0
      by (cases z,auto)
    with prems have lincnr: islinintterm (Add (Mult (Cst c) (Var 0)) r)
      by simp

    from prems z0 have ?case
      proof−
        show ?thesis
          using prems z0
      apply auto
      apply (rule exI[where x=I-intterm (a # ats) r])
      apply (rule allI)
      proof−
        fix x
        show x < I-intterm (a # ats) r ⟶ ¬ − x + I-intterm (x # ats) r ≤ 0
          by (simp add: intterm-novar0[OF lincnr, where x=a and y=x])
      qed
    qed
  }
  moreover
  {
    assume cpos: 0 ≤ c
    from prems have z0: z= Cst 0
      by (cases z) auto
    with prems have lincnr: islinintterm (Add (Mult (Cst c) (Var 0)) r)
      by simp
```

```
    from prems z0 have ?case
      proof−
        show ?thesis
          using prems z0
      apply auto
      apply (rule exI[where x=−(I-intterm (a # ats) r)])
      apply (rule allI)
      proof−
        fix x
        show x < − I-intterm (a # ats) r ⟶ x + I-intterm (x # ats) r ≤ 0
          by (simp add: intterm-novar0[OF lincnr, where x=a and y=x])
      qed
    qed
  }

    ultimately show ?case by blast
next
  case (2 c r z)
  from prems have z0: z= Cst 0
    by (cases z,auto)
  with prems have lincnr: islinintterm (Add (Mult (Cst c) (Var 0)) r)
    by simp
  have c <0 ∨ 0 ≤ c by arith
  moreover
  {
    assume cneg:  c < 0
    from prems z0 have ?case
      proof−
        show ?thesis
          using prems z0
      apply auto
      apply (rule exI[where x=I-intterm (a # ats) r])
      apply (rule allI)
      proof−
        fix x
        show x < I-intterm (a # ats) r ⟶ ¬ − x + I-intterm (x # ats) r = 0
          by (simp add: intterm-novar0[OF lincnr, where x=a and y=x])
      qed
    qed
  }
  moreover
  {
    assume cpos: 0 ≤ c
    from prems z0 have ?case
      proof−
        show ?thesis
          using prems z0
      apply auto
      apply (rule exI[where x=−(I-intterm (a # ats) r)])
```

```
      apply (rule allI)
      proof−
        fix x
        show x < − I-intterm (a # ats) r ⟶ x + I-intterm (x # ats) r ≠ 0
          by (simp add: intterm-novar0[OF lincnr, where x=a and y=x])
      qed
    qed
  }

    ultimately show ?case by blast
next
  case (3 c r z)
  from prems have z0: z= Cst 0
    by (cases z,auto)
  with prems have lincnr: islinintterm (Add (Mult (Cst c) (Var 0)) r)
    by simp
  have c <0 ∨ 0 ≤ c by arith
  moreover
  {
    assume cneg:  c < 0
    from prems z0 have ?case
      proof−
        show ?thesis
          using prems z0
      apply auto
      apply (rule exI[where x=I-intterm (a # ats) r])
      apply (rule allI)
      proof−
        fix x
        show x < I-intterm (a # ats) r ⟶ ¬ − x + I-intterm (x # ats) r = 0
          by (simp add: intterm-novar0[OF lincnr, where x=a and y=x])
      qed
    qed
  }
  moreover
  {
    assume cpos: 0 ≤ c
    from prems z0 have ?case
      proof−
        show ?thesis
          using prems z0
      apply auto
      apply (rule exI[where x=−(I-intterm (a # ats) r)])
      apply (rule allI)
      proof−
        fix x
        show x < − I-intterm (a # ats) r ⟶ x + I-intterm (x # ats) r ≠ 0
          by (simp add: intterm-novar0[OF lincnr, where x=a and y=x])
      qed
```

qed
}

ultimately show *?case* **by** *blast*
next

case (*4 f g*)
from *prems* **obtain** *zf* **where**
zf:$\forall x<zf$. *qinterp* (*x* # *ats*) *f* = *qinterp* (*x* # *ats*) (*minusinf f*) **by** *auto*
from *prems* **obtain** *zg* **where**
zg:$\forall x<zg$. *qinterp* (*x* # *ats*) *g* = *qinterp* (*x* # *ats*) (*minusinf g*) **by** *auto*
from *zf zg* **show** *?case*
apply *auto*
apply (*rule exI*[**where** *x*=*min zf zg*])
by *simp*

next case (*5 f g*)
from *prems* **obtain** *zf* **where**
zf:$\forall x<zf$. *qinterp* (*x* # *ats*) *f* = *qinterp* (*x* # *ats*) (*minusinf f*) **by** *auto*
from *prems* **obtain** *zg* **where**
zg:$\forall x<zg$. *qinterp* (*x* # *ats*) *g* = *qinterp* (*x* # *ats*) (*minusinf g*) **by** *auto*
from *zf zg* **show** *?case*
apply *auto*
apply (*rule exI*[**where** *x*=*min zf zg*])
by *simp*

qed *simp-all*

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*)
using *alldvd unity unified-islinform*[*OF unity*]
proof(*induct p rule*: *islinform.induct*, *simp-all*)
case (*goal1 t a*)
show *?case*
using *prems*
apply (*cases t*, *simp-all add*: *nth-pos2*)
apply (*case-tac intterm1*, *simp-all*)
apply (*case-tac intterm1a*,*simp-all*)
by (*case-tac intterm2a*,*simp-all*)
(*case-tac nat*,*simp-all add*: *nth-pos2 intterm-novar0*[**where** *x*=*x* **and** *y*=*x*+*c*∗*d*])
next
case (*goal2 t a*)
show *?case*
using *prems*
apply (*cases t*, *simp-all add*: *nth-pos2*)
apply (*case-tac intterm1*, *simp-all*)

apply (*case-tac intterm1a*,*simp-all*)
by (*case-tac intterm2a*,*simp-all*)
(*case-tac nat*,*simp-all add*: *nth-pos2 intterm-novar0*[**where** $x=x$ **and** $y=x+c*d$])
next
case (*goal3 a t*)
show *?case* **using** *prems*

proof(*induct t rule*: *islinintterm.induct*, *simp-all add*: *nth-pos2*)
case (*goal1 i n i'*)
show *?case*
using *prems*
proof(*cases n*, *simp-all*, *case-tac* $i=1$, *simp*,
simp add: *dvd-period*[**where** $a=a$ **and** $d=d$ **and** $x=x$ **and** $c=c$])
case *goal1*
from *prems* **have** ($abs\ i = 1$) $\wedge$ $i \neq 1$ **by** *auto*
then have *im1*: $i=-1$ **by** *arith*
then have (a *dvd* $i*x + i'$) = (a *dvd* $x + (-i')$)
by (*simp add*: *uminus-dvd-conv'*[**where** $d=a$ **and** $t=-x+i'$])
moreover
from *im1* **have** (a *dvd* $i*x + (i*(c*d)) + i'$) = (a *dvd* ($x + c*d - i'$))
apply *simp*
apply (*simp add*: *uminus-dvd-conv'*[**where** $d=a$ **and** $t=-x-c*d+i'$])
by (*simp add*: *zadd-ac*)
ultimately
have *eq1*:((a *dvd* $i*x + i'$) = (a *dvd* $i*x + (i*(c*d)) + i'$)) =
((a *dvd* $x + (-i')$) = (a *dvd* ($x + c*d - i'$))) **by** *simp*
moreover
have *dvd2*: (a *dvd* $x + (-i')$) = (a *dvd* $x + c*d + (-i')$)
by (*rule dvd-period*[**where** $a=a$ **and** $d=d$ **and** $x=x$ **and** $c=c$], *assumption*)
ultimately show *?case* **by** *simp*
qed
next
case (*goal2 i n i' n' r*)
have $n = 0 \vee 0 < n$ **by** *arith*
moreover
{
assume *npos*: $0 < n$
from *prems* **have** $n < n'$ **by** *simp* **then have** $0 < n'$ **by** *simp*
moreover from *prems*
have *linr*: *islinintterm* (*Add* (*Mult* (*Cst* i') (*Var* n')) r) **by** *simp*
ultimately have *?case*
using *prems npos*
by (*simp add*: *nth-pos2 intterm-novar0*[*OF linr*,**where** $x=x$ **and** $y=x+c*d$])
}
moreover
{
assume *n0*: $n=0$
from *prems* **have** *lin2*: *islinintterm* (*Add* (*Mult* (*Cst* i') (*Var* n')) r) **by** *simp*

from *prems* **have** $n < n'$ **by** *simp* **then have** *npos'*: $0 < n'$ **by** *simp*
with *prems* **have** *?case*
proof(*simp add*: *intterm-novar0*[*OF lin2*, **where** $x=x$ **and** $y=x+c*d$]
nth-pos2 dvd-period,*case-tac* $i=1$,
simp add: *dvd-period*[**where** $a=a$ **and** $d=d$ **and** $x=x$ **and** $c=c$], *simp*)
case *goal1*
from *prems* **have** *abs* $i = 1 \wedge i \neq 1$ **by** *auto*
then have *mi*: $i = -1$ **by** *arith*
have (a *dvd* $-x + (i' * ats\ !\ (n' - Suc\ 0) + I\text{-}intterm\ ((x + c * d)\ \#\ ats)$
$r)) =$
(a *dvd* $x + (-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#\ ats)\ r))$
by (*simp add*:
uminus-dvd-conv'[**where** $d=a$ **and**
$t=-x + (i' * ats\ !\ (n' - Suc\ 0) + I\text{-}intterm\ ((x + c * d)\ \#\ ats)\ r)$])
also
have (a *dvd* $x + (-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#\ ats)$
$r)) =$
(a *dvd* $x + c*d + (-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#$
$ats)\ r))$
by (*rule dvd-period*[**where** $a=a$ **and** $d=d$ **and** $x=x$ **and** $c=c$], *assumption*)
also
have (a *dvd* $x + c*d +$
$(-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#\ ats)\ r)) =$
(a *dvd* $-(x + c*d +$
$(-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#\ ats)\ r)))$
by (*rule uminus-dvd-conv'*[**where** $d=a$ **and**
$t=x + c*d + (-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#\ ats)$
$r)$])
also
have (a *dvd* $-(x + c*d +$
$(-i' * ats\ !\ (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d)\ \#\ ats)\ r)))$
$=$ (a *dvd*
$- x - c * d + (i' * ats\ !\ (n' - Suc\ 0) + I\text{-}intterm\ ((x + c * d)\ \#\ ats)$
$r))$
by (*auto*,*simp-all add*: *zadd-ac*)
finally show *?case* **using** *mi* **by** *auto*
qed
}
ultimately show *?case* **by** *blast*
qed
next
case (*goal4 a t*)
show *?case* **using** *prems*
proof(*induct t rule*: *islinintterm.induct*, *simp-all*,*case-tac* $n=0$,
simp-all add: *nth-pos2*)
case (*goal1 i n i'*)
show *?case*
using *prems*
proof(*case-tac* $i=1$, *simp*,

simp add: *dvd-period*[**where** *a=a* **and** *d=d* **and** *x=x* **and** *c=c*])
case *goal1*
from *prems* **have** *abs i = 1 ∧ i≠1* **by** *auto*
then have *im1*: *i=−1* **by** *arith*
then have (*a dvd i∗x + i′*) = (*a dvd x +* (*−i′*))
by (*simp add*: *uminus-dvd-conv′*[**where** *d=a* **and** *t=−x +i′*])
moreover
from *im1* **have** (*a dvd i∗x +* (*i∗*(*c ∗ d*)) *+ i′*) = (*a dvd* (*x + c∗d − i′*))
apply *simp*
apply (*simp add*: *uminus-dvd-conv′*[**where** *d=a* **and** *t=−x − c ∗ d + i′*])
by (*simp add*: *zadd-ac*)
ultimately
have *eq1*:((*a dvd i∗x + i′*) = (*a dvd i∗x +* (*i∗*(*c ∗ d*)) *+ i′*)) =
((*a dvd x +* (*−i′*)) = (*a dvd* (*x + c∗d − i′*))) **by** *simp*
moreover
have *dvd2*: (*a dvd x +* (*−i′*)) = (*a dvd x + c ∗ d +* (*−i′*))
by (*rule dvd-period*[**where** *a=a* **and** *d=d* **and** *x=x* **and** *c=c*], *assumption*)
ultimately show *?thesis* **by** *simp*
qed
next
case (*goal2 i n i′ n′ r*)
have *n = 0 ∨ 0 < n* **by** *arith*
moreover
{
assume *npos*: *0 < n*
from *prems* **have** *n < n′* **by** *simp* **then have** *0 < n′* **by** *simp*
moreover from *prems*
have *linr*: *islinintterm* (*Add* (*Mult* (*Cst i′*) (*Var n′*)) *r*) **by** *simp*
ultimately have *?case*
using *prems npos*
by (*simp add*: *nth-pos2 intterm-novar0*[*OF linr*,**where** *x=x* **and** *y=x + c∗d*])
}
moreover
{
assume *n0*: *n=0*
from *prems* **have** *lin2*: *islinintterm* (*Add* (*Mult* (*Cst i′*) (*Var n′*)) *r*) **by** *simp*
from *prems* **have** *n < n′* **by** *simp* **then have** *npos′*: *0 < n′* **by** *simp*
with *prems* **have** *?case*
proof(*simp add*: *intterm-novar0*[*OF lin2*, **where** *x=x* **and** *y=x+c∗d*]
nth-pos2 dvd-period,*case-tac i=1*,
simp add: *dvd-period*[**where** *a=a* **and** *d=d* **and** *x=x* **and** *c=c*], *simp*)
case *goal1*
from *prems* **have** *abs i = 1 ∧ i≠1* **by** *auto*
then have *mi*: *i = −1* **by** *arith*
have (*a dvd −x +* (*i′ ∗ ats !* (*n′ − Suc 0*) *+ I-intterm* ((*x + c ∗ d*) *# ats*) *r*)) =
(*a dvd x +* (*−i′ ∗ ats !* (*n′ − Suc 0*) *− I-intterm* ((*x + c ∗ d*) *# ats*) *r*))
by (*simp add*:

uminus-dvd-conv'[**where** $d=a$ **and**
$t=-x + (i' * ats ! (n' - Suc\ 0) + I\text{-}intterm\ ((x + c * d) \# ats)\ r)$])
also
have (a *dvd* $x + (-i' * ats ! (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d) \# ats)\ r)) =$
(a *dvd* $x + c*d + (-i' * ats ! (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d) \# ats)\ r))$
by (*rule dvd-period*[**where** $a=a$ **and** $d=d$ **and** $x=x$ **and** $c=c$], *assumption*)
also
have (a *dvd* $x + c*d +$
$(-i' * ats ! (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d) \# ats)\ r)) =$
(a *dvd* $-(x + c*d +$
$(-i' * ats ! (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d) \# ats)\ r)))$
by (*rule uminus-dvd-conv'*[**where** $d=a$ **and**
$t=x + c*d + (-i' * ats ! (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d) \# ats)\ r)$])
also
have (a *dvd* $-(x + c*d +$
$(-i' * ats ! (n' - Suc\ 0) - I\text{-}intterm\ ((x + c * d) \# ats)\ r)))$
$=$ (a *dvd*
$- x - c * d + (i' * ats ! (n' - Suc\ 0) + I\text{-}intterm\ ((x + c * d) \# ats)\ r))$
by (*auto*,*simp-all add*: *zadd-ac*)
finally show *?case* **using** *mi* **by** *auto*
qed
}
ultimately show *?case* **by** *blast*
qed
next
case (*goal5 t a*)
show *?case*
using *prems*
apply (*cases t*, *simp-all add*: *nth-pos2*)
apply (*case-tac intterm1*, *simp-all*)
apply (*case-tac intterm1a*,*simp-all*)
by (*case-tac intterm2a*,*simp-all*)
(*case-tac nat*,*simp-all add*: *nth-pos2 intterm-novar0*[**where** $x=x$ **and** $y=x+c*d$])
qed

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(*rule allI*, *rule allI*)
fix x k
show $?P\ x = ?P\ (x - k*d)$
proof−

have *?P x* = *?P* (*x* + (−*k*)∗*d*) **by** (*rule minusinf-repeats*[*OF alldvd unity*])
then have *?P x* = *?P* (*x* − (*k*∗*d*)) **by** *simp*
then show *?thesis* **by** *blast*
qed
qed

lemma *minusinf-lemma*:
assumes *unifp*: *isunified p*
and *exminf*: ∃ *j* ∈ {*1* ..*d*}. *qinterp* (*j*#*ats*) (*minusinf p*) (**is** ∃ *j* ∈ {*1* .. *d*}. *?P1 j*)
shows ∃ *x*. *qinterp* (*x*#*ats*) *p* (**is** ∃ *x*. *?P x*)
proof−
from *exminf* **obtain** *j* **where** *P1j*: *?P1 j* **by** *blast*
have *ePeqP1*: ∃*z*. ∀ *x*. *x* < *z* ⟶ (*?P x* = *?P1 x*)
by (*rule minusinf-eq*[*OF unifp*])
then obtain *z* **where** *P1eqP* : ∀ *x*. *x* < *z* ⟶ (*?P x* = *?P1 x*) **by** *blast*
let *?d* = *divlcm p*
have *alldvd*: *alldivide* (*?d*,*p*) **using** *unified-islinform*[*OF unifp*] *divlcm-corr*
by *auto*
have *dpos*: *0* < *?d* **using** *unified-islinform*[*OF unifp*] *divlcm-pos*
by *simp*
have *P1eqP1* : ∀ *x k*. *?P1 x* = *?P1* (*x* − *k*∗(*?d*))
by (*rule minusinf-repeats2*[*OF alldvd unifp*])
let *?w* = *j* − (*abs* (*j*−*z*) +*1*)∗ *?d*
show ∃ *x*. *?P x*
proof
have *w*: *?w* < *z*
by (*rule decr-lemma*[*OF dpos*])

have *?P1 j* = *?P1 ?w* **using** *P1eqP1* **by** *blast*
also have . . . = *?P ?w* **using** *w P1eqP* **by** *blast*
finally show *?P ?w* **using** *P1j* **by** *blast*
qed
qed

lemma *minusinf-disj*:
assumes *unifp*: *isunified p*
shows (∃ *x*. *qinterp* (*x*#*ats*) (*minusinf p*)) =
(∃ *j* ∈ { *1*.. *divlcm p*}. *qinterp* (*j*#*ats*) (*minusinf p*))
(**is** (∃ *x*. *?P x*) = (∃ *j* ∈ { *1*.. *?d*}. *?P j*))
proof
have *linp*: *islinform p* **by** (*rule unified-islinform*[*OF unifp*])
have *dpos*: *0* < *?d* **by** (*rule divlcm-pos*[*OF linp*])
have *alldvd*: *alldivide*(*?d*,*p*) **by** (*rule divlcm-corr*[*OF linp*])
{
assume ∃ *j*∈ {*1* .. *?d*}. *?P j*

then show ∃ *x*. *?P x* **using** *dpos* **by** *auto*
next
assume ∃ *x*. *?P x*
then obtain *x* **where** *P*: *?P x* **by** *blast*
have *modd*: ∀ *x k*. *?P x* = *?P* (*x* − *k*∗*?d*)
by (*rule minusinf-repeats2*[*OF alldvd unifp*])

have *x mod ?d* = *x* − (*x div ?d*)∗*?d*
by(*simp add*:*zmod-zdiv-equality mult-ac eq-diff-eq*)
hence *Pmod*: *?P x* = *?P* (*x mod ?d*) **using** *modd* **by** *simp*
show ∃ *j*∈ {*1* .. *?d*}. *?P j*
proof (*cases*)
assume *x mod ?d* = *0*
hence *?P 0* **using** *P Pmod* **by** *simp*
moreover have *?P 0* = *?P* (*0* − (−*1*)∗*?d*) **using** *modd* **by** *blast*
ultimately have *?P ?d* **by** *simp*
moreover have *?d* ∈ {*1* .. *?d*} **using** *dpos*
by (*simp add*:*atLeastAtMost-iff*)
ultimately show ∃ *j*∈ {*1* .. *?d*}. *?P j* **..**
next
assume *not0*: *x mod ?d* ≠ *0*
have *?P*(*x mod ?d*) **using** *dpos P Pmod* **by**(*simp add*:*pos-mod-sign pos-mod-bound*)
moreover have *x mod ?d* : {*1* .. *?d*}
proof −
have *0* ≤ *x mod ?d* **by**(*rule pos-mod-sign*[*OF dpos*])
moreover have *x mod ?d* < *?d* **by**(*rule pos-mod-bound*[*OF dpos*])
ultimately show *?thesis* **using** *not0* **by**(*simp add*:*atLeastAtMost-iff*)
qed
ultimately show ∃ *j*∈ {*1* .. *?d*}. *?P j* **..**
qed
}
qed

lemma *minusinf-qfree*:
assumes *linp* : *islinform p*
shows *isqfree* (*minusinf p*)
using *linp*
by (*induct p rule*: *minusinf.induct*) *auto*

lemma *bset-lin*:
assumes *unifp*: *isunified p*
shows ∀ *b* ∈ *set* (*bset p*). *islinintterm b*
using *unifp unified-islinform*[*OF unifp*]
proof (*induct p rule*: *bset.induct*, *auto*)
case (*goal1 c r z*)
from *prems* **have** *z* = *Cst 0* **by** (*cases z*, *simp-all*)

then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *islinintterm* (*Cst* −*1*) **by** *simp*
then show *?case* **using** *linr lin-add-lin* **by** *simp*
next
case (*goal2 c r z*)
from *prems* **have** $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
show *?case* **by** (*rule linr*)
next
case (*goal3 c r z*)
from *prems* **have** $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *islinintterm* (*Cst* −*1*) **by** *simp*
then show *?case* **using** *linr lin-add-lin lin-neg-lin* **by** *simp*
next
case (*goal4 c r z*)
from *prems* **have** $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *islinintterm* (*Cst* −*1*) **by** *simp*
then show *?case* **using** *linr lin-add-lin lin-neg-lin* **by** *simp*
next
case (*goal5 c r z*)
from *prems* **have** $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *islinintterm* (*Cst* −*1*) **by** *simp*
then show *?case* **using** *linr lin-add-lin lin-neg-lin* **by** *simp*
next
case (*goal6 c r z*)
from *prems* **have** $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *islinintterm* (*Cst* −*1*) **by** *simp*
then show *?case* **using** *linr lin-add-lin lin-neg-lin* **by** *simp*
next
case (*goal7 c r z*)
from *prems* **have** $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then have *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **using** *prems* **by** *simp*

have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *islinintterm* (*Cst −1*) **by** *simp*
then show *?case* **using** *linr lin-add-lin lin-neg-lin* **by** *simp*
qed

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*)
using *unifp nob alldvd unified-islinform*[*OF unifp*]
proof (*induct p rule*: *islinform.induct*,*auto*)
case (*goal1 t*)
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have (∃ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*) ∨ (∃ *i*. *t* = *Cst i*)
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ **assume** ∃ *i*. *t* = *Cst i* **then have** *?case* **using** *prems* **by** *auto* }
moreover
{ **assume** ∃ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
then obtain *i n r* **where**
inr-def: *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
by *blast*
with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])
have *n=0* ∨ *n>0* **by** *arith*
moreover {**assume** *n>0* **then have** *?case*
using *prems*
by (*simp add*: *nth-pos2*
intterm-novar0[*OF lininr*, **where** *x=x* **and** *y=x−d*]) }
moreover
{**assume** *nz*: *n* = *0*
from *prems* **have** *abs i* = *1* **by** *auto*
then have *i* = *−1* ∨ *i* =*1* **by** *arith*
moreover
{
assume *i1*: *i=1*
have *?case* **using** *dpos prems*
by (*auto simp add*: *intterm-novar0*[*OF lininr*, **where** *x=x* **and** *y=x − d*])
}
moreover
{

assume *im1*: $i = -1$
have *?case*
using *prems*
proof(*auto simp add*: *intterm-novar0*[*OF lininr*, **where** $x=x - d$ **and** $y=x$], *cases*)
assume $- x + d + ?I\ x\ r \leq 0$
then show $- x + d + ?I\ x\ r \leq 0$ **.**
next
assume *np*: $\neg - x + d + ?I\ x\ r \leq 0$
then have *ltd*:$x - ?I\ x\ r \leq d - 1$ **by** *simp*
from *prems* **have** $-x + ?I\ x\ r \leq 0$ **by** *simp*
then have *ge0*: $x - ?I\ x\ r \geq 0$
by *simp*
from *ltd ge0* **have** $x - ?I\ x\ r = 0 \vee (1 \leq x - ?I\ x\ r \wedge x - ?I\ x\ r \leq d - 1)$ **by** *arith*
moreover
{
assume $x - ?I\ x\ r = 0$
then have *xeqr*: $x = ?I\ x\ r$ **by** *simp*
from *prems* **have** $?Q\ x$ **by** *simp*
with *xeqr* **have** *qr*:$?Q\ (?I\ x\ r)$ **by** *simp*
from *prems* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var 0*)) *r*) **by** *simp*
have *islinintterm r* **by** (*rule islinintterm-subt*[*OF lininr*])
from *prems*
have $\forall j \in \{1..d\}.\ \neg\ ?Q\ (?I\ a\ r + -1 + j)$
using *linr* **by** (*auto simp add*: *lin-add-corr*)
moreover from *dpos* **have** $1 \in \{1..d\}$ **by** *simp*
ultimately have $\neg\ ?Q\ (?I\ a\ r + -1 + 1)$ **by** *blast*
with *dpos linr* **have** $\neg\ ?Q\ (?I\ x\ r)$
by (*simp add*: *intterm-novar0*[*OF lininr*, **where** $x=x$ **and** $y=a$] *lin-add-corr*)
with *qr* **have** $- x + d + ?I\ x\ r \leq 0$ **by** *simp*
}
moreover
{
assume *gt0*: $1 \leq x - ?I\ x\ r \wedge x - ?I\ x\ r \leq d - 1$
then have $\exists\ j \in \{1\ ..\ d - 1\}.\ x - ?I\ x\ r = j$ **by** *simp*
then have $\exists\ j \in \{1\ ..\ d\}.\ x - ?I\ x\ r = j$ **by** *auto*
then obtain j **where** *con*: $1 \leq j \wedge j \leq d \wedge x - ?I\ x\ r = j$ **by** *auto*
then have *xeqr*: $x = ?I\ x\ r + j$ **by** *auto*
with *prems* **have** $?Q\ (?I\ x\ r + j)$ **by** *simp*
with *con* **have** *qrpj*: $\exists\ j \in \{1\ ..\ d\}.\ ?Q\ (?I\ x\ r + j)$ **by** *auto*
from *prems* **have** $\forall j \in \{1..d\}.\ \neg\ ?Q\ (?I\ a\ r + j)$ **by** *auto*
then have $\neg\ (\exists\ j \in \{1..d\}.\ ?Q\ (?I\ x\ r + j))$
by (*simp add*: *intterm-novar0*[*OF lininr*, **where** $x=x$ **and** $y=a$])
with *qrpj prems* **have** $- x + d + ?I\ x\ r \leq 0$ **by** *simp*

}

```
        ultimately show − x + d + ?I x r ≤ 0 by blast
       qed
      }
      ultimately have ?case by blast
    }
    ultimately have ?case by blast
  }
  ultimately show ?case by blast
next
  case (goal3 a t)
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have n=0 ∨ n>0 by arith
    moreover {assume n>0 then have ?case using prems
        by (simp add: nth-pos2
          intterm-novar0[OF lininr, where x=x and y=x−d]) }
    moreover {
      assume nz: n=0
      from prems have abs i = 1 by auto
      then have ipm: i=1 ∨ i = −1 by arith
      from nz prems have advdixr: a dvd (i ∗ x) + I-intterm (x # ats) r
        by simp
      from prems have a dvd d by simp
      then have advdid: a dvd i∗d using ipm by auto
      have ?case
      using prems ipm
        by (auto simp add: intterm-novar0[OF lininr, where x=x−d and y=x]
dvd-period[OF advdid, where x=i∗x and c=−1])
  }
  ultimately have ?case by blast
  } ultimately show ?case by blast
next

  case (goal4 a t)
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
```

by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ **assume** $\exists$ *i*. *t* = *Cst i* **then have** *?case* **using** *prems* **by** *auto* }
moreover
{ **assume** $\exists$ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
then obtain *i n r* **where**
inr-def: *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
by *blast*
with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])

have *n=0* $\vee$ *n>0* **by** *arith*
moreover {**assume** *n>0* **then have** *?case* **using** *prems*
by (*simp add*: *nth-pos2*
intterm-novar0[*OF lininr*, **where** *x=x* **and** *y=x−d*]) }
moreover {
assume *nz*: *n=0*
from *prems* **have** *abs i* = *1* **by** *auto*
then have *ipm*: *i* =*1* $\vee$ *i* = −*1* **by** *arith*
from *nz prems* **have** *advdixr*: ¬ (*a dvd* (*i* ∗ *x*) + *I-intterm* (*x* # *ats*) *r*)
by *simp*
from *prems* **have** *a dvd d* **by** *simp*
then have *advdid*: *a dvd i∗d* **using** *ipm* **by** *auto*
have *?case*
using *prems ipm*
by (*auto simp add*: *intterm-novar0*[*OF lininr*, **where** *x=x−d* **and** *y=x*]
dvd-period[*OF advdid*, **where** *x=i∗x* **and** *c=−1*])
}
ultimately have *?case* **by** *blast*
} **ultimately show** *?case* **by** *blast*
next
case (*goal2 t*)
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have ($\exists$ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*) $\vee$ ($\exists$ *i*. *t* = *Cst i*)
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ **assume** $\exists$ *i*. *t* = *Cst i* **then have** *?case* **using** *prems* **by** *auto* }
moreover
{ **assume** $\exists$ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
then obtain *i n r* **where**
inr-def: *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
by *blast*
with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])
have *n=0* $\vee$ *n>0* **by** *arith*
moreover {**assume** *n>0* **then have** *?case*

using *prems*
by (*simp add*: *nth-pos2*
intterm-novar0[*OF lininr*, **where** $x{=}x$ **and** $y{=}x{-}d$]) **}**
moreover
{assume *nz*: $n = 0$
from *prems* **have** *abs* $i = 1$ **by** *auto*
then have $i = -1 \lor i = 1$ **by** *arith*
moreover
{
assume *i1*: $i{=}1$
with *prems* **have** *px*: $x + ?I\ x\ r = 0$ **by** *simp*
then have $x = (-\ ?I\ x\ r - 1) + 1$ **by** *simp*
hence *q1*: $?Q\ ((-\ ?I\ x\ r - 1) + 1)$ **by** *simp*
from *prems* **have** $\neg\ (?Q\ ((?I\ a\ (\mathit{lin\text{-}add}(\mathit{lin\text{-}neg}\ r,\ \mathit{Cst}\ -1))) + 1))$
by *auto*
hence $\neg\ (?Q\ ((-\ ?I\ a\ r - 1) + 1))$
using *lin-add-corr lin-neg-corr linr lin-neg-lin*
by *simp*
hence $\neg\ (?Q\ ((-\ ?I\ x\ r - 1) + 1))$
using *intterm-novar0*[*OF lininr*, **where** $x{=}x$ **and** $y{=}a$]
by *simp*
with *q1* **have** *?case* **by** *simp*
}
moreover
{
assume *im1*: $i = -1$
with *prems* **have** *px*: $-x + ?I\ x\ r = 0$ **by** *simp*
then have $x = ?I\ x\ r$ **by** *simp*
hence *q1*: $?Q\ (?I\ x\ r)$ **by** *simp*
from *prems* **have** $\neg\ (?Q\ ((?I\ a\ (\mathit{lin\text{-}add}(r,\ \mathit{Cst}\ -1))) + 1))$
by *auto*
hence $\neg\ (?Q\ (?I\ a\ r))$
using *lin-add-corr lin-neg-corr linr lin-neg-lin*
by *simp*
hence $\neg\ (?Q\ (?I\ x\ r\ ))$
using *intterm-novar0*[*OF lininr*, **where** $x{=}x$ **and** $y{=}a$]
by *simp*
with *q1* **have** *?case* **by** *simp*
}
ultimately have *?case* **by** *blast*
}
ultimately have *?case* **by** *blast*
}
ultimately show *?case* **by** *blast*
next
case (*goal5 t*)
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have $(\exists\ i\ n\ r.\ t = \mathit{Add}\ (\mathit{Mult}\ (\mathit{Cst}\ i)\ (\mathit{Var}\ n)\ )\ r) \lor (\exists\ i.\ t = \mathit{Cst}\ i)$

```
  by (induct t rule: islinintterm.induct) auto
moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
moreover
{ assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
  then obtain i n r where
    inr-def: t = Add (Mult (Cst i) (Var n) ) r
    by blast
  with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
    by simp
  have linr: islinintterm r
    by (rule islinintterm-subt[OF lininr])
  have n=0 ∨ n>0 by arith
  moreover {assume n>0 then have ?case
      using prems
      by (simp add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=x−d]) }
  moreover
  {assume nz: n = 0
    from prems have abs i = 1 by auto
    then have i = −1 ∨ i =1 by arith
    moreover
    {
      assume i1: i=1
      with prems have px: x −d + ?I (x−d) r = 0 by simp
      hence x = (− ?I x r) + d
        using intterm-novar0[OF lininr, where x=x and y=x−d]
        by simp
      hence q1: ?Q (− ?I x r + d) by simp
      from prems have ¬ (?Q ((?I a (lin-neg r)) + d))
        by auto
      hence ¬ (?Q (− ?I a r + d))
        using lin-neg-corr linr by simp
      hence ¬ (?Q ((− ?I x r + d)))
        using intterm-novar0[OF lininr, where x=x and y=a]
        by simp
      with q1 have  ?case by simp
    }
    moreover
    {
      assume im1: i = −1
      with prems have px: − (x −d) + ?I (x − d) r = 0 by simp
      then have x = ?I x r + d
        using intterm-novar0[OF lininr, where x=x and y=x−d]
        by simp
      hence q1: ?Q (?I x r + d) by simp
      from prems have ¬ (?Q ((?I a r) + d))
        by auto
      hence ¬ (?Q (?I x r + d))
        using intterm-novar0[OF lininr, where x=x and y=a]
```

by *simp*
with *q1* **have** *?case* **by** *simp*
}
ultimately have *?case* **by** *blast*
}
ultimately have *?case* **by** *blast*
}
ultimately show *?case* **by** *blast*

qed

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

shows $\forall$ *x*. $\neg(\exists j\in$ {*1* .. (*divlcm p*)}. $\exists$ *b* $\in$ *set* (*bset p*).
(*qinterp* (((*I-intterm* (*a#ats*) *b*) + *j*)#*ats*) *p*))
$\longrightarrow$ (*qinterp* (*x#ats*) *p*) $\longrightarrow$ (*qinterp* ((*x* − (*divlcm p*))#*ats*) *p*)
(**is** $\forall$ *x*. $\neg(\exists$ *j*$\in$ {*1* .. *?d*}. $\exists$ *b*$\in$ *?B*. *?P* (*?I a b* + *j*)) $\longrightarrow$ *?P x* $\longrightarrow$ *?P* (*x* − *?d*))
proof
fix *x*
have *linp*: *islinform p* **by** (*rule unified-islinform*[*OF unifp*])
have *dpos*: *?d* > *0* **by** (*rule divlcm-pos*[*OF linp*])
have *alldvd*: *alldivide*(*?d*,*p*) **by** (*rule divlcm-corr*[*OF linp*])
show $\neg(\exists$ *j*$\in$ {*1* .. *?d*}. $\exists$ *b*$\in$ *?B*. *?P* (*?I a b* + *j*)) $\longrightarrow$ *?P x* $\longrightarrow$ *?P* (*x* − *?d*)
using *prems bset-disj-repeat*[*OF unifp alldvd dpos*]
by *blast*
qed

lemma *cooper-mi-eq*:
assumes *unifp* : *isunified p*
shows ($\exists$ *x*. *qinterp* (*x#ats*) *p*) =
(($\exists$ *j* $\in$ {*1* .. (*divlcm p*)}. *qinterp* (*j#ats*) (*minusinf p*)) $\vee$
($\exists$ *j* $\in$ {*1* .. (*divlcm p*)}. $\exists$ *b* $\in$ *set* (*bset p*).
qinterp (((*I-intterm* (*a#ats*) *b*) + *j*)#*ats*) *p*))
(**is** ($\exists$ *x*. *?P x*) = (($\exists$ *j*$\in$ {*1* .. *?d*}. *?MP j*) $\vee$ ($\exists$ *j* $\in$ *?D*. $\exists$ *b*$\in$ *?B*. *?P* (*?I a b* + *j*))))
proof−
have *linp* :*islinform p* **by** (*rule unified-islinform*[*OF unifp*])
have *dpos*: *?d* > *0* **by** (*rule divlcm-pos*[*OF linp*])
have *alldvd*: *alldivide*(*?d*,*p*) **by** (*rule divlcm-corr*[*OF linp*])
have *eMPimpeP*: ($\exists$*j* $\in$ *?D*. *?MP j*) $\longrightarrow$ ($\exists$*x*. *?P x*)
by (*simp add*: *minusinf-lemma*[*OF unifp*, **where** *d*=*?d* **and** *ats*=*ats*])
have *ePimpeP*: ($\exists$ *j* $\in$ *?D*. $\exists$ *b*$\in$ *?B*. *?P* (*?I a b* + *j*)) $\longrightarrow$ ($\exists$ *x*. *?P x*)
by *blast*
have *bst-rep*: $\forall$ *x*. $\neg$ ($\exists$ *j* $\in$ *?D*. $\exists$ *b* $\in$ *?B*. *?P* (*?I a b* + *j*)) $\longrightarrow$ *?P x* $\longrightarrow$ *?P* (*x* − *?d*)

by (*rule bset-disj-repeat2*[*OF unifp*])
have *MPrep*: $\forall$ *x k*. *?MP x* = *?MP* (*x*− *k*∗*?d*)
by (*rule minusinf-repeats2*[*OF alldvd unifp*])
have *MPeqP*: $\exists$ *z*. $\forall$ *x* < *z*. *?P x* = *?MP x*
by (*rule minusinf-eq*[*OF unifp*])
let *?B'*= {*?I a b*| *b*. *b*∈ *?B*}
from *bst-rep* **have** *bst-rep2*: $\forall$ *x*. ¬ ($\exists$ *j*∈*?D*. $\exists$ *b*∈ *?B'*. *?P* (*b*+*j*)) ⟶ *?P x* ⟶ *?P* (*x* − *?d*)
by *auto*
show *?thesis*
using *cpmi-eq*[*OF dpos MPeqP bst-rep2 MPrep*]
by *auto*
qed

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*) **by** *arith*
lemma *mirror-unified*:
assumes *unif*: *isunified p*
shows *isunified* (*mirror p*)
using *unif*
proof (*induct p rule*: *mirror.induct*, *simp-all*)
case (*goal1 c r z*)
from *prems* **have** *zz*: *z* = *Cst 0* **by** (*cases z*, *simp-all*)
then show *?case* **using** *prems*
by (*auto simp add*: *islinintterm-eq-islint islint-def*)
next
case (*goal2 c r z*)
from *prems* **have** *zz*: *z* = *Cst 0* **by** (*cases z*, *simp-all*)
then show *?case* **using** *prems*
by (*auto simp add*: *islinintterm-eq-islint islint-def*)
next

case (*goal3 d c r*) **show** *?case* **using** *prems* **by** (*auto simp add*: *islinintterm-eq-islint islint-def*)
next
case (*goal4 d c r*) **show** *?case* **using** *prems* **by** (*auto simp add*: *islinintterm-eq-islint islint-def*)
next
case (*goal5 c r z*)
from *prems* **have** *zz*: $z = Cst\ 0$ **by** (*cases z*, *simp-all*)
then show *?case* **using** *prems*
by (*auto simp add*: *islinintterm-eq-islint islint-def*)
qed

lemma *plusinf-eq-minusinf-mirror*:
assumes *unifp*: *isunified p*
shows (*qinterp* ($x\#ats$) (*plusinf p*)) = (*qinterp* (($-\ x$)$\#ats$) (*minusinf* (*mirror p*)))
using *unifp unified-islinform*[*OF unifp*]
proof (*induct p rule*: *islinform.induct*, *simp-all*)
case (*goal1 t z*)
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have ($\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r) \vee (\exists\ i.\ t = Cst\ i)$
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ assume $\exists\ i.\ t = Cst\ i$ **then have** *?case* **using** *prems* **by** *auto* **}**
moreover
{ assume $\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
then obtain *i n r* **where**
inr-def: $t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
by *blast*
with *lint* **have** *lininr*: *islinintterm* ($Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])
have *?case* **using** *prems*
by (*cases n*, *auto simp add*: *nth-pos2*
intterm-novar0[*OF lininr*, **where** x=x **and** y=$-x$])**}**
ultimately show *?case* **by** *blast*

next
case (*goal2 t z*)
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have ($\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r) \vee (\exists\ i.\ t = Cst\ i)$
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ assume $\exists\ i.\ t = Cst\ i$ **then have** *?case* **using** *prems* **by** *auto* **}**
moreover
{ assume $\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
then obtain *i n r* **where**

```
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case using prems
      by (cases n, auto simp add: nth-pos2
          intterm-novar0[OF lininr, where x=x and y=−x] )}
  ultimately show ?case by blast
next
  case (goal3 d t)

 from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])

    have ?case using prems
      by (cases n, simp-all add: nth-pos2
          intterm-novar0[OF lininr, where x=x and y=−x] )}
  ultimately show ?case by blast
next

  case (goal4 d t)

 from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
```

```
    by (rule islinintterm-subt[OF lininr])

   have ?case using prems
    by (cases n, simp-all add: nth-pos2
       intterm-novar0[OF lininr, where x=x and y=−x] )}
  ultimately show ?case by blast
next
  case (goal5 t z)
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
   by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
   then obtain i n r where
    inr-def: t = Add (Mult (Cst i) (Var n) ) r
    by blast
   with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
    by simp
   have linr: islinintterm r
    by (rule islinintterm-subt[OF lininr])
   have ?case using prems
    by (cases n, auto simp add: nth-pos2
       intterm-novar0[OF lininr, where x=x and y=−x] )}
  ultimately show ?case by blast
qed


lemma aset-eq-bset-mirror:
  assumes unifp: isunified p
  shows set (aset p) = set (map lin-neg (bset (mirror p)))
using unifp
proof(induct p rule: mirror.induct)
  case (1 c r z)
  from prems have zz: z = Cst 0
   by (cases z, auto)
  from prems zz have lincnr: islinintterm (Add (Mult (Cst c) (Var 0)) r) by
simp
  have linr: islinintterm r by (rule islinintterm-subt[OF lincnr])
  have neg1eqm1: Cst 1 = lin-neg (Cst −1) by (simp add: lin-neg-def)
  have negm1eq1: Cst −1 = lin-neg (Cst 1) by (simp add: lin-neg-def)
  show ?case  using prems linr zz apply (auto simp add: lin-neg-lin-add-distrib
lin-neg-idemp neg1eqm1)
   by (simp add: negm1eq1 lin-neg-idemp sym[OF lin-neg-lin-add-distrib] lin-add-lin)
next
  case (2 c r z)  from prems have zz: z = Cst 0
   by (cases z, auto)
  from prems zz have lincnr: islinintterm (Add (Mult (Cst c) (Var 0)) r) by
```

simp
  have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
  have *neg1eqm1*: *Cst 1* = *lin-neg* (*Cst* −*1*) **by** (*simp add*: *lin-neg-def*)
  have *negm1eq1*: *Cst* −*1* = *lin-neg* (*Cst 1*) **by** (*simp add*: *lin-neg-def*)
  show *?case* **using** *prems linr zz*
    by (*auto simp add*: *lin-neg-lin-add-distrib lin-neg-idemp neg1eqm1*)
  (*simp add*: *negm1eq1 lin-neg-idemp sym*[*OF lin-neg-lin-add-distrib*] *lin-add-lin lin-neg-lin*)

next
  case (*5 c r z*) **from** *prems* **have** *zz*: *z* = *Cst 0*
    by (*cases z*, *auto*)
  from *prems zz* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
  have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
  have *neg1eqm1*: *Cst 1* = *lin-neg* (*Cst* −*1*) **by** (*simp add*: *lin-neg-def*)
  have *negm1eq1*: *Cst* −*1* = *lin-neg* (*Cst 1*) **by** (*simp add*: *lin-neg-def*)
  show *?case* **using** *prems linr zz*
    by(*auto simp add*: *lin-neg-lin-add-distrib lin-neg-idemp neg1eqm1*)

qed *simp-all*

lemma *aset-eq-bset-mirror2*:
  assumes *unifp*: *isunified p*
  shows *aset p* = *map lin-neg* (*bset* (*mirror p*))
using *unifp*
proof(*induct p rule*: *mirror.induct*)
  case (*1 c r z*)
  from *prems* **have** *zz*: *z* = *Cst 0*
    by (*cases z*, *auto*)
  from *prems zz* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
  have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
  have *neg1eqm1*: *Cst 1* = *lin-neg* (*Cst* −*1*) **by** (*simp add*: *lin-neg-def*)
  have *negm1eq1*: *Cst* −*1* = *lin-neg* (*Cst 1*) **by** (*simp add*: *lin-neg-def*)
  show *?case* **using** *prems linr zz*
    apply (*simp add*: *lin-neg-lin-add-distrib lin-neg-idemp neg1eqm1*)
    apply (*simp add*: *negm1eq1 lin-neg-idemp sym*[*OF lin-neg-lin-add-distrib*] *lin-add-lin*)
    by *arith*
next
  case (*2 c r z*) **from** *prems* **have** *zz*: *z* = *Cst 0*
    by (*cases z*, *auto*)
  from *prems zz* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
  have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
  have *neg1eqm1*: *Cst 1* = *lin-neg* (*Cst* −*1*) **by** (*simp add*: *lin-neg-def*)
  have *negm1eq1*: *Cst* −*1* = *lin-neg* (*Cst 1*) **by** (*simp add*: *lin-neg-def*)
  show *?case* **using** *prems linr zz*

by(*auto simp add*: *lin-neg-lin-add-distrib lin-neg-idemp neg1eqm1*)
(*simp add*: *negm1eq1 lin-neg-idemp sym*[*OF lin-neg-lin-add-distrib*] *lin-add-lin lin-neg-lin*)

next
case (*5 c r z*) **from** *prems* **have** *zz*: $z = Cst\ 0$
by (*cases z*, *auto*)
from *prems zz* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *neg1eqm1*: *Cst 1* = *lin-neg* (*Cst* −*1*) **by** (*simp add*: *lin-neg-def*)
have *negm1eq1*: *Cst* −*1* = *lin-neg* (*Cst 1*) **by** (*simp add*: *lin-neg-def*)
show *?case* **using** *prems linr zz*
by(*auto simp add*: *lin-neg-lin-add-distrib lin-neg-idemp neg1eqm1*)

qed *simp-all*

lemma *divlcm-mirror-eq*:
assumes *unifp*: *isunified p*
shows *divlcm p* = *divlcm* (*mirror p*)
using *unifp*
by (*induct p rule*: *mirror.induct*) *auto*

lemma *mirror-interp*:
assumes *unifp*: *isunified p*
shows (*qinterp* (*x#ats*) *p*) = (*qinterp* ((− *x*)#*ats*) (*mirror p*)) (**is** *?P x* = *?MP* (−*x*))
using *unifp unified-islinform*[*OF unifp*]
proof (*induct p rule*: *islinform.induct*)
case (*1 t z*)
from *prems* **have** *zz*: *z* = *0* **by** *simp*
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have (∃ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*) ∨ (∃ *i*. *t* = *Cst i*)
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ **assume** ∃ *i*. *t* = *Cst i* **then have** *?case* **using** *prems* **by** *auto* }
moreover
{ **assume** ∃ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
then obtain *i n r* **where**
inr-def: *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
by *blast*
with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])
have *?case* **using** *prems zz*
by (*cases n*) (*simp-all add*: *nth-pos2*

```
      intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
next
  case (2 t z)
  from prems have zz: z = 0 by simp
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case using prems zz
      by (cases n) (simp-all add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
next
  case (3 d t)
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case
      using prems linr
      by (cases n) (simp-all add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
next
```

```
  case (6 d t)
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case
      using prems linr
      by (cases n) (simp-all add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
next
  case (7 t z)
  from prems have zz: z = 0 by simp
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case using prems zz
      by (cases n) (simp-all add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
qed simp-all


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

```
(−x))
using unifp
proof (induct p rule: islinform.induct)
  case (1 t z)
  from prems have zz: z = 0 by simp
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case using prems zz
      by (cases n) (simp-all add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
next
  case (2 t z)
  from prems have zz: z = 0 by simp
  from prems
  have lint: islinintterm t by simp
  then have (∃ i n r. t = Add (Mult (Cst i) (Var n) ) r) ∨ (∃ i. t = Cst i)
    by (induct t rule: islinintterm.induct) auto
  moreover{ assume ∃ i. t = Cst i then have ?case using prems by auto }
  moreover
  { assume ∃ i n r. t = Add (Mult (Cst i) (Var n) ) r
    then obtain i n r where
      inr-def: t = Add (Mult (Cst i) (Var n) ) r
      by blast
    with lint have lininr: islinintterm (Add (Mult (Cst i) (Var n) ) r)
      by simp
    have linr: islinintterm r
      by (rule islinintterm-subt[OF lininr])
    have ?case using prems zz
      by (cases n) (simp-all add: nth-pos2
        intterm-novar0[OF lininr, where x=x and y=−x])
  }
  ultimately show ?case by blast
next
  case (3 d t)
  from prems
```

have *lint*: *islinintterm t* **by** *simp*
then have $(\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r) \vee (\exists\ i.\ t = Cst\ i)$
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ assume $\exists\ i.\ t = Cst\ i$ **then have** *?case* **using** *prems* **by** *auto* **}**
moreover
{ assume $\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
then obtain *i n r* **where**
inr-def: $t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
by *blast*
with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])
have *?case*
using *prems linr*
by (*cases n*) (*simp-all add*: *nth-pos2*
intterm-novar0[*OF lininr*, **where** $x{=}x$ **and** $y{=}{-}x$])
}
ultimately show *?case* **by** *blast*
next

case (*6 d t*)
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have $(\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r) \vee (\exists\ i.\ t = Cst\ i)$
by (*induct t rule*: *islinintterm.induct*) *auto*
moreover{ assume $\exists\ i.\ t = Cst\ i$ **then have** *?case* **using** *prems* **by** *auto* **}**
moreover
{ assume $\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
then obtain *i n r* **where**
inr-def: $t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r$
by *blast*
with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
by *simp*
have *linr*: *islinintterm r*
by (*rule islinintterm-subt*[*OF lininr*])
have *?case*
using *prems linr*
by (*cases n*) (*simp-all add*: *nth-pos2*
intterm-novar0[*OF lininr*, **where** $x{=}x$ **and** $y{=}{-}x$])
}
ultimately show *?case* **by** *blast*
next
case (*7 t z*)
from *prems* **have** *zz*: $z = 0$ **by** *simp*
from *prems*
have *lint*: *islinintterm t* **by** *simp*
then have $(\exists\ i\ n\ r.\ t = Add\ (Mult\ (Cst\ i)\ (Var\ n)\ )\ r) \vee (\exists\ i.\ t = Cst\ i)$
by (*induct t rule*: *islinintterm.induct*) *auto*

moreover{ **assume** ∃ *i*. *t* = *Cst i* **then have** *?case* **using** *prems* **by** *auto* }
moreover
{ **assume** ∃ *i n r*. *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
  then obtain *i n r* **where**
    inr-def: *t* = *Add* (*Mult* (*Cst i*) (*Var n*)) *r*
    by *blast*
  with *lint* **have** *lininr*: *islinintterm* (*Add* (*Mult* (*Cst i*) (*Var n*)) *r*)
    by *simp*
  have *linr*: *islinintterm r*
    by (*rule islinintterm-subt*[*OF lininr*])
  have *?case* **using** *prems zz*
    by (*cases n*) (*simp-all add*: *nth-pos2*
      intterm-novar0[*OF lininr*, **where** *x*=*x* **and** *y*=−*x*])
}
ultimately show *?case* **by** *blast*
qed *simp-all*

lemma *mirror-ex*:
  assumes *unifp*: *isunified p*
  shows (∃ *x*. (*qinterp* (*x#ats*) *p*)) = (∃ *y*. (*qinterp* (*y#ats*) (*mirror p*)))
  (**is** (∃ *x*. *?P x*) = (∃ *y*. *?MP y*))
proof
  assume ∃ *x*. *?P x*
  then obtain *x* **where** *px*:*?P x* **by** *blast*
  have *?MP* (−*x*)
    using *px*
    by(*simp add*: *mirror-interp*[*OF unifp*, **where** *x*=*x*])
  then show ∃ *y*. *?MP y* **by** *blast*
next
  assume ∃ *y*. *?MP y*
  then obtain *y* **where** *mpy*: *?MP y* **by** *blast*
  have *?P* (−*y*)
    using *mpy*
    by (*simp add*: *mirror-interp*[*OF unifp*, **where** *x*=−*y*])
  then show ∃ *x*. *?P x* **by** *blast*
qed

lemma *mirror-ex2*:
  assumes *unifp*: *isunified p*
  shows *qinterp ats* (*QEx p*) = *qinterp ats* (*QEx* (*mirror p*))
using *mirror-ex*[*OF unifp*] **by** *simp*

lemma *cooper-pi-eq*:
  assumes *unifp* : *isunified p*
  shows (∃ *x*. *qinterp* (*x#ats*) *p*) =
  ((∃ *j* ∈ {*1* .. (*divlcm p*)}. *qinterp* (−*j#ats*) (*plusinf p*)) ∨

(∃ j ∈ {1 .. (divlcm p)}. ∃ b ∈ set (aset p).
qinterp (((I-intterm (a#ats) b) − j)#ats) p))
(**is** (∃ x. ?P x) = ((∃ j∈ {1 .. ?d}. ?PP (−j)) ∨ (∃ j ∈ ?D. ∃ b∈ ?A. ?P (?I a b − j))))
proof−
have unifmp: isunified (mirror p) **by** (rule mirror-unified[OF unifp])
have th1:
(∃ j∈ {1 .. ?d}. ?PP (−j)) = (∃ j∈ {1..?d}. qinterp (j # ats) (minusinf (mirror p)))
by (simp add: plusinf-eq-minusinf-mirror[OF unifp])
have dth: ?d = divlcm (mirror p)
by (rule divlcm-mirror-eq[OF unifp])
have (∃ j ∈ ?D. ∃ b∈ ?A. ?P (?I a b − j)) =
(∃ j∈ ?D. ∃ b ∈ set (map lin-neg (bset (mirror p))). ?P (?I a b − j))
by (simp only: aset-eq-bset-mirror[OF unifp])
also have . . . = (∃ j∈ ?D. ∃ b ∈ set (bset (mirror p)). ?P (?I a (lin-neg b) − j))
by simp
also have . . . = (∃ j∈ ?D. ∃ b ∈ set (bset (mirror p)). ?P (−(?I a b + j)))
proof
assume ∃j∈{1..divlcm p}.
∃b∈set (bset (mirror p)). qinterp ((I-intterm (a # ats) (lin-neg b) − j) # ats) p
then
obtain j **and** b **where**
pbmj: j∈ ?D ∧ b∈ set (bset (mirror p)) ∧ ?P (?I a (lin-neg b) − j) **by** blast
then have linb: islinintterm b
by (auto simp add:bset-lin[OF unifmp])
from linb pbmj **have** ?P (−(?I a b + j)) **by** (simp add: lin-neg-corr)
then show ∃ j∈ ?D. ∃ b ∈ set (bset (mirror p)). ?P (−(?I a b + j))
using pbmj
by auto
next
assume ∃ j∈ ?D. ∃ b ∈ set (bset (mirror p)). ?P (−(?I a b + j))
then obtain j **and** b **where**
pbmj: j∈ ?D ∧ b∈ set (bset (mirror p)) ∧ ?P (−(?I a b + j))
by blast
then have linb: islinintterm b
by (auto simp add:bset-lin[OF unifmp])
from linb pbmj **have** ?P (?I a (lin-neg b) − j)
by (simp add: lin-neg-corr)
then show ∃ j∈ ?D. ∃ b ∈ set (bset (mirror p)). ?P (?I a (lin-neg b) − j)
using pbmj **by** auto
qed
finally
have bth: (∃ j∈ ?D. ∃ b∈ ?A. ?P (?I a b − j)) =
(∃j∈ ?D. ∃ b∈set (bset (mirror p)).
qinterp ((I-intterm (a # ats) b + j) # ats) (mirror p))
by (simp add: mirror-interp[OF unifp] zadd-ac)

from *bth dth th1*
have $(\exists\ x.\ ?P\ x) = (\exists\ x.\ qinterp\ (x\#ats)\ (mirror\ p))$
by (*simp add*: *mirror-ex*[*OF unifp*])
also have $\ldots = ((\exists j{\in}\{1..divlcm\ (mirror\ p)\}.\ qinterp\ (j\ \#\ ats)\ (minusinf\ (mirror\ p))) \vee$
$(\exists j{\in}\{1..divlcm\ (mirror\ p)\}.$
$\exists b{\in}set\ (bset\ (mirror\ p)).\ qinterp\ ((I\text{-}intterm\ (a\ \#\ ats)\ b + j)\ \#\ ats)\ (mirror\ p)))$
(**is** $(\exists\ x.\ ?MP\ x) = ((\exists\ j{\in}\ ?DM.\ ?MPM\ j) \vee (\exists\ j\ \in\ ?DM.\ \exists\ b{\in}\ ?BM.\ ?MP\ (?I\ a\ b + j))))$
by (*rule cooper-mi-eq*[*OF unifmp*])
also
have $\ldots = ((\exists\ j{\in}\ ?D.\ ?PP\ (-j)) \vee (\exists\ j\ \in\ ?D.\ \exists\ b{\in}\ ?BM.\ ?MP\ (?I\ a\ b + j)))$
using *bth th1 dth* **by** *simp*
finally show *?thesis* **using** *sym*[*OF bth*] **by** *simp*
qed

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-it i it*)
subst-it i (*Add it1 it2*) = *Add* (*subst-it i it1*) (*subst-it i it2*)
subst-it i (*Sub it1 it2*) = *Sub* (*subst-it i it1*) (*subst-it i it2*)
subst-it i (*Mult it1 it2*) = *Mult* (*subst-it i it1*) (*subst-it i it2*)

lemma *subst-it-corr*:
I-intterm (*a#ats*) (*subst-it i t*) = *I-intterm* ((*I-intterm* (*a#ats*) *i*)#*ats*) *t*
by (*induct t rule*: *subst-it.induct*, *simp-all add*: *nth-pos2*)

consts *subst-p*:: *intterm* ⇒ *QF* ⇒ *QF*
primrec
subst-p i (*Le it1 it2*) = *Le* (*subst-it i it1*) (*subst-it i it2*)
subst-p i (*Lt it1 it2*) = *Lt* (*subst-it i it1*) (*subst-it i it2*)
subst-p i (*Ge it1 it2*) = *Ge* (*subst-it i it1*) (*subst-it i it2*)
subst-p i (*Gt it1 it2*) = *Gt* (*subst-it i it1*) (*subst-it i it2*)
subst-p i (*Eq it1 it2*) = *Eq* (*subst-it i it1*) (*subst-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*
  using *qf*
by (*induct p rule*: *subst-p.induct*) (*simp-all add*: *subst-it-corr*)

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*
using *nov0*
by (*induct x*) (*auto simp add*: *nth-pos2*)

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−
  have *qinterp* (*a*#*ats*) (*subst-p i p*) = *qinterp* (*I-intterm* (*a*#*ats*) *i*#*ats*) *p*
    by (*rule subst-p-corr*[*OF qfp*])
  moreover have *I-intterm* (*a*#*ats*) *i*#*ats* = *I-intterm* (*b*#*ats*) *i*#*ats*
    by (*simp add*: *I-intterm-novar0*[*OF nov0*, **where** *a*=*a* **and** *b*=*b*])
  ultimately show *?thesis* **by** *simp*
qed

lemma *lin-novar0*:
  assumes *linx*: *islinintterm x*
  and *nov0*: *novar0I x*
  shows ∃ *n* > *0*. *islintn*(*n*,*x*)
using *linx nov0*
by (*induct x rule*: *islinintterm.induct*) *auto*

lemma *lintnpos-novar0*:
 assumes *npos*: *n* > *0*
  and *linx*: *islintn*(*n*,*x*)
  shows *novar0I x*
using *npos linx*
by (*induct n x rule*: *islintn.induct*) *auto*

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−
  have ∃ *na* > *0*. *islintn*(*na*, *a*) **by** (*rule lin-novar0*[*OF lina nov0a*])
  then obtain *na* **where** *na*: *na* > *0* ∧ *islintn*(*na*,*a*) **by** *blast*
  have ∃ *nb* > *0*. *islintn*(*nb*, *b*) **by** (*rule lin-novar0*[*OF linb nov0b*])
  then obtain *nb* **where** *nb*: *nb* > *0* ∧ *islintn*(*nb*,*b*) **by** *blast*
  from *na* **have** *napos*: *na* > *0* **by** *simp*
  from *na* **have** *linna*: *islintn*(*na*,*a*) **by** *simp*
  from *nb* **have** *nbpos*: *nb* > *0* **by** *simp*
  from *nb* **have** *linnb*: *islintn*(*nb*,*b*) **by** *simp*
  have *min na nb* ≤ *min na nb* **by** *simp*
  then have *islintn* (*min na nb*, *lin-add*(*a*,*b*)) **by** (*simp add*: *lin-add-lint*[*OF linna linnb*])
  moreover have *min na nb* > *0* **using** *napos nbpos* **by** (*simp add*: *min-def*)
  ultimately show *?thesis* **by** (*simp only*: *lintnpos-novar0*)
qed

lemma *lin-mul-novar0*:
  assumes *linx*: *islinintterm x*

and *nov0*: *novar0I x*
shows *novar0I* (*lin-mul*(*i*,*x*))
using *linx nov0*
proof (*induct i x rule*: *lin-mul.induct*, *auto*)
case (*goal1 c c′ n r*)
from *prems* **have** *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c′*) (*Var n*)) *r*) **by** *simp*
have *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
then show *?case* **using** *prems* **by** *simp*
qed

lemma *lin-neg-novar0*:
assumes *linx*: *islinintterm x*
and *nov0*: *novar0I x*
shows *novar0I* (*lin-neg x*)
by (*auto simp add*: *lin-mul-novar0 linx nov0 lin-neg-def*)

lemma *intterm-subt-novar0*:
assumes *lincnr*: *islinintterm* (*Add* (*Mult* (*Cst c*) (*Var n*)) *r*)
shows *novar0I r*
proof−
have *cnz*: $c \neq 0$ **by** (*rule islinintterm-cnz*[*OF lincnr*])
have *islintn*(*0*,*Add* (*Mult* (*Cst c*) (*Var n*)) *r*) **using** *lincnr*
by (*simp only*: *islinintterm-eq-islint islint-def*)
then have *islintn* (*n+1*,*r*) **by** *auto*
moreover have *n+1 >0* **by** *arith*
ultimately show *?thesis*
using *lintnpos-novar0*
by *auto*
qed

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*)
using *nov0*
by (*induct t*) (*auto simp add*: *nth-pos2*)

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*)
by (*induct p rule*: *isqfree.induct*, *auto*)

lemma *novar0-qfree*: *novar0 p* ⟹ *isqfree p*
by (*induct p*) *auto*

lemma *qinterp-novar0*:
  assumes *nov0*: *novar0 p*
  shows *qinterp* (*a#ats*) *p* = *qinterp ats* (*decrvars p*)
using *nov0*
by(*induct p*) (*simp-all add*: *intterm-decrvarsI*)

lemma *bset-novar0*:
  assumes *unifp*: *isunified p*
  shows ∀ *b*∈ *set* (*bset p*). *novar0I b*
  using *unifp*
proof(*induct p rule*: *bset.induct*)
  case (*1 c r z*)
  from *prems* **have** *zz*: *z* = *Cst 0* **by** (*cases z*, *auto*)
    from *prems zz* **have** *lincnr*: *islinintterm*(*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
    have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
    have *novar0r*: *novar0I r* **by** (*rule intterm-subt-novar0*[*OF lincnr*])
    from *prems zz* **have** *c* = *1* ∨ *c* = −*1* **by** *auto*
    moreover
    {
      assume *c1*: *c=1*
      have *lin1*: *islinintterm* (*Cst 1*) **by** *simp*
      have *novar01*: *novar0I* (*Cst 1*) **by** *simp*
      then have *?case*
        using *prems zz novar0r lin1 novar01*

by (*auto simp add*: *lin-add-novar0 lin-neg-novar0 linr lin-neg-lin*)
}
moreover
{
assume *c1*: *c*= −*1*
have *lin1*: *islinintterm* (*Cst* −*1*) **by** *simp*
have *novar01*: *novar0I* (*Cst* −*1*) **by** *simp*
then have *?case*
using *prems zz novar0r lin1 novar01*
by (*auto simp add*: *lin-add-novar0 lin-neg-novar0 linr lin-neg-lin*)
}
ultimately show *?case* **by** *blast*
next
case (*2 c r z*)
from *prems* **have** *zz*: *z* = *Cst 0* **by** (*cases z*, *auto*)
from *prems zz* **have** *lincnr*: *islinintterm*(*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *novar0r*: *novar0I r* **by** (*rule intterm-subt-novar0*[*OF lincnr*])
from *prems zz* **have** *c* = *1* ∨ *c* = −*1* **by** *auto*
moreover
{
assume *c1*: *c*=*1*
have *lin1*: *islinintterm* (*Cst 1*) **by** *simp*
have *novar01*: *novar0I* (*Cst 1*) **by** *simp*
then have *?case*
using *prems zz novar0r lin1 novar01*
by (*auto simp add*: *lin-add-novar0 lin-neg-novar0 linr lin-neg-lin*)
}
moreover
{
assume *c1*: *c*= −*1*
have *lin1*: *islinintterm* (*Cst* −*1*) **by** *simp*
have *novar01*: *novar0I* (*Cst* −*1*) **by** *simp*
then have *?case*
using *prems zz novar0r lin1 novar01*
by (*auto simp add*: *lin-add-novar0 lin-neg-novar0 linr lin-neg-lin*)
}
ultimately show *?case* **by** *blast*
next
case (*3 c r z*)
from *prems* **have** *zz*: *z* = *Cst 0* **by** (*cases z*, *auto*)
from *prems zz* **have** *lincnr*: *islinintterm*(*Add* (*Mult* (*Cst c*) (*Var 0*)) *r*) **by** *simp*
have *linr*: *islinintterm r* **by** (*rule islinintterm-subt*[*OF lincnr*])
have *novar0r*: *novar0I r* **by** (*rule intterm-subt-novar0*[*OF lincnr*])
from *prems zz* **have** *c* = *1* ∨ *c* = −*1* **by** *auto*
moreover
{

```
      assume c1: c=1
      have lin1: islinintterm (Cst 1) by simp
      have novar01: novar0I (Cst 1) by simp
      then have ?case
        using prems zz novar0r lin1 novar01
        by (auto simp add: lin-add-novar0 lin-neg-novar0 linr lin-neg-lin)
    }
    moreover
    {
      assume c1: c= −1
      have lin1: islinintterm (Cst −1) by simp
      have novar01: novar0I (Cst −1) by simp
      then have ?case
        using prems zz novar0r lin1 novar01
        by (auto simp add: lin-add-novar0 lin-neg-novar0 linr lin-neg-lin)
    }
    ultimately show ?case by blast
qed auto


lemma subst-it-novar0:
  assumes nov0x: novar0I x
  shows novar0I (subst-it x t)
  using nov0x
  by (induct t) auto

lemma subst-p-novar0:
  assumes nov0x:novar0I x
  and qfp: isqfree p
  shows novar0 (subst-p x p)
  using nov0x qfp
  by (induct p rule: novar0.induct) (simp-all add: subst-it-novar0)


lemma linearize-novar0:
  assumes nov0t: novar0I t
  shows ⋀ t'. linearize t = Some t' ⟹ novar0I t'
using nov0t
proof(induct t rule: novar0I.induct)
  case (Neg a)
  let ?la = linearize a
  from prems have ∃ a'. ?la = Some a' by (cases ?la, auto)
  then obtain a' where ?la = Some a' by blast
  with prems have nv0a':novar0I a' by simp
  have islinintterm a' using prems by (simp add: linearize-linear)
  with nv0a' have novar0I (lin-neg a')
    by (simp add: lin-neg-novar0)
  then
  show ?case using prems by simp
```

```
next
  case (Add a b)
  let ?la = linearize a
  let ?lb = linearize b
  from prems have linab: linearize (Add a b) = Some t' by simp
  then have ∃ a'. ?la = Some a' by (cases ?la) auto
  then obtain a' where ?la = Some a' by blast
  with prems have nv0a':novar0I a' by simp
  have lina': islinintterm a' using prems by (simp add: linearize-linear)
  from linab have ∃ b'. ?lb = Some b'
    by (cases ?la, auto simp add: measure-def inv-image-def) (cases ?lb, auto)
  then obtain b' where ?lb = Some b' by blast
  with prems have nv0b':novar0I b' by simp
  have linb': islinintterm b' using prems by (simp add: linearize-linear)
  then show ?case using prems lina' linb' nv0a' nv0b'
    by (auto simp add: measure-def inv-image-def lin-add-novar0)
next
  case (Sub a b)
    let ?la = linearize a
  let ?lb = linearize b
  from prems have linab: linearize (Sub a b) = Some t' by simp
  then have ∃ a'. ?la = Some a' by (cases ?la) auto
  then obtain a' where ?la = Some a' by blast
  with prems have nv0a':novar0I a' by simp
  have lina': islinintterm a' using prems by (simp add: linearize-linear)
  from linab have ∃ b'. ?lb = Some b'
    by (cases ?la, auto simp add: measure-def inv-image-def) (cases ?lb, auto)
  then obtain b' where ?lb = Some b' by blast
  with prems have nv0b':novar0I b' by simp
  have linb': islinintterm b' using prems by (simp add: linearize-linear)
  then show ?case using prems lina' linb' nv0a' nv0b'
    by (auto simp add:
      measure-def inv-image-def lin-add-novar0 lin-neg-novar0 lin-neg-lin)
next
  case (Mult a b)
  let ?la = linearize a
  let ?lb = linearize b
  from prems have linab: linearize (Mult a b) = Some t' by simp
  then have ∃ a'. ?la = Some a'
    by (cases ?la, auto simp add: measure-def inv-image-def)
  then obtain a' where ?la = Some a' by blast
  with prems have nv0a':novar0I a' by simp
  have lina': islinintterm a' using prems by (simp add: linearize-linear)
  from prems linab have ∃ b'. ?lb = Some b'
    apply (cases ?la, auto simp add: measure-def inv-image-def)
    by (cases a',auto simp add: measure-def inv-image-def) (cases ?lb, auto)+
  then obtain b' where ?lb = Some b' by blast
  with prems have nv0b':novar0I b' by simp
  have linb': islinintterm b' using prems by (simp add: linearize-linear)
```

```
  then show ?case using prems lina' linb' nv0a' nv0b'
    by (cases a',auto simp add: measure-def inv-image-def lin-mul-novar0)
  (cases b',auto simp add: measure-def inv-image-def lin-mul-novar0)
qed auto


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(*induct p rule*: *psimpl.induct*)

case (*1 l r*)
have (∃ *lx*. *linearize* (*Sub l r*) = *Some lx*) ∨ (*linearize* (*Sub l r*) = *None*) **by** *auto*
moreover
{
assume *lin*: ∃ *lx*. *linearize* (*Sub l r*) = *Some lx*
from *lin* **obtain** *lx* **where** *lx*: *linearize* (*Sub l r*) = *Some lx* **by** *blast*
from *lx* **have** *I-intterm ats* (*Sub l r*) = *I-intterm ats lx*
by (*rule linearize-corr*[**where** *t*=*Sub l r* **and** *t'*= *lx*])
then have *feq*: *qinterp ats* (*Le l r*) = *qinterp ats* (*Le lx* (*Cst 0*)) **by** (*simp* , *arith*)
from *lx* **have** *lxlin*: *islinintterm lx* **by** (*rule linearize-linear*)
from *lxlin feq* **have** *?case*
proof−
have (∃ *i*. *lx* = *Cst i*) ∨ (¬ (∃ *i*. *lx* = *Cst i*)) **by** *blast*
moreover
{
assume *lxcst*: ∃ *i*. *lx* = *Cst i*
from *lxcst* **obtain** *i* **where** *lxi*: *lx* = *Cst i* **by** *blast*
with *feq* **have** *qinterp ats* (*Le l r*) = (*i* ≤ *0*) **by** *simp*
then have *?case* **using** *prems* **by** (*simp add*: *measure-def inv-image-def*)
}
moreover
{
assume (¬ (∃ *i*. *lx* = *Cst i*))
then have (*case lx of*
Cst i ⇒ (*if i* ≤ *0 then T else F*)
| - ⇒ (*Le lx* (*Cst 0*))) = (*Le lx* (*Cst 0*))
by (*case-tac lx*::*intterm*, *auto*)
with *prems lxlin feq* **have** *?case* **by** (*auto simp add*: *measure-def inv-image-def*)
}
ultimately show *?thesis* **by** *blast*
qed
}
moreover
{
assume *linearize* (*Sub l r*) = *None*
then have *?case* **using** *prems* **by** *simp*
}
ultimately show *?case* **by** *blast*

next
case (*2 l r*)
have (∃ *lx*. *linearize* (*Sub l r*) = *Some lx*) ∨ (*linearize* (*Sub l r*) = *None*) **by** *auto*
moreover
{
assume *lin*: ∃ *lx*. *linearize* (*Sub l r*) = *Some lx*

from *lin* **obtain** *lx* **where** *lx*: *linearize* (*Sub l r*) = *Some lx* **by** *blast*
from *lx* **have** *I-intterm ats* (*Sub l r*) = *I-intterm ats lx*
by (*rule linearize-corr*[**where** *t*=*Sub l r* **and** *t′*= *lx*])
then have *feq*: *qinterp ats* (*Eq l r*) = *qinterp ats* (*Eq lx* (*Cst 0*)) **by** (*simp* , *arith*)
from *lx* **have** *lxlin*: *islinintterm lx* **by** (*rule linearize-linear*)
from *lxlin feq* **have** *?case*
proof−
have (∃ *i*. *lx* = *Cst i*) ∨ (¬ (∃ *i*. *lx* = *Cst i*)) **by** *blast*
moreover
{
assume *lxcst*: ∃ *i*. *lx* = *Cst i*
from *lxcst* **obtain** *i* **where** *lxi*: *lx* = *Cst i* **by** *blast*
with *feq* **have** *qinterp ats* (*Eq l r*) = (*i* = *0*) **by** *simp*
then have *?case* **using** *prems* **by** (*simp add*: *measure-def inv-image-def*)
}
moreover
{
assume (¬ (∃ *i*. *lx* = *Cst i*))
then have (*case lx of*
Cst i ⇒ (*if i* = *0 then T else F*)
| - ⇒ (*Eq lx* (*Cst 0*))) = (*Eq lx* (*Cst 0*))
by (*case-tac lx*::*intterm*, *auto*)
with *prems lxlin feq* **have** *?case* **by** (*auto simp add*: *measure-def inv-image-def*)
}
ultimately show *?thesis* **by** *blast*
qed
}
moreover
{
assume *linearize* (*Sub l r*) = *None*
then have *?case* **using** *prems* **by** *simp*
}
ultimately show *?case* **by** *blast*

next

case (*3 d t*)
have (∃ *lt*. *linearize t* = *Some lt*) ∨ (*linearize t* = *None*) **by** *auto*
moreover
{
assume *lin*: ∃ *lt*. *linearize t* = *Some lt*
from *lin* **obtain** *lt* **where** *lt*: *linearize t* = *Some lt* **by** *blast*
from *lt* **have** *I-intterm ats t* = *I-intterm ats lt*
by (*rule linearize-corr*[**where** *t*=*t* **and** *t′*= *lt*])
then have *feq*: *qinterp ats* (*Divides* (*Cst d*) *t*) = *qinterp ats* (*Divides* (*Cst d*) *lt*) **by** (*simp*)
from *lt* **have** *ltlin*: *islinintterm lt* **by** (*rule linearize-linear*)

from *ltlin feq* **have** *?case* **using** *prems* **apply** *simp* **by** (*case-tac lt::intterm*, *simp-all*)
}
moreover
{
assume *linearize t* = *None*
then have *?case* **using** *prems* **by** *simp*
}
ultimately show *?case* **by** *blast*

next
case (*4 f g*)

let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case* **using** *prems*
by (*cases ?sf*, *simp-all add*: *Let-def measure-def inv-image-def*)
(*cases ?sg*, *simp-all*)+
next
case (*5 f g*)
let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case* **using** *prems*
apply (*cases ?sf*, *simp-all add*: *Let-def measure-def inv-image-def*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply *blast*
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply (*cases ?sg*, *simp-all*)
apply *blast*
apply (*cases ?sg*, *simp-all*)
by (*cases ?sg*, *simp-all*) (*cases ?sg*, *simp-all*)
next
case (*6 f g*)
let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case* **using** *prems*
apply(*simp add*: *Let-def measure-def inv-image-def*)
apply(*cases ?sf*,*simp-all*)
apply (*simp-all add*: *Let-def measure-def inv-image-def*)
apply(*cases ?sg*, *simp-all*)
apply(*cases ?sg*, *simp-all*)
apply(*cases ?sg*, *simp-all*)

```
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply blast
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    apply(cases ?sg, simp-all)
    done
next
  case (7 f g)
  let ?sf = psimpl f
  let ?sg = psimpl g
  show ?case
    using prems
    by (cases ?sf, simp-all add: Let-def) (cases ?sg, simp-all)+
next
  case (8 f) show ?case
    using prems
    apply (simp add: Let-def)
    by (case-tac psimpl f, simp-all)
qed simp-all


lemma psimpl-novar0:
  assumes nov0p: novar0 p
  shows novar0 (psimpl p)
  using nov0p
proof (induct p rule: psimpl.induct)
  case (1 l r)
  let ?ls = linearize (Sub l r)
  have ?ls = None ∨ (∃ x. ?ls = Some x) by auto
  moreover
  {
```

```
    assume ?ls = None then have ?case
      using prems by (simp add: measure-def inv-image-def)
  }
  moreover {
    assume ∃ x. ?ls = Some x
    then obtain x where ls-d: ?ls = Some x by blast
    from prems have novar0I l by simp
    moreover from prems have novar0I r by simp
    ultimately have nv0s: novar0I (Sub l r) by simp
    from prems have novar0I x
      by (simp add: linearize-novar0[OF nv0s, where t'=x])
    then have ?case
      using prems
      by (cases x) (auto simp add: measure-def inv-image-def)
  }
  ultimately show ?case by blast
next
  case (2 l r)
  let ?ls = linearize (Sub l r)
  have ?ls = None ∨ (∃ x. ?ls = Some x) by auto
  moreover
  {
    assume ?ls = None then have ?case
      using prems by (simp add: measure-def inv-image-def)
  }
  moreover {
    assume ∃ x. ?ls = Some x
    then obtain x where ls-d: ?ls = Some x by blast
    from prems have novar0I l by simp
    moreover from prems have novar0I r by simp
    ultimately have nv0s: novar0I (Sub l r) by simp
    from prems have novar0I x
      by (simp add: linearize-novar0[OF nv0s, where t'=x])
    then have ?case
      using prems
      by (cases x) (auto simp add: measure-def inv-image-def)
  }
  ultimately show ?case by blast
next
  case (3 d t)
  let ?lt = linearize t
  have ?lt = None ∨ (∃ x. ?lt = Some x)  by auto
  moreover
  { assume ?lt = None then have ?case using prems by simp }
  moreover {
    assume ∃x. ?lt = Some x
    then obtain x where x-d: ?lt = Some x by blast
    from prems have nv0t: novar0I t by simp
    with x-d have novar0I x
```

by (*simp add*: *linearize-novar0*[*OF nv0t*])
with *prems* **have** *?case*
by (*cases x*) *simp-all*
}
ultimately show *?case* **by** *blast*
next
case (*4 f g*)
let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case*
using *prems*
by (*cases ?sf*, *simp-all add*: *Let-def measure-def inv-image-def*)
(*cases ?sg*,*simp-all*)+
next
case (*5 f g*)
let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case*
using *prems*
by (*cases ?sf*, *simp-all add*: *Let-def measure-def inv-image-def*)
(*cases ?sg*,*simp-all*)+
next
case (*6 f g*)
let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case*
using *prems*
by (*cases ?sf*, *simp-all add*: *Let-def measure-def inv-image-def*)
(*cases ?sg*,*simp-all*)+
next
case (*7 f g*)
let *?sf* = *psimpl f*
let *?sg* = *psimpl g*
show *?case*
using *prems*
by (*cases ?sf*, *simp-all add*: *Let-def measure-def inv-image-def*)
(*cases ?sg*,*simp-all*)+

next
case (*8 f*)
let *?sf* = *psimpl f*
from *prems* **have** *nv0sf*:*novar0 ?sf* **by** *simp*
show *?case* **using** *prems nv0sf*
by (*cases ?sf*, *auto simp add*: *Let-def measure-def inv-image-def*)
qed *simp-all*

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))))
  using qfp
proof−
  let ?pi = psimpl (subst-p i p)
  have pi: qinterp (x#xs) ?pi = qinterp (x#xs) (subst-p i p)
    by (simp add: psimpl-corr[where p=(subst-p i p)])
  let ?dp = explode-disj(is,p)
  show ?thesis using pi
  proof (cases)
    assume ?pi= T ∨ ?pi = F
    then show ?thesis using pi by (case-tac ?pi::QF, auto)

  next
    assume notTF: ¬ (?pi = T ∨ ?pi = F)
    let ?dp = explode-disj(is,p)
    have dp-cases: explode-disj(i#is,p) =
      (case (explode-disj(is,p)) of
      T ⇒ T
      | F ⇒ psimpl (subst-p i p)
      | - ⇒ Or (psimpl (subst-p i p)) (explode-disj(is,p))) using notTF
      by (cases ?pi)
    (simp-all add: Let-def cong del: QF.weak-case-cong)
    show ?thesis using pi dp-cases notTF
    proof(cases)
      assume ?dp = T ∨ ?dp = F
      then show ?thesis
        using pi dp-cases
        by (cases ?dp) auto
    next
      assume ¬ (?dp = T ∨ ?dp = F)
      then show ?thesis using pi dp-cases notTF
        by (cases ?dp) auto
    qed
```

qed
qed

lemma *explode-disj-corr*:
assumes *qfp*: *isqfree p*
shows $(\exists\ x \in set\ xs.\ qinterp\ (a\#ats)\ (subst\text{-}p\ x\ p)) =$
$(qinterp\ (a\#ats)\ (explode\text{-}disj(xs,p)))$ (**is** $(\exists\ x \in set\ xs.\ ?P\ x) = (?DP\ a\ xs\ ))$
using *qfp*
proof (*induct xs*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons y ys*)
have $(\exists\ x \in set\ (y\#ys).\ ?P\ x) = (?P\ y \lor (\exists\ x\in set\ ys.\ ?P\ x))$
by *auto*
also have $\ldots = (?P\ y \lor ?DP\ a\ ys)$ **using** *Cons.hyps qfp* **by** *auto*
also have $\ldots = ?DP\ a\ (y\#ys)$ **using** *explode-disj-disj*[*OF qfp*] **by** *auto*
finally show *?case* **by** *simp*
qed

lemma *explode-disj-novar0*:
assumes *nov0xs*: $\forall x \in set\ xs.\ novar0I\ x$
and *qfp*: *isqfree p*
shows *novar0* (*explode-disj* (*xs*,*p*))
using *nov0xs qfp*
proof (*induct xs*, *auto simp add*: *Let-def*)
case (*goal1 a as*)
let $?q = subst\text{-}p\ a\ p$
let $?qs = psimpl\ ?q$
have $?qs = T \lor ?qs = F \lor (?qs \neq T \lor ?qs \neq F)$ **by** *simp*
moreover
{ assume $?qs = T$ **then have** *?case* **by** *simp* **}**
moreover
{ assume $?qs = F$ **then have** *?case* **by** *simp* **}**
moreover
{
assume *qsnTF*: $?qs \neq T \land ?qs \neq F$
let $?r = explode\text{-}disj\ (as,p)$
have *nov0qs*: *novar0 ?qs*
using *prems*
by (*auto simp add*: *psimpl-novar0 subst-p-novar0*)
have $?r = T \lor ?r = F \lor (?r \neq T \lor ?r \neq F)$ **by** *simp*
moreover
{ assume $?r = T$ **then have** *?case* **by** (*cases ?qs*) *auto* **}**
moreover
{ assume $?r = F$ **then have** *?case* **using** *nov0qs* **by** (*cases ?qs*, *auto*) **}**
moreover
{ assume $?r \neq T \land ?r \neq F$ **then have** *?case* **using** *nov0qs prems qsnTF*

by (*cases ?qs*, *auto simp add*: *Let-def*) (*cases ?r*,*auto*)+
}
ultimately have *?case* **by** *blast*
}
ultimately show *?case* **by** *blast*
qed

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−
let *?result* =
(*case f of*
$T \Rightarrow T$
| $F \Rightarrow g$
| - $\Rightarrow$ (*case g of*
$T \Rightarrow T$
| $F \Rightarrow f$
| - $\Rightarrow$ *Or f g*))
have $f = T \vee f = F \vee (f \neq T \wedge f \neq F)$ **by** *auto*
moreover
{
assume *fT*: $f = T$
then have *?thesis* **by** *auto*
}
moreover
{
assume $f=F$
then have *?thesis* **by** *auto*
}
moreover
{
assume *fnT*: $f \neq T$
and *fnF*: $f \neq F$
have $g = T \vee g = F \vee (g \neq T \wedge g \neq F)$ **by** *auto*
moreover
{
assume $g=T$
then have *?thesis* **using** *fnT fnF* **by** (*cases f*, *auto*)
}
moreover
{
assume $g=F$

then have *?thesis* **using** *fnT fnF* **by** (*cases f*, *auto*)
}
moreover
{
assume *gnT*: $g \neq T$
and *gnF*: $g \neq F$
then have *?result* = (*case g of*
$T \Rightarrow T$
| $F \Rightarrow f$
| - $\Rightarrow$ *Or f g*)
using *fnT fnF*
by (*cases f*, *auto*)
also have ... = *Or f g*
using *gnT gnF*
by (*cases g*, *auto*)
finally have *?result* = *Or f g* **by** *simp*
then
have *?thesis* **by** *simp*
}
ultimately have *?thesis* **by** *blast*

}

ultimately show *?thesis* **by** *blast*
qed

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*))
using *fnTF gnTF f0 g0*
by (*cases f*, *auto*) (*cases g*, *auto*)+

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*)
by(*induct xs*) (*auto simp add*: *list-insert-def*)

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)
  by(induct xs ys rule: list-union.induct, auto simp add:list-insert-set)


consts list-set ::'a list ⇒ '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)
by (induct xs) (auto simp add: list-insert-set)

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(induct rule: iupto.induct)
  case (1 a b)
  show ?case
    using prems by (simp add: simp-from-to)
qed

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
  using nov0xs linxs
proof(induct d xs rule: all-sums.induct)
  case 1 show ?case by simp
next
  case (2 j a as)
  have lina: islinintterm a using 2.prems by auto
  have nov0a: novar0I a using 2.prems by auto
  let ?ys = map (λx. lin-add (a,(Cst x))) (iupto(1,j))
  have nov0ys: ∀ y∈ set ?ys. novar0I y
  proof−
    have linx: ∀ x ∈ set (iupto(1,j)). islinintterm (Cst x) by simp
    have nov0x: ∀ x ∈ set (iupto(1,j)). novar0I (Cst x) by simp
    with nov0a lina linx have ∀ x∈ set (iupto(1,j)). novar0I (lin-add (a,Cst x))
```

    by (*simp add*: *lin-add-novar0*)
   then show *?thesis* **by** *auto*
  qed
  from *2.prems*
  have *linas*: ∀ *u*∈*set as*. *islinintterm u* **by** *auto*
  from *2.prems* **have** *nov0as*: ∀ *u*∈*set as*. *novar0I u* **by** *auto*
  from *2.hyps linas nov0as* **have** *nov0alls*: ∀ *u*∈*set* (*all-sums* (*j*, *as*)). *novar0I u* **by** *simp*
  from *nov0alls nov0ys* **have**
   cs: (∀ *u*∈ *set* (*?ys*@ (*all-sums* (*j*,*as*))). *novar0I u*)
   by (*simp only*: *sym*[*OF list-all-iff*]) *auto*

  have *all-sums*(*j*,*a*#*as*) = *?ys*@(*all-sums*(*j*,*as*))
   by *simp*
  then
  have *?case* = (∀ *x*∈ *set* (*?ys*@ (*all-sums* (*j*,*as*))). *novar0I x*)
   by *auto*
  with *cs* **show** *?case* **by** *blast*
qed

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(*induct d xs rule*: *all-sums.induct*)
  case (*1 a*) **show** *?case* **by** *simp*
next
  case (*2 a y ys*)
  have (∃ *x*∈ *set* (*map* (λ*x*. *lin-add* (*y*,(*Cst x*))) (*iupto*(*1*,*a*))) . *P x*) =
   (∃ *j*∈ *set* (*iupto*(*1*,*a*)). *P* (*lin-add*(*y*,*Cst j*)))
   by *auto*
  also have ... = (∃ *j*∈ {*1*..*a*}. *P* (*lin-add*(*y*,*Cst j*)))
   by (*simp only* : *iupto-set*)
  finally
  have *dsj1*:(∃*j*∈{*1*..*a*}. *P* (*lin-add* (*y*, *Cst j*))) = (∃*x*∈*set* (*map* (λ*x*. *lin-add* (*y*, *Cst x*)) (*iupto* (*1*, *a*))). *P x*) **by** *simp*

  from *prems* **have** (∃ *j*∈ {*1*..*a*}. ∃ *b*∈ (*set* (*y*#*ys*)). *P* (*lin-add*(*b*,*Cst j*))) =
   ((∃ *j*∈ {*1*..*a*}. *P* (*lin-add*(*y*,*Cst j*))) ∨ (∃ *j*∈ {*1*..*a*}. ∃ *b* ∈ *set ys*. *P* (*lin-add*(*b*,*Cst j*)))) **by** *auto*
  also
  have ... = ((∃ *j*∈ {*1*..*a*}. *P* (*lin-add*(*y*,*Cst j*))) ∨ (∃ *x*∈ *set* (*all-sums*(*a*, *ys*)). *P x*)) **using** *prems* **by** *simp*
  also have ... = ((∃*x*∈*set* (*map* (λ*x*. *lin-add* (*y*, *Cst x*)) (*iupto* (*1*, *a*))). *P x*) ∨ (∃*x*∈*set* (*all-sums* (*a*, *ys*)). *P x*)) **using** *dsj1* **by** *simp*
  also have ... = (∃ *x*∈ (*set* (*map* (λ*x*. *lin-add* (*y*, *Cst x*)) (*iupto* (*1*, *a*))) ∪ (*set* (*all-sums*(*a*, *ys*)))). *P x*) **by** *blast*
  finally show *?case* **by** *simp*
qed

```
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−
  let ?d = divlcm p
  let ?pm = minusinf p
  let ?dj1 = explode-disj (map Cst (iupto(1,?d)),?pm)

   have qfpm: isqfree ?pm  using unified-islinform[OF unifp] minusinf-qfree by
simp
  have dpos: ?d >0 using unified-islinform[OF unifp] divlcm-pos by simp
  have ∀ x∈ set (map Cst (iupto(1,?d))). novar0I x by auto
  then have dj1-nov0: novar0 ?dj1 using qfpm explode-disj-novar0 by simp

  let ?dj2 = explode-disj (all-sums (?d,B),p)
  have
    bstlin: ∀ b∈set B. islinintterm b
    using bset-lin[OF unifp] bst
    by simp

  have bstnov0: ∀ b∈set B. novar0I b
    using bst bset-novar0[OF unifp] by simp
  have allsnov0: ∀ x∈set (all-sums(?d,B)). novar0I x
    by (simp add:all-sums-novar0[OF bstnov0 bstlin] )
  then have dj2-nov0: novar0 ?dj2
    using explode-disj-novar0 unified-isqfree[OF unifp] bst by simp
  have ?dj1 = T ∨ ?dj1 = F ∨ (?dj1 ≠ T ∧ ?dj1 ≠ F) by auto
```

```
  moreover
  { assume ?dj1 = T then have ?thesis by simp }
  moreover
   { assume ?dj1 = F then have ?thesis using bst dj2-nov0 by (simp add:
Let-def)}
  moreover
  {
    assume dj1nFT:?dj1 ≠ T ∧ ?dj1 ≠ F

    have ?dj2 = T ∨ ?dj2 = F ∨ (?dj2 ≠ T ∧ ?dj2 ≠ F) by auto
    moreover
    { assume ?dj2 = T then have ?thesis by (cases ?dj1) simp-all }
    moreover
    { assume ?dj2 = F then have ?thesis using dj1-nov0 bst
        by (cases ?dj1) (simp-all add: Let-def)}
    moreover
    {
      assume dj2-nTF:?dj2 ≠ T ∧ ?dj2 ≠ F
      let ?res = λf. λg. (case f of T ⇒ T | F ⇒ g
        | - ⇒ (case g of T ⇒ T| F ⇒ f| - ⇒ Or f g))
      have expth: explode-minf (p,B) = ?res ?dj1 ?dj2
        by (simp add: Let-def del: iupto.simps split del: split-if
          cong del: QF.weak-case-cong)
      then have ?thesis
        using prems or-case-novar0 [OF dj1nFT dj2-nTF dj1-nov0 dj2-nov0]
        by (simp add: Let-def del: iupto.simps cong del: QF.weak-case-cong)
    }
    ultimately have ?thesis by blast
  }
  ultimately show ?thesis by blast
qed


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−
  let ?d = divlcm p
  let ?pm = minusinf p
  let ?dj1 = explode-disj (map Cst (iupto(1,?d)),?pm)
   have qfpm: isqfree ?pm  using unified-islinform[OF unifp] minusinf-qfree by
simp
  have nnfp: isnnf p by (rule unified-isnnf[OF unifp])

  have (∃j∈{1..?d}. qinterp (j # ats) (minusinf p))
    = (∃j∈ set (iupto(1,?d)). qinterp (j#ats) (minusinf p))
    (is (∃ j∈ {1..?d}. ?QM j) = ...)
```

by (*simp add*: *sym*[*OF iupto-set*])
also
have ... =(∃ *j*∈ *set* (*iupto*(*1*,*?d*)). *qinterp* ((*I-intterm* (*a*#*ats*) (*Cst j*))#*ats*) (*minusinf p*))
by *simp*
also have
... = (∃ *j*∈ *set* (*map Cst* (*iupto*(*1*,*?d*))). *qinterp* ((*I-intterm* (*a*#*ats*) *j*)#*ats*) (*minusinf p*)) **by** *simp*
also have
... =
(∃ *j*∈ *set* (*map Cst* (*iupto*(*1*,*?d*))). *qinterp* (*a*#*ats*) (*subst-p j* (*minusinf p*)))
by (*simp add*: *subst-p-corr*[*OF qfpm*])
finally have *dj1-thm*:
(∃ *j*∈ {*1*..*?d*}. *?QM j*) = (*qinterp* (*a*#*ats*) *?dj1*)
by (*simp only*: *explode-disj-corr*[*OF qfpm*])
let *?dj2* = *explode-disj* (*all-sums* (*?d*,*B*),*p*)
have
bstlin: ∀ *b*∈*set B*. *islinintterm b*
using *bst* **by** (*simp add*: *bset-lin*[*OF unifp*])
have *bstnov0*: ∀ *b*∈*set B*. *novar0I b*
using *bst* **by** (*simp add*: *bset-novar0*[*OF unifp*])
have *allsnov0*: ∀ *x*∈*set* (*all-sums*(*?d*,*B*)). *novar0I x*
by (*simp add*:*all-sums-novar0*[*OF bstnov0 bstlin*])
have (∃ *j*∈ {*1*..*?d*}. ∃ *b*∈ *set B*. *?P* (*I-intterm* (*a*#*ats*) *b* + *j*)) =
(∃ *j*∈ {*1*..*?d*}. ∃ *b*∈ *set B*. *?P* (*I-intterm* (*a*#*ats*) (*lin-add*(*b*,*Cst j*))))
using *bst* **by** (*auto simp add*: *lin-add-corr bset-lin*[*OF unifp*])
also have ... = (∃ *x* ∈ *set* (*all-sums* (*?d*, *B*)). *?P* (*I-intterm* (*a*#*ats*) *x*))
by (*simp add*: *all-sums-ex*[**where** *P*=λ *t*. *?P* (*I-intterm* (*a*#*ats*) *t*)])
finally
have (∃ *j*∈ {*1*..*?d*}. ∃ *b*∈ *set B*. *?P* (*I-intterm* (*a*#*ats*) *b* + *j*)) =
(∃ *x* ∈ *set* (*all-sums* (*?d*, *B*)). *qinterp* (*a*#*ats*) (*subst-p x p*))
using *allsnov0 prems linform-isqfree unified-islinform*[*OF unifp*]
by (*simp add*: *all-sums-ex subst-p-corr*)
also have ... = (*qinterp* (*a*#*ats*) *?dj2*)
using *linform-isqfree unified-islinform*[*OF unifp*]
by (*simp add*: *explode-disj-corr*)
finally have *dj2th*:
(∃ *j*∈ {*1*..*?d*}. ∃ *b*∈ *set B*. *?P* (*I-intterm* (*a*#*ats*) *b* + *j*)) =
(*qinterp* (*a*#*ats*) *?dj2*) **by** *simp*
let *?result* = λ*f*. λ*g*.
(*case f of*
T ⇒ *T*
| *F* ⇒ *g*
| - ⇒ (*case g of*
T ⇒ *T*
| *F* ⇒ *f*
| - ⇒ *Or f g*))
have *?EXP a p* = *qinterp* (*a*#*ats*) (*?result ?dj1 ?dj2*)
by (*simp only*: *explode-minf*.*simps Let-def*)

```
  also
  have ... = (qinterp (a#ats) ?dj1 ∨ qinterp (a#ats) ?dj2)
    by (rule eval-Or-cases[where f=?dj1 and g=?dj2 and a=a and ats=ats])
  also
  have ... = ((∃ j∈ {1..?d}. ?QM j) ∨
    (∃ j∈ {1..?d}. ∃ b∈ set B. ?P (I-intterm (a#ats) b + j)))
    by (simp add: dj1-thm dj2th)
  also
  have ... = (∃ x. ?P x)
    using bst sym[OF cooper-mi-eq[OF unifp]] by simp
  finally show ?thesis by simp
qed


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−
  have ?P = (∃x. qinterp (x#ats) p) by simp
  also have ...  = (qinterp (a # ats) (explode-minf (p,B)))
    using unifp bst explode-minf-corr by simp
  finally have ex: ?P = (qinterp (a # ats) (explode-minf (p,B))) .
  have nv0: novar0 (explode-minf (p,B))
    by (rule explode-minf-novar0[OF unifp])
  show ?thesis
    using qinterp-novar0[OF nv0] ex by simp
qed


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−
  have length (list-set(bset (unitycoeff p))) ≤ length (list-set (aset (unitycoeff p)))
```

∨ *length* (*list-set*(*bset* (*unitycoeff p*))) > *length* (*list-set* (*aset* (*unitycoeff p*))) **by** *arith*
  moreover
  {
    assume *length* (*list-set*(*bset* (*unitycoeff p*))) ≤ *length* (*list-set* (*aset* (*unitycoeff p*)))
    then have *fst* (*unify p*) = *unitycoeff p* **using** *unify-def* **by** (*simp add*: *Let-def*)
    then have *?thesis* **using** *unitycoeff-corr*[*OF linp*]
      by *simp*
  }
  moreover
  {
    assume *length* (*list-set*(*bset* (*unitycoeff p*))) > *length* (*list-set* (*aset* (*unitycoeff p*)))
    then have *unif*: *fst*(*unify p*) = *mirror* (*unitycoeff p*)
      using *unify-def* **by** (*simp add*: *Let-def*)
    let *?q* =*unitycoeff p*
    have *unifq*: *isunified ?q* **by**(*rule unitycoeff-unified*[*OF linp*])
    have *linq*: *islinform ?q* **by** (*rule unified-islinform*[*OF unifq*])
    have *qinterp ats* (*QEx ?q*) = *qinterp ats* (*QEx* (*mirror ?q*))
      by (*rule mirror-ex2*[*OF unifq*])
    moreover have *qinterp ats* (*QEx p*) = *qinterp ats* (*QEx ?q*)
      using *unitycoeff-corr linp* **by** *simp*
    ultimately have *?thesis* **using** *prems unif* **by** *simp*
  }
  ultimately show *?thesis* **by** *blast*
qed

lemma *unify-unified*:
  assumes *linp*: *islinform p*
  shows *isunified* (*fst* (*unify p*))
  using *linp unitycoeff-unified mirror-unified unify-def unified-islinform*
  by (*auto simp add*: *Let-def*)

lemma *unify-qfree*:
  assumes *linp*: *islinform p*
  shows *isqfree* (*fst*(*unify p*))
  using *linp unify-unified unified-isqfree* **by** *simp*

lemma *unify-bst*:
  assumes *linp*: *islinform p*
  and *unif*: *unify p* = (*q*,*B*)
  shows *set B* = *set* (*bset q*)
proof−
  let *?q* = *unitycoeff p*
  let *?a* = *aset ?q*

let *?b = bset ?q*
let *?la = list-set ?a*
let *?lb = list-set ?b*
have *length ?lb ≤ length ?la ∨ length ?lb > length ?la* **by** *arith*
moreover
{
assume *length ?lb ≤ length ?la*
then
have *unify p = (?q,?lb)***using** *unify-def prems* **by** *(simp add: Let-def)*
then
have *?thesis* **using** *prems* **by** *(simp add: sym[OF list-set-set])*
}
moreover
{ **assume** *length ?lb > length ?la*
have *r: unify p = (mirror ?q,map lin-neg ?la)***using** *unify-def prems* **by** *(simp add: Let-def)*
have *lin: ∀ x∈ set (bset (mirror ?q)). islinintterm x*
using *bset-lin mirror-unified unitycoeff-unified[OF linp]* **by** *auto*
with *r prems aset-eq-bset-mirror lin-neg-idemp unitycoeff-unified linp*
have *set B = set (map lin-neg (map lin-neg (bset (mirror (unitycoeff p)))))*
by *(simp add: sym[OF list-set-set])*
also have *... = set (map (λx. lin-neg (lin-neg x)) (bset (mirror (unitycoeff p))))*
by *auto*
also have *... = set (bset (mirror (unitycoeff p)))*
using *lin lin-neg-idemp* **by** *(auto simp add: map-idI)*
finally
have *?thesis* **using** *r prems aset-eq-bset-mirror lin-neg-idemp unitycoeff-unified linp*
by *(simp add: sym[OF list-set-set])***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *explode-minf-unify-novar0*:
assumes *linp: islinform p*
shows *novar0 (explode-minf (unify p))*
proof−
have *∃ q B. unify p = (q,B)* **by** *simp*
then obtain *q B* **where** *qB-def: unify p = (q,B)* **by** *blast*
have *unifq: isunified q* **using** *unify-unified[OF linp] qB-def* **by** *simp*
have *bst: set B = set (bset q)* **using** *unify-bst linp qB-def* **by** *simp*
from *unifq bst explode-minf-novar0* **show** *?thesis*
using *qB-def* **by** *simp*
qed

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

have $\exists$ *q B. unify p* = (*q*,*B*) **by** *simp*
then obtain *q B* **where** *qB-def*: *unify p* = (*q*,*B*) **by** *blast*
have *unifq*: *isunified q* **using** *unify-unified*[*OF linp*] *qB-def* **by** *simp*
have *bst*: *set* (*bset q*) = *set B* **using** *unify-bst linp qB-def* **by** *simp*
from *explode-minf-corr2*[*OF unifq bst*] *unify-ex*[*OF linp*] **show** *?thesis*
using *qB-def* **by** *simp*
qed

constdefs *cooper*:: *QF* $\Rightarrow$ *QF option*
cooper p $\equiv$ *lift-un* (λ*q*. *decrvars*(*explode-minf* (*unify q*))) (*linform* (*nnf p*))

lemma *cooper-qfree*: ($\bigwedge$ *q q*$'$. $[\![$*isqfree q* ; *cooper q* = *Some q*$']\!]$ $\Longrightarrow$ *isqfree q*$'$)
proof−
fix *q q*$'$
assume *qfq*: *isqfree q*
and *qeq*: *cooper q* = *Some q*$'$
from *qeq* **have** $\exists$ *p*. *linform* (*nnf q*) = *Some p*
by (*cases linform* (*nnf q*)) (*simp-all add*: *cooper-def*)
then obtain *p* **where** *p-def*: *linform* (*nnf q*) = *Some p* **by** *blast*
have *linp*: *islinform p* **using** *p-def linform-lin nnf-isnnf qfq*
by *auto*
have *nnfq*: *isnnf* (*nnf q*) **using** *nnf-isnnf qfq* **by** *simp*
then have *nnfp*: *isnnf p* **using** *linform-nnf*[*OF nnfq*] *p-def* **by** *auto*
have *qfp*: *isqfree p* **using** *linp linform-isqfree* **by** *simp*
have *cooper q* = *Some* (*decrvars*(*explode-minf* (*unify p*))) **using** *p-def*
by (*simp add*: *cooper-def del*: *explode-minf*.*simps*)
then have *q*$'$ = *decrvars* (*explode-minf* (*unify p*)) **using** *qeq* **by** *simp*
with *linp qfp nnfp unify-unified unify-qfree unified-islinform*
show *isqfree q*$'$
using *novar0-qfree explode-minf-unify-novar0 decrvars-qfree*
by *simp*
qed

lemma *cooper-corr*: ($\bigwedge$ *q q*$'$ *ats*. $[\![$*isqfree q* ; *cooper q* = *Some q*$']\!]$ $\Longrightarrow$ (*qinterp ats* (*QEx q*)) = (*qinterp ats q*$'$)) (**is** $\bigwedge$ *q q*$'$ *ats*. $[\![$ - ; - $]\!]$ $\Longrightarrow$ (*?P ats* (*QEx q*) = *?P ats q*$'$))
proof−
fix *q q*$'$ *ats*
assume *qfq*: *isqfree q*
and *qeq*: *cooper q* = *Some q*$'$
from *qeq* **have** $\exists$ *p*. *linform* (*nnf q*) = *Some p*
by (*cases linform* (*nnf q*)) (*simp-all add*: *cooper-def*)
then obtain *p* **where** *p-def*: *linform* (*nnf q*) = *Some p* **by** *blast*
have *linp*: *islinform p* **using** *p-def linform-lin nnf-isnnf qfq* **by** *auto*
have *qfp*: *isqfree p* **using** *linp linform-isqfree* **by** *simp*
have *nnfq*: *isnnf* (*nnf q*) **using** *nnf-isnnf qfq* **by** *simp*
then have *nnfp*: *isnnf p* **using** *linform-nnf*[*OF nnfq*] *p-def* **by** *auto*

have ∀ *ats*. *?P ats q* = *?P ats* (*nnf q*) **using** *nnf-corr qfq* **by** *auto*
then have *qeqp*: ∀ *ats*. *?P ats q* = *?P ats p*
using *linform-corr p-def nnf-isnnf qfq*
by *auto*

have *cooper q* = *Some* (*decrvars* (*explode-minf* (*unify p*))) **using** *p-def*
by (*simp add*: *cooper-def del*: *explode-minf.simps*)
then have *decr*: q' = *decrvars*(*explode-minf* (*unify p*)) **using** *qeq* **by** *simp*
have *eqq*:*?P ats* (*QEx q*) = *?P ats* (*QEx p*) **using** *qeqp* **by** *auto*
with *decr explode-minf-unify-corr2 unified-islinform unify-unified linp*
show *?P ats* (*QEx q*) = *?P ats* q' **by** *simp*
qed

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

lemma *psimpl-qfree*: *isqfree p* ⟹ *isqfree* (*psimpl p*)
apply(*induct p rule*: *isqfree.induct*)
apply(*auto simp add*: *Let-def measure-def inv-image-def*)
apply (*simp-all cong del*: *QF.weak-case-cong add*: *Let-def*)
apply (*case-tac psimpl p*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl p*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl p*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)

apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl p*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)
apply (*case-tac psimpl q*, *auto*)

apply (*case-tac psimpl p*, *auto*)
apply (*case-tac lift-bin* (λ*x y*. *lin-add* (*x*, *lin-neg y*), *linearize y*,
linearize z), *auto*)
apply (*case-tac a*,*auto*)
apply (*case-tac lift-bin* (λ*x y*. *lin-add* (*x*, *lin-neg y*), *linearize ac*,
linearize ad), *auto*)
apply (*case-tac a*,*auto*)
apply (*case-tac ae*, *auto*)
apply (*case-tac linearize af*, *auto*)
by (*case-tac a*, *auto*)

theorem *pa-qfree*: $\bigwedge p'$. *pa p* = *Some* $p' \Longrightarrow$ *isqfree* p'
proof(*simp only*: *pa-def*)
fix p'
assume *qep*: *lift-un psimpl* (*qelim* (*cooper*, *p*)) = *Some* p'
then have $\exists$ *q*. *qelim* (*cooper*, *p*) = *Some q*
by (*cases qelim*(*cooper*, *p*)) *auto*
then obtain *q* **where** *q-def*: *qelim* (*cooper*, *p*) = *Some q* **by** *blast*
have $\bigwedge q\ q'$. $[\![$*isqfree q*; *cooper q* = *Some* $q'$$]\!] \Longrightarrow$ *isqfree* q' **using** *cooper-qfree* **by** *blast*
with *q-def*
have *isqfree q* **using** *qelim-qfree* **by** *blast*
then have *isqfree* (*psimpl q*) **using** *psimpl-qfree*
by *auto*
then show *isqfree* p'
using *prems*

by *simp*

qed

theorem *pa-corr*:
$\bigwedge$ *p'*. *pa p* = *Some p'* $\Longrightarrow$ (*qinterp ats p* = *qinterp ats p'*)
proof(*simp only*: *pa-def*)
fix *p'*
assume *qep*: *lift-un psimpl* (*qelim*(*cooper*, *p*)) = *Some p'*
then have $\exists$ *q*. *qelim* (*cooper*, *p*) = *Some q*
by (*cases qelim*(*cooper*, *p*)) *auto*
then obtain *q* **where** *q-def*: *qelim* (*cooper*, *p*) = *Some q* **by** *blast*
have *cp1*:$\bigwedge$*q q' ats*.
⟦*isqfree q*; *cooper q* = *Some q'*⟧ $\Longrightarrow$ *qinterp ats* (*QEx q*) = *qinterp ats q'*
using *cooper-corr* **by** *blast*
moreover have *cp2*: $\bigwedge$*q q'*. ⟦*isqfree q*; *cooper q* = *Some q'*⟧ $\Longrightarrow$ *isqfree q'*
using *cooper-qfree* **by** *blast*
ultimately have *qinterp ats p* = *qinterp ats q* **using** *qelim-corr qep psimpl-corr q-def*
by *blast*
then have *qinterp ats p* = *qinterp ats* (*psimpl q*) **using** *psimpl-corr q-def*
by *auto*
then show *qinterp ats p* = *qinterp ats p'* **using** *prems*
by *simp*
qed

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*))))
by (*simp add*: *measure-def inv-image-def*)

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′*))))

by (*simp add*: *measure-def inv-image-def*)

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′*))))

by (*simp add*: *measure-def inv-image-def*)

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′*))))
by (*simp add*: *measure-def inv-image-def*)

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*
apply (*induct t*)
apply *auto*
done

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

The famous relationship between the numbers of leaves and nodes.

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

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

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

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

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

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

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

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 × 'a) *set* => 'a *set*
inductive *acc r*
  intros
    accI: (!!*y*. (*y*, *x*) ∈ *r* ==> *y* ∈ *acc r*) ==> *x* ∈ *acc r*

syntax
  termi :: ('a × 'a) *set* => 'a *set*
translations
  termi r == *acc* (r^{-1})

26.2 Induction rules

theorem *acc-induct*:
  a ∈ *acc r* ==>
    (!!*x*. *x* ∈ *acc r* ==> ∀ *y*. (*y*, *x*) ∈ *r* --> *P y* ==> *P x*) ==> *P a*
proof −
  assume *major*: *a* ∈ *acc r*
  assume *hyp*: !!*x*. *x* ∈ *acc r* ==> ∀ *y*. (*y*, *x*) ∈ *r* --> *P y* ==> *P x*
  show *?thesis*
    apply (*rule major* [*THEN acc.induct*])
    apply (*rule hyp*)
     apply (*rule accI*)
     apply *fast*
    apply *fast*
    done
qed

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

theorem *acc-downward*: *b* ∈ *acc r* ==> (*a*, *b*) ∈ *r* ==> *a* ∈ *acc r*
  apply (*erule acc.elims*)
  apply *fast*
  done

lemma *acc-downwards-aux*: (*b*, *a*) ∈ r^* ==> *a* ∈ *acc r* --> *b* ∈ *acc r*
  apply (*erule rtrancl-induct*)
   apply *blast*
  apply (*blast dest*: *acc-downward*)
  done

theorem *acc-downwards*: *a* ∈ *acc r* ==> (*b*, *a*) ∈ r^* ==> *b* ∈ *acc r*
  apply (*blast dest*: *acc-downwards-aux*)
  done

theorem *acc-wfI*: ∀ *x*. *x* ∈ *acc r* ==> *wf r*
  apply (*rule wfUNIVI*)
  apply (*induct-tac P x rule*: *acc-induct*)
   apply *blast*

```
  apply blast
  done

theorem acc-wfD: wf r ==> x ∈ acc r
  apply (erule wf-induct)
  apply (rule accI)
  apply blast
  done

theorem wf-acc-iff: wf r = (∀ x. x ∈ acc r)
  apply (blast intro: acc-wfI dest: acc-wfD)
  done

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
  show (λx. 0::nat) ∈ ?multiset by simp
qed

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*} **..**

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* $\Rightarrow$ $'a$ *multiset* $\Rightarrow$ $'a$ *multiset* (**infixl** #$\cap$ 70)
multiset-inter A $B \equiv A - (A - B)$

Preservation of the representing set *multiset*.

lemma *const0-in-multiset* [*simp*]: (λa. 0) $\in$ *multiset*
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset* [*simp*]: (λb. *if* $b = a$ *then* 1 *else* 0) $\in$ *multiset*
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset* [*simp*]:
$M \in$ *multiset* ==> $N \in$ *multiset* ==> (λa. M a + N a) $\in$ *multiset*
apply (*simp add*: *multiset-def*)
apply (*drule* (*1*) *finite-UnI*)
apply (*simp del*: *finite-Un add*: *Un-def*)
done

lemma *diff-preserves-multiset* [*simp*]:
$M \in$ *multiset* ==> (λa. M a − N a) $\in$ *multiset*
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*)
apply *auto*
done

27.2 Algebraic properties of multisets

27.2.1 Union

lemma *union-empty* [*simp*]: M + {#} = M $\wedge$ {#} + M = M
by (*simp add*: *union-def Mempty-def*)

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

by (*simp add*: *union-def add-ac*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
by (*simp add*: *union-def add-ac*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
proof −
have $M + (N + K) = (N + K) + M$
by (*rule union-commute*)
also have $\ldots = N + (K + M)$
by (*rule union-assoc*)
also have $K + M = M + K$
by (*rule union-commute*)
finally show *?thesis* **.**
qed

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

instance *multiset* :: (*type*) *comm-monoid-add*
proof
fix $a\ b\ c$ :: *'a multiset*
show $(a + b) + c = a + (b + c)$ **by** (*rule union-assoc*)
show $a + b = b + a$ **by** (*rule union-commute*)
show $0 + a = a$ **by** *simp*
qed

27.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by (*simp add*: *Mempty-def diff-def*)

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
by (*simp add*: *union-def diff-def*)

27.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}$ $a = 0$
by (*simp add*: *count-def Mempty-def*)

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

lemma *count-union* [*simp*]: *count* $(M + N)$ a = *count* M a + *count* N a
by (*simp add*: *count-def union-def*)

lemma *count-diff* [*simp*]: *count* $(M - N)$ a = *count* M a − *count* N a
by (*simp add*: *count-def diff-def*)

27.2.4 Set of elements

lemma *set-of-empty* [*simp*]: *set-of* {#} = {}
 by (*simp add*: *set-of-def*)

lemma *set-of-single* [*simp*]: *set-of* {#*b*#} = {*b*}
 by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
 by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
 by (*auto simp add*: *set-of-def Mempty-def count-def expand-fun-eq*)

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
 by (*auto simp add*: *set-of-def*)

27.2.5 Size

lemma *size-empty* [*simp*]: *size* {#} = *0*
 by (*simp add*: *size-def*)

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
 by (*simp add*: *size-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
 using *Rep-multiset* [*of M*]
 by (*simp add*: *multiset-def set-of-def count-def*)

lemma *setsum-count-Int*:
  finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
 apply (*erule finite-induct*)
  apply *simp*
 apply (*simp add*: *Int-insert-left set-of-def*)
 done

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
 apply (*unfold size-def*)
 apply (*subgoal-tac count* (*M* + *N*) = (λ*a*. *count M a* + *count N a*))
  prefer *2*
  apply (*rule ext*, *simp*)
 apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-nat setsum-addf setsum-count-Int*)
 apply (*subst Int-commute*)
 apply (*simp* (*no-asm-simp*) *add*: *setsum-count-Int*)
 done

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
 apply (*unfold size-def Mempty-def count-def*, *auto*)
 apply (*simp add*: *set-of-def count-def expand-fun-eq*)
 done

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ==> ∃ *a*. *a* :# *M*
  apply (*unfold size-def*)
  apply (*drule setsum-SucD*, *auto*)
  done

27.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: (*M* = *N*) = (∀ *a*. *count M a* = *count N a*)
  by (*simp add*: *count-def expand-fun-eq*)

lemma *single-not-empty* [*simp*]: {#*a*#} ≠ {#} ∧ {#} ≠ {#*a*#}
  by (*simp add*: *single-def Mempty-def expand-fun-eq*)

lemma *single-eq-single* [*simp*]: ({#*a*#} = {#*b*#}) = (*a* = *b*)
  by (*auto simp add*: *single-def expand-fun-eq*)

lemma *union-eq-empty* [*iff*]: (*M* + *N* = {#}) = (*M* = {#} ∧ *N* = {#})
  by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *empty-eq-union* [*iff*]: ({#} = *M* + *N*) = (*M* = {#} ∧ *N* = {#})
  by (*auto simp add*: *union-def Mempty-def expand-fun-eq*)

lemma *union-right-cancel* [*simp*]: (*M* + *K* = *N* + *K*) = (*M* = (*N*::'*a multiset*))
  by (*simp add*: *union-def expand-fun-eq*)

lemma *union-left-cancel* [*simp*]: (*K* + *M* = *K* + *N*) = (*M* = (*N*::'*a multiset*))
  by (*simp add*: *union-def expand-fun-eq*)

lemma *union-is-single*:
    (*M* + *N* = {#*a*#}) = (*M* = {#*a*#} ∧ *N*={#} ∨ *M* = {#} ∧ *N* = {#*a*#})
  apply (*simp add*: *Mempty-def single-def union-def add-is-1 expand-fun-eq*)
  apply *blast*
  done

lemma *single-is-union*:
    ({#*a*#} = *M* + *N*) = ({#*a*#} = *M* ∧ *N* = {#} ∨ *M* = {#} ∧ {#*a*#} = *N*)
  apply (*unfold Mempty-def single-def union-def*)
  apply (*simp add*: *add-is-1 one-is-add expand-fun-eq*)
  apply (*blast dest*: *sym*)
  done

lemma *add-eq-conv-diff*:
  (*M* + {#*a*#} = *N* + {#*b*#}) =
   (*M* = *N* ∧ *a* = *b* ∨ *M* = *N* − {#*a*#} + {#*b*#} ∧ *N* = *M* − {#*b*#} + {#*a*#})
  apply (*unfold single-def union-def diff-def*)
  apply (*simp* (*no-asm*) *add*: *expand-fun-eq*)

apply (*rule conjI*, *force*, *safe*, *simp-all*)
apply (*simp add*: *eq-sym-conv*)
done

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

27.2.7 Intersection

lemma *multiset-inter-count*:
count (A #∩ B) x = *min* (*count A x*) (*count B x*)
by (*simp add*: *multiset-inter-def min-def*)

lemma *multiset-inter-commute*: A #∩ B = B #∩ A
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-max.below-inf.inf-commute*)

lemma *multiset-inter-assoc*: A #∩ (B #∩ C) = A #∩ B #∩ C
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-max.below-inf.inf-assoc*)

lemma *multiset-inter-left-commute*: A #∩ (B #∩ C) = B #∩ (A #∩ C)
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*)

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

lemma *multiset-union-diff-commute*: B #∩ C = {#} $\Longrightarrow$ $A + B - C = A - C + B$
apply (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def split*: *split-if-asm*)
apply *clarsimp*
apply (*erule-tac* $x = a$ **in** *allE*)
apply *auto*
done

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*)
apply (*erule finite-induct*, *auto*)
apply (*drule-tac* $a = a$ **in** *mk-disjoint-insert*, *auto*)
done

lemma *rep-multiset-induct-aux*:
assumes *P* (λ*a*. (*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 -
  note premises = prems [unfolded multiset-def]
  show ?thesis
    apply (unfold multiset-def)
    apply (induct-tac n, simp, clarify)
     apply (subgoal-tac f = (λa.0))
      apply simp
      apply (rule premises)
     apply (rule ext, force, clarify)
    apply (frule setsum-SucD, clarify)
    apply (rename-tac a)
    apply (subgoal-tac finite {x. 0 < (f (a := f a - 1)) x})
     prefer 2
     apply (rule finite-subset)
      prefer 2
      apply assumption
     apply simp
     apply blast
    apply (subgoal-tac f = (f (a := f a - 1))(a := (f (a := f a - 1)) a + 1))
     prefer 2
     apply (rule ext)
     apply (simp (no-asm-simp))
     apply (erule ssubst, rule premises, blast)
    apply (erule allE, erule impE, erule-tac [2] mp, blast)
    apply (simp (no-asm-simp) add: setsum-decr del: fun-upd-apply One-nat-def)
    apply (subgoal-tac {x. x ≠ a --> 0 < f x} = {x. 0 < f x})
     prefer 2
     apply blast
    apply (subgoal-tac {x. x ≠ a ∧ 0 < f x} = {x. 0 < f x} - {a})
     prefer 2
     apply blast
    apply (simp add: le-imp-diff-is-add setsum-diff1-nat cong: conj-cong)
    done
qed

theorem rep-multiset-induct:
  f ∈ multiset ==> P (λa. 0) ==>
    (!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
  using rep-multiset-induct-aux by blast

theorem multiset-induct [induct type: multiset]:
  assumes prem1: P {#}
    and prem2: !!M x. P M ==> P (M + {#x#})
  shows P M
proof -
  note defns = union-def single-def Mempty-def
  show ?thesis
    apply (rule Rep-multiset-inverse [THEN subst])
    apply (rule Rep-multiset [THEN rep-multiset-induct])
```

apply (*rule prem1* [*unfolded defns*])
apply (*subgoal-tac* $f(b := f\ b + 1) = (\lambda a.\ f\ a + (\textit{if } a{=}b \textit{ then } 1 \textit{ else } 0)))$
prefer *2*
apply (*simp add*: *expand-fun-eq*)
apply (*erule ssubst*)
apply (*erule Abs-multiset-inverse* [*THEN subst*])
apply (*erule prem2* [*unfolded defns*, *simplified*])
done
qed

lemma *MCollect-preserves-multiset*:
$M \in multiset ==> (\lambda x.\ \textit{if } P\ x \textit{ then } M\ x \textit{ else } 0) \in multiset$
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*, *auto*)
done

lemma *count-MCollect* [*simp*]:
count $\{\#\ x{:}M.\ P\ x\ \#\}\ a = (\textit{if } P\ a \textit{ then count } M\ a \textit{ else } 0)$
by (*simp add*: *count-def MCollect-def MCollect-preserves-multiset*)

lemma *set-of-MCollect* [*simp*]: *set-of* $\{\#\ x{:}M.\ P\ x\ \#\} =$ *set-of* $M \cap \{x.\ P\ x\}$
by (*auto simp add*: *set-of-def*)

lemma *multiset-partition*: $M = \{\#\ x{:}M.\ P\ x\ \#\} + \{\#\ x{:}M.\ \neg\ P\ x\ \#\}$
by (*subst multiset-eq-conv-count-eq*, *auto*)

lemma *add-eq-conv-ex*:
$(M + \{\#a\#\} = N + \{\#b\#\}) =$
$(M = N \wedge a = b \vee (\exists K.\ M = K + \{\#b\#\} \wedge N = K + \{\#a\#\}))$
by (*auto simp add*: *add-eq-conv-diff*)

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

27.4 Multiset orderings

27.4.1 Well-foundedness

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

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

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
by (*simp add*: *mult1-def*)

lemma *less-add*: $(N,\ M0 + \{\#a\#\}) \in mult1\ r ==>$

```
    (∃M. (M, M0) ∈ mult1 r ∧ N = M + {#a#}) ∨
    (∃K. (∀b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K)
  (concl is ?case1 (mult1 r) ∨ ?case2)
proof (unfold mult1-def)
  let ?r = λK a. ∀b. b :# K --> (b, a) ∈ r
  let ?R = λN M. ∃a M0 K. M = M0 + {#a#} ∧ N = M0 + K ∧ ?r K a
  let ?case1 = ?case1 {(N, M). ?R N M}

  assume (N, M0 + {#a#}) ∈ {(N, M). ?R N M}
  hence ∃a' M0' K.
      M0 + {#a#} = M0' + {#a'#} ∧ N = M0' + K ∧ ?r K a' by simp
  thus ?case1 ∨ ?case2
  proof (elim exE conjE)
    fix a' M0' K
    assume N: N = M0' + K and r: ?r K a'
    assume M0 + {#a#} = M0' + {#a'#}
    hence M0 = M0' ∧ a = a' ∨
        (∃K'. M0 = K' + {#a'#} ∧ M0' = K' + {#a#})
      by (simp only: add-eq-conv-ex)
    thus ?thesis
    proof (elim disjE conjE exE)
      assume M0 = M0' a = a'
      with N r have ?r K a ∧ N = M0 + K by simp
      hence ?case2 .. thus ?thesis ..
    next
      fix K'
      assume M0' = K' + {#a#}
      with N have n: N = K' + K + {#a#} by (simp add: union-ac)

      assume M0 = K' + {#a'#}
      with r have ?R (K' + K) M0 by blast
      with n have ?case1 by simp thus ?thesis ..
    qed
  qed
qed

lemma all-accessible: wf r ==> ∀M. M ∈ acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = acc ?R
  {
    fix M M0 a
    assume M0: M0 ∈ ?W
      and wf-hyp: !!b. (b, a) ∈ r ==> (∀M ∈ ?W. M + {#b#} ∈ ?W)
      and acc-hyp: ∀M. (M, M0) ∈ ?R --> M + {#a#} ∈ ?W
    have M0 + {#a#} ∈ ?W
    proof (rule accI [of M0 + {#a#}])
      fix N
      assume (N, M0 + {#a#}) ∈ ?R
```

```
    hence ((∃ M. (M, M0) ∈ ?R ∧ N = M + {#a#}) ∨
        (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K))
      by (rule less-add)
    thus N ∈ ?W
    proof (elim exE disjE conjE)
      fix M assume (M, M0) ∈ ?R and N: N = M + {#a#}
      from acc-hyp have (M, M0) ∈ ?R --> M + {#a#} ∈ ?W ..
      hence M + {#a#} ∈ ?W ..
      thus N ∈ ?W by (simp only: N)
    next
      fix K
      assume N: N = M0 + K
      assume ∀ b. b :# K --> (b, a) ∈ r
      have ?this --> M0 + K ∈ ?W (is ?P K)
      proof (induct K)
        from M0 have M0 + {#} ∈ ?W by simp
        thus ?P {#} ..

        fix K x assume hyp: ?P K
        show ?P (K + {#x#})
        proof
          assume a: ∀ b. b :# (K + {#x#}) --> (b, a) ∈ r
          hence (x, a) ∈ r by simp
          with wf-hyp have b: ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast

          from a hyp have M0 + K ∈ ?W by simp
          with b have (M0 + K) + {#x#} ∈ ?W ..
          thus M0 + (K + {#x#}) ∈ ?W by (simp only: union-assoc)
        qed
      qed
      hence M0 + K ∈ ?W ..
      thus N ∈ ?W by (simp only: N)
    qed
  qed
} note tedious-reasoning = this

assume wf: wf r
fix M
show M ∈ ?W
proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
    fix b assume (b, {#}) ∈ ?R
    with not-less-empty show b ∈ ?W by contradiction
  qed

  fix M a assume M ∈ ?W
  from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
  proof induct
```

fix a
assume $!!b.\ (b,\ a) \in r \ {=}{=}{>}\ (\forall M \in\ ?W.\ M + \{\#b\#\} \in\ ?W)$
show $\forall M \in\ ?W.\ M + \{\#a\#\} \in\ ?W$
proof
fix M **assume** $M \in\ ?W$
thus $M + \{\#a\#\} \in\ ?W$
by (*rule acc-induct*) (*rule tedious-reasoning*)
qed
qed
thus $M + \{\#a\#\} \in\ ?W$ **..**
qed
qed

theorem *wf-mult1*: *wf* r ==> *wf* (*mult1* r)
by (*rule acc-wfI*, *rule all-accessible*)

theorem *wf-mult*: *wf* r ==> *wf* (*mult* r)
by (*unfold mult-def*, *rule wf-trancl*, *rule wf-mult1*)

27.4.2 Closure-free presentation

lemma *diff-union-single-conv*: a :# J ==> $I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
by (*simp add*: *multiset-eq-conv-count-eq*)

One direction.

lemma *mult-implies-one-step*:
trans r ==> $(M,\ N) \in$ *mult* r ==>
$\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
$(\forall k \in$ *set-of* $K.\ \exists j \in$ *set-of* $J.\ (k,\ j) \in r)$
apply (*unfold mult-def mult1-def set-of-def*)
apply (*erule converse-trancl-induct*, *clarify*)
apply (*rule-tac* $x = M0$ **in** *exI*, *simp*, *clarify*)
apply (*case-tac* a :# K)
apply (*rule-tac* $x = I$ **in** *exI*)
apply (*simp* (*no-asm*))
apply (*rule-tac* $x = (K - \{\#a\#\}) + Ka$ **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *union-assoc* [*symmetric*])
apply (*drule-tac* $f = \lambda M.\ M - \{\#a\#\}$ **in** *arg-cong*)
apply (*simp add*: *diff-union-single-conv*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply *blast*
apply (*subgoal-tac* a :# I)
apply (*rule-tac* $x = I - \{\#a\#\}$ **in** *exI*)
apply (*rule-tac* $x = J + \{\#a\#\}$ **in** *exI*)
apply (*rule-tac* $x = K + Ka$ **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
apply (*rule conjI*)
apply (*drule-tac* $f = \lambda M.\ M - \{\#a\#\}$ **in** *arg-cong*, *simp*)

apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply *blast*
apply (*subgoal-tac a* :# (*M0* + {#*a*#}))
apply *simp*
apply (*simp* (*no-asm*))
done

lemma *elem-imp-eq-diff-union*: *a* :# *M* ==> *M* = *M* − {#*a*#} + {#*a*#}
by (*simp add*: *multiset-eq-conv-count-eq*)

lemma *size-eq-Suc-imp-eq-union*: *size M* = *Suc n* ==> ∃ *a N*. *M* = *N* + {#*a*#}
apply (*erule size-eq-Suc-imp-elem* [*THEN exE*])
apply (*drule elem-imp-eq-diff-union*, *auto*)
done

lemma *one-step-implies-mult-aux*:
trans r ==>
∀ *I J K*. (*size J* = *n* ∧ *J* ≠ {#} ∧ (∀ *k* ∈ *set-of K*. ∃ *j* ∈ *set-of J*. (*k*, *j*) ∈ *r*))
−−> (*I* + *K*, *I* + *J*) ∈ *mult r*
apply (*induct-tac n*, *auto*)
apply (*frule size-eq-Suc-imp-eq-union*, *clarify*)
apply (*rename-tac J′*, *simp*)
apply (*erule notE*, *auto*)
apply (*case-tac J′* = {#})
apply (*simp add*: *mult-def*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*, *blast*)

Now we know *J′* ≠ {#}.

apply (*cut-tac M* = *K* **and** *P* = λ*x*. (*x*, *a*) ∈ *r* **in** *multiset-partition*)
apply (*erule-tac P* = ∀ *k* ∈ *set-of K*. *?P k* **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
((*I* + {# *x* : *K*. (*x*, *a*) ∈ *r* #}) + {# *x* : *K*. (*x*, *a*) ∉ *r* #},
(*I* + {# *x* : *K*. (*x*, *a*) ∈ *r* #}) + *J′*) ∈ *mult r*)
prefer *2*
apply *force*
apply (*simp* (*no-asm-use*) *add*: *union-assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*)
apply (*rule-tac x* = *a* **in** *exI*)
apply (*rule-tac x* = *I* + *J′* **in** *exI*)
apply (*simp add*: *union-ac*)
done

lemma *one-step-implies-mult*:

trans r ==> $J \neq \{\#\}$ ==> $\forall\, k \in$ *set-of* $K.\ \exists\, j \in$ *set-of* $J.\ (k, j) \in r$
==> $(I + K, I + J) \in$ *mult* r
apply (*insert one-step-implies-mult-aux*, *blast*)
done

27.4.3 Partial-order properties

instance *multiset* :: (*type*) *ord* ..

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

lemma *trans-base-order*: *trans* $\{(x', x).\ x' < (x::'a::order)\}$
apply (*unfold trans-def*)
apply (*blast intro*: *order-less-trans*)
done

Irreflexivity.

lemma *mult-irrefl-aux*:
finite A ==> $(\forall\, x \in A.\ \exists\, y \in A.\ x < (y::'a::order))$ --> $A = \{\}$
apply (*erule finite-induct*)
apply (*auto intro*: *order-less-trans*)
done

lemma *mult-less-not-refl*: $\neg\ M < (M::'a::order\ multiset)$
apply (*unfold less-multiset-def*, *auto*)
apply (*drule trans-base-order* [*THEN mult-implies-one-step*], *auto*)
apply (*drule finite-set-of* [*THEN mult-irrefl-aux* [*rule-format* (*no-asm*)]])
apply (*simp add*: *set-of-eq-empty-iff*)
done

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset)$ ==> R
by (*insert mult-less-not-refl*, *fast*)

Transitivity.

theorem *mult-less-trans*: $K < M$ ==> $M < N$ ==> $K < (N::'a::order\ multiset)$
apply (*unfold less-multiset-def mult-def*)
apply (*blast intro*: *trancl-trans*)
done

Asymmetry.

theorem *mult-less-not-sym*: $M < N$ ==> $\neg\ N < (M::'a::order\ multiset)$
apply *auto*
apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

theorem *mult-less-asym*:
  $M < N ==> (\neg\ P ==> N < (M::'a::order\ multiset)) ==> P$
 by (*insert mult-less-not-sym*, *blast*)

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
by (*unfold le-multiset-def*, *auto*)

Anti-symmetry.

theorem *mult-le-antisym*:
  $M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
 apply (*unfold le-multiset-def*)
 apply (*blast dest*: *mult-less-not-sym*)
 done

Transitivity.

theorem *mult-le-trans*:
  $K <= M ==> M <= N ==> K <= (N::'a::order\ multiset)$
 apply (*unfold le-multiset-def*)
 apply (*blast intro*: *mult-less-trans*)
 done

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
by (*unfold le-multiset-def*, *auto*)

Partial order.

instance *multiset* :: (*order*) *order*
 apply *intro-classes*
   apply (*rule mult-le-refl*)
  apply (*erule mult-le-trans*, *assumption*)
  apply (*erule mult-le-antisym*, *assumption*)
 apply (*rule mult-less-le*)
 done

27.4.4 Monotonicity of multiset union

lemma *mult1-union*:
  $(B, D) \in mult1\ r ==> trans\ r ==> (C + B, C + D) \in mult1\ r$
 apply (*unfold mult1-def*, *auto*)
 apply (*rule-tac* $x = a$ **in** *exI*)
 apply (*rule-tac* $x = C + M0$ **in** *exI*)
 apply (*simp add*: *union-assoc*)
 done

lemma *union-less-mono2*: $B < D ==> C + B < C + (D::'a::order\ multiset)$
 apply (*unfold less-multiset-def mult-def*)
 apply (*erule trancl-induct*)
  apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl*)
 apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl trancl-trans*)
 done

lemma *union-less-mono1*: $B < D ==> B + C < D + (C::'a::order\ multiset)$
  apply (*subst union-commute* [*of B C*])
  apply (*subst union-commute* [*of D C*])
  apply (*erule union-less-mono2*)
  done

lemma *union-less-mono*:
    $A < C ==> B < D ==> A + B < C + (D::'a::order\ multiset)$
  apply (*blast intro*!: *union-less-mono1 union-less-mono2 mult-less-trans*)
  done

lemma *union-le-mono*:
    $A <= C ==> B <= D ==> A + B <= C + (D::'a::order\ multiset)$
  apply (*unfold le-multiset-def*)
  apply (*blast intro*: *union-less-mono union-less-mono1 union-less-mono2*)
  done

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
  apply (*unfold le-multiset-def less-multiset-def*)
  apply (*case-tac* $M = \{\#\}$)
   prefer *2*
   apply (*subgoal-tac* $(\{\#\} + \{\#\}, \{\#\} + M) \in$ *mult* (*Collect* (*split op* $<$)))
    prefer *2*
    apply (*rule one-step-implies-mult*)
      apply (*simp only*: *trans-def*, *auto*)
  done

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
proof −
  have $A + \{\#\} <= A + B$ **by** (*blast intro*: *union-le-mono*)
  thus *?thesis* **by** *simp*
qed

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
by (*subst union-commute*, *rule union-upper1*)

27.5 Link with lists

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

lemma *multiset-of-zero-iff* [*simp*]: (*multiset-of* $x = \{\#\}) = (x = [])$
  by (*induct-tac* x, *auto*)

lemma *multiset-of-zero-iff-right*[*simp*]: $(\{\#\} =$ *multiset-of* $x) = (x = [])$

by (*induct-tac x*, *auto*)

lemma *set-of-multiset-of*[*simp*]: *set-of*(*multiset-of x*) = *set x*
by (*induct-tac x*, *auto*)

lemma *mem-set-multiset-eq*: $x \in$ *set xs* = (x :# *multiset-of xs*)
by (*induct xs*) *auto*

lemma *multiset-of-append*[*simp*]:
multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
by (*rule-tac x=ys* **in** *spec*, *induct-tac xs*, *auto simp*: *union-ac*)

lemma *surj-multiset-of*: *surj multiset-of*
apply (*unfold surj-def*, *rule allI*)
apply (*rule-tac M=y* **in** *multiset-induct*, *auto*)
apply (*rule-tac x = x # xa* **in** *exI*, *auto*)
done

lemma *set-count-greater-0*: *set x* = {*a*. *0* < *count* (*multiset-of x*) *a*}
by (*induct-tac x*, *auto*)

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* $\in$ *set x then 1 else 0*))
apply (*induct-tac x*, *simp*, *rule iffI*, *simp-all*)
apply (*rule conjI*)
apply (*simp-all add*: *set-of-multiset-of* [*THEN sym*] *del*: *set-of-multiset-of*)
apply (*erule-tac x=a* **in** *allE*, *simp*, *clarify*)
apply (*erule-tac x=aa* **in** *allE*, *simp*)
done

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of ys* $\Longrightarrow$ *set xs* = *set ys*
by (*rule*) (*auto simp add*:*multiset-eq-conv-count-eq set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
⟦*distinct x*; *distinct y*⟧
$\Longrightarrow$ (*set x* = *set y*) = (*multiset-of x* = *multiset-of y*)
by (*auto simp*: *multiset-eq-conv-count-eq distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups*[*THEN distinct-remdups*
[*THEN set-eq-iff-multiset-of-eq-distinct*[*THEN iffD2*]]])
apply *simp*
done

lemma *multiset-of-compl-union*[*simp*]:

multiset-of $[x{\in}xs.\ P\ x]$ + *multiset-of* $[x{\in}xs.\ \neg P\ x]$ = *multiset-of xs*
by (*induct xs*) (*auto simp*: *union-ac*)

lemma *count-filter*:
count (*multiset-of xs*) *x* = *length* $[y \in xs.\ y = x]$
by (*induct xs*, *auto*)

27.6 Pointwise ordering induced by count

consts
mset-le :: $['a\ multiset,\ 'a\ multiset] \Rightarrow bool$

syntax
-mset-le :: $'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (- ≤# - [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*
by (*unfold mset-le-def*) *auto*

lemma *mset-le-trans*: ⟦ *xs* ≤# *ys*; *ys* ≤# *zs* ⟧ ⟹ *xs* ≤# *zs*
by (*unfold mset-le-def*) (*fast intro*: *order-trans*)

lemma *mset-le-antisym*: ⟦ *xs*≤# *ys*; *ys* ≤# *xs*⟧ ⟹ *xs* = *ys*
apply (*unfold mset-le-def*)
apply (*rule multiset-eq-conv-count-eq*[*THEN iffD2*])
apply (*blast intro*: *order-antisym*)
done

lemma *mset-le-exists-conv*:
(*xs* ≤# *ys*) = (∃ *zs*. *ys* = *xs* + *zs*)
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac x* = *ys* − *xs* **in** *exI*)
apply (*auto intro*: *multiset-eq-conv-count-eq* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel*[*simp*]: (*xs* + *zs* ≤# *ys* + *zs*) = (*xs* ≤# *ys*)
by (*unfold mset-le-def*) *auto*

lemma *mset-le-mono-add-left-cancel*[*simp*]: (*zs* + *xs* ≤# *zs* + *ys*) = (*xs* ≤# *ys*)
by (*unfold mset-le-def*) *auto*

lemma *mset-le-mono-add*: ⟦ *xs* ≤# *ys*; *vs* ≤# *ws* ⟧ ⟹ *xs* + *vs* ≤# *ys* + *ws*
apply (*unfold mset-le-def*)
apply *auto*
apply (*erule-tac x*=*a* **in** *allE*)+
apply *auto*

done

lemma *mset-le-add-left*[*simp*]: *xs* $\leq$# *xs* + *ys*
by (*unfold mset-le-def*) *auto*

lemma *mset-le-add-right*[*simp*]: *ys* $\leq$# *xs* + *ys*
by (*unfold mset-le-def*) *auto*

lemma *multiset-of-remdups-le*: *multiset-of* (*remdups x*) $\leq$# *multiset-of x*
apply (*induct x*)
apply *auto*
apply (*rule mset-le-trans*)
apply *auto*
done

end

28 Sorting: Basic Theory

theory *Sorting*
imports *Main Multiset*
begin

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

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

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

constdefs
total :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool*) => *bool*
total r == ($\forall$ *x y*. *r x y* | *r y x*)

transf :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool*) => *bool*
transf f == ($\forall$ *x y z*. *f x y* & *f y z* --> *f x z*)

lemma *sorted1-is-sorted*: *transf*(*le*) ==> *sorted1 le xs* = *sorted le xs*
apply(*induct xs*)
 apply *simp*
apply(*simp split*: *list.split*)
apply(*unfold transf-def*)
apply(*blast*)
done

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

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*)
  by (*induct xs*) (*auto simp*: *union-ac*)

theorem *insort-permutes*[*simp*]:
 $\bigwedge x$. *multiset-of* (*insort le xs*) = *multiset-of xs*
  by (*induct xs*) *auto*

lemma *set-ins* [*simp*]: *set*(*ins le x xs*) = *insert x* (*set xs*)
  by (*simp add*: *set-count-greater-0*) *fast*

lemma *sorted-ins*[*simp*]:
 $[\![$ *total le*; *transf le* $]\!] \Longrightarrow$ *sorted le* (*ins le x xs*) = *sorted le xs*
apply (*induct xs*)

apply *simp-all*
apply (*unfold Sorting.total-def Sorting.transf-def*)
apply *blast*
done

theorem *sorted-insort*:
[| *total*(*le*); *transf*(*le*) |] ==> *sorted le* (*insort le xs*)
by (*induct xs*) *auto*

end

30 Quicksort

theory *Qsort*
imports *Sorting*
begin

30.1 Version 1: higher-order

consts *qsort* :: ('*a* ⇒ '*a* => *bool*) ∗ '*a list* ⇒ '*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*
by (*induct le xs rule*: *qsort.induct*) (*auto simp*: *union-ac*)

lemma *set-qsort* [*simp*]: *set* (*qsort*(*le*,*xs*)) = *set xs*
by(*simp add*: *set-count-greater-0*)

lemma *sorted-qsort*:
  total(*le*) ==> *transf*(*le*) ==> *sorted le* (*qsort*(*le*,*xs*))
apply (*induct le xs rule*: *qsort.induct*)
apply *simp*
apply *simp*
apply(*unfold Sorting.total-def Sorting.transf-def*)
apply *blast*
done

30.2 Version 2:type classes

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

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

lemma *quickSort-permutes*[*simp*]:
multiset-of (*quickSort xs*) = *multiset-of xs*
by (*induct xs rule*: *quickSort.induct*) (*auto simp*: *union-ac*)

lemma *set-quickSort*[*simp*]: *set* (*quickSort xs*) = *set xs*
by(*simp add*: *set-count-greater-0*)

theorem *sorted-quickSort*: *sorted* (*op* $\le$) (*quickSort xs*)
by (*induct xs rule*: *quickSort.induct*, *auto*)

end

31 Merge Sort

theory *MergeSort*
imports *Sorting*
begin

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

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

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

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

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

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

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

apply(*blast intro*: *order-trans*)
done

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

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

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

end

32 A question from "Bundeswettbewerb Mathematik"

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

consts *f* :: *nat* => *nat*

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

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

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

lemma *lemma2*: $f(n) < f(Suc(n))$
by (*blast intro*: *le-less-trans lemma1*)

lemma *f-mono* [*rule-format* (*no-asm*)]: $m <= n --> f(m) <= f(n)$
apply (*induct-tac n*)
 apply *simp*
apply (*rule impI*)
apply (*erule le-SucE*)
 apply (*cut-tac n = n* **in** *lemma2*, *auto*)
done

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

end

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

— once a slow step, but now (2001) just three seconds!
lemma *Lagrange-lemma*:
 !!*x1*::*'a*::*comm-ring*.
  $(sq\ x1 + sq\ x2 + sq\ x3 + sq\ x4) * (sq\ y1 + sq\ y2 + sq\ y3 + sq\ y4) =$
  $sq(x1*y1 - x2*y2 - x3*y3 - x4*y4) +$
  $sq(x1*y2 + x2*y1 + x3*y4 - x4*y3) +$
  $sq(x1*y3 - x2*y4 + x3*y1 + x4*y2) +$
  $sq(x1*y4 + x2*y3 - x3*y2 + x4*y1)$
by(*simp add*: *sq-def ring-eq-simps*)

A challenge by John Harrison. Takes about 74s on a 2.5GHz Apple G5.

lemma !!*p1*::*'a*::*comm-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*)
oops

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*)ˆ*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* ˆ *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* **by** (*cases n*) *auto*
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*)))
by (*cases n*) *simp-all*

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*)
by (*induct B*) (*auto simp add*: *mkPinj-def ring-eq-simps*)

mkPX preserves semantics

lemma *mkPX-ci*: *Ipol l* (*mkPX A b C*) = *Ipol l* (*PX A b C*)
by (*cases A*) (*auto simp add*: *mkPX-def mkPinj-ci power-add ring-eq-simps*)

Correctness theorems for the implemented operations

Negation

lemma *neg-ci*: ⋀*l*. *Ipol l* (*neg P*) = −(*Ipol l P*)
by (*induct P*) *auto*

Addition

lemma *add-ci*: $\bigwedge l.$ *Ipol l* (*add* (*P*, *Q*)) = *Ipol l P* + *Ipol l Q*
proof (*induct P Q rule*: *add.induct*)
  case (*6 x P y Q*)
  show *?case*
  proof (*rule linorder-cases*)
    assume $x < y$
    with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci ring-eq-simps*)
  next
    assume $x = y$
    with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci*)
  next
    assume $x > y$
    with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci ring-eq-simps*)
  qed
next
  case (*7 x P Q y R*)
  have $x = 0 \lor x = 1 \lor x > 1$ **by** *arith*
  moreover
  { assume $x = 0$ **with** *7* **have** *?case* **by** *simp* **}**
  moreover
  { assume $x = 1$ **with** *7* **have** *?case* **by** (*simp add*: *ring-eq-simps*) **}**
  moreover
  { assume $x > 1$ **from** *7* **have** *?case* **by** (*cases x*) *simp-all* **}**
  ultimately show *?case* **by** *blast*
next
  case (*8 P x R y Q*)
  have $y = 0 \lor y = 1 \lor y > 1$ **by** *arith*
  moreover
  { assume $y = 0$ **with** *8* **have** *?case* **by** *simp* **}**
  moreover
  { assume $y = 1$ **with** *8* **have** *?case* **by** *simp* **}**
  moreover
  { assume $y > 1$ **with** *8* **have** *?case* **by** *simp* **}**
  ultimately show *?case* **by** *blast*
next
  case (*9 P1 x P2 Q1 y Q2*)
  show *?case*
  proof (*rule linorder-cases*)
    assume *a*: $x < y$ **hence** *EX d*. $d + x = y$ **by** *arith*
    with *9 a* **show** *?case* **by** (*auto simp add*: *mkPX-ci power-add ring-eq-simps*)
  next
    assume *a*: $y < x$ **hence** *EX d*. $d + y = x$ **by** *arith*
    with *9 a* **show** *?case* **by** (*auto simp add*: *power-add mkPX-ci ring-eq-simps*)
  next
    assume $x = y$
    with *9* **show** *?case* **by** (*simp add*: *mkPX-ci ring-eq-simps*)
  qed
qed (*auto simp add*: *ring-eq-simps*)

Multiplication

lemma *mul-ci*: $\bigwedge l.$ *Ipol l* (*mul* (*P*, *Q*)) = *Ipol l P* ∗ *Ipol l Q*
by (*induct P Q rule*: *mul.induct*)
(*simp-all add*: *mkPX-ci mkPinj-ci ring-eq-simps add-ci power-add*)

Substraction

lemma *sub-ci*: *Ipol l* (*sub p q*) = *Ipol l p* − *Ipol l q*
by (*simp add*: *add-ci neg-ci sub-def*)

Square

lemma *sqr-ci*:$\bigwedge ls.$ *Ipol ls* (*sqr p*) = *Ipol ls p* ∗ *Ipol ls p*
by (*induct p*) (*simp-all add*: *add-ci mkPinj-ci mkPX-ci mul-ci ring-eq-simps power-add*)

Power

lemma *even-pow*:*even n* ⟹ *pow* (*p*, *n*) = *pow* (*sqr p*, *n div 2*) **by** (*induct n*) *simp-all*

lemma *pow-ci*: $\bigwedge p.$ *Ipol ls* (*pow* (*p*, *n*)) = (*Ipol ls p*) ˆ *n*
proof (*induct n rule*: *nat-less-induct*)
case (*1 k*)
have *two*:*2* = *Suc* (*Suc 0*) **by** *simp*
show *?case*
proof (*cases k*)
case (*Suc l*)
show *?thesis*
proof *cases*
assume *EL*: *even l*
have *Suc l div 2* = *l div 2*
by (*simp add*: *nat-number even-nat-plus-one-div-two* [*OF EL*])
moreover
from *Suc* **have** *l* < *k* **by** *simp*
with *1* **have** ∀ *p*. *Ipol ls* (*pow* (*p*, *l*)) = *Ipol ls p* ˆ *l* **by** *simp*
moreover
note *Suc EL even-nat-plus-one-div-two* [*OF EL*]
ultimately show *?thesis* **by** (*auto simp add*: *mul-ci power-Suc even-pow*)
next
assume *OL*: *odd l*
with *prems* **have** ⟦∀ *m*<*Suc l*. ∀ *p*. *Ipol ls* (*pow* (*p*, *m*)) = *Ipol ls p* ˆ *m*; *k* = *Suc l*; *odd l*⟧ ⟹ ∀ *p*. *Ipol ls* (*sqr p*) ˆ (*Suc l div 2*) = *Ipol ls p* ˆ *Suc l*
proof(*cases l*)
case (*Suc w*)
from *prems* **have** *EW*: *even w* **by** *simp*
from *two* **have** *two-times*:(*2* ∗ (*w div 2*))= *w*
by (*simp only*: *even-nat-div-two-times-two*[*OF EW*])
have *A*: $\bigwedge p.$ (*Ipol ls p* ∗ *Ipol ls p*) = (*Ipol ls p*) ˆ (*Suc* (*Suc 0*))
by (*simp add*: *power-Suc*)
from *A two* [*symmetric*] **have** *ALL p*.(*Ipol ls p* ∗ *Ipol ls p*) = (*Ipol ls p*) ˆ *2*
by *simp*
with *prems* **show** *?thesis*

    by (*auto simp add*: *power-mult*[*symmetric*, *of* - *2* -] *two-times mul-ci sqr-ci*)
   qed *simp*
   with *prems* **show** *?thesis* **by** *simp*
  qed
 next
  case *0*
  then show *?thesis* **by** *simp*
 qed
qed

Normalization preserves semantics

lemma *norm-ci*:*Ipolex l Pe* = *Ipol l* (*norm Pe*)
 by (*induct Pe*) (*simp-all add*: *add-ci sub-ci mul-ci neg-ci pow-ci*)

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 −
 from *eq* **have** *Ipol l* (*norm P1*) = *Ipol l* (*norm P2*) **by** *simp*
 thus *?thesis* **by** (*simp only*: *norm-ci*)
qed

Code generation

use *comm-ring.ML*
setup *CommRing.setup*

end

35 Some examples demonstrating the comm-ring method

theory *Commutative-RingEx*
imports *Commutative-Ring*
begin

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

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

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

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

lemma $((x::int) - y)\hat{\ }2 = x\hat{\ }2 + y\hat{\ }2 - 2*x*y$
by *comm-ring*

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

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

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

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

lemma $(a::int)\hat{\ }3 - b\hat{\ }3 = (a - b) * (a\hat{\ }2 + a*b + b\hat{\ }2)$
by *comm-ring*

lemma $(a::int)\hat{\ }3 + b\hat{\ }3 = (a + b) * (a\hat{\ }2 - a*b + b\hat{\ }2)$
by *comm-ring*

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

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

end

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*
by(*cases P*, *auto*)

lemma *norm-PX-0-False*:*isnorm* (*PX* (*Pc 0*) *i Q*) = *False*
by(*cases i*, *auto*)

lemma *norm-Pinj*:*isnorm* (*Pinj i Q*) ⟹ *isnorm Q*
by(*cases i*,*simp add*: *norm-Pinj-0-False norm-PX-0-False*,*cases Q*) *auto*

lemma *norm-PX2*:*isnorm* (*PX P i Q*) ⟹ *isnorm Q*
by(*cases i*, *auto*, *cases P*, *auto*, *case-tac pol2*, *auto*)

lemma *norm-PX1*:*isnorm* (*PX P i Q*) ⟹ *isnorm P*
by(*cases i*, *auto*, *cases P*, *auto*, *case-tac pol2*, *auto*)

lemma *mkPinj-cn*:⟦*y*~=*0*; *isnorm Q*⟧ ⟹ *isnorm* (*mkPinj y Q*)
apply(*auto simp add*: *mkPinj-def norm-Pinj-0-False split*: *pol.split*)
apply(*case-tac nat*, *auto simp add*: *norm-Pinj-0-False*)
by(*case-tac pol*, *auto*) (*case-tac y*, *auto*)

lemma *norm-PXtrans*:
  assumes *A*:*isnorm* (*PX P x Q*) **and** *isnorm Q2*
  shows *isnorm* (*PX P x Q2*)
proof(*cases P*)
  case (*PX p1 y p2*) **from** *prems* **show** *?thesis* **by**(*cases x*, *auto*, *cases p2*, *auto*)
next
  case *Pc* **from** *prems* **show** *?thesis* **by**(*cases x*, *auto*)
next
  case *Pinj* **from** *prems* **show** *?thesis* **by**(*cases x*, *auto*)
qed

lemma *norm-PXtrans2*: **assumes** *A*:*isnorm* (*PX P x Q*) **and** *isnorm Q2* **shows** *isnorm* (*PX P* (*Suc* (*n*+*x*)) *Q2*)
proof(*cases P*)
  case (*PX p1 y p2*)
  from *prems* **show** *?thesis* **by**(*cases x*, *auto*, *cases p2*, *auto*)
next
  case *Pc*
  from *prems* **show** *?thesis* **by**(*cases x*, *auto*)
next
  case *Pinj*
  from *prems* **show** *?thesis* **by**(*cases x*, *auto*)

qed

lemma *mkPX-cn*:
assumes $x \neq 0$ **and** *isnorm P* **and** *isnorm Q*
shows *isnorm* (*mkPX P x Q*)
proof(*cases P*)
case (*Pc c*)
from *prems* **show** *?thesis* **by** (*cases x*) (*auto simp add*: *mkPinj-cn mkPX-def*)
next
case (*Pinj i Q*)
from *prems* **show** *?thesis* **by** (*cases x*) (*auto simp add*: *mkPinj-cn mkPX-def*)
next
case (*PX P1 y P2*)
from *prems* **have** *Y0*:$y>0$ **by**(*cases y*, *auto*)
from *prems* **have** *isnorm P1 isnorm P2* **by** (*auto simp add*: *norm-PX1*[*of P1 y P2*] *norm-PX2*[*of P1 y P2*])
with *prems Y0* **show** *?thesis* **by** (*cases x*, *auto simp add*: *mkPX-def norm-PXtrans2*[*of P1 y - Q -*], *cases P2*, *auto*)
qed

lemma *add-cn*:⟦*isnorm P*; (*isnorm Q*)⟧ $\Longrightarrow$ *isnorm* (*add* (*P*, *Q*))
proof(*induct P Q rule*: *add.induct*)
case (*2 c i P2*) **thus** *?case* **by** (*cases P2*, *simp-all*, *cases i*, *simp-all*)
next
case (*3 i P2 c*) **thus** *?case* **by** (*cases P2*, *simp-all*, *cases i*, *simp-all*)
next
case (*4 c P2 i Q2*)
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp only*: *norm-PX1*[*of P2 i Q2*] *norm-PX2*[*of P2 i Q2*])
with *prems* **show** *?case* **by**(*cases i*, *simp*, *cases P2*, *auto*, *case-tac pol2*, *auto*)
next
case (*5 P2 i Q2 c*)
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp only*: *norm-PX1*[*of P2 i Q2*] *norm-PX2*[*of P2 i Q2*])
with *prems* **show** *?case* **by**(*cases i*, *simp*, *cases P2*, *auto*, *case-tac pol2*, *auto*)
next
case (*6 x P2 y Q2*)
from *prems* **have** *Y0*:$y>0$ **by** (*cases y*, *auto simp add*: *norm-Pinj-0-False*)
from *prems* **have** *X0*:$x>0$ **by** (*cases x*, *auto simp add*: *norm-Pinj-0-False*)
have $x < y \vee x = y \vee x > y$ **by** *arith*
moreover
{ assume $x<y$ **hence** *EX d*. $y=d+x$ **by** *arith*
then obtain *d* **where** $y=d+x$..
moreover
note *prems X0*
moreover
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp add*: *norm-Pinj*[*of -*

P2] *norm-Pinj*[*of* - *Q2*])
moreover
with *prems* **have** *isnorm* (*Pinj d Q2*) **by** (*cases d*, *simp*, *cases Q2*, *auto*)
ultimately have *?case* **by** (*simp add*: *mkPinj-cn*)**}**
moreover
{ assume *x=y*
moreover
from *prems* **have** *isnorm P2 isnorm Q2* **by**(*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-Pinj*[*of* - *Q2*])
moreover
note *prems Y0*
moreover
ultimately have *?case* **by** (*simp add*: *mkPinj-cn*) **}**
moreover
{ assume *x>y* **hence** *EX d. x=d+y* **by** *arith*
then obtain *d* **where** *x=d+y*..
moreover
note *prems Y0*
moreover
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-Pinj*[*of* - *Q2*])
moreover
with *prems* **have** *isnorm* (*Pinj d P2*) **by** (*cases d*, *simp*, *cases P2*, *auto*)
ultimately have *?case* **by** (*simp add*: *mkPinj-cn*)**}**
ultimately show *?case* **by** *blast*
next
case (*7 x P2 Q2 y R*)
have $x=0 \lor (x = 1) \lor (x > 1)$ **by** *arith*
moreover
{ assume *x=0* **with** *prems* **have** *?case* **by** (*auto simp add*: *norm-Pinj-0-False*)**}**
moreover
{ assume *x=1*
from *prems* **have** *isnorm R isnorm P2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-PX2*[*of Q2 y R*])
with *prems* **have** *isnorm* (*add* (*R*, *P2*)) **by** *simp*
with *prems* **have** *?case* **by** (*simp add*: *norm-PXtrans*[*of Q2 y* -]) **}**
moreover
{ assume $x > 1$ **hence** *EX d. x=Suc* (*Suc d*) **by** *arith*
then obtain *d* **where** *X*:*x=Suc* (*Suc d*) ..
from *prems* **have** *NR*:*isnorm R isnorm P2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-PX2*[*of Q2 y R*])
with *prems* **have** *isnorm* (*Pinj* $(x - 1)$ *P2*) **by**(*cases P2*, *auto*)
with *prems NR* **have** *isnorm*(*add* (*R*, *Pinj* $(x - 1)$ *P2*)) *isnorm*(*PX Q2 y R*) **by** *simp*
with *X* **have** *?case* **by** (*simp add*: *norm-PXtrans*[*of Q2 y* -]) **}**
ultimately show *?case* **by** *blast*
next
case (*8 Q2 y R x P2*)
have $x=0 \lor (x = 1) \lor (x > 1)$ **by** *arith*

moreover
{ assume *x=0* **with** *prems* **have** *?case* **by** (*auto simp add*: *norm-Pinj-0-False*)**}**
moreover
{ assume *x=1*
from *prems* **have** *isnorm R isnorm P2* **by** (*auto simp add*: *norm-Pinj*[*of - P2*] *norm-PX2*[*of Q2 y R*])
with *prems* **have** *isnorm* (*add* (*R*, *P2*)) **by** *simp*
with *prems* **have** *?case* **by** (*simp add*: *norm-PXtrans*[*of Q2 y -*]) **}**
moreover
{ assume $x > 1$ **hence** *EX d*. *x=Suc* (*Suc d*) **by** *arith*
then obtain *d* **where** *X*:*x=Suc* (*Suc d*) **..**
from *prems* **have** *NR*:*isnorm R isnorm P2* **by** (*auto simp add*: *norm-Pinj*[*of - P2*] *norm-PX2*[*of Q2 y R*])
with *prems* **have** *isnorm* (*Pinj* ($x - 1$) *P2*) **by**(*cases P2*, *auto*)
with *prems NR* **have** *isnorm*(*add* (*R*, *Pinj* ($x - 1$) *P2*)) *isnorm*(*PX Q2 y R*) **by** *simp*
with *X* **have** *?case* **by** (*simp add*: *norm-PXtrans*[*of Q2 y -*]) **}**
ultimately show *?case* **by** *blast*
next
case (*9 P1 x P2 Q1 y Q2*)
from *prems* **have** *Y0*:*y>0* **by**(*cases y*, *auto*)
from *prems* **have** *X0*:*x>0* **by**(*cases x*, *auto*)
from *prems* **have** *NP1*:*isnorm P1* **and** *NP2*:*isnorm P2* **by** (*auto simp add*: *norm-PX1*[*of P1 - P2*] *norm-PX2*[*of P1 - P2*])
from *prems* **have** *NQ1*:*isnorm Q1* **and** *NQ2*:*isnorm Q2* **by** (*auto simp add*: *norm-PX1*[*of Q1 - Q2*] *norm-PX2*[*of Q1 - Q2*])
have $y < x \vee x = y \vee x < y$ **by** *arith*
moreover
{assume *sm1*:$y < x$ **hence** *EX d*. *x=d+y* **by** *arith*
then obtain *d* **where** *sm2*:*x=d+y***..**
note *prems NQ1 NP1 NP2 NQ2 sm1 sm2*
moreover
have *isnorm* (*PX P1 d* (*Pc 0*))
proof(*cases P1*)
case (*PX p1 y p2*)
with *prems* **show** *?thesis* **by**(*cases d*, *simp*,*cases p2*, *auto*)
next case *Pc* **from** *prems* **show** *?thesis* **by**(*cases d*, *auto*)
next case *Pinj* **from** *prems* **show** *?thesis* **by**(*cases d*, *auto*)
qed
ultimately have *isnorm* (*add* (*P2*, *Q2*)) *isnorm* (*add* (*PX P1* ($x - y$) (*Pc 0*), *Q1*)) **by** *auto*
with *Y0 sm1 sm2* **have** *?case* **by** (*simp add*: *mkPX-cn*)**}**
moreover
{assume *x=y*
from *prems NP1 NP2 NQ1 NQ2* **have** *isnorm* (*add* (*P2*, *Q2*)) *isnorm* (*add* (*P1*, *Q1*)) **by** *auto*
with *Y0 prems* **have** *?case* **by** (*simp add*: *mkPX-cn*) **}**
moreover
{assume *sm1*:*x<y* **hence** *EX d*. *y=d+x* **by** *arith*

then obtain *d* **where** *sm2*:*y*=*d*+*x*..
note *prems NQ1 NP1 NP2 NQ2 sm1 sm2*
moreover
have *isnorm* (*PX Q1 d* (*Pc 0*))
proof(*cases Q1*)
case (*PX p1 y p2*)
with *prems* **show** *?thesis* **by**(*cases d*, *simp*,*cases p2*, *auto*)
next case *Pc* **from** *prems* **show** *?thesis* **by**(*cases d*, *auto*)
next case *Pinj* **from** *prems* **show** *?thesis* **by**(*cases d*, *auto*)
qed
ultimately have *isnorm* (*add* (*P2*, *Q2*)) *isnorm* (*add* (*PX Q1* (*y* − *x*) (*Pc 0*), *P1*)) **by** *auto*
with *X0 sm1 sm2* **have** *?case* **by** (*simp add*: *mkPX-cn*)**}**
ultimately show *?case* **by** *blast*
qed(*simp*)

lemma *mul-cn* :⟦*isnorm P*; (*isnorm Q*)⟧ ⟹ *isnorm* (*mul* (*P*, *Q*))
proof(*induct P Q rule*: *mul.induct*)
case (*2 c i P2*) **thus** *?case*
by (*cases P2*, *simp-all*) (*cases i*,*simp-all add*: *mkPinj-cn*)
next
case (*3 i P2 c*) **thus** *?case*
by (*cases P2*, *simp-all*) (*cases i*,*simp-all add*: *mkPinj-cn*)
next
case (*4 c P2 i Q2*)
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp only*: *norm-PX1*[*of P2 i Q2*] *norm-PX2*[*of P2 i Q2*])
with *prems* **show** *?case*
by − (*case-tac c*=*0*,*simp-all*,*case-tac i*=*0*,*simp-all add*: *mkPX-cn*)
next
case (*5 P2 i Q2 c*)
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp only*: *norm-PX1*[*of P2 i Q2*] *norm-PX2*[*of P2 i Q2*])
with *prems* **show** *?case*
by − (*case-tac c*=*0*,*simp-all*,*case-tac i*=*0*,*simp-all add*: *mkPX-cn*)
next
case (*6 x P2 y Q2*)
have *x* < *y* ∨ *x* = *y* ∨ *x* > *y* **by** *arith*
moreover
{ assume *x*<*y* **hence** *EX d*. *y*=*d*+*x* **by** *arith*
then obtain *d* **where** *y*=*d*+*x*..
moreover
note *prems*
moreover
from *prems* **have** *x*>*0* **by** (*cases x*, *auto simp add*: *norm-Pinj-0-False*)
moreover
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-Pinj*[*of* - *Q2*])

moreover
with *prems* **have** *isnorm* (*Pinj d Q2*) **by** (*cases d*, *simp*, *cases Q2*, *auto*)
ultimately have *?case* **by** (*simp add*: *mkPinj-cn*)**}**
moreover
{ assume $x=y$
moreover
from *prems* **have** *isnorm P2 isnorm Q2* **by**(*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-Pinj*[*of* - *Q2*])
moreover
with *prems* **have** $y>0$ **by** (*cases y*, *auto simp add*: *norm-Pinj-0-False*)
moreover
note *prems*
moreover
ultimately have *?case* **by** (*simp add*: *mkPinj-cn*) **}**
moreover
{ assume $x>y$ **hence** *EX d*. $x=d+y$ **by** *arith*
then obtain *d* **where** $x=d+y$**..**
moreover
note *prems*
moreover
from *prems* **have** $y>0$ **by** (*cases y*, *auto simp add*: *norm-Pinj-0-False*)
moreover
from *prems* **have** *isnorm P2 isnorm Q2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-Pinj*[*of* - *Q2*])
moreover
with *prems* **have** *isnorm* (*Pinj d P2*) **by** (*cases d*, *simp*, *cases P2*, *auto*)
ultimately have *?case* **by** (*simp add*: *mkPinj-cn*) **}**
ultimately show *?case* **by** *blast*
next
case (*7 x P2 Q2 y R*)
from *prems* **have** *Y0*:$y>0$ **by**(*cases y*, *auto*)
have $x=0 \lor (x = 1) \lor (x > 1)$ **by** *arith*
moreover
{ assume $x=0$ **with** *prems* **have** *?case* **by** (*auto simp add*: *norm-Pinj-0-False*)**}**
moreover
{ assume $x=1$
from *prems* **have** *isnorm R isnorm P2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-PX2*[*of Q2 y R*])
with *prems* **have** *isnorm* (*mul* (*R*, *P2*)) *isnorm Q2* **by** (*auto simp add*: *norm-PX1*[*of Q2 y R*])
with *Y0 prems* **have** *?case* **by** (*simp add*: *mkPX-cn*)**}**
moreover
{ assume $x > 1$ **hence** *EX d*. $x=Suc\ (Suc\ d)$ **by** *arith*
then obtain *d* **where** *X*:$x=Suc\ (Suc\ d)$ **..**
from *prems* **have** *NR*:*isnorm R isnorm Q2* **by** (*auto simp add*: *norm-PX2*[*of Q2 y R*] *norm-PX1*[*of Q2 y R*])
moreover
from *prems* **have** *isnorm* (*Pinj* $(x - 1)$ *P2*) **by**(*cases P2*, *auto*)
moreover

from *prems* **have** *isnorm* (*Pinj x P2*) **by**(*cases P2*, *auto*)
moreover
note *prems*
ultimately have *isnorm* (*mul* (*R*, *Pinj* (*x* − *1*) *P2*)) *isnorm* (*mul* (*Pinj x P2*, *Q2*)) **by** *auto*
with *Y0 X* **have** *?case* **by** (*simp add*: *mkPX-cn*)**}**
ultimately show *?case* **by** *blast*
next
case (*8 Q2 y R x P2*)
from *prems* **have** *Y0*:*y>0* **by**(*cases y*, *auto*)
have *x=0* ∨ (*x* = *1*) ∨ (*x* > *1*) **by** *arith*
moreover
{ assume *x=0* **with** *prems* **have** *?case* **by** (*auto simp add*: *norm-Pinj-0-False*)**}**
moreover
{ assume *x=1*
from *prems* **have** *isnorm R isnorm P2* **by** (*auto simp add*: *norm-Pinj*[*of* - *P2*] *norm-PX2*[*of Q2 y R*])
with *prems* **have** *isnorm* (*mul* (*R*, *P2*)) *isnorm Q2* **by** (*auto simp add*: *norm-PX1*[*of Q2 y R*])
with *Y0 prems* **have** *?case* **by** (*simp add*: *mkPX-cn*) **}**
moreover
{ assume *x* > *1* **hence** *EX d*. *x=Suc* (*Suc d*) **by** *arith*
then obtain *d* **where** *X*:*x=Suc* (*Suc d*) **..**
from *prems* **have** *NR*:*isnorm R isnorm Q2* **by** (*auto simp add*: *norm-PX2*[*of Q2 y R*] *norm-PX1*[*of Q2 y R*])
moreover
from *prems* **have** *isnorm* (*Pinj* (*x* − *1*) *P2*) **by**(*cases P2*, *auto*)
moreover
from *prems* **have** *isnorm* (*Pinj x P2*) **by**(*cases P2*, *auto*)
moreover
note *prems*
ultimately have *isnorm* (*mul* (*R*, *Pinj* (*x* − *1*) *P2*)) *isnorm* (*mul* (*Pinj x P2*, *Q2*)) **by** *auto*
with *Y0 X* **have** *?case* **by** (*simp add*: *mkPX-cn*) **}**
ultimately show *?case* **by** *blast*
next
case (*9 P1 x P2 Q1 y Q2*)
from *prems* **have** *X0*:*x>0* **by**(*cases x*, *auto*)
from *prems* **have** *Y0*:*y>0* **by**(*cases y*, *auto*)
note *prems*
moreover
from *prems* **have** *isnorm P1 isnorm P2* **by** (*auto simp add*: *norm-PX1*[*of P1 x P2*] *norm-PX2*[*of P1 x P2*])
moreover
from *prems* **have** *isnorm Q1 isnorm Q2* **by** (*auto simp add*: *norm-PX1*[*of Q1 y Q2*] *norm-PX2*[*of Q1 y Q2*])
ultimately have *isnorm* (*mul* (*P1*, *Q1*)) *isnorm* (*mul* (*P2*, *Q2*)) *isnorm* (*mul* (*P1*, *mkPinj 1 Q2*)) *isnorm* (*mul* (*Q1*, *mkPinj 1 P2*))
by (*auto simp add*: *mkPinj-cn*)

with *prems X0 Y0* **have** *isnorm* (*mkPX* (*mul* (*P1*, *Q1*)) (*x* + *y*) (*mul* (*P2*, *Q2*))) *isnorm* (*mkPX* (*mul* (*P1*, *mkPinj* (*Suc 0*) *Q2*)) *x* (*Pc 0*))
isnorm (*mkPX* (*mul* (*Q1*, *mkPinj* (*Suc 0*) *P2*)) *y* (*Pc 0*))
by (*auto simp add*: *mkPX-cn*)
thus *?case* **by** (*simp add*: *add-cn*)
qed(*simp*)

lemma *neg-cn*: *isnorm P* $\Longrightarrow$ *isnorm* (*neg P*)
proof(*induct P rule*: *neg.induct*)
case (*Pinj i P2*)
from *prems* **have** *isnorm P2* **by** (*simp add*: *norm-Pinj*[*of i P2*])
with *prems* **show** *?case* **by**(*cases P2*, *auto*, *cases i*, *auto*)
next
case (*PX P1 x P2*)
from *prems* **have** *isnorm P2 isnorm P1* **by** (*auto simp add*: *norm-PX1*[*of P1 x P2*] *norm-PX2*[*of P1 x P2*])
with *prems* **show** *?case*
proof(*cases P1*)
case (*PX p1 y p2*)
with *prems* **show** *?thesis* **by**(*cases x*, *auto*, *cases p2*, *auto*)
next
case *Pinj*
with *prems* **show** *?thesis* **by**(*cases x*, *auto*)
qed(*cases x*, *auto*)
qed(*simp*)

lemma *sub-cn*:⟦*isnorm p*; *isnorm q*⟧ $\Longrightarrow$ *isnorm* (*sub p q*)
by (*simp add*: *sub-def add-cn neg-cn*)

lemma *sqr-cn*:*isnorm P* $\Longrightarrow$ *isnorm* (*sqr P*)
proof(*induct P*)
case (*Pinj i Q*)
from *prems* **show** *?case* **by**(*cases Q*, *auto simp add*: *mkPX-cn mkPinj-cn*, *cases i*, *auto simp add*: *mkPX-cn mkPinj-cn*)
next
case (*PX P1 x P2*)
from *prems* **have** *x+x~=0 isnorm P2 isnorm P1* **by** (*cases x*, *auto simp add*: *norm-PX1*[*of P1 x P2*] *norm-PX2*[*of P1 x P2*])
with *prems* **have** *isnorm* (*mkPX* (*mul* (*mul* (*Pc* ((*1*::′*a*) + (*1*::′*a*)), *P1*), *mkPinj* (*Suc 0*) *P2*)) *x* (*Pc* (*0*::′*a*)))
and *isnorm* (*mkPX* (*sqr P1*) (*x* + *x*) (*sqr P2*)) **by**(*auto simp add*: *add-cn mkPX-cn mkPinj-cn mul-cn*)
thus *?case* **by**(*auto simp add*: *add-cn mkPX-cn mkPinj-cn mul-cn*)
qed(*simp*)

lemma *pow-cn*:!! P. $[\![isnorm\ P]\!] \Longrightarrow isnorm\ (pow\ (P,\ n))$
proof(*induct n rule*: *nat-less-induct*)
  case (*1 k*)
  show *?case*
  proof(*cases k=0*)
    case *False*
    hence *K2*:*k div 2* < *k* **by** (*cases k*, *auto*)
    from *prems* **have** *isnorm* (*sqr P*) **by** (*simp add*: *sqr-cn*)
    with *prems K2* **show** *?thesis* **by**(*simp add*: *allE*[*of* - (*k div 2*) -] *allE*[*of* - (*sqr P*) -], *cases k*, *auto simp add*: *mul-cn*)
  qed(*simp*)
qed

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 $(X = Y \cup Z) =$
    $(Y \subseteq X \wedge Z \subseteq X \wedge (\forall V.\ Y \subseteq V \wedge Z \subseteq V \longrightarrow X \subseteq V))$
  by *blast*

lemma $(X = Y \cap Z) =$
    $(X \subseteq Y \wedge X \subseteq Z \wedge (\forall V.\ V \subseteq Y \wedge V \subseteq Z \longrightarrow V \subseteq X))$
  by *blast*

Trivial example of term synthesis: apparently hard for some provers!

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

37.1 Examples for the *blast* paper

lemma $(\bigcup x \in C.\ f\ x \cup g\ x) = \bigcup (f\ `\ C) \ \cup\ \bigcup (g\ `\ C)$
  — Union-image, called *Un-Union-image* in Main HOL
  by *blast*

lemma $(\bigcap x \in C.\ f\ x \cap g\ x) = \bigcap (f\ `\ C) \cap \bigcap (g\ `\ C)$
  — Inter-image, called *Int-Inter-image* in Main HOL
  by *blast*

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*:
$\bigwedge S::'a\ set\ set.\ \forall x \in S.\ \forall y \in S.\ x \subseteq y \Longrightarrow \exists z.\ S \subseteq \{z\}$
by (*meson subsetI subset-antisym insertCI*)

lemma *singleton-example-2*:
$\forall x \in S.\ \bigcup S \subseteq x \Longrightarrow \exists z.\ S \subseteq \{z\}$
— Variant of the problem above.
by (*meson subsetI subset-antisym insertCI UnionI*)

lemma $\exists !x.\ f\ (g\ x) = x \Longrightarrow \exists !y.\ g\ (f\ y) = y$
— A unique fixpoint theorem — *fast*/*best*/*meson* all fail.
apply (*erule ex1E*, *rule ex1I*, *erule arg-cong*)
apply (*rule subst*, *assumption*, *erule allE*, *rule arg-cong*, *erule mp*)
apply (*erule arg-cong*)
done

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.
by *best*

lemma $\forall f::\ 'a \Rightarrow 'a\ set.\ \forall x.\ f\ x \neq\ ?S\ f$
— This form displays the diagonal term.
by *best*

lemma $?S \notin range\ (f\ ::\ 'a \Rightarrow 'a\ set)$
— This form exploits the set constructs.
by (*rule notI*, *erule rangeE*, *best*)

lemma $?S \notin range\ (f\ ::\ 'a \Rightarrow 'a\ set)$
— Or just this!
by *best*

37.3 The Schröder-Berstein Theorem

lemma *disj-lemma*: $-\ (f\ `\ X) = g\ `\ (-X) \Longrightarrow f\ a = g\ b \Longrightarrow a \in X \Longrightarrow b \in X$
by *blast*

lemma *surj-if-then-else*:
$-(f\ `\ X) = g\ `\ (-X) \Longrightarrow surj\ (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g\ z)$
by (*simp add*: *surj-def*) *blast*

lemma *bij-if-then-else*:
$inj\text{-}on\ f\ X \Longrightarrow inj\text{-}on\ g\ (-X) \Longrightarrow -(f\ `\ X) = g\ `\ (-X) \Longrightarrow$
$h = (\lambda z.\ if\ z \in X\ then\ f\ z\ else\ g\ z) \Longrightarrow inj\ h \land surj\ h$

apply (*unfold inj-on-def*)
apply (*simp add*: *surj-if-then-else*)
apply (*blast dest*: *disj-lemma sym*)
done

lemma *decomposition*: $\exists X.\ X = -\ (g\ `\ (-\ (f\ `\ X)))$
apply (*rule exI*)
apply (*rule lfp-unfold*)
apply (*rule monoI*, *blast*)
done

theorem *Schroeder-Bernstein*:
$inj\ (f :: 'a \Rightarrow 'b) \Longrightarrow inj\ (g :: 'b \Rightarrow 'a)$
$\Longrightarrow \exists h:: 'a \Rightarrow 'b.\ inj\ h \wedge surj\ h$
apply (*rule decomposition* [**where** $f=f$ **and** $g=g$, *THEN exE*])
apply (*rule-tac* $x = (\lambda z.\ if\ z \in x\ then\ f\ z\ else\ inv\ g\ z)$ **in** *exI*)
— The term above can be synthesized by a sufficiently detailed proof.
apply (*rule bij-if-then-else*)
apply (*rule-tac* [*4*] *refl*)
apply (*rule-tac* [*2*] *inj-on-inv*)
apply (*erule subset-inj-on* [*OF* - *subset-UNIV*])
apply *blast*
apply (*erule ssubst*, *subst double-complement*, *erule inv-image-comp* [*symmetric*])
done

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.
by *force*

lemma $D \in F \Longrightarrow \exists G.\ \forall A \in G.\ \exists B \in F.\ A \subseteq B$
— Example 2.
by *force*

lemma $P\ a \Longrightarrow \exists A.\ (\forall x \in A.\ P\ x) \wedge (\exists y.\ y \in A)$
— Example 3.
by *force*

lemma $a < b \wedge b < (c::int) \Longrightarrow \exists A.\ a \notin A \wedge b \in A \wedge c \notin A$
— Example 4.
by *force*

lemma $P\ (f\ b) \Longrightarrow \exists s\ A.\ (\forall x \in A.\ P\ x) \wedge f\ s \in A$
— Example 5, page 298.
by *force*

lemma $P\ (f\ b) \Longrightarrow \exists\, s\ A.\ (\forall\, x \in A.\ P\ x) \wedge f\ s \in A$
— Example 6.
by *force*

lemma $\exists\, A.\ a \notin A$
— Example 7.
by *force*

lemma $(\forall\, u\ v.\ u < (0::int) \longrightarrow u \neq abs\ v)$
$\longrightarrow (\exists\, A::int\ set.\ (\forall\, y.\ abs\ y \notin A) \wedge -2 \in A)$
— Example 8 now needs a small hint.
by (*simp add*: *abs-if*, *force*)
— not *blast*, which can't simplify $-2 < 0$

Example 9 omitted (requires the reals).

The paper has no Example 10!

lemma $(\forall\, A.\ 0 \in A \wedge (\forall\, x \in A.\ Suc\ x \in A) \longrightarrow n \in A) \wedge$
$P\ 0 \wedge (\forall\, x.\ P\ x \longrightarrow P\ (Suc\ x)) \longrightarrow P\ n$
— Example 11: needs a hint.
apply *clarify*
apply (*drule-tac* $x = \{x.\ P\ x\}$ **in** *spec*)
apply *force*
done

lemma
$(\forall\, A.\ (0,\ 0) \in A \wedge (\forall\, x\ y.\ (x,\ y) \in A \longrightarrow (Suc\ x,\ Suc\ y) \in A) \longrightarrow (n,\ m) \in A)$
$\wedge\ P\ n \longrightarrow P\ m$
— Example 12.
by *auto*

lemma
$(\forall\, x.\ (\exists\, u.\ x = 2 * u) = (\neg\ (\exists\, v.\ Suc\ x = 2 * v))) \longrightarrow$
$(\exists\, A.\ \forall\, x.\ (x \in A) = (Suc\ x \notin A))$
— Example EO1: typo in article, and with the obvious fix it seems to require arithmetic reasoning.
apply *clarify*
apply (*rule-tac* $x = \{x.\ \exists\, u.\ x = 2 * u\}$ **in** *exI*, *auto*)
apply (*case-tac* v, *auto*)
apply (*drule-tac* $x = Suc\ v$ **and** $P = \lambda x.\ ?a\ x \neq ?b\ x$ **in** *spec*, *force*)
done

end

theory *MT*
imports *Main*
begin

```
typedecl Const

typedecl ExVar
typedecl Ex

typedecl TyConst
typedecl Ty

typedecl Clos
typedecl Val

typedecl ValEnv
typedecl TyEnv

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

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

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

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

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

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

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

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

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

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

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

axioms

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

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

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

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

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

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

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

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

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

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

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

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

defs

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

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



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

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



  isof-env-def:
```

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

axioms
isof-app: [| *c1 isof t1*−>*t2*; *c2 isof t1* |] ==> *c-app c1 c2 isof t2*

defs

hasty-fun-def:
hasty-fun(*r*) ==
{ *p*.
(*? c t*. *p* = (*v-const*(*c*),*t*) & *c isof t*) |
(*? ev e ve t te*.
p = (*v-clos*(<|*ev*,*e*,*ve*|>),*t*) &
te |− *fn ev* => *e* ===> *t* &
ve-dom(*ve*) = *te-dom*(*te*) &
(! *ev1*. *ev1*:*ve-dom*(*ve*) −−> (*ve-app ve ev1*,*te-app te ev1*) : *r*)
)
}

hasty-rel-def: *hasty-rel* == *gfp*(*hasty-fun*)
hasty-def: *v hasty t* == (*v*,*t*) : *hasty-rel*
hasty-env-def:
ve hastyenv te ==
ve-dom(*ve*) = *te-dom*(*te*) &
(! *x*. *x*: *ve-dom*(*ve*) −−> *ve-app ve x hasty te-app te x*)

ML ⟪ *use-legacy-bindings* (*the-context* ()) ⟫

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. ∀y∈S. (y,x): order po) po

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

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

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

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

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

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

lemma (**in** *PO*) *reflE*: [| *refl A r*; *x* ∈ *A*|] ==> (*x*, *x*) ∈ *r*
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def refl-def*)
done

lemma (**in** *PO*) *antisymE*: [| *antisym r*; (*a*, *b*) ∈ *r*; (*b*, *a*) ∈ *r* |] ==> *a* = *b*
apply (*insert cl-po*)
apply (*simp add*: *PartialOrder-def antisym-def*)
done

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

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

lemma (**in** *PO*) *po-subset-po*:
S <= *A* ==> (| *pset* = *S*, *order* = *induced S r* |) ∈ *PartialOrder*
apply (*simp* (*no-asm*) *add*: *PartialOrder-def*)

apply *auto*
— refl
apply (*simp add*: *refl-def induced-def*)
apply (*blast intro*: *PO-imp-refl* [*THEN reflE*])
— antisym
apply (*simp add*: *antisym-def induced-def*)
apply (*blast intro*: *PO-imp-sym* [*THEN antisymE*])
— trans
apply (*simp add*: *trans-def induced-def*)
apply (*blast intro*: *PO-imp-trans* [*THEN transE*])
done

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

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

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

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

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

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

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

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

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

lemma *Rdual*:
  $\forall$ *S*. ($S <= A$ −−>($\exists$ *L*. *isLub S* (| *pset* = *A*, *order* = *r*|) *L*))
  ==> $\forall$ *S*. ($S <= A$ −−> ($\exists$ *G*. *isGlb S* (| *pset* = *A*, *order* = *r*|) *G*))
apply *safe*
apply (*rule-tac x* = *lub* {*y*. $y \in A$ & ($\forall k \in S$. $(y, k) \in r$)}
    (|*pset* = *A*, *order* = *r*|) **in** *exI*)

apply (*drule-tac* $x = \{y.\ y \in A\ \&\ (\forall\, k \in S.\ (y,k) \in r)\ \}$ **in** *spec*)
apply (*drule mp*, *fast*)
apply (*simp add*: *isLub-lub isGlb-def*)
apply (*simp add*: *isLub-def*, *blast*)
done

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

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

lemma *CL-subset-PO*: *CompleteLattice* <= *PartialOrder*
by (*simp add*: *PartialOrder-def CompleteLattice-def*, *fast*)

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*
by (*rule PO-imp-refl*)

lemma (**in** *CL*) *CO-antisym*: *antisym r*
by (*rule PO-imp-sym*)

lemma (**in** *CL*) *CO-trans*: *trans r*
by (*rule PO-imp-trans*)

lemma *CompleteLatticeI*:
  [| *po* ∈ *PartialOrder*; (∀ *S*. *S* <= *pset po* −−> (∃ *L*. *isLub S po L*));
    (∀ *S*. *S* <= *pset po* −−> (∃ *G*. *isGlb S po G*))|]
  ==> *po* ∈ *CompleteLattice*
apply (*unfold CompleteLattice-def*, *blast*)
done

lemma (**in** *CL*) *CL-dualCL*: *dual cl* ∈ *CompleteLattice*
apply (*insert cl-co*)
apply (*simp add*: *CompleteLattice-def dual-def*)
apply (*fold dual-def*)
apply (*simp add*: *isLub-dual-isGlb* [*symmetric*] *isGlb-dual-isLub* [*symmetric*]
    dualPO)
done

lemma (**in** *PO*) *dualA-iff*: *pset* (*dual cl*) = *pset cl*
by (*simp add*: *dual-def*)

lemma (**in** *PO*) *dualr-iff*: ((*x*, *y*) ∈ (*order*(*dual cl*))) = ((*y*, *x*) ∈ *order cl*)

by (*simp add*: *dual-def*)

lemma (**in** *PO*) *monotone-dual*:
  monotone f (*pset cl*) (*order cl*)
  ==> *monotone f* (*pset* (*dual cl*)) (*order*(*dual cl*))
by (*simp add*: *monotone-def dualA-iff dualr-iff*)

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

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

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

lemma (**in** *PO*) *left-in-interval*:
  [| $a \in A$; $b \in A$; *interval r a b* $\neq$ {} |] ==> $a \in$ *interval r a b*
apply (*simp* (*no-asm-simp*) *add*: *interval-def*)
apply (*simp add*: *PO-imp-trans interval-not-empty*)
apply (*simp add*: *PO-imp-refl* [*THEN reflE*])
done

lemma (**in** *PO*) *right-in-interval*:
  [| $a \in A$; $b \in A$; *interval r a b* $\neq$ {} |] ==> $b \in$ *interval r a b*
apply (*simp* (*no-asm-simp*) *add*: *interval-def*)
apply (*simp add*: *PO-imp-trans interval-not-empty*)
apply (*simp add*: *PO-imp-refl* [*THEN reflE*])
done

38.2 sublattice

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

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

38.3 lub

lemma (**in** *CL*) *lub-unique*: [| *S* <= *A*; *isLub S cl x*; *isLub S cl L*|] ==> *x* = *L*

apply (*rule antisymE*)
apply (*rule CO-antisym*)
apply (*auto simp add*: *isLub-def r-def*)
done

lemma (**in** *CL*) *lub-upper*: $[|S <= A; x \in S|] ==> (x, lub\ S\ cl) \in r$
apply (*rule CL-imp-ex-isLub* [*THEN exE*], *assumption*)
apply (*unfold lub-def least-def*)
apply (*rule some-equality* [*THEN ssubst*])
  apply (*simp add*: *isLub-def*)
 apply (*simp add*: *lub-unique A-def isLub-def*)
apply (*simp add*: *isLub-def r-def*)
done

lemma (**in** *CL*) *lub-least*:
  $[|\ S <= A;\ L \in A;\ \forall x \in S.\ (x,L) \in r\ |] ==> (lub\ S\ cl,\ L) \in r$
apply (*rule CL-imp-ex-isLub* [*THEN exE*], *assumption*)
apply (*unfold lub-def least-def*)
apply (*rule-tac s=x* **in** *some-equality* [*THEN ssubst*])
  apply (*simp add*: *isLub-def*)
 apply (*simp add*: *lub-unique A-def isLub-def*)
apply (*simp add*: *isLub-def r-def A-def*)
done

lemma (**in** *CL*) *lub-in-lattice*: $S <= A ==> lub\ S\ cl \in A$
apply (*rule CL-imp-ex-isLub* [*THEN exE*], *assumption*)
apply (*unfold lub-def least-def*)
apply (*subst some-equality*)
apply (*simp add*: *isLub-def*)
prefer *2* **apply** (*simp add*: *isLub-def A-def*)
apply (*simp add*: *lub-unique A-def isLub-def*)
done

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

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

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

by (*simp add*: *isLub-def A-def*)

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

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

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*
by (*simp add*: *isLub-def A-def r-def*)

38.4 glb

lemma (**in** *CL*) *glb-in-lattice*: *S* <= *A* ==> *glb S cl* ∈ *A*
apply (*subst glb-dual-lub*)
apply (*simp add*: *A-def*)
apply (*rule dualA-iff* [*THEN subst*])
apply (*rule Tarski.lub-in-lattice*)
apply (*rule dualPO*)
apply (*rule CL-dualCL*)
apply (*simp add*: *dualA-iff*)
done

lemma (**in** *CL*) *glb-lower*: [|*S* <= *A*; *x* ∈ *S*|] ==> (*glb S cl*, *x*) ∈ *r*
apply (*subst glb-dual-lub*)
apply (*simp add*: *r-def*)
apply (*rule dualr-iff* [*THEN subst*])
apply (*rule Tarski.lub-upper* [*rule-format*])
apply (*rule dualPO*)
apply (*rule CL-dualCL*)
apply (*simp add*: *dualA-iff A-def*, *assumption*)
done

Reduce the sublattice property by using substructural properties; abandoned see *Tarski-4.ML*.

lemma (**in** *CLF*) [*simp*]:
f: *pset cl* −> *pset cl* & *monotone f* (*pset cl*) (*order cl*)
apply (*insert f-cl*)
apply (*simp add*: *CLF-def*)
done

declare (**in** *CLF*) *f-cl* [*simp*]

lemma (**in** *CLF*) *f-in-funcset*: *f* ∈ *A* −> *A*
by (*simp add*: *A-def*)

lemma (**in** *CLF*) *monotone-f*: *monotone f A r*
by (*simp add*: *A-def r-def*)

lemma (**in** *CLF*) *CLF-dual*: (*cl*,*f*) ∈ *CLF* ==> (*dual cl*, *f*) ∈ *CLF*
apply (*simp add*: *CLF-def CL-dualCL monotone-dual*)
apply (*simp add*: *dualA-iff*)
done

38.5 fixed points

lemma *fix-subset*: *fix f A* <= *A*
by (*simp add*: *fix-def*, *fast*)

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

lemma *fixf-subset*:
    [| *A* <= *B*; *x* ∈ *fix* (%*y*: *A*. *f y*) *A* |] ==> *x* ∈ *fix f B*
apply (*simp add*: *fix-def*, *auto*)
done

38.6 lemmas for Tarski, lub

lemma (**in** *CLF*) *lubH-le-flubH*:
    H = {*x*. (*x*, *f x*) ∈ *r* & *x* ∈ *A*} ==> (*lub H cl*, *f* (*lub H cl*)) ∈ *r*
apply (*rule lub-least*, *fast*)
apply (*rule f-in-funcset* [*THEN funcset-mem*])
apply (*rule lub-in-lattice*, *fast*)
— ∀ *x*:*H*. (*x*, *f* (*lub H r*)) ∈ *r*
apply (*rule ballI*)
apply (*rule transE*)
apply (*rule CO-trans*)
— instantiates (*x*, *???z*) ∈ *order cl to* (*x*, *f x*),
— because of the def of *H*
apply *fast*
— so it remains to show (*f x*, *f* (*lub H cl*)) ∈ *r*
apply (*rule-tac f* = *f* **in** *monotoneE*)
apply (*rule monotone-f*, *fast*)
apply (*rule lub-in-lattice*, *fast*)
apply (*rule lub-upper*, *fast*)
apply *assumption*
done

lemma (**in** *CLF*) *flubH-le-lubH*:
    [| *H* = {*x*. (*x*, *f x*) ∈ *r* & *x* ∈ *A*} |] ==> (*f* (*lub H cl*), *lub H cl*) ∈ *r*
apply (*rule lub-upper*, *fast*)
apply (*rule-tac t* = *H* **in** *ssubst*, *assumption*)
apply (*rule CollectI*)
apply (*rule conjI*)

apply (*rule-tac* [*2*] *f-in-funcset* [*THEN funcset-mem*])
apply (*rule-tac* [*2*] *lub-in-lattice*)
prefer *2* **apply** *fast*
apply (*rule-tac* $f = f$ **in** *monotoneE*)
apply (*rule monotone-f*)
  apply (*blast intro*: *lub-in-lattice*)
 apply (*blast intro*: *lub-in-lattice f-in-funcset* [*THEN funcset-mem*])
apply (*simp add*: *lubH-le-flubH*)
done

lemma (**in** *CLF*) *lubH-is-fixp*:
    $H = \{x.\ (x, f\,x) \in r\ \&\ x \in A\}$ $==>$ *lub H cl* $\in$ *fix f A*
apply (*simp add*: *fix-def*)
apply (*rule conjI*)
apply (*rule lub-in-lattice*, *fast*)
apply (*rule antisymE*)
apply (*rule CO-antisym*)
apply (*simp add*: *flubH-le-lubH*)
apply (*simp add*: *lubH-le-flubH*)
done

lemma (**in** *CLF*) *fix-in-H*:
    [| $H = \{x.\ (x, f\,x) \in r\ \&\ x \in A\}$; $x \in P$ |] $==>$ $x \in H$
by (*simp add*: *P-def fix-imp-eq* [*of - f A*] *reflE CO-refl*
                    fix-subset [*of f A*, *THEN subsetD*])

lemma (**in** *CLF*) *fixf-le-lubH*:
    $H = \{x.\ (x, f\,x) \in r\ \&\ x \in A\}$ $==>$ $\forall\, x \in$ *fix f A*. (x, *lub H cl*) $\in r$
apply (*rule ballI*)
apply (*rule lub-upper*, *fast*)
apply (*rule fix-in-H*)
apply (*simp-all add*: *P-def*)
done

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

38.7 Tarski fixpoint theorem 1, first part

lemma (**in** *CLF*) *T-thm-1-lub*: *lub P cl* = *lub* $\{x.\ (x, f\,x) \in r\ \&\ x \in A\}$ *cl*
apply (*rule sym*)
apply (*simp add*: *P-def*)
apply (*rule lubI*)

apply (*rule fix-subset*)
apply (*rule lub-in-lattice*, *fast*)
apply (*simp add*: *fixf-le-lubH*)
apply (*simp add*: *lubH-least-fixf*)
done

lemma (**in** *CLF*) *glbH-is-fixp*: $H = \{x.\ (f\ x,\ x) \in r\ \&\ x \in A\}$ ==> *glb* H *cl* $\in P$
— Tarski for glb
apply (*simp add*: *glb-dual-lub P-def A-def r-def*)
apply (*rule dualA-iff* [*THEN subst*])
apply (*rule Tarski.lubH-is-fixp*)
apply (*rule dualPO*)
apply (*rule CL-dualCL*)
apply (*rule f-cl* [*THEN CLF-dual*])
apply (*simp add*: *dualr-iff dualA-iff*)
done

lemma (**in** *CLF*) *T-thm-1-glb*: *glb P cl* = *glb* $\{x.\ (f\ x,\ x) \in r\ \&\ x \in A\}$ *cl*
apply (*simp add*: *glb-dual-lub P-def A-def r-def*)
apply (*rule dualA-iff* [*THEN subst*])
apply (*simp add*: *Tarski.T-thm-1-lub* [*of - f*, *OF dualPO CL-dualCL*]
dualPO CL-dualCL CLF-dual dualr-iff)
done

38.8 interval

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

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

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

lemma (**in** *CLF*) *interval-lemma1*:
[| S <= *interval r a b*; $x \in S$ |] ==> $(a,\ x) \in r$
apply (*unfold interval-def*, *fast*)
done

lemma (**in** *CLF*) *interval-lemma2*:
[| S <= *interval r a b*; $x \in S$ |] ==> $(x,\ b) \in r$
apply (*unfold interval-def*, *fast*)

done

lemma (**in** *CLF*) *a-less-lub*:
  [| $S <= A$; $S \neq \{\}$;
    $\forall x \in S.\ (a,x) \in r$; $\forall y \in S.\ (y,\ L) \in r$ |] ==> $(a,L) \in r$
by (*blast intro*: *transE PO-imp-trans*)

lemma (**in** *CLF*) *glb-less-b*:
  [| $S <= A$; $S \neq \{\}$;
    $\forall x \in S.\ (x,b) \in r$; $\forall y \in S.\ (G,\ y) \in r$ |] ==> $(G,b) \in r$
by (*blast intro*: *transE PO-imp-trans*)

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

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*
apply (*rule intervalI*)
apply (*rule a-less-lub*)
prefer *2* **apply** *assumption*
apply (*simp add*: *S-intv-cl*)
apply (*rule ballI*)
apply (*simp add*: *interval-lemma1*)
apply (*simp add*: *isLub-upper*)
— $(L,\ b) \in r$
apply (*simp add*: *isLub-least interval-lemma2*)
done

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

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

lemma (**in** *CLF*) *intv-CL-lub*:
 [| $a \in A$; $b \in A$; *interval r a b* $\neq \{\}$ |]
  ==> $\forall S.\ S <=$ *interval r a b* -->

$(\exists L.\ isLub\ S\ (|\ pset = interval\ r\ a\ b,$
$order = induced\ (interval\ r\ a\ b)\ r\ |)\ \ L)$

apply (*intro strip*)
apply (*frule S-intv-cl* [*THEN CL-imp-ex-isLub*])
prefer *2* **apply** *assumption*
apply *assumption*
apply (*erule exE*)
— define the lub for the interval as
apply (*rule-tac x = if S = {} then a else L* **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *isLub-def split del*: *split-if*)
apply (*intro impI conjI*)
— (*if S* = {} *then a else L*) ∈ *interval r a b*
apply (*simp add*: *CL-imp-PO L-in-interval*)
apply (*simp add*: *left-in-interval*)
— lub prop 1
apply (*case-tac S* = {})
— *S* = {}, *y* ∈ *S* = *False* => *everything*
apply *fast*
— *S* ≠ {}
apply *simp*
— ∀ *y*:*S*. (*y*, *L*) ∈ *induced* (*interval r a b*) *r*
apply (*rule ballI*)
apply (*simp add*: *induced-def L-in-interval*)
apply (*rule conjI*)
apply (*rule subsetD*)
apply (*simp add*: *S-intv-cl*, *assumption*)
apply (*simp add*: *isLub-upper*)
— ∀ *z*:*interval r a b*. (∀ *y*:*S*. (*y*, *z*) ∈ *induced* (*interval r a b*) *r* ⟶ (*if S* = {} *then a else L*, *z*) ∈ *induced* (*interval r a b*) *r*
apply (*rule ballI*)
apply (*rule impI*)
apply (*case-tac S* = {})
— *S* = {}
apply *simp*
apply (*simp add*: *induced-def interval-def*)
apply (*rule conjI*)
apply (*rule reflE*)
apply (*rule CO-refl*, *assumption*)
apply (*rule interval-not-empty*)
apply (*rule CO-trans*)
apply (*simp add*: *interval-def*)
— *S* ≠ {}
apply *simp*
apply (*simp add*: *induced-def L-in-interval*)
apply (*rule isLub-least*, *assumption*)
apply (*rule subsetD*)
prefer *2* **apply** *assumption*
apply (*simp add*: *S-intv-cl*, *fast*)
done

lemmas (**in** *CLF*) *intv-CL-glb* = *intv-CL-lub* [*THEN Rdual*]

lemma (**in** *CLF*) *interval-is-sublattice*:
[| $a \in A$; $b \in A$; *interval r a b* $\neq$ {} |]
==> *interval r a b* <<= *cl*
apply (*rule sublatticeI*)
apply (*simp add*: *interval-subset*)
apply (*rule CompleteLatticeI*)
apply (*simp add*: *intervalPO*)
apply (*simp add*: *intv-CL-lub*)
apply (*simp add*: *intv-CL-glb*)
done

lemmas (**in** *CLF*) *interv-is-compl-latt* =
interval-is-sublattice [*THEN sublattice-imp-CL*]

38.9 Top and Bottom

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

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

lemma (**in** *CLF*) *Bot-in-lattice*: *Bot cl* $\in$ *A*
apply (*simp add*: *Bot-def least-def*)
apply (*rule someI2*)
apply (*fold A-def*)
apply (*erule-tac* [*2*] *conjunct1*)
apply (*rule conjI*)
apply (*rule glb-in-lattice*)
apply (*rule subset-refl*)
apply (*fold r-def*)
apply (*simp add*: *glb-lower*)
done

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

lemma (**in** *CLF*) *Top-prop*: $x \in A$ ==> (*x*, *Top cl*) $\in$ *r*
apply (*simp add*: *Top-def greatest-def*)
apply (*rule someI2*)
apply (*fold r-def A-def*)
prefer *2* **apply** *fast*
apply (*intro conjI ballI*)

apply (*rule-tac* [*2*] *lub-upper*)
apply (*auto simp add*: *lub-in-lattice*)
done

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

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

lemma (**in** *CLF*) *Bot-intv-not-empty*: $x \in A$ ==> *interval r* (*Bot cl*) *x* $\neq$ {}
apply (*simp add*: *Bot-dual-Top*)
apply (*subst interval-dual*)
prefer *2* **apply** *assumption*
apply (*simp add*: *A-def*)
apply (*rule dualA-iff* [*THEN subst*])
apply (*blast intro*!: *Tarski.Top-in-lattice*
f-cl dualPO CL-dualCL CLF-dual)
apply (*simp add*: *Tarski.Top-intv-not-empty* [*of - f*]
dualA-iff A-def dualPO CL-dualCL CLF-dual)
done

38.10 fixed points form a partial order

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

lemma (**in** *Tarski*) *Y-subset-A*: *Y* <= *A*
apply (*rule subset-trans* [*OF - fix-subset*])
apply (*rule Y-ss* [*simplified P-def*])
done

lemma (**in** *Tarski*) *lubY-in-A*: *lub Y cl* $\in$ *A*
by (*simp add*: *Y-subset-A* [*THEN lub-in-lattice*])

lemma (**in** *Tarski*) *lubY-le-flubY*: (*lub Y cl*, *f* (*lub Y cl*)) $\in r$
apply (*rule lub-least*)
apply (*rule Y-subset-A*)
apply (*rule f-in-funcset* [*THEN funcset-mem*])
apply (*rule lubY-in-A*)
— *Y* <= *P* ==> *f x* = *x*

apply (*rule ballI*)
apply (*rule-tac t* = *x* **in** *fix-imp-eq* [*THEN subst*])
apply (*erule Y-ss* [*simplified P-def*, *THEN subsetD*])
— *reduce* (*f x*, *f* (*lub Y cl*)) $\in$ *r to* (*x*, *lub Y cl*) $\in$ *r* by monotonicity
apply (*rule-tac f* = *f* **in** *monotoneE*)
apply (*rule monotone-f*)
apply (*simp add*: *Y-subset-A* [*THEN subsetD*])
apply (*rule lubY-in-A*)
apply (*simp add*: *lub-upper Y-subset-A*)
done

lemma (**in** *Tarski*) *intY1-subset*: *intY1* <= *A*
apply (*unfold intY1-def*)
apply (*rule interval-subset*)
apply (*rule lubY-in-A*)
apply (*rule Top-in-lattice*)
done

lemmas (**in** *Tarski*) *intY1-elem* = *intY1-subset* [*THEN subsetD*]

lemma (**in** *Tarski*) *intY1-f-closed*: *x* $\in$ *intY1* $\Longrightarrow$ *f x* $\in$ *intY1*
apply (*simp add*: *intY1-def interval-def*)
apply (*rule conjI*)
apply (*rule transE*)
apply (*rule CO-trans*)
apply (*rule lubY-le-flubY*)
— (*f* (*lub Y cl*), *f x*) $\in$ *r*
apply (*rule-tac f*=*f* **in** *monotoneE*)
apply (*rule monotone-f*)
apply (*rule lubY-in-A*)
apply (*simp add*: *intY1-def interval-def intY1-elem*)
apply (*simp add*: *intY1-def interval-def*)
— (*f x*, *Top cl*) $\in$ *r*
apply (*rule Top-prop*)
apply (*rule f-in-funcset* [*THEN funcset-mem*])
apply (*simp add*: *intY1-def interval-def intY1-elem*)
done

lemma (**in** *Tarski*) *intY1-func*: (%*x*: *intY1*. *f x*) $\in$ *intY1* −> *intY1*
apply (*rule restrictI*)
apply (*erule intY1-f-closed*)
done

lemma (**in** *Tarski*) *intY1-mono*:
monotone (%*x*: *intY1*. *f x*) *intY1* (*induced intY1 r*)
apply (*auto simp add*: *monotone-def induced-def intY1-f-closed*)
apply (*blast intro*: *intY1-elem monotone-f* [*THEN monotoneE*])
done

lemma (**in** *Tarski*) *intY1-is-cl*:
  (| *pset* = *intY1*, *order* = *induced intY1 r* |) ∈ *CompleteLattice*
apply (*unfold intY1-def*)
apply (*rule interv-is-compl-latt*)
apply (*rule lubY-in-A*)
apply (*rule Top-in-lattice*)
apply (*rule Top-intv-not-empty*)
apply (*rule lubY-in-A*)
done

lemma (**in** *Tarski*) *v-in-P*: *v* ∈ *P*
apply (*unfold P-def*)
apply (*rule-tac A* = *intY1* **in** *fixf-subset*)
apply (*rule intY1-subset*)
apply (*simp add*: *Tarski.glbH-is-fixp* [*OF* - *intY1-is-cl*, *simplified*]
    v-def CL-imp-PO intY1-is-cl CLF-def intY1-func intY1-mono)
done

lemma (**in** *Tarski*) *z-in-interval*:
  [| *z* ∈ *P*; ∀ *y*∈*Y*. (*y*, *z*) ∈ *induced P r* |] ==> *z* ∈ *intY1*
apply (*unfold intY1-def P-def*)
apply (*rule intervalI*)
prefer *2*
apply (*erule fix-subset* [*THEN subsetD*, *THEN Top-prop*])
apply (*rule lub-least*)
apply (*rule Y-subset-A*)
apply (*fast elim*!: *fix-subset* [*THEN subsetD*])
apply (*simp add*: *induced-def*)
done

lemma (**in** *Tarski*) *f'z-in-int-rel*: [| *z* ∈ *P*; ∀ *y*∈*Y*. (*y*, *z*) ∈ *induced P r* |]
   ==> ((%*x*: *intY1*. *f x*) *z*, *z*) ∈ *induced intY1 r*
apply (*simp add*: *induced-def intY1-f-closed z-in-interval P-def*)
apply (*simp add*: *fix-imp-eq* [*of* - *f A*] *fix-subset* [*of f A*, *THEN subsetD*]
    CO-refl [*THEN reflE*])
done

lemma (**in** *Tarski*) *tarski-full-lemma*:
  ∃ *L*. *isLub Y* (| *pset* = *P*, *order* = *induced P r* |) *L*
apply (*rule-tac x* = *v* **in** *exI*)
apply (*simp add*: *isLub-def*)
— *v* ∈ *P*
apply (*simp add*: *v-in-P*)
apply (*rule conjI*)
— *v* is lub
— *1*. ∀ *y*:*Y*. (*y*, *v*) ∈ *induced P r*
apply (*rule ballI*)
apply (*simp add*: *induced-def subsetD v-in-P*)
apply (*rule conjI*)

```
apply (erule Y-ss [THEN subsetD])
apply (rule-tac b = lub Y cl in transE)
apply (rule CO-trans)
apply (rule lub-upper)
apply (rule Y-subset-A, assumption)
apply (rule-tac b = Top cl in interval-imp-mem)
apply (simp add: v-def)
apply (fold intY1-def)
apply (rule Tarski.glb-in-lattice [OF - intY1-is-cl, simplified])
 apply (simp add: CL-imp-PO intY1-is-cl, force)
— v is LEAST ub
apply clarify
apply (rule indI)
  prefer 3 apply assumption
 prefer 2 apply (simp add: v-in-P)
apply (unfold v-def)
apply (rule indE)
apply (rule-tac [2] intY1-subset)
apply (rule Tarski.glb-lower [OF - intY1-is-cl, simplified])
  apply (simp add: CL-imp-PO intY1-is-cl)
 apply force
apply (simp add: induced-def intY1-f-closed z-in-interval)
apply (simp add: P-def fix-imp-eq [of - f A]
                fix-subset [of f A, THEN subsetD]
                CO-refl [THEN reflE])
done

lemma CompleteLatticeI-simp:
    [| (| pset = A, order = r |) ∈ PartialOrder;
        ∀S. S <= A --> (∃L. isLub S (| pset = A, order = r |)  L) |]
   ==> (| pset = A, order = r |) ∈ CompleteLattice
by (simp add: CompleteLatticeI Rdual)

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

end
```

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*

oracle
  svc-oracle (*term*) = *Svc.oracle*

end

40 Examples for the 'refute' command

theory *Refute-Examples* **imports** *Main*

begin

lemma $P \wedge Q$
  apply (*rule conjI*)
  refute *1* — refutes P
  refute *2* — refutes Q
  refute — equivalent to 'refute 1'
    — here 'refute 3' would cause an exception, since we only have 2 subgoals
  refute [*maxsize=5*] — we can override parameters ...
  refute [*satsolver=dpll*] *2* — ... and specify a subgoal at the same time
oops

41 Examples and Test Cases

41.1 Propositional logic

lemma *True*
  refute
  apply *auto*
done

lemma *False*
  refute
oops

lemma P
  refute
oops

lemma $\sim P$
refute
oops

lemma $P \;\&\; Q$
refute
oops

lemma $P \mid Q$
refute
oops

lemma $P \longrightarrow Q$
refute
oops

lemma $(P{::}bool) = Q$
refute
oops

lemma $(P \mid Q) \longrightarrow (P \;\&\; Q)$
refute
oops

41.2 Predicate logic

lemma $P\ x\ y\ z$
refute
oops

lemma $P\ x\ y \longrightarrow P\ y\ x$
refute
oops

lemma $P\ (f\ (f\ x)) \longrightarrow P\ x \longrightarrow P\ (f\ x)$
refute
oops

41.3 Equality

lemma $P = \mathit{True}$
refute
oops

lemma $P = \mathit{False}$
refute
oops

lemma $x = y$

refute
oops

lemma $f\ x = g\ x$
refute
oops

lemma $(f::'a \Rightarrow 'b) = g$
refute
oops

lemma $(f::('d \Rightarrow 'd) \Rightarrow ('c \Rightarrow 'd)) = g$
refute
oops

lemma *distinct* $[a,b]$
refute
apply *simp*
refute
oops

41.4 First-Order Logic

lemma $\exists\, x.\ P\ x$
refute
oops

lemma $\forall\, x.\ P\ x$
refute
oops

lemma $EX!\ x.\ P\ x$
refute
oops

lemma *Ex P*
refute
oops

lemma *All P*
refute
oops

lemma *Ex1 P*
refute
oops

lemma $(\exists\, x.\ P\ x) \longrightarrow (\forall\, x.\ P\ x)$
refute

oops

lemma $(\forall x.\ \exists y.\ P\ x\ y) \longrightarrow (\exists y.\ \forall x.\ P\ x\ y)$
  refute
oops

lemma $(\exists x.\ P\ x) \longrightarrow (EX!\ x.\ P\ x)$
  refute
oops

A true statement (also testing names of free and bound variables being identical)

lemma $(\forall x\ y.\ P\ x\ y \longrightarrow P\ y\ x) \longrightarrow (\forall x.\ P\ x\ y) \longrightarrow P\ y\ x$
  refute
  apply *fast*
done

"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
oops

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
oops

"Every reflexive and symmetric relation is transitive."

lemma $[\![\ \forall x.\ P\ x\ x;\ \forall x\ y.\ P\ x\ y \longrightarrow P\ y\ x\ ]\!] \Longrightarrow P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z$
  refute
oops

The "Drinker's theorem" ...

lemma $\exists x.\ f\ x = g\ x \longrightarrow f = g$
  refute [*maxsize=4*]
  apply (*auto simp add*: *ext*)
done

... and an incorrect version of it

lemma $(\exists x.\ f\ x = g\ x) \longrightarrow f = g$
  refute
oops

"Every function has a fixed point."

lemma $\exists x.\ f\ x = x$
  refute

oops

"Function composition is commutative."

lemma $f\ (g\ x) = g\ (f\ x)$
  refute
oops

"Two functions that are equivalent wrt. the same predicate 'P' are equal."

lemma $((P{::}('a \Rightarrow 'b) \Rightarrow bool)\ f = P\ g) \longrightarrow (f\ x = g\ x)$
  refute
oops

41.5 Higher-Order Logic

lemma $\exists\, P.\ P$
  refute
  apply *auto*
done

lemma $\forall\, P.\ P$
  refute
oops

lemma $EX!\ P.\ P$
  refute
  apply *auto*
done

lemma $EX!\ P.\ P\ x$
  refute
oops

lemma $P\ Q\ |\ Q\ x$
  refute
oops

lemma $P\ All$
  refute
oops

lemma $P\ Ex$
  refute
oops

lemma $P\ Ex1$
  refute
oops

"The transitive closure 'T' of an arbitrary relation 'P' is non-empty."

constdefs
  trans :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  trans P == $(ALL\ x\ y\ z.\ P\ x\ y \longrightarrow P\ y\ z \longrightarrow P\ x\ z)$
  subset :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  subset P Q == $(ALL\ x\ y.\ P\ x\ y \longrightarrow Q\ x\ y)$
  trans-closure :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  trans-closure P Q == $(subset\ Q\ P)\ \&\ (trans\ P)\ \&\ (ALL\ R.\ subset\ Q\ R \longrightarrow trans\ R \longrightarrow subset\ P\ R)$

lemma *trans-closure T P* $\longrightarrow (\exists\, x\ y.\ T\ x\ y)$
  refute
oops

"The union of transitive closures is equal to the transitive closure of unions."

lemma $(\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow T\ x\ y) \longrightarrow trans\ T \longrightarrow (\forall\, Q.\ (\forall\, x\ y.\ (P\ x\ y \mid R\ x\ y) \longrightarrow Q\ x\ y) \longrightarrow trans\ Q \longrightarrow subset\ T\ Q)$
    $\longrightarrow$ *trans-closure TP P*
    $\longrightarrow$ *trans-closure TR R*
    $\longrightarrow (T\ x\ y = (TP\ x\ y \mid TR\ x\ y))$
  refute
oops

"Every surjective function is invertible."

lemma $(\forall\, y.\ \exists\, x.\ y = f\ x) \longrightarrow (\exists\, g.\ \forall\, x.\ g\ (f\ x) = x)$
  refute
oops

"Every invertible function is surjective."

lemma $(\exists\, g.\ \forall\, x.\ g\ (f\ x) = x) \longrightarrow (\forall\, y.\ \exists\, x.\ y = f\ x)$
  refute
oops

Every point is a fixed point of some function.

lemma $\exists\, f.\ f\ x = x$
  refute [*maxsize=4*]
  apply (*rule-tac x*=$\lambda x.\ x$ **in** *exI*)
  apply *simp*
done

Axiom of Choice: first an incorrect version ...

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (EX!f.\ \forall\, x.\ P\ x\ (f\ x))$
  refute
oops

... and now two correct ones

lemma $(\forall\, x.\ \exists\, y.\ P\ x\ y) \longrightarrow (\exists\, f.\ \forall\, x.\ P\ x\ (f\ x))$
  refute [*maxsize=4*]
  apply (*simp add*: *choice*)

done

lemma $(\forall x.\ EX!y.\ P\ x\ y) \longrightarrow (EX!f.\ \forall x.\ P\ x\ (f\ x))$
 refute [*maxsize=2*]
 apply *auto*
  apply (*simp add*: *ex1-implies-ex choice*)
 apply (*fast intro*: *ext*)
done

41.6 Meta-logic

lemma $!!x.\ P\ x$
 refute
oops

lemma $f\ x == g\ x$
 refute
oops

lemma $P \Longrightarrow Q$
 refute
oops

lemma $[\![\ P;\ Q;\ R\ ]\!] \Longrightarrow S$
 refute
oops

41.7 Schematic variables

lemma *?P*
 refute
 apply *auto*
done

lemma $x = ?y$
 refute
 apply *auto*
done

41.8 Abstractions

lemma $(\lambda x.\ x) = (\lambda x.\ y)$
 refute
oops

lemma $(\lambda f.\ f\ x) = (\lambda f.\ True)$
 refute
oops

lemma $(\lambda x.\ x) = (\lambda y.\ y)$

refute
apply *simp*
done

41.9 Sets

lemma $P\ (A::'a\ set)$
refute
oops

lemma $P\ (A::'a\ set\ set)$
refute
oops

lemma $\{x.\ P\ x\} = \{y.\ P\ y\}$
refute
apply *simp*
done

lemma $x : \{x.\ P\ x\}$
refute
oops

lemma $P\ op:$
refute
oops

lemma $P\ (op:\ x)$
refute
oops

lemma $P\ Collect$
refute
oops

lemma $A\ Un\ B = A\ Int\ B$
refute
oops

lemma $(A\ Int\ B)\ Un\ C = (A\ Un\ C)\ Int\ B$
refute
oops

lemma $Ball\ A\ P \longrightarrow Bex\ A\ P$
refute
oops

41.10 arbitrary

lemma *arbitrary*

refute
oops

lemma *P arbitrary*
refute
oops

lemma *arbitrary x*
refute
oops

lemma *arbitrary arbitrary*
refute
oops

41.11 The

lemma *The P*
refute
oops

lemma *P The*
refute
oops

lemma $P\ (The\ P)$
refute
oops

lemma $(THE\ x.\ x{=}y) = z$
refute
oops

lemma $Ex\ P \longrightarrow P\ (The\ P)$
refute
oops

41.12 Eps

lemma *Eps P*
refute
oops

lemma *P Eps*
refute
oops

lemma $P\ (Eps\ P)$
refute
oops

lemma $(SOME\ x.\ x=y) = z$
  refute
oops

lemma $Ex\ P \longrightarrow P\ (Eps\ P)$
  refute $[maxsize=3]$
  apply $(auto\ simp\ add\colon someI)$
done

41.13 Subtypes (typedef), typedecl

A completely unspecified non-empty subset of $'a$:

typedef $'a\ myTdef = insert\ (arbitrary::'a)\ (arbitrary::'a\ set)$
  by $auto$

lemma $(x::'a\ myTdef) = y$
  refute
oops

typedecl $myTdecl$

typedef $'a\ T\text{-}bij = \{(f::'a \Rightarrow 'a).\ \forall y.\ \exists!x.\ f\ x = y\}$
  by $auto$

lemma $P\ (f::(myTdecl\ myTdef)\ T\text{-}bij)$
  refute
oops

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 $\langle\langle$ *reset quick-and-dirty*; $\rangle\rangle$

41.14.1 unit

lemma $P\ (x::unit)$
  refute
oops

lemma $\forall x::unit.\ P\ x$
  refute
oops

lemma $P\ ()$
  refute

oops

lemma P (*unit-rec u x*)
 refute
oops

lemma P (*case x of* () $\Rightarrow$ *u*)
 refute
oops

41.14.2 option

lemma P (x::$'a$ *option*)
 refute
oops

lemma $\forall x$::$'a$ *option*. P x
 refute
oops

lemma P *None*
 refute
oops

lemma P (*Some x*)
 refute
oops

lemma P (*option-rec n s x*)
 refute
oops

lemma P (*case x of None* $\Rightarrow$ *n* | *Some u* $\Rightarrow$ *s u*)
 refute
oops

41.14.3 *

lemma P (x::$'a*'b$)
 refute
oops

lemma $\forall x$::$'a*'b$. P x
 refute
oops

lemma P (x,y)
 refute
oops

lemma P (*fst* x)
  refute
oops

lemma P (*snd* x)
  refute
oops

lemma P *Pair*
  refute
oops

lemma P (*prod-rec* p x)
  refute
oops

lemma P (*case* x *of Pair* a $b \Rightarrow p$ a b)
  refute
oops

41.14.4 +

lemma P $(x::'a+'b)$
  refute
oops

lemma $\forall x::'a+'b.\ P\ x$
  refute
oops

lemma P (*Inl* x)
  refute
oops

lemma P (*Inr* x)
  refute
oops

lemma P *Inl*
  refute
oops

lemma P (*sum-rec* l r x)
  refute
oops

lemma P (*case* x *of Inl* $a \Rightarrow l\ a$ | *Inr* $b \Rightarrow r\ b$)
  refute
oops

41.14.5 Non-recursive datatypes

datatype $T1 = A \mid B$

lemma $P\ (x{::}T1)$
  refute
oops

lemma $\forall\, x{::}T1.\ P\ x$
  refute
oops

lemma $P\ A$
  refute
oops

lemma $P\ (T1\text{-}rec\ a\ b\ x)$
  refute
oops

lemma $P\ (case\ x\ of\ A \Rightarrow a \mid B \Rightarrow b)$
  refute
oops

datatype $'a\ T2 = C\ T1 \mid D\ 'a$

lemma $P\ (x{::}'a\ T2)$
  refute
oops

lemma $\forall\, x{::}'a\ T2.\ P\ x$
  refute
oops

lemma $P\ D$
  refute
oops

lemma $P\ (T2\text{-}rec\ c\ d\ x)$
  refute
oops

lemma $P\ (case\ x\ of\ C\ u \Rightarrow c\ u \mid D\ v \Rightarrow d\ v)$
  refute
oops

datatype $('a,'b)\ T3 = E\ 'a \Rightarrow 'b$

lemma $P\ (x{::}('a,'b)\ T3)$
  refute

oops

lemma $\forall x::('a,'b)\ T3.\ P\ x$
  refute
oops

lemma $P\ E$
  refute
oops

lemma $P\ (T3\text{-}rec\ e\ x)$
  refute
oops

lemma $P\ (case\ x\ of\ E\ f \Rightarrow e\ f)$
  refute
oops

41.14.6 Recursive datatypes

nat

lemma $P\ (x::nat)$
  refute
oops

lemma $\forall x::nat.\ P\ x$
  refute
oops

lemma $P\ (Suc\ 0)$
  refute
oops

lemma $P\ Suc$
  refute — *Suc* is a partial function (regardless of the size of the model), hence *P Suc* is undefined, hence no model will be found
oops

lemma $P\ (nat\text{-}rec\ zero\ suc\ x)$
  refute
oops

lemma $P\ (case\ x\ of\ 0 \Rightarrow zero\ |\ Suc\ n \Rightarrow suc\ n)$
  refute
oops

'a list

lemma $P\ (xs::'a\ list)$
  refute

oops

lemma $\forall xs::'a\ list.\ P\ xs$
  refute
oops

lemma $P\ [x,\ y]$
  refute
oops

lemma $P\ (list\text{-}rec\ nil\ cons\ xs)$
  refute
oops

lemma $P\ (case\ x\ of\ Nil \Rightarrow nil\ |\ Cons\ a\ b \Rightarrow cons\ a\ b)$
  refute
oops

lemma $(xs::'a\ list) = ys$
  refute
oops

lemma $a\ \#\ xs = b\ \#\ xs$
  refute
oops

datatype $'a\ BinTree = Leaf\ 'a\ |\ Node\ 'a\ BinTree\ 'a\ BinTree$

lemma $P\ (x::'a\ BinTree)$
  refute
oops

lemma $\forall x::'a\ BinTree.\ P\ x$
  refute
oops

lemma $P\ (Node\ (Leaf\ x)\ (Leaf\ y))$
  refute
oops

lemma $P\ (BinTree\text{-}rec\ l\ n\ x)$
  refute
oops

lemma $P\ (case\ x\ of\ Leaf\ a \Rightarrow l\ a\ |\ Node\ a\ b \Rightarrow n\ a\ b)$
  refute
oops

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
oops

lemma ∀ *x*::*'a aexp*. *P x*
 refute
oops

lemma *P* (*ITE* (*Equal* (*Number x*) (*Number y*)) (*Number x*) (*Number y*))
 refute
oops

lemma *P* (*x*::*'a bexp*)
 refute
oops

lemma ∀ *x*::*'a bexp*. *P x*
 refute
oops

lemma *P* (*aexp-bexp-rec-1 number ite equal x*)
 refute
oops

lemma *P* (*case x of Number a* ⇒ *number a* | *ITE b a1 a2* ⇒ *ite b a1 a2*)
 refute
oops

lemma *P* (*aexp-bexp-rec-2 number ite equal x*)
 refute
oops

lemma *P* (*case x of Equal a1 a2* ⇒ *equal a1 a2*)
 refute
oops

41.14.8 Other datatype examples

datatype *Trie* = *TR Trie list*

lemma *P* (*x*::*Trie*)
 refute
oops

lemma ∀ *x*::*Trie*. *P x*

refute
oops

lemma P (*TR* [*TR* []])
refute
oops

lemma P (*Trie-rec-1 a b c x*)
refute
oops

lemma P (*Trie-rec-2 a b c x*)
refute
oops

datatype *InfTree* = *Leaf* | *Node nat* ⇒ *InfTree*

lemma P (*x*::*InfTree*)
refute
oops

lemma ∀ *x*::*InfTree*. P *x*
refute
oops

lemma P (*Node* (λ*n*. *Leaf*))
refute
oops

lemma P (*InfTree-rec leaf node x*)
refute
oops

datatype *'a lambda* = *Var 'a* | *App 'a lambda 'a lambda* | *Lam 'a* ⇒ *'a lambda*

lemma P (*x*::*'a lambda*)
refute
oops

lemma ∀ *x*::*'a lambda*. P *x*
refute
oops

lemma P (*Lam* (λ*a*. *Var a*))
refute
oops

lemma P (*lambda-rec v a l x*)
refute

oops

Taken from "Inductive datatypes in HOL", p.8:

datatype $('a, 'b)\ T = C\ 'a \Rightarrow bool \mid D\ 'b\ list$
datatype $'c\ U = E\ ('c, 'c\ U)\ T$

lemma $P\ (x::'c\ U)$
  refute
oops

lemma $\forall x::'c\ U.\ P\ x$
  refute
oops

lemma $P\ (E\ (C\ (\lambda a.\ True)))$
  refute
oops

lemma $P\ (U\text{-}rec\text{-}1\ e\ f\ g\ h\ i\ x)$
  refute
oops

lemma $P\ (U\text{-}rec\text{-}2\ e\ f\ g\ h\ i\ x)$
  refute
oops

lemma $P\ (U\text{-}rec\text{-}3\ e\ f\ g\ h\ i\ x)$
  refute
oops

41.15 Records

record $('a, 'b)\ point =$
  $xpos :: 'a$
  $ypos :: 'b$

lemma $(x::('a, 'b)\ point) = y$
  refute
oops

record $('a, 'b, 'c)\ extpoint = ('a, 'b)\ point +$
  $ext :: 'c$

lemma $(x::('a, 'b, 'c)\ extpoint) = y$
  refute
oops

41.16 Inductively defined sets

consts

arbitrarySet :: *'a set*
inductive *arbitrarySet*
intros
arbitrary : *arbitrarySet*

lemma *x* : *arbitrarySet*
refute
oops

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
oops

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
oops

41.17 Examples involving special functions

lemma $card\ x = 0$
refute
oops

lemma *finite x*
refute — no finite countermodel exists
oops

lemma $(x::nat) + y = 0$
refute
oops

lemma $(x::nat) = x + x$
refute

oops

lemma $(x::nat) - y + y = x$
  refute
oops

lemma $(x::nat) = x * x$
  refute
oops

lemma $(x::nat) < x + y$
  refute
oops

lemma a @ [] = b @ []
  refute
oops

lemma a @ b = b @ a
  refute
oops

lemma f (*lfp* f) = *lfp* f
  refute
oops

lemma f (*gfp* f) = *gfp* f
  refute
oops

lemma *lfp* f = *gfp* f
  refute
oops

41.18 Axiomatic type classes and overloading

A type class without axioms:

axclass *classA*

lemma P (x::$'a$::*classA*)
  refute
oops

The axiom of this type class does not contain any type variables, but is internally converted into one that does:

axclass *classB*
  classB-ax: P | ~ P

lemma $P\ (x::'a::classB)$
  refute
oops

An axiom with a type variable (denoting types which have at least two elements):

axclass $classC < type$
  $classC\text{-}ax$: $\exists\, x\ y.\ x \neq y$

lemma $P\ (x::'a::classC)$
  refute
oops

lemma $\exists\, x\ y.\ (x::'a::classC) \neq y$
  refute — no countermodel exists
oops

A type class for which a constant is defined:

consts
  $classD\text{-}const :: 'a \Rightarrow 'a$

axclass $classD < type$
  $classD\text{-}ax$: $classD\text{-}const\ (classD\text{-}const\ x) = classD\text{-}const\ x$

lemma $P\ (x::'a::classD)$
  refute
oops

A type class with multiple superclasses:

axclass $classE < classC,\ classD$

lemma $P\ (x::'a::classE)$
  refute
oops

lemma $P\ (x::'a::\{classB,\ classE\})$
  refute
oops

OFCLASS:

lemma $OFCLASS('a::type,\ type\text{-}class)$
  refute — no countermodel exists
  apply *intro-classes*
done

lemma $OFCLASS('a::classC,\ type\text{-}class)$
  refute — no countermodel exists
  apply *intro-classes*

done

lemma $OFCLASS('a,\ classB\text{-}class)$
  refute — no countermodel exists
  apply *intro-classes*
  apply *simp*
done

lemma $OFCLASS('a{::}type,\ classC\text{-}class)$
  refute
oops

Overloading:

consts $inverse :: 'a \Rightarrow 'a$

defs (**overloaded**)
  $inverse\text{-}bool$: $inverse\ (b{::}bool)\quad == \sim\ b$
  $inverse\text{-}set$: $inverse\ (S{::}'a\ set) == -S$
  $inverse\text{-}pair$: $inverse\ p\qquad\qquad == (inverse\ (fst\ p),\ inverse\ (snd\ p))$

lemma $inverse\ b$
  refute
oops

lemma $P\ (inverse\ (S{::}'a\ set))$
  refute
oops

lemma $P\ (inverse\ (p{::}'a{\times}'b))$
  refute
oops

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
  oops

theorem *map g (map f xs) = map (f o g) xs*
  quickcheck
  oops

theorem *rev (xs @ ys) = rev ys @ rev xs*
  quickcheck
  oops

theorem *rev (xs @ ys) = rev xs @ rev ys*
  quickcheck
  oops

theorem *rev (rev xs) = xs*
  quickcheck
  oops

theorem *rev xs = xs*
  quickcheck
  oops

consts
  occurs :: *'a ⇒ 'a list ⇒ nat*
primrec
  occurs a [] = 0
  occurs a (x#xs) = (if (x=a) then Suc(occurs a xs) else occurs a xs)

consts
  del1 :: *'a ⇒ 'a list ⇒ 'a list*
primrec
  del1 a [] = []
  del1 a (x#xs) = (if (x=a) then xs else (x#del1 a xs))

lemma *Suc (occurs a (del1 a xs)) = occurs a xs*
  — Wrong. Precondition needed.
  quickcheck
  oops

lemma *xs ~= [] ⟶ Suc (occurs a (del1 a xs)) = occurs a xs*
  quickcheck
    — Also wrong.
  oops

lemma *0 < occurs a xs ⟶ Suc (occurs a (del1 a xs)) = occurs a xs*
  quickcheck
  apply (*induct-tac xs*)
  apply *auto*
    — Correct!

done

consts
replace :: $'a \Rightarrow 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
primrec
$replace\ a\ b\ [] = []$
$replace\ a\ b\ (x\#xs) = (if\ (x{=}a)\ then\ (b\#(replace\ a\ b\ xs))$
$\qquad else\ (x\#(replace\ a\ b\ xs)))$

lemma $occurs\ a\ xs = occurs\ b\ (replace\ a\ b\ xs)$
quickcheck
— Wrong. Precondition needed.
oops

lemma $occurs\ b\ xs = 0 \lor a{=}b \longrightarrow occurs\ a\ xs = occurs\ b\ (replace\ a\ b\ xs)$
quickcheck
apply (*induct-tac xs*)
apply *auto*
done

42.2 Trees

datatype $'a\ tree = Twig \mid\ Leaf\ 'a \mid Branch\ 'a\ tree\ 'a\ tree$

consts
leaves :: $'a\ tree \Rightarrow 'a\ list$
primrec
$leaves\ Twig = []$
$leaves\ (Leaf\ a) = [a]$
$leaves\ (Branch\ l\ r) = (leaves\ l)\ @\ (leaves\ r)$

consts
plant :: $'a\ list \Rightarrow 'a\ tree$
primrec
$plant\ [] = Twig$
$plant\ (x\#xs) = Branch\ (Leaf\ x)\ (plant\ xs)$

consts
mirror :: $'a\ tree \Rightarrow 'a\ tree$
primrec
$mirror\ (Twig) = Twig$
$mirror\ (Leaf\ a) = Leaf\ a$
$mirror\ (Branch\ l\ r) = Branch\ (mirror\ r)\ (mirror\ l)$

theorem $plant\ (rev\ (leaves\ xt)) = mirror\ xt$
quickcheck
— Wrong!
oops

theorem *plant*((*leaves xt*) @ (*leaves yt*)) = *Branch xt yt*
  quickcheck
    — Wrong!
  oops

datatype *'a ntree* = *Tip 'a* | *Node 'a 'a ntree 'a ntree*

consts
  inOrder :: *'a ntree* ⇒ *'a list*
primrec
  inOrder (*Tip a*)= [*a*]
  inOrder (*Node f x y*) = (*inOrder x*)@[*f*]@(*inOrder y*)

consts
  root :: *'a ntree* ⇒ *'a*
primrec
  root (*Tip a*) = *a*
  root (*Node f x y*) = *f*

theorem *hd*(*inOrder xt*) = *root xt*
  quickcheck
    — Wrong!
  oops

end

43 Implementation of carry chain incrementor and adder

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

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

lemma *bv-to-nat-helper'*: *bv* ≠ [] ==> *bv-to-nat bv* = *bitval* (*hd bv*) ∗ *2* ˆ (*length bv* − *1*) + *bv-to-nat* (*tl bv*)
  by (*cases bv*,*simp-all add*: *bv-to-nat-helper*)

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*
  apply (*simp add*: *half-adder-def*)
  apply (*cases a*, *auto*)
  apply (*cases b*, *auto*)
  done

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

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*
apply (*simp add*: *full-adder-def Let-def*)
apply (*cases a*, *auto*)
apply (*case-tac*[!] *b*, *auto*)
apply (*case-tac*[!] *c*, *auto*)
done

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

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

primrec
carry-chain-inc [] *c* = [*c*]
carry-chain-inc (*a#as*) *c* = (*let chain* = *carry-chain-inc as c*
in half-adder a (*hd chain*) @ *tl chain*)

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

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

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

consts
carry-chain-adder :: [*bit list*,*bit list*,*bit*] => *bit list*

primrec
carry-chain-adder [] *bs c* = [*c*]
carry-chain-adder (*a#as*) *bs c* =

(*let chain* = *carry-chain-adder as* (*tl bs*) *c*
in full-adder a (*hd bs*) (*hd chain*) @ *tl chain*)

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

lemma *cca-length* [*rule-format*]:
∀ *bs*. *length as* = *length bs* −−>
length (*carry-chain-adder as bs c*) = *Suc* (*length bs*)
(**is** *?P as*)
proof (*induct as*,*auto simp add*: *Let-def*)
fix *as* :: *bit list*
fix *xs* :: *bit list*
assume *ind*: *?P as*
assume *len*: *Suc* (*length as*) = *length xs*
thus *Suc* (*length* (*carry-chain-adder as* (*tl xs*) *c*) − *Suc 0*) = *length xs*
proof (*cases xs*,*simp-all*)
fix *b bs*
assume [*simp*]: *xs* = *b* # *bs*
assume *length as* = *length bs*
with *ind*
show *length* (*carry-chain-adder as bs c*) − *Suc 0* = *length bs*
by *auto*
qed
qed

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 (*induct as*,*auto simp add*: *Let-def*)
fix *a* :: *bit*
fix *as* :: *bit list*
fix *xs* :: *bit list*
assume *ind*: *?P as*
assume *len*: *Suc* (*length as*) = *length xs*
thus *bv-to-nat* (*full-adder a* (*hd xs*) (*hd* (*carry-chain-adder as* (*tl xs*) *c*)) @ *tl* (*carry-chain-adder as* (*tl xs*) *c*)) = *bv-to-nat* (*a* # *as*) + *bv-to-nat xs* + *bitval c*
proof (*cases xs*,*simp-all*)
fix *b bs*
assume [*simp*]: *xs* = *b* # *bs*
assume *len*: *length as* = *length bs*
with *ind*
have *bv-to-nat* (*carry-chain-adder as bs c*) = *bv-to-nat as* + *bv-to-nat bs* + *bitval c*
by *blast*
with *len*
show *bv-to-nat* (*full-adder a b* (*hd* (*carry-chain-adder as bs c*)) @ *tl* (*carry-chain-adder*

as bs c)) = *bv-to-nat* (*a* # *as*) + *bv-to-nat* (*b* # *bs*) + *bitval c*
by (*subst bv-to-nat-dist-append*,*simp add*: *full-adder-correct bv-to-nat-helper′* [*OF cca-nonnull*] *ring-distrib bv-to-nat-helper cca-length*)
qed
qed

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.