First-Order Logic According to Harrison

Alexander Birch Jensen, Anders Schlichtkrull &
Jørgen Villadsen, DTU Compute, Denmark

6 February 2026

Abstract

We present a certified declarative first-order prover with equality based on John Harrison's Handbook of Practical Logic and Automated Reasoning, Cambridge University Press, 2009. ML code reflection is used such that the entire prover can be executed within Isabelle as a very simple interactive proof assistant. As examples we consider Pelletier's problems 1-46.

Contents

Preample 1

FOL-Harrison 2

Module Proven 2

ML Code Reflection 11

Main Examples 57

Other Examples 58

Acknowledgements 69

Preample

Preliminary formalizations are described here:

Alexander Birch Jensen: *Development and Verification of a Proof Assistant.* Master's Thesis, Technical University of Denmark, 2016. http://findit.dtu.dk/en/catalog/2345011633

Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen: *Verification of an LCF-Style First-Order Prover with Equality.* Isabelle Workshop 2016. https://github.com/logic-tools/sml-handbook

FOL-Harrison

```
theory FOL_Harrison
imports
  Main
begin
```

Module Proven

Syntax of first-order logic

```
type_synonym id = String.literal

datatype tm = Var id | Fn id "tm list"

datatype 'a fm = Truth | Falsity | Atom 'a | Imp "'a fm" "'a fm" | Iff "'a fm" "'a fm" |
    And "'a fm" "'a fm" | Or "'a fm" "'a fm" | Not "'a fm" | Exists id "'a fm" | Forall id "'a fm"

datatype fol = Rl id "tm list"

datatype "thm" = Thm (concl: "fol fm")
```

Definition of rules and axioms

```
abbreviation (input) "fail_thm ≡ Thm Truth"

definition fol_equal :: "fol fm ⇒ fol fm ⇒ bool"
where
  "fol_equal p q ≡ p = q"

definition zip_eq :: "tm list ⇒ tm list ⇒ fol fm list"
where
  "zip_eq l l' ≡ map (λ(t, t'). Atom (Rl (STR ''='') [t, t'])) (zip l l')"

primrec occurs_in :: "id ⇒ tm ⇒ bool" and occurs_in_list :: "id ⇒ tm list ⇒ bool"
where
  "occurs_in i (Var x) = (i = x)" |
  "occurs_in i (Fn _ l) = occurs_in_list i l" |
  "occurs_in_list _ [] = False" |
  "occurs_in_list i (h # t) = (occurs_in i h ∨ occurs_in_list i t)"

primrec free_in :: "id ⇒ fol fm ⇒ bool"
where
  "free_in _ Truth = False" |
  "free_in _ Falsity = False" |
  "free_in i (Atom a) = (case a of Rl _ l ⇒ occurs_in_list i l)" |
  "free_in i (Imp p q) = (free_in i p ∨ free_in i q)" |
  "free_in i (Iff p q) = (free_in i p ∨ free_in i q)" |
  "free_in i (And p q) = (free_in i p ∨ free_in i q)" |
  "free_in i (Or p q) = (free_in i p ∨ free_in i q)" |
  "free_in i (Not p) = free_in i p" |
  "free_in i (Exists x p) = (i ≠ x ∧ free_in i p)" |
  "free_in i (Forall x p) = (i ≠ x ∧ free_in i p)"

primrec equal_length :: "tm list ⇒ tm list ⇒ bool"
where
  "equal_length l [] = (case l of [] ⇒ True | _ # _ ⇒ False)" |
  "equal_length l (_ # r') = (case l of [] ⇒ False | _ # l' ⇒ equal_length l' r')"

definition modusponens :: "thm ⇒ thm ⇒ thm"
```

```
where
  "modusponens s s' ≡ case concl s of Imp p q ⇒
      let p' = concl s' in if fol_equal p p' then Thm q else fail_thm | _ ⇒ fail_thm"

definition gen :: "id ⇒ thm ⇒ thm"
where
  "gen x s ≡ Thm (Forall x (concl s))"

definition axiom_addimp :: "fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_addimp p q ≡ Thm (Imp p (Imp q p))"

definition axiom_distribimp :: "fol fm ⇒ fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_distribimp p q r ≡ Thm (Imp (Imp p (Imp q r)) (Imp (Imp p q) (Imp p r )))"

definition axiom_doubleneg :: "fol fm ⇒ thm"
where
  "axiom_doubleneg p ≡ Thm (Imp (Imp (Imp p Falsity) Falsity) p)"

definition axiom_allimp :: "id ⇒ fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_allimp x p q ≡ Thm (Imp (Forall x (Imp p q)) (Imp (Forall x p) (Forall x q)))"

definition axiom_impall :: "id ⇒ fol fm ⇒ thm"
where
  "axiom_impall x p ≡ if ¬ free_in x p then Thm (Imp p (Forall x p)) else fail_thm"

definition axiom_existseq :: "id ⇒ tm ⇒ thm"
where
  "axiom_existseq x t ≡ if ¬ occurs_in x t
      then Thm (Exists x (Atom (Rl (STR ''='') [Var x, t]))) else fail_thm"

definition axiom_eqrefl :: "tm ⇒ thm"
where
  "axiom_eqrefl t ≡ Thm (Atom (Rl (STR ''='') [t, t]))"

definition axiom_funcong :: "id ⇒ tm list ⇒ tm list ⇒ thm"
where
  "axiom_funcong i l l' ≡ if equal_length l l'
      then Thm (foldr Imp (zip_eq l l') (Atom (Rl (STR ''='') [Fn i l, Fn i l']))) else fail_thm"

definition axiom_predcong :: "id ⇒ tm list ⇒ tm list ⇒ thm"
where
  "axiom_predcong i l l' ≡ if equal_length l l'
      then Thm (foldr Imp (zip_eq l l') (Imp (Atom (Rl i l)) (Atom (Rl i l')))) else fail_thm"

definition axiom_iffimp1 :: "fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_iffimp1 p q ≡ Thm (Imp (Iff p q) (Imp p q))"

definition axiom_iffimp2 :: "fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_iffimp2 p q ≡ Thm (Imp (Iff p q) (Imp q p))"

definition axiom_impiff :: "fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_impiff p q ≡ Thm (Imp (Imp p q) (Imp (Imp q p) (Iff p q)))"

definition axiom_true :: "thm"
where
  "axiom_true ≡ Thm (Iff Truth (Imp Falsity Falsity))"

definition axiom_not :: "fol fm ⇒ thm"
```

```
where
  "axiom_not p ≡ Thm (Iff (Not p) (Imp p Falsity))"

definition axiom_and :: "fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_and p q ≡ Thm (Iff (And p q) (Imp (Imp p (Imp q Falsity)) Falsity))"

definition axiom_or :: "fol fm ⇒ fol fm ⇒ thm"
where
  "axiom_or p q ≡ Thm (Iff (Or p q) (Not (And (Not p) (Not q))))"

definition axiom_exists :: "id ⇒ fol fm ⇒ thm"
where
  "axiom_exists x p ≡ Thm (Iff (Exists x p) (Not (Forall x (Not p))))"
```

Code generation for rules and axioms

```
export_code
  modusponens gen axiom_addimp axiom_distribimp axiom_doubleneg axiom_allimp axiom_impall
  axiom_existseq axiom_eqrefl axiom_funcong axiom_predcong axiom_iffimp1 axiom_iffimp2
  axiom_impiff axiom_true axiom_not axiom_and axiom_or axiom_exists concl
in SML module_name Proven

code_printing constant fol_equal ⇀ (SML) "_ = _" — More efficient

export_code
  modusponens gen axiom_addimp axiom_distribimp axiom_doubleneg axiom_allimp axiom_impall
  axiom_existseq axiom_eqrefl axiom_funcong axiom_predcong axiom_iffimp1 axiom_iffimp2
  axiom_impiff axiom_true axiom_not axiom_and axiom_or axiom_exists concl
in SML module_name Proven

code_printing constant fol_equal ⇀ (SML) — Delete

export_code
  modusponens gen axiom_addimp axiom_distribimp axiom_doubleneg axiom_allimp axiom_impall
  axiom_existseq axiom_eqrefl axiom_funcong axiom_predcong axiom_iffimp1 axiom_iffimp2
  axiom_impiff axiom_true axiom_not axiom_and axiom_or axiom_exists concl
in SML module_name Proven
```

Semantics of first-order logic

```
definition length2 :: "tm list ⇒ bool"
where
  "length2 l ≡ case l of [_,_] ⇒ True | _ ⇒ False"

primrec — Semantics of terms
  semantics_term :: "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm ⇒ 'a" and
  semantics_list :: "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ tm list ⇒ 'a list"
where
  "semantics_term e _ (Var x) = e x" |
  "semantics_term e f (Fn i l) = f i (semantics_list e f l)" |
  "semantics_list _ _ [] = []" |
  "semantics_list e f (t # l) = semantics_term e f t # semantics_list e f l"

primrec — Semantics of formulas
  semantics :: "(id ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ 'a) ⇒ (id ⇒ 'a list ⇒ bool) ⇒ fol fm ⇒ bool"
where
  "semantics _ _ _ Truth = True" |
  "semantics _ _ _ Falsity = False" |
  "semantics e f g (Atom a) = (case a of Rl i l ⇒ if i = STR ''='' ∧ length2 l
      then (semantics_term e f (hd l) = semantics_term e f (hd (tl l)))
      else g i (semantics_list e f l))" |
  "semantics e f g (Imp p q) = (semantics e f g p ⟶ semantics e f g q)" |
```

```
  "semantics e f g (Iff p q) = (semantics e f g p ⟷ semantics e f g q)" |
  "semantics e f g (And p q) = (semantics e f g p ∧ semantics e f g q)" |
  "semantics e f g (Or p q) = (semantics e f g p ∨ semantics e f g q)" |
  "semantics e f g (Not p) = (¬ semantics e f g p)" |
  "semantics e f g (Exists x p) = (∃v. semantics (e(x := v)) f g p)" |
  "semantics e f g (Forall x p) = (∀v. semantics (e(x := v)) f g p)"
```

Definition of proof system

```
inductive OK :: "fol fm ⇒ bool" (⊂⊢ _⊃ 0)
where
  modusponens:
                    "⊢ concl s ⟹ ⊢ concl s' ⟹ ⊢ concl (modusponens s s')" |
  gen:
                    "⊢ concl s ⟹ ⊢ concl (gen _ s)" |
  axiom_addimp:
                    "⊢ concl (axiom_addimp _ _)" |
  axiom_distribimp:
                    "⊢ concl (axiom_distribimp _ _ _)" |
  axiom_doubleneg:
                    "⊢ concl (axiom_doubleneg _)" |
  axiom_allimp:
                    "⊢ concl (axiom_allimp _ _ _)" |
  axiom_impall:
                    "⊢ concl (axiom_impall _ _)" |
  axiom_existseq:
                    "⊢ concl (axiom_existseq _ _)" |
  axiom_eqrefl:
                    "⊢ concl (axiom_eqrefl _)" |
  axiom_funcong:
                    "⊢ concl (axiom_funcong _ _ _)" |
  axiom_predcong:
                    "⊢ concl (axiom_predcong _ _ _)" |
  axiom_iffimp1:
                    "⊢ concl (axiom_iffimp1 _ _)" |
  axiom_iffimp2:
                    "⊢ concl (axiom_iffimp2 _ _)" |
  axiom_impiff:
                    "⊢ concl (axiom_impiff _ _)" |
  axiom_true:
                    "⊢ concl axiom_true" |
  axiom_not:
                    "⊢ concl (axiom_not _)" |
  axiom_and:
                    "⊢ concl (axiom_and _ _)" |
  axiom_or:
                    "⊢ concl (axiom_or _ _)" |
  axiom_exists:
                    "⊢ concl (axiom_exists _ _)"



proposition "⊢ Imp p p"
proof -
  have 1: "⊢ concl (Thm (Imp (Imp p (Imp (Imp p p) p)) (Imp (Imp p (Imp p p)) (Imp p p))))"
  using axiom_distribimp
  unfolding axiom_distribimp_def
  by simp

  have 2: "⊢ concl (Thm (Imp p (Imp (Imp p p) p)))"
  using axiom_addimp
  unfolding axiom_addimp_def
  by simp
```

```
  have 3: "⊢ concl (Thm (Imp (Imp p (Imp p p)) (Imp p p)))"
  using 1 2 modusponens
  unfolding modusponens_def fol_equal_def
  by fastforce

  have 4: "⊢ concl (Thm (Imp p (Imp p p)))"
  using axiom_addimp
  unfolding axiom_addimp_def
  by simp

  have 5: "⊢ concl (Thm (Imp p p))"
  using 3 4 modusponens
  unfolding modusponens_def fol_equal_def
  by fastforce

  show ?thesis
  using 5
  by simp
qed
```

Soundness of proof system

```
lemma map':
  "¬ occurs_in x t ⟹ semantics_term e f t = semantics_term (e(x := v)) f t"
  "¬ occurs_in_list x l ⟹ semantics_list e f l = semantics_list (e(x := v)) f l"
by (induct t and l rule: semantics_term.induct semantics_list.induct) simp_all

lemma map:
  "¬ free_in x p ⟹ semantics e f g p ⟷ semantics (e(x := v)) f g p"
proof (induct p arbitrary: e)
  fix e
  show "¬ free_in x Truth ⟹ semantics e f g Truth ⟷ semantics (e(x := v)) f g Truth"
  by simp
next
  fix e
  show "¬ free_in x Falsity ⟹ semantics e f g Falsity ⟷ semantics (e(x := v)) f g Falsity"
  by simp
next
  fix a e
  show "¬ free_in x (Atom a) ⟹ semantics e f g (Atom a) ⟷ semantics (e(x := v)) f g (Atom a)"
  proof (cases a)
    fix i l
    show "¬ free_in x (Atom a) ⟹ a = Rl i l ⟹
        semantics e f g (Atom a) ⟷ semantics (e(x := v)) f g (Atom a)"
    proof -
      assume assm: "¬ free_in x (Atom a)" "a = Rl i l"
      then have fresh: "¬ occurs_in_list x l"
      by simp
      show "semantics e f g (Atom a) ⟷ semantics (e(x := v)) f g (Atom a)"
      proof cases
        assume eq: "i = STR ''='' ∧ length2 l"
        then have "semantics e f g (Atom (Rl i l)) ⟷
            semantics_term e f (hd l) = semantics_term e f (hd (tl l))"
        by simp
        also have "... ⟷
            semantics_term (e(x := v)) f (hd l) = semantics_term (e(x := v)) f (hd (tl l))"
        using map'(1) fresh occurs_in_list.simps(2) eq list.case_eq_if list.collapse
        unfolding length2_def
        by metis
        finally show ?thesis
        using eq assm(2)
        by simp
      next
        assume not_eq: "¬ (i = STR ''='' ∧ length2 l)"
```

```
        then have "semantics e f g (Atom (Rl i l)) ⟷ g i (semantics_list e f l)"
        by simp iprover
        also have "... ⟷ g i (semantics_list (e(x := v)) f l)"
        using map'(2) fresh
        by metis
        finally show ?thesis
        using not_eq assm(2)
        by simp iprover
      qed
    qed
  qed
next
  fix p1 p2 e
  assume assm1: "¬ free_in x p1 ⟹ semantics e f g p1 ⟷ semantics (e(x := v)) f g p1" for e
  assume assm2: "¬ free_in x p2 ⟹ semantics e f g p2 ⟷ semantics (e(x := v)) f g p2" for e
  show "¬ free_in x (Imp p1 p2) ⟹
      semantics e f g (Imp p1 p2) ⟷ semantics (e(x := v)) f g (Imp p1 p2)"
  using assm1 assm2
  by simp
next
  fix p1 p2 e
  assume assm1: "¬ free_in x p1 ⟹ semantics e f g p1 ⟷ semantics (e(x := v)) f g p1" for e
  assume assm2: "¬ free_in x p2 ⟹ semantics e f g p2 ⟷ semantics (e(x := v)) f g p2" for e
  show "¬ free_in x (Iff p1 p2) ⟹
      semantics e f g (Iff p1 p2) ⟷ semantics (e(x := v)) f g (Iff p1 p2)"
  using assm1 assm2
  by simp
next
  fix p1 p2 e
  assume assm1: "¬ free_in x p1 ⟹ semantics e f g p1 ⟷ semantics (e(x := v)) f g p1" for e
  assume assm2: "¬ free_in x p2 ⟹ semantics e f g p2 ⟷ semantics (e(x := v)) f g p2" for e
  show "¬ free_in x (And p1 p2) ⟹
      semantics e f g (And p1 p2) ⟷ semantics (e(x := v)) f g (And p1 p2)"
  using assm1 assm2
  by simp
next
  fix p1 p2 e
  assume assm1: "¬ free_in x p1 ⟹ semantics e f g p1 ⟷ semantics (e(x := v)) f g p1" for e
  assume assm2: "¬ free_in x p2 ⟹ semantics e f g p2 ⟷ semantics (e(x := v)) f g p2" for e
  show "¬ free_in x (Or p1 p2) ⟹
      semantics e f g (Or p1 p2) ⟷ semantics (e(x := v)) f g (Or p1 p2)"
  using assm1 assm2
  by simp
next
  fix p e
  assume "¬ free_in x p ⟹ semantics e f g p ⟷ semantics (e(x := v)) f g p" for e
  then show "¬ free_in x (Not p) ⟹ semantics e f g (Not p) ⟷ semantics (e(x := v)) f g (Not p)"
  by simp
next
  fix x1 p e
  assume "¬ free_in x p ⟹ semantics e f g p ⟷ semantics (e(x := v)) f g p" for e
  then show "¬ free_in x (Exists x1 p) ⟹
      semantics e f g (Exists x1 p) ⟷ semantics (e(x := v)) f g (Exists x1 p)"
  by simp (metis fun_upd_twist fun_upd_upd)
next
  fix x1 p e
  assume "¬ free_in x p ⟹ semantics e f g p ⟷ semantics (e(x := v)) f g p" for e
  then show "¬ free_in x (Forall x1 p) ⟹
      semantics e f g (Forall x1 p) ⟷ semantics (e(x := v)) f g (Forall x1 p)"
  by simp (metis fun_upd_twist fun_upd_upd)
qed

lemma length2_equiv:
  "length2 l ⟷ [hd l, hd (tl l)] = l"
```

```
proof -
  have "length2 l ⟹ [hd l, hd (tl l)] = l"
  unfolding length2_def
  using list.case_eq_if list.exhaust_sel
  by metis
  then show ?thesis
  unfolding length2_def
  using list.case list.case_eq_if
  by metis
qed

lemma equal_length_sym:
  "equal_length l l' ⟹ equal_length l' l"
proof (induct l' arbitrary: l)
  fix l
  assume "equal_length l []"
  then show "equal_length [] l"
  using equal_length.simps list.case_eq_if
  by metis
next
  fix l l' a
  assume sym: "equal_length l l' ⟹ equal_length l' l" for l
  assume "equal_length l (a # l')"
  then show "equal_length (a # l') l"
  using equal_length.simps list.case_eq_if list.collapse list.inject sym
  by metis
qed

lemma equal_length2:
  "equal_length l l' ⟹ length2 l ⟷ length2 l'"
proof -
  assume assm: "equal_length l l'"
  have "equal_length l [t, t'] ⟹ length2 l" for t t'
  unfolding length2_def
  using equal_length.simps list.case_eq_if
  by metis
  moreover have "equal_length [t, t'] l' ⟹ length2 l'" for t t'
  unfolding length2_def
  using equal_length.simps list.case_eq_if equal_length_sym
  by metis
  ultimately show ?thesis
  using assm length2_equiv
  by metis
qed

lemma imp_chain_equiv:
  "semantics e f g (foldr Imp l p) ⟷ (∀q ∈ set l. semantics e f g q) ⟶ semantics e f g p"
using imp_conjL
by (induct l) simp_all

lemma imp_chain_zip_eq:
  "equal_length l l' ⟹
      semantics e f g (foldr Imp (zip_eq l l') p) ⟷
      semantics_list e f l = semantics_list e f l' ⟶ semantics e f g p"
proof -
  assume "equal_length l l'"
  then have "(∀q ∈ set (zip_eq l l'). semantics e f g q) ⟷
      semantics_list e f l = semantics_list e f l'"
  unfolding zip_eq_def
  using length2_def
  by (induct l l' rule: list_induct2') simp_all
  then show ?thesis
  using imp_chain_equiv
  by iprover
```

```
qed

lemma funcong:
  "equal_length l l' ⟹
      semantics e f g (foldr Imp (zip_eq l l') (Atom (Rl (STR ''='') [Fn i l, Fn i l'])))"
proof -
  assume assm: "equal_length l l'"
  show ?thesis
  proof cases
    assume "semantics_list e f l = semantics_list e f l'"
    then have "semantics e f g (Atom (Rl (STR ''='') [Fn i l, Fn i l']))"
    using length2_def
    by simp
    then show ?thesis
    using imp_chain_equiv
    by iprover
  next
    assume "semantics_list e f l ≠ semantics_list e f l'"
    then show ?thesis
    using assm imp_chain_zip_eq
    by iprover
  qed
qed

lemma predcong:
  "equal_length l l' ⟹
      semantics e f g (foldr Imp (zip_eq l l') (Imp (Atom (Rl i l)) (Atom (Rl i l'))))"
proof -
  assume assm: "equal_length l l'"
  show ?thesis
  proof cases
    assume eq: "i = STR ''='' ∧ length2 l ∧ length2 l'"
    show ?thesis
    proof cases
      assume "semantics_list e f l = semantics_list e f l'"
      then have "semantics_list e f [hd l, hd (tl l)] = semantics_list e f [hd l', hd (tl l')]"
      using eq length2_equiv
      by simp
      then have "semantics e f g (Imp (Atom (Rl (STR ''='') l)) (Atom (Rl (STR ''='') l')))"
      using eq
      by simp
      then show ?thesis
      using eq imp_chain_equiv
      by iprover
    next
      assume "semantics_list e f l ≠ semantics_list e f l'"
      then show ?thesis
      using assm imp_chain_zip_eq
      by iprover
    qed
  next
    assume not_eq: "¬ (i = STR ''='' ∧ length2 l ∧ length2 l')"
    show ?thesis
    proof cases
      assume "semantics_list e f l = semantics_list e f l'"
      then have "semantics e f g (Imp (Atom (Rl i l)) (Atom (Rl i l')))"
      using assm not_eq equal_length2
      by simp iprover
      then show ?thesis
      using imp_chain_equiv
      by iprover
    next
      assume "semantics_list e f l ≠ semantics_list e f l'"
      then show ?thesis
```

```
      using assm imp_chain_zip_eq
      by iprover
    qed
  qed
qed

theorem soundness:
  "⊢ p ⟹ semantics e f g p"
proof (induct arbitrary: e rule: OK.induct)
  fix e s s'
  assume "semantics e f g (concl s)" "semantics e f g (concl s')" for e
  then show "semantics e f g (concl (modusponens s s'))"
  unfolding modusponens_def
  proof (cases s)
    fix r
    assume "semantics e f g (concl s)" "semantics e f g (concl s')" "s = Thm r" for e
    then show "semantics e f g (concl (case (concl s) of Imp p q ⇒
        let p' = concl s' in if fol_equal p p' then Thm q else fail_thm | _ ⇒ fail_thm))"
    unfolding fol_equal_def
    by (cases r) simp_all
  qed
next
  fix e x s
  assume "semantics e f g (concl s)" for e
  then show "semantics e f g (concl (gen x s))"
  unfolding gen_def
  by simp
next
  fix e p q
  show "semantics e f g (concl (axiom_addimp p q))"
  unfolding axiom_addimp_def
  by simp
next
  fix e p q r
  show "semantics e f g (concl (axiom_distribimp p q r))"
  unfolding axiom_distribimp_def
  by simp
next
  fix e g p
  show "semantics e f g (concl (axiom_doubleneg p))"
  unfolding axiom_doubleneg_def
  by simp
next
  fix e x p q
  show "semantics e f g (concl (axiom_allimp x p q))"
  unfolding axiom_allimp_def
  by simp
next
  fix e x p
  show "semantics e f g (concl (axiom_impall x p))"
  unfolding axiom_impall_def
  using map
  by simp iprover
next
  fix e x t
  show "semantics e f g (concl (axiom_existseq x t))"
  unfolding axiom_existseq_def
  using map'(1) length2_def
  by simp iprover
next
  fix e t
  show "semantics e f g (concl (axiom_eqrefl t))"
  unfolding axiom_eqrefl_def
  using length2_def
```

```
  by simp
next
  fix e i l l'
  show "semantics e f g (concl (axiom_funcong i l l'))"
  unfolding axiom_funcong_def
  using funcong
  by simp standard
next
  fix e i l l'
  show "semantics e f g (concl (axiom_predcong i l l'))"
  unfolding axiom_predcong_def
  using predcong
  by simp standard
next
  fix e p q
  show "semantics e f g (concl (axiom_iffimp1 p q))"
  unfolding axiom_iffimp1_def
  by simp
next
  fix e p q
  show "semantics e f g (concl (axiom_iffimp2 p q))"
  unfolding axiom_iffimp2_def
  by simp
next
  fix e p q
  show "semantics e f g (concl (axiom_impiff p q))"
  unfolding axiom_impiff_def
  by (simp add: iffI)
next
  fix e
  show "semantics e f g (concl (axiom_true))"
  unfolding axiom_true_def
  by simp
next
  fix e p
  show "semantics e f g (concl (axiom_not p))"
  unfolding axiom_not_def
  by simp
next
  fix e p q
  show "semantics e f g (concl (axiom_and p q))"
  unfolding axiom_and_def
  by simp
next
  fix e p q
  show "semantics e f g (concl (axiom_or p q))"
  unfolding axiom_or_def
  by simp
next
  fix e x p
  show "semantics e f g (concl (axiom_exists x p))"
  unfolding axiom_exists_def
  by simp
qed

corollary "¬ (⊢ Falsity)"
using soundness
by fastforce
```

ML Code Reflection

```
code_reflect
  Proven
```

datatypes
```
  fm = Falsity | Truth | Atom | Imp | Iff | And | Or | Not | Exists | Forall
```
and
```
  tm = Var | Fn
```
and
```
  fol = Rl
```
functions
```
  modusponens gen axiom_addimp axiom_distribimp axiom_doubleneg axiom_allimp axiom_impall
  axiom_existseq axiom_eqrefl axiom_funcong axiom_predcong axiom_iffimp1 axiom_iffimp2
  axiom_impiff axiom_true axiom_not axiom_and axiom_or axiom_exists concl
```

ML `<open Proven>`

ML `<val print = writeln (* Should not add newline but only used for testing (see XXX label) *)>`

ML `<"format_simple.sml";`

```
fun set_margin _ = ();

fun print_string x = print x;

fun open_box _ = ();

fun close_box () = ();

fun print_space () = print " ";

fun print_break _ _ = ();

fun open_hbox () = ();

fun print_flush () = ();

fun print_newline () = print "\n";

fun print_int n = print (Int.toString n);

fun open_hvbox _ = ();

>
```

ML `<"lib.sml";`

```
(* ========================================================================= *)
(* Misc library functions to set up a nice environment.                      *)
(* ========================================================================= *)

fun str_ord s1 s2 =
    case String.compare(s1,s2) of
      EQUAL => 0    | GREATER => 1    | LESS  => ~1;

fun sip_ord (f1,a1) (f2,a2) =
    case str_ord f1 f2 of
      0 => if a1>a2 then 1 else ~1
    | n => n
;

infix 6 lxor
infix 6 land

fun to_int_fun f = fn a => fn b => Word.toIntX (f ((Word.fromInt a),(Word.fromInt b) ) );

fun a lxor b = to_int_fun Word.xorb a b;
fun a land b = to_int_fun Word.andb a b;
```

```
fun list_hash elem_hash l=
  let fun hash_code sval l =
        case l of
         [] => sval
        | e::l' =>
          let val e_hash = Word.fromInt (elem_hash e) in
          hash_code (Word.+(Word.*(sval,0wx31),(e_hash))) l'
          end
  in
  Word.toIntX(hash_code 0wx0 l)
  end
;

fun str_hash str = list_hash Char.ord (String.explode str);

fun fst (x,_) = x;
fun snd (_,y) = y;

(* ========================================================================= *)
(* Misc library functions to set up a nice environment.                      *)
(* ========================================================================= *)

fun identity x = x;

(* ------------------------------------------------------------------------- *)
(* A useful idiom for "non contradictory" etc.                               *)
(* ------------------------------------------------------------------------- *)

fun non p x = not(p x);

(* ------------------------------------------------------------------------- *)
(* Kind of assertion checking.                                               *)
(* ------------------------------------------------------------------------- *)

fun check p x = if p(x) then x else raise Fail "check";

(* ------------------------------------------------------------------------- *)
(* Repetition of a function.                                                 *)
(* ------------------------------------------------------------------------- *)

fun funpow n f x =
    if n < 1 then x
    else funpow (n - 1) f (f x);

fun can f x = (f x; true) handle Fail _ => false;

fun repeat f x = repeat f (f x) handle Fail _ => x;

(* ------------------------------------------------------------------------- *)
(* Handy list operations.                                                    *)
(* ------------------------------------------------------------------------- *)

infix 6 --
fun m -- n = if m > n then [] else m::((m + 1) -- n);

fun map2 f l1 l2 =
  case (l1,l2) of
    ([],[]) => []
  | ((h1::t1),(h2::t2)) => let val h = f h1 h2 in h::(map2 f t1 t2) end
  | _ => raise Fail "map2: length mismatch";

fun itlist f l b = List.foldr (fn (x,y) => f x y) b l;
```

```
fun end_itlist f l =
  case l of
        []     => raise Fail "end_itlist"
      | [x]    => x
      | (h::t) => f h (end_itlist f t);

fun itlist2 f l1 l2 b =
  case (l1,l2) of
    ([],[]) => b
  | (h1::t1,h2::t2) => f h1 h2 (itlist2 f t1 t2 b)
  | _ => raise Fail "itlist2";

fun zip l1 l2 =
  case (l1,l2) of
        ([],[]) => []
      | (h1::t1,h2::t2) => (h1,h2)::(zip t1 t2)
      | _ => raise Fail "zip";

fun chop_list n l =
  if n = 0 then ([],l) else
  let val (m,l') = chop_list (n-1) (tl l) in ((hd l)::m,l') end
  handle Fail _ => raise Fail "chop_list";

fun index x =
  let fun ind n l =
    case l of
      [] => raise Fail "index"
    | (h::t) => if x = h then n else ind (n + 1) t
  in
    ind 0
  end;

fun unzip l =
  case l of
    [] => ([],[])
  | (x,y)::t =>
      let val (xs,ys) = unzip t in (x::xs,y::ys) end;

(* ------------------------------------------------------------------------- *)
(* Association lists.                                                        *)
(* ------------------------------------------------------------------------- *)

fun assoc a l =
  case l of
    (x,y)::t => if x = a then y else assoc a t
  | [] => raise Fail "find";

(* ------------------------------------------------------------------------- *)
(* Merging of sorted lists (maintaining repetitions).                        *)
(* ------------------------------------------------------------------------- *)

fun merge ord l1 l2 =
  case l1 of
    [] => l2
  | h1::t1 => case l2 of
                [] => l1
              | h2::t2 => if ord h1 h2 then h1::(merge ord t1 l2)
                          else h2::(merge ord l1 t2);

(* ------------------------------------------------------------------------- *)
(* Bottom-up mergesort.                                                      *)
(* ------------------------------------------------------------------------- *)

fun sort ord =
```

```
  let fun mergepairs l1 l2 =
    case (l1,l2) of
        ([s],[]) => s
      | (l,[]) => mergepairs [] l
      | (l,[s1]) => mergepairs (s1::l) []
      | (l,(s1::s2::ss)) => mergepairs ((merge ord s1 s2)::l) ss in
  fn l => if l = [] then [] else mergepairs [] (List.map (fn x => [x]) l)
  end;

(* ------------------------------------------------------------------------- *)
(* Common measure predicates to use with "sort".                             *)
(* ------------------------------------------------------------------------- *)

fun increasing f x y = (f x) < (f y);

fun decreasing f x y = (f x) > (f y) ;

(* ------------------------------------------------------------------------- *)
(* Eliminate repetitions of adjacent elements, with and without counting.    *)
(* ------------------------------------------------------------------------- *)

fun uniq l =
    case l of
      x :: (t as y :: ys ) =>
        let val t' = uniq t in
            if x = y then t'
            else
                x :: t'
        end
    | _ => l;

fun tryfind f l =
  case l of
      [] => raise Fail "tryfind"
    | (h::t) =>
      ((f h) handle Fail _ => tryfind f t);

(* ------------------------------------------------------------------------- *)
(* Set operations on ordered lists.                                          *)
(* ------------------------------------------------------------------------- *)

fun setify ord l=
  let fun canonical lis =
     case lis of
       x::(rest as y::_) => ord x y < 0 andalso canonical rest
     | _ => true in
  if canonical l then l
  else uniq (sort (fn x => fn y => ord x y <= 0) l)
  end;

fun union ord s1 s2=
  let fun union l1 l2 =
    case (l1,l2) of
        ([],l2) => l2
      | (l1,[]) => l1
      | ((l1 as h1::t1),(l2 as h2::t2)) =>
          if h1 = h2 then h1::(union t1 t2)
          else if ord h1 h2 = ~1 then h1::(union t1 l2)
          else h2::(union l1 t2) in
  union (setify ord s1) (setify ord s2)
  end;

 fun union_str s1 s2 = union str_ord s1 s2;
 fun union_sip p1 p2 = union sip_ord p1 p2;
```

```
fun subtract ord s1 s2=
  let fun subtract l1 l2 =
    case (l1,l2) of
        ([],l2) => []
      | (l1,[]) => l1
      | ((l1 as h1::t1),(l2 as h2::t2)) =>
          if h1 = h2 then subtract t1 t2
          else if ord h1 h2 = ~1 then h1::(subtract t1 l2)
          else subtract l1 t2 in
  subtract (setify ord s1) (setify ord s2)
  end;

fun subtract_str s1 s2 = subtract str_ord s1 s2;

fun insert ord x s = union ord [x] s;

fun insert_str x s = insert str_ord x s;

(* ------------------------------------------------------------------------- *)
(* Union of a family of sets.                                                *)
(* ------------------------------------------------------------------------- *)

fun unions ord s =
   let fun concat a b = a @ b in
   setify ord (itlist concat s [])
   end;

fun unions_str s = unions str_ord s;

(* ------------------------------------------------------------------------- *)
(* List membership. This does *not* assume the list is a set.                *)
(* ------------------------------------------------------------------------- *)

fun mem x lis =
    case lis of
      [] => false
    | hd :: tl => hd = x orelse mem x tl;

(* ------------------------------------------------------------------------- *)
(* Timing; useful for documentation but not logically necessary.             *)
(* ------------------------------------------------------------------------- *)

fun time f x =
    let val timer = Timer.startRealTimer()
        val result = f x
        val time = Timer.checkRealTimer timer
    in (
    (* XXX print_string ("CPU time (user): " ^ (Real.toString (Time.toReal time)));
    print_newline(); *)
    result
    ) end;

(* ------------------------------------------------------------------------- *)
(* Polymorphic finite partial functions via Patricia trees.                  *)
(*                                                                           *)
(* The point of this strange representation is that it is canonical (equal   *)
(* functions have the same encoding) yet reasonably efficient on average.    *)
(*                                                                           *)
(* Idea due to Diego Olivier Fernandez Pons (OCaml list, 2003/11/10).        *)
(* ------------------------------------------------------------------------- *)

datatype ('a,'b)func =
   Empty
```

```
 | Leaf of int * ('a*'b)list
 | Branch of int * int * ('a,'b)func * ('a,'b)func;

(* ------------------------------------------------------------------------ *)
(* Undefined function.                                                      *)
(* ------------------------------------------------------------------------ *)

val undefined = Empty;

(* ------------------------------------------------------------------------ *)
(* In case of equality comparison worries, better use this.                 *)
(* ------------------------------------------------------------------------ *)

fun is_undefined f =
  case f of
    Empty => true
  | _ => false;

(* ------------------------------------------------------------------------ *)
(* Operation analogous to "map" for lists.                                  *)
(* ------------------------------------------------------------------------ *)

local
  fun map_list f l =
        case l of
          [] => []
        | (x,y)::t => (x,f(y))::(map_list f t)
in
  fun mapf f t =
        case t of
          Empty => Empty
        | Leaf(h,l) => Leaf(h,map_list f l)
        | Branch(p,b,l,r) => Branch(p,b,mapf f l,mapf f r)
end;

(* ------------------------------------------------------------------------ *)
(* Application.                                                             *)
(* ------------------------------------------------------------------------ *)

fun applyd ord hash f d x=
  let fun apply_listd l d x =
        case l of
          (a,b)::t => if x = a then b else if ord x a > 0 then apply_listd t d x else d x
        | [] => d x
      val k = hash x
      fun look t =
        case t of
          Leaf(h,l) =>
            if (h = k) then
              apply_listd l d x
            else d x
        | Branch(p,b,l,r) =>
            if ((k lxor p) land (b - 1)) = 0 then
              look (if k land b = 0 then l else r)
            else d x
        | _ => d x
  in
  look f
  end
;

fun apply ord hash f = applyd ord hash f (fn x => raise Fail "apply");

fun apply_str f = apply str_ord str_hash f;
```

```
fun tryapplyd ord hash f a d = applyd ord hash f (fn x => d) a;

fun tryapplyd_str f a d = tryapplyd str_ord str_hash f a d;

fun tryapplyl ord hash f x = tryapplyd ord hash f x [];

fun defined ord hash f x = (apply ord hash f x; true) handle Fail _ => false;

fun defined_str f x = defined str_ord str_hash f x;

(* ------------------------------------------------------------------------ *)
(* Undefinition.                                                            *)
(* ------------------------------------------------------------------------ *)

local
  fun undefine_list ord x l =
    case l of
      (ab as (a,b))::t =>
          let val c = ord x a in
          if c = 0 then
            t
          else if c < 0 then
            l
          else
            let val t' = undefine_list ord x t in
            ab::t'
            end
          end
    | [] => []
in
  fun undefine ord hash x =
    let val k = hash x
        fun und t =
          case t of
            Leaf(h,l) =>
              if h=k then (
                let val l' = undefine_list ord x l in
                if l' = l then t
                else if l' = [] then Empty
                else Leaf(h,l')
                end
              ) else t
          | Branch(p,b,l,r) =>
              if k land (b - 1) = p then (
                if k land b = 0 then
                  let val l' = und l in
                  if l' = l then t
                  else (case l' of Empty => r | _ => Branch(p,b,l',r))
                  end
                else
                  let val r' = und r in
                  if r' = r then t
                  else (case r' of Empty => l | _ => Branch(p,b,l,r'))
                  end
              ) else t
          | _ => t
    in
    und
    end
end;

fun undefine_str x t = undefine str_ord str_hash x t
```

```
(* ------------------------------------------------------------------------ *)
(* Redefinition and combination.                                            *)
(* ------------------------------------------------------------------------ *)

infix 6 |->

local
  fun newbranch p1 t1 p2 t2 =
        let val zp = p1 lxor p2
            val b = zp land (~zp)
            val p = p1 land (b - 1) in
        if p1 land b = 0 then Branch(p,b,t1,t2)
        else Branch(p,b,t2,t1)
        end
  fun define_list ord (xy as (x,y)) l =
        case l of
          (ab as (a,b))::t =>
              let val c = ord x a in
              if c = 0 then xy::t
              else if c < 0 then xy::l
              else ab::(define_list ord xy t)
              end
        | [] => [xy]
  fun combine_list ord op' z l1 l2 =
        case (l1,l2) of
          ([],_) => l2
        | (_,[]) => l1
        | ((xy1 as (x1,y1))::t1,(xy2 as (x2,y2))::t2) =>
              let val c = ord x1 x2 in
              if c < 0 then
                xy1::(combine_list ord op' z t1 l2)
              else if c > 0 then
                xy2::(combine_list ord op' z l1 t2)
              else
                let val y = op' y1 y2
                    val l = combine_list ord op' z t1 t2 in
                if z(y) then l else (x1,y)::l
                end
              end
  in
  fun (x |-> y) t ord hash =
        let val k = hash x
            fun upd t =
              case t of
                Empty => Leaf (k,[(x,y)])
              | Leaf(h,l) =>
                   if h = k then Leaf(h,define_list ord (x,y) l)
                   else newbranch h t k (Leaf(k,[(x,y)]))
              | Branch(p,b,l,r) =>
                  if k land (b - 1) <> p then newbranch p t k (Leaf(k,[(x,y)]))
                  else if k land b = 0 then Branch(p,b,upd l,r)
                  else Branch(p,b,l,upd r) in
        upd t
        end
  fun combine ord op' z t1 t2 =
        case (t1,t2) of
          (Empty,_) => t2
        | (_,Empty) => t1
        | (Leaf(h1,l1),Leaf(h2,l2)) =>
              if h1 = h2 then
                let val l = combine_list ord op' z l1 l2 in
                if l = [] then Empty else Leaf(h1,l)
                end
              else newbranch h1 t1 h2 t2
```

```
        | ((lf as Leaf(k,lis)),(br as Branch(p,b,l,r))) =>
              if k land (b - 1) = p then
                if k land b = 0 then
                  (case combine ord op' z lf l of
                     Empty => r | l' => Branch(p,b,l',r))
                else
                  (case combine ord op' z lf r of
                     Empty => l | r' => Branch(p,b,l,r'))
              else
                newbranch k lf p br
        | ((br as Branch(p,b,l,r)),(lf as Leaf(k,lis))) =>
              if k land (b - 1) = p then
                if k land b = 0 then
                  (case combine ord op' z l lf of
                    Empty => r | l' => Branch(p,b,l',r))
                else
                  (case combine ord op' z r lf of
                     Empty => l | r' => Branch(p,b,l,r'))
              else
                newbranch p br k lf
        | (Branch(p1,b1,l1,r1),Branch(p2,b2,l2,r2)) =>
              if b1 < b2 then
                if p2 land (b1 - 1) <> p1 then newbranch p1 t1 p2 t2
                else if p2 land b1 = 0 then
                  (case combine ord op' z l1 t2 of
                     Empty => r1 | l => Branch(p1,b1,l,r1))
                else
                  (case combine ord op' z r1 t2 of
                     Empty => l1 | r => Branch(p1,b1,l1,r))
              else if b2 < b1 then
                if p1 land (b2 - 1) <> p2 then newbranch p1 t1 p2 t2
                else if p1 land b2 = 0 then
                  (case combine ord op' z t1 l2 of
                     Empty => r2 | l => Branch(p2,b2,l,r2))
                else
                  (case combine ord op' z t1 r2 of
                     Empty => l2 | r => Branch(p2,b2,l2,r))
              else if p1 = p2 then
               (case (combine ord op' z l1 l2,combine ord op' z r1 r2) of
                  (Empty,r) => r | (l,Empty) => l | (l,r) => Branch(p1,b1,l,r))
              else
                newbranch p1 t1 p2 t2
end ;

infix 6 |--> (* For strings *)

fun (x |--> y) t = (x |-> y) t str_ord str_hash;

(* ------------------------------------------------------------------------- *)
(* Special case of point function.                                           *)
(* ------------------------------------------------------------------------- *)

infix 6 |=>

fun x |=> y = (x |-> y) undefined;

infix 6 |==> (* For strings *)

fun x |==> y = (x |=> y) str_ord str_hash;

>

ML <"intro.sml";
```

```
(* ========================================================================= *)
(* Simple algebraic expression example from the introductory chapter.        *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Lexical analysis.                                                         *)
(* ------------------------------------------------------------------------- *)

fun matches s =
    let val chars = String.explode s in
    fn c => mem c chars
    end;

val space = matches " \t\n\r";
val punctuation = matches "()[]{},";
val symbolic = matches "~`!@#$%^&*-+=|\\:;<>.?/";
val numeric = matches "0123456789";
val alphanumeric = matches
  "abcdefghijklmnopqrstuvwxyz_'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

fun lexwhile prop inp =
  if inp <> [] andalso prop (List.hd inp) then
     let val (tok,rest) = lexwhile prop (List.tl inp) in
     ((str (List.hd inp))^tok,rest)
     end
  else
     ("",inp);

fun lex inp =
  case snd(lexwhile space inp) of
    [] => []
  | c::cs => let val prop = if alphanumeric(c) then alphanumeric
                        else if symbolic(c) then symbolic
                        else fn c => false
                 val (toktl,rest) = lexwhile prop cs in
             ((str c)^toktl)::lex rest
             end;

(* ------------------------------------------------------------------------- *)
(* Generic function to impose lexing and exhaustion checking on a parser.    *)
(* ------------------------------------------------------------------------- *)

fun make_parser pfn s =
  let val (expr,rest) = pfn (lex(String.explode s)) in
  if rest = [] then expr else raise Fail "Unparsed input"
  end;

>
```

ML `<"formulas.sml";`

```
(* ========================================================================= *)
(* Polymorphic type of formulas with parser and printer.                     *)
(* ========================================================================= *)

fun fm_ord at_ord fm1 fm2 =
    case (fm1,fm2) of
      (Falsity,Falsity) => 0
    | (Falsity,_) =>  1
    | (_,Falsity) => ~1

    | (Truth,Truth) => 0
    | (Truth,_)  =>  1
    | (_,Truth)  => ~1
```

```
    | (Atom a, Atom b) => at_ord a b
    | (Atom(_),_) => 1
    | (_,Atom(_)) => ~1

    | (Not a,Not b) => fm_ord at_ord a b
    | (Not(_),_) => 1
    | (_,Not(_)) => ~1

    | (And(a1,a2),And(b1,b2)) => fm_pair_ord at_ord (a1,a2) (b1,b2)
    | (And(_,_),_) =>  1
    | (_,And(_,_)) => ~1

    | (Or(a1,a2),Or(b1,b2)) => fm_pair_ord at_ord (a1,a2) (b1,b2)
    | (Or(_,_),_) =>  1
    | (_,Or(_,_)) => ~1

    | (Imp(a1,a2),Imp(b1,b2)) => fm_pair_ord at_ord (a1,a2) (b1,b2)
    | (Imp(_,_),_) =>  1
    | (_,Imp(_,_)) => ~1

    | (Iff(a1,a2),Iff(b1,b2)) => fm_pair_ord at_ord (a1,a2) (b1,b2)
    | (Iff(_,_),_) =>  1
    | (_,Iff(_,_)) => ~1

    | (Forall(x1,a),Forall(x2,b)) => fm_quant_ord at_ord (x1,a) (x2,b)
    | (Forall (_,_), _) => 1
    | (_, Forall (_,_)) => ~1

    | (Exists(x1,a),Exists(x2,b)) => fm_quant_ord at_ord (x1,a) (x2,b)
and fm_pair_ord at_ord (a1,a2) (b1,b2) =
    case fm_ord at_ord a1 b1 of
     0 => fm_ord at_ord a2 b2
    |n => n
and fm_quant_ord at_ord (x1,a) (x2,b) =
    case str_ord x1 x2 of
     0 => fm_ord at_ord a b
    |n => n
;

(* ------------------------------------------------------------------------- *)
(* General parsing of iterated infixes.                                      *)
(* ------------------------------------------------------------------------- *)

fun parse_ginfix opsym opupdate sof subparser inp =
  let val (e1,inp1) = subparser inp in
  if inp1 <> [] andalso List.hd inp1 = opsym then
     parse_ginfix opsym opupdate (opupdate sof e1) subparser (List.tl inp1)
  else (sof e1,inp1)
  end;

fun parse_left_infix opsym opcon =
  parse_ginfix opsym (fn f => fn e1 => fn e2 => opcon(f e1,e2)) (fn x => x);

fun parse_right_infix opsym opcon =
  parse_ginfix opsym (fn f => fn e1 => fn e2 => f(opcon(e1,e2))) (fn x => x);

fun parse_list opsym =
  parse_ginfix opsym (fn f => fn e1 => fn e2 => (f e1)@[e2]) (fn x => [x]);

(* ------------------------------------------------------------------------- *)
(* Other general parsing combinators.                                        *)
(* ------------------------------------------------------------------------- *)
```

```
fun papply f (ast,rest) = (f ast,rest);

fun nextin inp tok = inp <> [] andalso List.hd inp = tok;

fun parse_bracketed subparser cbra inp =
  let val(ast,rest) = subparser inp in
  if nextin rest cbra then (ast,List.tl rest)
  else raise Fail "Closing bracket expected"
  end;

(* ------------------------------------------------------------------------- *)
(* Parsing of formulas, parametrized by atom parser "pfn".                   *)
(* ------------------------------------------------------------------------- *)

fun parse_atomic_formula (ifn,afn) vs inp =
  case inp of
    [] => raise Fail "formula expected"
  | "false"::rest => (Falsity,rest)
  | "true"::rest => (Truth,rest)
  | "("::rest => ( (ifn vs inp) handle Fail _ =>
                  parse_bracketed (parse_formula (ifn,afn) vs) ")" rest)
  | "~"::rest => papply (fn p => Not p)
                        (parse_atomic_formula (ifn,afn) vs rest)
  | "forall"::x::rest =>
        parse_quant (ifn,afn) (x::vs) (fn (x,p) => Forall(x,p)) x rest
  | "exists"::x::rest =>
        parse_quant (ifn,afn) (x::vs) (fn (x,p) => Exists(x,p)) x rest
  | _ => afn vs inp

and parse_quant (ifn,afn) vs qcon x inp =
   case inp of
     [] => raise Fail "Body of quantified term expected"
   | y::rest =>
        papply (fn fm => qcon(x,fm))
               (if y = "." then parse_formula (ifn,afn) vs rest
                else parse_quant (ifn,afn) (y::vs) qcon y rest)

and parse_formula (ifn,afn) vs inp =
   parse_right_infix "<=>" (fn (p,q) => Iff(p,q))
     (parse_right_infix "==>" (fn (p,q) => Imp(p,q))
         (parse_right_infix "\\/" (fn (p,q) => Or(p,q))
             (parse_right_infix "/\\" (fn (p,q) => And(p,q))
                  (parse_atomic_formula (ifn,afn) vs)))) inp;

(* ------------------------------------------------------------------------- *)
(* Printing of formulas, parametrized by atom printer.                       *)
(* ------------------------------------------------------------------------- *)

fun bracket p n f x y = (
    (if p then print_string "(" else ());
    open_box n; f x y; close_box();
    (if p then print_string ")" else ())
);

fun strip_quant fm =
    case fm of
      Forall (x, (Forall (y, p))) =>
        let val (xs, q) = strip_quant (Forall (y, p)) in
        ((x :: xs), q)
        end
    | Exists (x, (Exists (y, p))) =>
        let val (xs, q) = strip_quant (Exists (y, p)) in
        ((x :: xs), q)
        end
```

```
    | Forall (x, p) =>
        ([x],p)
    | Exists (x, p) =>
        ([x],p)
    | _ =>
        ([], fm);

fun print_formula_aux pfn =
    let fun print_formula pr fm =
        case fm of
          Falsity =>
            print_string "false"
        | Truth =>
            print_string "true"
        | Atom pargs =>
            pfn pr pargs
        | Not p =>
            bracket (pr > 10) 1 (print_prefix 10) "~" p
        | And (p, q) =>
            bracket (pr > 8) 0 (print_infix 8 "/\\") p q
        | Or (p, q) =>
            bracket (pr > 6) 0 (print_infix 6 "\\/") p q
        | Imp (p, q) =>
            bracket (pr > 4) 0 (print_infix 4 "==>") p q
        | Iff (p, q) =>
            bracket (pr > 2) 0 (print_infix 2 "<=>") p q
        | Forall (x, p) =>
            bracket (pr > 0) 2 print_qnt "forall" (strip_quant fm)
        | Exists (x, p) =>
            bracket (pr > 0) 2 print_qnt "exists" (strip_quant fm)

    and print_qnt qname (bvs, bod) = (
        print_string qname;
        List.app (fn v => (print_string " "; print_string v)) bvs;
        print_string "."; print_space(); open_box 0;
        print_formula 0 bod;
        close_box()
    )

    and print_prefix newpr sym p = (
        print_string sym ; print_formula (newpr + 1) p
    )

    and print_infix newpr sym p q = (
        print_formula (newpr + 1) p ;
        print_string (" "^sym); print_space();
        print_formula newpr q
    ) in
    print_formula 0
    end
    ;

fun print_formula pfn fm = (print_formula_aux pfn fm; print_flush ());

fun print_qformula_aux pfn fm = (
  open_box 0; print_string "<!";
  open_box 0; print_formula_aux pfn fm; close_box();
  print_string "!>"; close_box()
);

fun print_qformula pfn fm = (print_qformula_aux pfn fm; print_flush ());

fun mk_and p q = And (p, q)
```

```
fun mk_or p q = Or (p, q)

fun mk_imp p q = Imp (p, q)

fun mk_iff p q = Iff (p, q)

fun mk_forall x p = Forall (x, p)

fun mk_exists x p = Exists (x, p)

(* ------------------------------------------------------------------------- *)
(* Destructors.                                                              *)
(* ------------------------------------------------------------------------- *)

fun dest_iff fm =
  case fm of Iff(p,q) => (p,q) | _ => raise Fail "dest_iff";

fun dest_and fm =
  case fm of And(p,q) => (p,q) | _ => raise Fail "dest_and";

fun conjuncts fm =
  case fm of And(p,q) => conjuncts p @ conjuncts q | _ => [fm];

fun dest_or fm =
  case fm of Or(p,q) => (p,q) | _ => raise Fail "dest_or";

fun disjuncts fm =
  case fm of Or(p,q) => disjuncts p @ disjuncts q | _ => [fm];

fun dest_imp fm =
  case fm of Imp(p,q) => (p,q) | _ => raise Fail "dest_imp";

fun antecedent fm = fst (dest_imp fm);
fun consequent fm = snd (dest_imp fm);

(* ------------------------------------------------------------------------- *)
(* Apply a function to the atoms, otherwise keeping structure.               *)
(* ------------------------------------------------------------------------- *)

fun onatoms f fm =
  case fm of
    Atom a => f a
  | Not(p) => Not(onatoms f p)
  | And(p,q) => And(onatoms f p,onatoms f q)
  | Or(p,q) => Or(onatoms f p,onatoms f q)
  | Imp(p,q) => Imp(onatoms f p,onatoms f q)
  | Iff(p,q) => Iff(onatoms f p,onatoms f q)
  | Forall(x,p) => Forall(x,onatoms f p)
  | Exists(x,p) => Exists(x,onatoms f p)
  | _ => fm;

(* ------------------------------------------------------------------------- *)
(* Formula analog of list iterator "itlist".                                 *)
(* ------------------------------------------------------------------------- *)

fun overatoms f fm b =
  case fm of
    Atom(a) => f a b
  | Not(p) => overatoms f p b
  | And(p,q) => overatoms f p (overatoms f q b)
  | Or(p,q)  => overatoms f p (overatoms f q b)
  | Imp(p,q) => overatoms f p (overatoms f q b)
  | Iff(p,q) => overatoms f p (overatoms f q b)
  | Forall(x,p) => overatoms f p b
```

```
  | Exists(x,p) => overatoms f p b
  | _ => b;

(* ------------------------------------------------------------------------- *)
(* Special case of a union of the results of a function over the atoms.      *)
(* ------------------------------------------------------------------------- *)

fun atom_union ord f fm = setify ord (overatoms (fn h => fn t => f(h)@t) fm []);

fun atom_union_sip f fm = atom_union sip_ord f fm;

>
```

ML `<"prop.sml";`

```
(* ========================================================================= *)
(* Basic stuff for propositional logic: datatype, parsing and printing.      *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Disjunctive normal form (DNF) via truth tables.                           *)
(* ------------------------------------------------------------------------- *)

fun list_conj l = if l = [] then Truth else end_itlist mk_and l;

>
```

ML `<"fol.sml";`

```
(* ========================================================================= *)
(* Basic stuff for first order logic.                                        *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Terms.                                                                    *)
(* ------------------------------------------------------------------------- *)

fun t_ord t1 t2 =
    case (t1,t2) of
      (Var x1, Var x2 ) => str_ord x1 x2
    | (Var _, _) => 1
    | (_, Var _) => ~1
    | (Fn(f1,tl1),Fn(f2,tl2)) =>
       case str_ord f1 f2 of
         0 => tl_ord tl1 tl2
        |n => n
and tl_ord tl1 tl2 =
    case (tl1,tl2) of
      ([],[]) => 0
    | ([],_) => 1
    | (_,[]) => ~1
    | (t1::tl1',t2::tl2') =>
        case t_ord t1 t2 of
         0 => tl_ord tl1' tl2'
       | n => n
;

fun t_hash t =
    case t of
      Var x => str_hash x
    | Fn (f,tl) => Word.toIntX(Word.+(Word.*(0wx31,Word.fromInt(str_hash f)),
                                      Word.fromInt(list_hash t_hash tl)))
;
```

```
infix 6 |---> (* For terms *)

fun (x |---> y) t = (x |-> y) t t_ord t_hash;

infix 6 |===> (* For terms *)
fun x |===> y = (x |=> y) t_ord t_hash;

fun apply_t f = apply t_ord t_hash f;

(* ------------------------------------------------------------------------- *)
(* Abbreviation for FOL formula.                                             *)
(* ------------------------------------------------------------------------- *)

fun fol_ord (r1 as Rl(s1,tl1)) (r2 as Rl(s2,tl2)) =
    case str_ord s1 s2 of
      0 => tl_ord tl1 tl2
    | n => n
;

fun folfm_ord fm1 fm2 = fm_ord fol_ord fm1 fm2;

fun union_folfm s1 s2 = union folfm_ord s1 s2;

fun ftp_ord (fm1,t1) (fm2,t2) =
    case folfm_ord fm1 fm2 of
      0 => t_ord t1 t2
    | n => n;

fun setify_ftp s = setify ftp_ord s;

(* ------------------------------------------------------------------------- *)
(* Special case of applying a subfunction to the top *terms*.                *)
(* ------------------------------------------------------------------------- *)

fun onformula f = onatoms(fn (Rl(p,a)) => Atom(Rl(p,List.map f a)));

(* ------------------------------------------------------------------------- *)
(* Parsing of terms.                                                         *)
(* ------------------------------------------------------------------------- *)

fun is_const_name s = List.all numeric (String.explode s) orelse s = "nil";

fun parse_atomic_term vs inp =
  case inp of
    [] => raise Fail "term expected"
  | "("::rest => parse_bracketed (parse_term vs) ")" rest
  | "-"::rest => papply (fn t => Fn("-",[t])) (parse_atomic_term vs rest)
  | f::"("::")"::rest => (Fn(f,[]),rest)
  | f::"("::rest =>
      papply (fn args => Fn(f,args))
             (parse_bracketed (parse_list "," (parse_term vs)) ")" rest)
  | a::rest =>
      ((if is_const_name a andalso not(mem a vs) then Fn(a,[]) else Var a),rest)

and parse_term vs inp =
  parse_right_infix "::" (fn (e1,e2) => Fn("::",[e1,e2]))
    (parse_right_infix "+" (fn (e1,e2) => Fn("+",[e1,e2]))
       (parse_left_infix "-" (fn (e1,e2) => Fn("-",[e1,e2]))
          (parse_right_infix "*" (fn (e1,e2) => Fn("*",[e1,e2]))
             (parse_left_infix "/" (fn (e1,e2) => Fn("/",[e1,e2]))
                (parse_left_infix "^" (fn (e1,e2) => Fn("^",[e1,e2]))
                   (parse_atomic_term vs))))))) inp;

val parset = make_parser (parse_term []);
```

```
(* ------------------------------------------------------------------------- *)
(* Parsing of formulas.                                                      *)
(* ------------------------------------------------------------------------- *)

fun parse_infix_atom vs inp =
  let val (tm,rest) = parse_term vs inp in
  if List.exists (nextin rest) ["=", "<", "<=", ">", ">="] then
        papply (fn tm' => Atom(Rl(List.hd rest,[tm,tm'])))
               (parse_term vs (List.tl rest))
  else raise Fail ""
  end;

fun parse_atom vs inp =
  (parse_infix_atom vs inp) handle Fail _ =>
  case inp of
    p::"("::")"::rest => (Atom(Rl(p,[])),rest)
  | p::"("::rest =>
      papply (fn args => Atom(Rl(p,args)))
             (parse_bracketed (parse_list "," (parse_term vs)) ")" rest)
  | p::rest =>
      if p <> "(" then (Atom(Rl(p,[])),rest)
      else raise Fail "parse_atom"
  | _ => raise Fail "parse_atom";

val parse = make_parser
  (parse_formula (parse_infix_atom,parse_atom) []);

(* ------------------------------------------------------------------------- *)
(* Set up parsing of quotations.                                             *)
(* ------------------------------------------------------------------------- *)

val default_parser = parse;
datatype default_parser_end = !>;
fun <! s !> = default_parser s;

val secondary_parser = parset;
datatype secondary_parser_end = |!>;
fun <!| s |!> = secondary_parser s;

(* ------------------------------------------------------------------------- *)
(* Printing of terms.                                                        *)
(* ------------------------------------------------------------------------- *)

fun print_term_aux prec fm =
  case fm of
    Var x => print_string x
  | Fn("^",[tm1,tm2]) => print_infix_term_aux true prec 24 "^" tm1 tm2
  | Fn("/",[tm1,tm2]) => print_infix_term_aux true prec 22 " /" tm1 tm2
  | Fn("*",[tm1,tm2]) => print_infix_term_aux false prec 20 " *" tm1 tm2
  | Fn("-",[tm1,tm2]) => print_infix_term_aux true prec 18 " -" tm1 tm2
  | Fn("+",[tm1,tm2]) => print_infix_term_aux false prec 16 " +" tm1 tm2
  | Fn("::",[tm1,tm2]) => print_infix_term_aux false prec 14 "::" tm1 tm2
  | Fn(f,args) => print_fargs_aux f args

and print_fargs_aux f args = (
  print_string f;
  if args = [] then () else
   (print_string "(";
    open_box 0;
    print_term_aux 0 (List.hd args); print_break 0 0;
    List.app (fn t => (print_string ","; print_break 0 0 ; print_term_aux 0 t))
             (List.tl args);
    close_box ();
```

```
    print_string ")")
)

and print_infix_term_aux isleft oldprec newprec sym p q = (
  if oldprec > newprec then (print_string "("; open_box 0) else ();
  print_term_aux (if isleft then newprec else newprec+1) p;
  print_string sym;
  print_break (if String.substring (sym, 0, 1) = " " then 1 else 0) 0;
  print_term_aux (if isleft then newprec+1 else newprec) q;
  if oldprec > newprec then (close_box (); print_string ")") else ()
);

fun print_term prec fm = (print_term_aux prec fm; print_flush ())
and print_fargs f args = (print_fargs_aux f args; print_flush ())
and print_infix_term isleft oldprec newprec sym p q = (print_infix_term_aux isleft oldprec newprec sym p q; print_flus
());

fun printert_aux tm = (
  open_box 0; print_string "<!|";
  open_box 0; print_term_aux 0 tm; close_box();
  print_string "|!>"; close_box()
);

fun printert tm = (printert_aux tm; print_flush ());

(* ------------------------------------------------------------------------- *)
(* Printing of formulas.                                                     *)
(* ------------------------------------------------------------------------- *)

fun print_atom_aux prec (Rl (p, args)) =
    if mem p ["=", "<", "<=", ">", ">="] andalso List.length args = 2 then
        print_infix_term_aux false 12 12 (" " ^ p) (List.nth (args, 0)) (List.nth (args, 1))
    else
        print_fargs_aux p args;

fun print_atom prec rpa = (print_atom_aux prec rpa; print_flush ());

val print_fol_formula_aux = print_qformula_aux print_atom_aux;

fun print_fol_formula f = (print_fol_formula_aux f; print_flush ());

(* ------------------------------------------------------------------------- *)
(* Free variables in terms and formulas.                                     *)
(* ------------------------------------------------------------------------- *)

fun fvt tm =
    case tm of
      Var x => [x]
    | Fn (f, args) =>
        unions_str (List.map fvt args)
;

fun var fm =
    case fm of
      Falsity => []
    | Truth => []
    | Atom (Rl (p, args)) =>
        unions str_ord (List.map fvt args)
    | Not p => var p
    | And (p, q) => union_str (var p) (var q)
    | Or  (p, q) => union_str (var p) (var q)
    | Imp (p, q) => union_str (var p) (var q)
    | Iff (p, q) => union_str (var p) (var q)
    | Forall (x, p) => insert_str x (var p)
```

```
    | Exists (x, p) => insert_str x (var p)
;

fun fv fm =
    case fm of
      Falsity => []
    | Truth => []
    | Atom (Rl (p, args)) =>
        unions_str (List.map fvt args)
    | Not p => fv p
    | And (p, q) => union_str (fv p) (fv q)
    | Or  (p, q) => union_str (fv p) (fv q)
    | Imp (p, q) => union_str (fv p) (fv q)
    | Iff (p, q) => union_str (fv p) (fv q)
    | Forall (x, p) => subtract_str (fv p) [x]
    | Exists (x, p) => subtract_str (fv p) [x]
;

(* ------------------------------------------------------------------------- *)
(* Substitution within terms.                                                *)
(* ------------------------------------------------------------------------- *)

fun tsubst sfn tm =
  case tm of
    Var x => tryapplyd_str sfn x tm
  | Fn(f,args) => Fn(f,List.map (tsubst sfn) args);

fun variant x vars =
  if mem x vars then variant (x^"'") vars else x;

(* ------------------------------------------------------------------------- *)
(* Substitution in formulas, with variable renaming.                         *)
(* ------------------------------------------------------------------------- *)

fun subst subfn fm =
    case fm of
      Falsity => Falsity
    | Truth => Truth
    | Atom (Rl (p, args)) =>
        Atom (Rl (p, List.map (tsubst subfn) args))
    | Not p =>
        Not (subst subfn p)
    | And (p, q) =>
        And (subst subfn p, subst subfn q)
    | Or (p, q) =>
        Or  (subst subfn p, subst subfn q)
    | Imp (p, q) =>
        Imp (subst subfn p, subst subfn q)
    | Iff (p, q) =>
        Iff (subst subfn p, subst subfn q)
    | Forall (x, p) =>
        substq subfn mk_forall x p
    | Exists (x, p) =>
        substq subfn mk_exists x p
and substq subfn quant x p =
    let val x' =
        if List.exists (fn y => mem x (fvt (tryapplyd_str subfn y (Var y))))
                       (subtract_str (fv p) [x]) then
            variant x (fv (subst (undefine_str x subfn) p))
        else x
    in
    quant x' (subst ((x |--> Var x') subfn) p)
    end
;
```

```
>

ML <"skolem.sml";

(* ========================================================================= *)
(* Prenex and Skolem normal forms.                                           *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Get the functions in a term and formula.                                  *)
(* ------------------------------------------------------------------------- *)

fun funcs tm =
  case tm of
    Var x => []
  | Fn(f,args) => itlist (union_sip o funcs) args [(f,List.length args)];

fun functions fm =
  atom_union_sip (fn (Rl(p,a)) => itlist (union_sip o funcs) a []) fm;

>

ML <"unif.sml";

(* ========================================================================= *)
(* Unification for first order terms.                                        *)
(* ========================================================================= *)

fun istriv env x t =
  case t of
    Var y => y = x orelse defined_str env y andalso istriv env x (apply_str env y)
  | Fn(f,args) => List.exists (istriv env x) args andalso raise Fail "cyclic";

(* ------------------------------------------------------------------------- *)
(* Main unification procedure                                                *)
(* ------------------------------------------------------------------------- *)

fun unify env eqs =
  case eqs of
    [] => env
  | (Fn(f,fargs),Fn(g,gargs))::oth =>
        if f = g andalso length fargs = length gargs
        then unify env (zip fargs gargs @ oth)
        else raise Fail "impossible unification"
  | (Var x,t)::oth =>
        if defined_str env x then unify env ((apply_str env x,t)::oth)
        else unify (if istriv env x t then env else (x|-->t) env) oth
  | (t,Var x)::oth =>
        if defined_str env x then unify env ((apply_str env x,t)::oth)
        else unify (if istriv env x t then env else (x|-->t) env) oth;

(* ------------------------------------------------------------------------- *)
(* Solve to obtain a single instantiation.                                   *)
(* ------------------------------------------------------------------------- *)

fun solve env =
  let val env' = mapf (tsubst env) env in
  if env' = env then env else solve env'
  end;

(* ------------------------------------------------------------------------- *)
(* Unification reaching a final solved form (often this isn't needed).       *)
(* ------------------------------------------------------------------------- *)
```

```
fun fullunify eqs = solve (unify undefined eqs);

fun unify_and_apply eqs =
  let val i = fullunify eqs
      fun apply (t1,t2) = (tsubst i t1,tsubst i t2) in
  map apply eqs
  end;

>
```

ML `<"tableaux.sml";`

```
(* ========================================================================= *)
(* Tableaux, seen as an optimized version of a Prawitz-like procedure.       *)
(* ========================================================================= *)

fun deepen f n =
  ((* XXX print_string "Searching with depth limit ";
      print_int n; print_newline(); *) f n
  )
  handle Fail _ => deepen f (n + 1);

>
```

ML `<"resolution.sml";`

```
(* ========================================================================= *)
(* Resolution.                                                               *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Matching of terms and literals.                                           *)
(* ------------------------------------------------------------------------- *)

fun term_match env eqs =
    case eqs of
      [] => env
    | (Fn (f, fa), Fn(g, ga)) :: oth =>
        if (f = g andalso List.length fa = List.length ga) then
        term_match env (zip fa ga @ oth)
        else raise Fail "term_match"
    | (Var x, t) :: oth =>
        if not (defined_str env x) then
            term_match ((x |--> t) env) oth
        else if apply_str env x = t then
            term_match env oth
        else
            raise Fail "term_match"
    | _ =>
        raise Fail "term_match";

>
```

ML `<"equal.sml";`

```
(* ========================================================================= *)
(* First order logic with equality.                                          *)
(* ========================================================================= *)

fun mk_eq s t = Atom(Rl("=",[s,t]));

fun dest_eq fm =
  case fm of
```

```
    Atom(Rl("=",[s,t])) => (s,t)
  | _ => raise Fail "dest_eq: not an equation";

fun lhs eq = fst (dest_eq eq) and rhs eq = snd (dest_eq eq);

>
```

ML `<"order.sml";`

```
(* ========================================================================= *)
(* Term orderings.                                                           *)
(* ========================================================================= *)

fun termsize tm =
  case tm of
    Var x => 1
  | Fn(f,args) => itlist (fn t => fn n => termsize t + n) args 1;

>
```

ML `<"eqelim.sml";`

```
(* ========================================================================= *)
(* Equality elimination including Brand transformation and relatives.        *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Replacement (substitution for non-variable) in term and literal.          *)
(* ------------------------------------------------------------------------- *)

fun replacet rfn tm =
  apply_t rfn tm
  handle Fail _ =>
  case tm of
    Fn(f,args) => Fn(f,List.map (replacet rfn) args)
  | _ => tm;

>
```

ML `<"lcf.sml";`

```
fun print_thm_aux th = (
    open_box 0;
    print_string "|-"; print_space();
    open_box 0; print_formula_aux print_atom_aux (concl th); close_box();
    close_box()
)

fun print_thm th = (print_thm_aux th; print_flush ())

>
```

ML `<"lcfprop.sml";`

```
(* ========================================================================= *)
(* Propositional reasoning by derived rules atop the LCF core.               *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* |- p ==> p                                                                *)
(* ------------------------------------------------------------------------- *)

fun imp_refl p =
  modusponens (modusponens (axiom_distribimp p (Imp(p,p)) p)
```

```
                          (axiom_addimp p (Imp(p,p))))
              (axiom_addimp p p);

(* ------------------------------------------------------------------------- *)
(*                 |- p ==> p ==> q                                         *)
(*               -------------------- imp_unduplicate                       *)
(*                 |- p ==> q                                               *)
(* ------------------------------------------------------------------------- *)

fun imp_unduplicate th =
  let val (p,pq) = dest_imp(concl th)
      val q = consequent pq in
  modusponens (modusponens (axiom_distribimp p p q) th) (imp_refl p)
  end ;

(* ------------------------------------------------------------------------- *)
(* Some handy syntax operations.                                             *)
(* ------------------------------------------------------------------------- *)

fun negatef fm =
  case fm of
    Imp(p,Falsity) => p
  | p => Imp(p,Falsity);

fun negativef fm =
  case fm of
    Imp(p,Falsity) => true
  | _ => false;

(* ------------------------------------------------------------------------- *)
(*                           |- q                                           *)
(*         ------------------------------------------- add_assum (p)        *)
(*                          |- p ==> q                                      *)
(* ------------------------------------------------------------------------- *)

fun add_assum p th = modusponens (axiom_addimp (concl th) p) th;

(* ------------------------------------------------------------------------- *)
(*                   |- q ==> r                                             *)
(*         ---------------------------------- imp_add_assum p               *)
(*           |- (p ==> q) ==> (p ==> r)                                      *)
(* ------------------------------------------------------------------------- *)

fun imp_add_assum p th =
  let val (q,r) = dest_imp(concl th) in
  modusponens (axiom_distribimp p q r) (add_assum p th)
  end;

(* ------------------------------------------------------------------------- *)
(*            |- p ==> q              |- q ==> r                            *)
(*         ---------------------------------------- imp_trans                *)
(*                 |- p ==> r                                               *)
(* ------------------------------------------------------------------------- *)

fun imp_trans th1 th2 =
  let val p = antecedent(concl th1) in
  modusponens (imp_add_assum p th2) th1
  end;

(* ------------------------------------------------------------------------- *)
(*                 |- p ==> r                                               *)
(*         ------------------------- imp_insert q                           *)
(*              |- p ==> q ==> r                                            *)
(* ------------------------------------------------------------------------- *)
```

```
fun imp_insert q th =
  let val (p,r) = dest_imp(concl th) in
  imp_trans th (axiom_addimp r q)
  end ;

(* ------------------------------------------------------------------------- *)
(*                |- p ==> q ==> r                                           *)
(*             ---------------------- imp_swap                              *)
(*                |- q ==> p ==> r                                           *)
(* ------------------------------------------------------------------------- *)

fun imp_swap th =
  let val (p,qr) = dest_imp(concl th)
      val (q,r) = dest_imp qr in
  imp_trans (axiom_addimp q p)
            (modusponens (axiom_distribimp p q r) th)
  end;

(* ------------------------------------------------------------------------- *)
(* |- (q ==> r) ==> (p ==> q) ==> (p ==> r)                                  *)
(* ------------------------------------------------------------------------- *)

fun imp_trans_th p q r =
   imp_trans (axiom_addimp (Imp(q,r)) p)
             (axiom_distribimp p q r);

(* ------------------------------------------------------------------------- *)
(*                |- p ==> q                                                *)
(*         ------------------------------ imp_add_concl r                   *)
(*          |- (q ==> r) ==> (p ==> r)                                       *)
(* ------------------------------------------------------------------------- *)

fun imp_add_concl r th =
  let val (p,q) = dest_imp(concl th) in
  modusponens (imp_swap(imp_trans_th p q r)) th
  end;

(* ------------------------------------------------------------------------- *)
(* |- (p ==> q ==> r) ==> (q ==> p ==> r)                                    *)
(* ------------------------------------------------------------------------- *)

fun imp_swap_th p q r =
  imp_trans (axiom_distribimp p q r)
            (imp_add_concl (Imp(p,r)) (axiom_addimp q p));

(* ------------------------------------------------------------------------- *)
(*  |- (p ==> q ==> r) ==> (s ==> t ==> u)                                   *)
(* ----------------------------------------                                  *)
(*  |- (q ==> p ==> r) ==> (t ==> s ==> u)                                   *)
(* ------------------------------------------------------------------------- *)

fun imp_swap2 th =
  case concl th of
    Imp(Imp(p,Imp(q,r)),Imp(s,Imp(t,u))) =>
        imp_trans (imp_swap_th q p r) (imp_trans th (imp_swap_th s t u))
  | _ => raise Fail "imp_swap2";

(* ------------------------------------------------------------------------- *)
(* If |- p ==> q ==> r and |- p ==> q then |- p ==> r.                       *)
(* ------------------------------------------------------------------------- *)

fun right_mp ith th =
  imp_unduplicate(imp_trans th (imp_swap ith));
```

```
(* ------------------------------------------------------------------------- *)
(*                |- p <=> q                                                 *)
(*                ------------ iff_imp1                                      *)
(*                |- p ==> q                                                 *)
(* ------------------------------------------------------------------------- *)

fun iff_imp1 th =
  let val (p,q) = dest_iff(concl th) in
  modusponens (axiom_iffimp1 p q) th
  end;

(* ------------------------------------------------------------------------- *)
(*                |- p <=> q                                                 *)
(*                ------------ iff_imp2                                      *)
(*                |- q ==> p                                                 *)
(* ------------------------------------------------------------------------- *)

fun iff_imp2 th =
  let val (p,q) = dest_iff(concl th) in
  modusponens (axiom_iffimp2 p q) th
  end;

(* ------------------------------------------------------------------------- *)
(*         |- p ==> q      |- q ==> p                                        *)
(*        --------------------------- imp_antisym                           *)
(*              |- p <=> q                                                   *)
(* ------------------------------------------------------------------------- *)

fun imp_antisym th1 th2 =
  let val (p,q) = dest_imp(concl th1) in
  modusponens (modusponens (axiom_impiff p q) th1) th2
  end;

(* ------------------------------------------------------------------------- *)
(*         |- p ==> (q ==> false) ==> false                                  *)
(*       ---------------------------------- right_doubleneg                 *)
(*               |- p ==> q                                                  *)
(* ------------------------------------------------------------------------- *)

fun right_doubleneg th =
  case concl th of
    Imp(_,Imp(Imp(p,Falsity),Falsity)) => imp_trans th (axiom_doubleneg p)
  | _ => raise Fail "right_doubleneg";

(* ------------------------------------------------------------------------- *)
(*                                                                           *)
(*         ----------------------------------------- ex_falso (p)           *)
(*                |- false ==> p                                             *)
(* ------------------------------------------------------------------------- *)

fun ex_falso p = right_doubleneg(axiom_addimp Falsity (Imp(p,Falsity)));

(* ------------------------------------------------------------------------- *)
(*  |- p ==> q ==> r        |- r ==> s                                        *)
(* ----------------------------------- imp_trans2                            *)
(*      |- p ==> q ==> s                                                     *)
(* ------------------------------------------------------------------------- *)

fun imp_trans2 th1 th2 =
  let val Imp(p,Imp(q,r)) = concl th1
      val Imp(r',s) = concl th2
      val th = imp_add_assum p (modusponens (imp_trans_th q r s) th2) in
  modusponens th th1
```

```
  end;

(* ------------------------------------------------------------------------- *)
(*         |- p ==> q1   ...   |- p ==> qn   |- q1 ==> ... ==> qn ==> r      *)
(*        ------------------------------------------------------------     *)
(*                              |- p ==> r                                    *)
(* ------------------------------------------------------------------------- *)

fun imp_trans_chain ths th =
  itlist (fn a => fn b => imp_unduplicate (imp_trans a (imp_swap b)))
    (List.rev(List.tl ths)) (imp_trans (List.hd ths) th);

(* ------------------------------------------------------------------------- *)
(* |- (q ==> false) ==> p ==> (p ==> q) ==> false                            *)
(* ------------------------------------------------------------------------- *)

fun imp_truefalse p q =
  imp_trans (imp_trans_th p q Falsity) (imp_swap_th (Imp(p,q)) p Falsity);

(* ------------------------------------------------------------------------- *)
(*  |- (p' ==> p) ==> (q ==> q') ==> (p ==> q) ==> (p' ==> q')               *)
(* ------------------------------------------------------------------------- *)

fun imp_mono_th p p' q q' =
  let val th1 = imp_trans_th (Imp(p,q)) (Imp(p',q)) (Imp(p',q'))
      val th2 = imp_trans_th p' q q'
      val th3 = imp_swap(imp_trans_th p' p q) in
  imp_trans th3 (imp_swap(imp_trans th2 th1))
  end;

(* ------------------------------------------------------------------------- *)
(* |- true                                                                   *)
(* ------------------------------------------------------------------------- *)

val truth = modusponens (iff_imp2 axiom_true) (imp_refl Falsity);

(* ------------------------------------------------------------------------- *)
(*         |- p ==> q                                                        *)
(*      ----------------- contrapos                                          *)
(*         |- ~q ==> ~p                                                      *)
(* ------------------------------------------------------------------------- *)

fun contrapos th =
  let val (p,q) = dest_imp(concl th) in
  imp_trans (imp_trans (iff_imp1(axiom_not q)) (imp_add_concl Falsity th))
            (iff_imp2(axiom_not p))
  end;

(* ------------------------------------------------------------------------- *)
(* |- p /\ q ==> p                                                           *)
(* ------------------------------------------------------------------------- *)

fun and_left p q =
  let val th1 = imp_add_assum p (axiom_addimp Falsity q)
      val th2 = right_doubleneg(imp_add_concl Falsity th1) in
  imp_trans (iff_imp1(axiom_and p q)) th2
  end;

(* ------------------------------------------------------------------------- *)
(* |- p /\ q ==> q                                                           *)
(* ------------------------------------------------------------------------- *)

fun and_right p q =
  let val th1 = axiom_addimp (Imp(q,Falsity)) p
```

```
      val th2 = right_doubleneg(imp_add_concl Falsity th1) in
  imp_trans (iff_imp1(axiom_and p q)) th2
  end;

(* ------------------------------------------------------------------------- *)
(* |- p1 /\ ... /\ pn ==> pi for each 1 <= i <= n (input term right assoc)   *)
(* ------------------------------------------------------------------------- *)

fun conjths fm =
  let val (p,q) = dest_and fm in
      (and_left p q)::List.map (imp_trans (and_right p q)) (conjths q)
  end handle Fail _ => [imp_refl fm];

(* ------------------------------------------------------------------------- *)
(* |- p ==> q ==> p /\ q                                                     *)
(* ------------------------------------------------------------------------- *)

fun and_pair p q =
  let val th1 = iff_imp2(axiom_and p q)
      val th2 = imp_swap_th (Imp(p,Imp(q,Falsity))) q Falsity
      val th3 = imp_add_assum p (imp_trans2 th2 th1) in
  modusponens th3 (imp_swap (imp_refl (Imp(p,Imp(q,Falsity)))))
  end;

(* ------------------------------------------------------------------------- *)
(* If |- p /\ q ==> r then |- p ==> q ==> r                                  *)
(* ------------------------------------------------------------------------- *)

fun shunt th =
  let val (p,q) = dest_and(antecedent(concl th)) in
  modusponens (itlist imp_add_assum [p,q] th) (and_pair p q)
  end;

(* ------------------------------------------------------------------------- *)
(* If |- p ==> q ==> r then |- p /\ q ==> r                                  *)
(* ------------------------------------------------------------------------- *)

fun unshunt th =
  let val (p,qr) = dest_imp(concl th)
      val (q,r) = dest_imp qr in
  imp_trans_chain [and_left p q, and_right p q] th
  end;

(* ------------------------------------------------------------------------- *)
(* Produce |- (p <=> q) <=> (p ==> q) /\ (q ==> p)                           *)
(* ------------------------------------------------------------------------- *)

fun iff_def p q = (* Not in the book *)
  let val th1 = and_pair (Imp(p,q)) (Imp(q,p))
      val th2 = imp_trans_chain [axiom_iffimp1 p q, axiom_iffimp2 p q] th1 in
  imp_antisym th2 (unshunt (axiom_impiff p q))
  end;

fun iff_def p q =
  let val th = and_pair (Imp(p,q)) (Imp(q,p))
      val thl = [axiom_iffimp1 p q, axiom_iffimp2 p q] in
  imp_antisym (imp_trans_chain thl th) (unshunt (axiom_impiff p q))
  end;

(* ------------------------------------------------------------------------- *)
(* Produce "expansion" theorem for defined connectives.                      *)
(* ------------------------------------------------------------------------- *)

fun expand_connective fm =
```

```
  case fm of
    Truth => axiom_true
  | Not p => axiom_not p
  | And(p,q) => axiom_and p q
  | Or(p,q) => axiom_or p q
  | Iff(p,q) => iff_def p q
  | Exists(x,p) => axiom_exists x p
  | _ => raise Fail "expand_connective";

fun eliminate_connective fm =
  if not(negativef fm) then iff_imp1(expand_connective fm)
  else imp_add_concl Falsity (iff_imp2(expand_connective(negatef fm)));

(* ------------------------------------------------------------------------- *)
(*                                                                           *)
(*   ------------------------------------------------ imp_false_conseqs      *)
(*      [|- ((p ==> q) ==> false) ==> (q ==> false);                         *)
(*       |- ((p ==> q) ==> false) ==> p]                                     *)
(* ------------------------------------------------------------------------- *)

fun imp_false_conseqs p q =
 [right_doubleneg(imp_add_concl Falsity (imp_add_assum p (ex_falso q))),
  imp_add_concl Falsity (imp_insert p (imp_refl q))];

(* ------------------------------------------------------------------------- *)
(*         |- p ==> (q ==> false) ==> r                                      *)
(*        ------------------------------------ imp_false_rule                *)
(*             |- ((p ==> q) ==> false) ==> r                                *)
(* ------------------------------------------------------------------------- *)

fun imp_false_rule th =
  let val (p,r) = dest_imp (concl th) in
  imp_trans_chain (imp_false_conseqs p (funpow 2 antecedent r)) th
  end;

(* ------------------------------------------------------------------------- *)
(*         |- (p ==> false) ==> r          |- q ==> r                       *)
(*       ------------------------------------------------ imp_true_rule      *)
(*                      |- (p ==> q) ==> r                                   *)
(* ------------------------------------------------------------------------- *)

fun imp_true_rule th1 th2 =
  let val p = funpow 2 antecedent (concl th1)
      val q = antecedent(concl th2)
      val th3 = right_doubleneg(imp_add_concl Falsity th1)
      val th4 = imp_add_concl Falsity th2
      val th5 = imp_swap(imp_truefalse p q)
      val th6 = imp_add_concl Falsity (imp_trans_chain [th3, th4] th5)
      val th7 = imp_swap(imp_refl(Imp(Imp(p,q),Falsity))) in
  right_doubleneg(imp_trans th7 th6)
  end;

(* ------------------------------------------------------------------------- *)
(*                                 *                                         *)
(*                ------------------------------------ imp_contr           *)
(*                       |- p ==> -p ==> q                                   *)
(* ------------------------------------------------------------------------- *)

fun imp_contr p q =
  if negativef p then imp_add_assum (negatef p) (ex_falso q)
  else imp_swap (imp_add_assum p (ex_falso q));

(* ------------------------------------------------------------------------- *)
(*                                                                           *)
```

```
(* ------------------------------------------- imp_front (this antecedent) *)
(*  |- (p0 ==> p1 ==> ... ==> pn ==> q)                                     *)
(*     ==> pn ==> p0 ==> p1 ==> .. p(n-1) ==> q                            *)
(* ------------------------------------------------------------------------ *)

fun imp_front_th n fm =
  if n = 0 then imp_refl fm else
  let val (p,qr) = dest_imp fm
      val th1 = imp_add_assum p (imp_front_th (n - 1) qr)
      val (q',r') = dest_imp(funpow 2 consequent(concl th1)) in
  imp_trans th1 (imp_swap_th p q' r')
  end;

(* ------------------------------------------------------------------------ *)
(*           |- p0 ==> p1 ==> ... ==> pn ==> q                             *)
(*         ---------------------------------------- imp_front n            *)
(*           |- pn ==> p0 ==> p1 ==> .. p(n-1) ==> q                       *)
(* ------------------------------------------------------------------------ *)

fun imp_front n th = modusponens (imp_front_th n (concl th)) th;

(* ------------------------------------------------------------------------ *)
(* Propositional tableaux procedure.                                        *)
(* ------------------------------------------------------------------------ *)

fun is_false Falsity = true
  | is_false _ = false;

fun is_true (Imp(p,q)) = (p = q)
  | is_true _ = false;

fun is_conj (Imp(Imp(p,q),Falsity)) = true
  | is_conj _ = false

fun dest_conj fm =
    case fm of
      (Imp(Imp(p,q),Falsity)) => (p,q)
    | _ => raise Fail "dest_conj"

fun is_disj (Imp(p,q)) = (q <> Falsity)
  | is_disj _ = false

fun dest_disj fm = dest_imp fm;

fun is_prop_lit p =
  case p of
     Atom(_) => true
   | Forall(_,_) => true
   | Imp(Atom(_),Falsity) => true
   | Imp(Forall(_),Falsity) => true
   | _ => false ;

fun lcfptab fms lits =
    case fms of
      []     => raise Fail "lcfptab: no contradiction"
    | fm::fl =>
        if is_false fm then (
            ex_falso (itlist mk_imp (fl @ lits) Falsity)
        ) else if is_true fm then (
            add_assum fm (lcfptab fl lits)
        ) else if is_conj fm then (
            let val (p,q)=dest_conj fm in
            imp_false_rule(lcfptab (p::Imp(q,Falsity)::fl) lits)
            end
```

```
        ) else if is_disj fm then (
            let val (p,q)=dest_disj fm in
            imp_true_rule (lcfptab (Imp(p,Falsity)::fl) lits) (lcfptab (q::fl) lits)
            end
        ) else if is_prop_lit fm then (
            if mem (negatef fm) lits then
              let val (l1,l2) = chop_list (index (negatef fm) lits) lits
                  val th = imp_contr fm (itlist mk_imp (List.tl l2) Falsity ) in
              itlist imp_insert (fl @ l1) th
              end
            else imp_front (List.length fl) (lcfptab fl (fm::lits))
        ) else ( (* is nonprimitive *)
           let val th = eliminate_connective fm in
           imp_trans th (lcfptab (consequent(concl th)::fl) lits)
           end
        )
;

(* ------------------------------------------------------------------------- *)
(* In particular, this gives a tautology prover.                             *)
(* ------------------------------------------------------------------------- *)

fun lcftaut p =
  modusponens (axiom_doubleneg p) (lcfptab [negatef p] []);

>
```

ML `<"folderived.sml";`

```
(* ========================================================================= *)
(* First-order derived rules in the LCF setup.                               *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(*                          ******                                           *)
(*         ------------------------------------------ eq_sym                 *)
(*                       |- s = t ==> t = s                                  *)
(* ------------------------------------------------------------------------- *)

fun eq_sym s t =
  let val rth = axiom_eqrefl s in
  funpow 2 (fn th => (modusponens (imp_swap th) rth))
           (axiom_predcong "=" [s, s] [t, s])
  end;

(* ------------------------------------------------------------------------- *)
(* |- s = t ==> t = u ==> s = u.                                             *)
(* ------------------------------------------------------------------------- *)

fun eq_trans s t u =
  let val th1 = axiom_predcong "=" [t, u] [s, u]
      val th2 = modusponens (imp_swap th1) (axiom_eqrefl u) in
  imp_trans (eq_sym s t) th2
  end;

(* ------------------------------------------------------------------------- *)
(*         ---------------------------- icongruence                          *)
(*          |- s = t ==> tm[s] = tm[t]                                       *)
(* ------------------------------------------------------------------------- *)

fun icongruence s t stm ttm =
  if stm = ttm then add_assum (mk_eq s t) (axiom_eqrefl stm)
  else if stm = s andalso ttm = t then imp_refl (mk_eq s t) else
  case (stm,ttm) of
```

```
   (Fn(fs,sa),Fn(ft,ta)) =>
        if fs = ft andalso length sa = length ta then
            let val ths = map2 (icongruence s t) sa ta
                val ts = List.map (consequent o concl) ths in
            imp_trans_chain ths (axiom_funcong fs (List.map lhs ts) (List.map rhs ts))
            end
        else raise Fail "icongruence: not congruent"
  | _ => raise Fail "icongruence: not congruent";

(* ------------------------------------------------------------------------- *)
(* |- (forall x. p ==> q(x)) ==> p ==> (forall x. q(x))                      *)
(* ------------------------------------------------------------------------- *)

fun gen_right_th x p q =
  imp_swap(imp_trans (axiom_impall x p) (imp_swap(axiom_allimp x p q)));

(* ------------------------------------------------------------------------- *)
(*                       |- p ==> q                                         *)
(*         ------------------------------------ genimp "x"                  *)
(*           |- (forall x. p) ==> (forall x. q)                              *)
(* ------------------------------------------------------------------------- *)

fun genimp x th =
  let val (p,q) = dest_imp(concl th) in
  modusponens (axiom_allimp x p q) (gen x th)
  end;

(* ------------------------------------------------------------------------- *)
(* If |- p ==> q[x] then |- p ==> forall x. q[x]                             *)
(* ------------------------------------------------------------------------- *)

fun gen_right x th =
  let val (p,q) = dest_imp(concl th) in
  modusponens (gen_right_th x p q) (gen x th)
  end;

(* ------------------------------------------------------------------------- *)
(* |- (forall x. p(x) ==> q) ==> (exists x. p(x)) ==> q                      *)
(* ------------------------------------------------------------------------- *)

fun exists_left_th x p q =
  let val  p' = Imp(p,Falsity)
      val  q' = Imp(q,Falsity)
      val th1 = genimp x (imp_swap(imp_trans_th p q Falsity))
      val th2 = imp_trans th1 (gen_right_th x q' p')
      val th3 = imp_swap(imp_trans_th q' (Forall(x,p')) Falsity)
      val th4 = imp_trans2 (imp_trans th2 th3) (axiom_doubleneg q)
      val th5 = imp_add_concl Falsity (genimp x (iff_imp2 (axiom_not p)))
      val th6 = imp_trans (iff_imp1 (axiom_not (Forall(x,Not p)))) th5
      val th7 = imp_trans (iff_imp1(axiom_exists x p)) th6 in
  imp_swap(imp_trans th7 (imp_swap th4))
  end;

(* ------------------------------------------------------------------------- *)
(* If |- p(x) ==> q then |- (exists x. p(x)) ==> q                           *)
(* ------------------------------------------------------------------------- *)

fun exists_left x th =
  let val (p,q) = dest_imp(concl th) in
  modusponens (exists_left_th x p q) (gen x th)
  end;

(* ------------------------------------------------------------------------- *)
(*    |- x = t ==> p ==> q    [x not in t and not free in q]                 *)
```

```
(*  ------------------------------------------------------------- subspec  *)
(*                 |- (forall x. p) ==> q                                  *)
(* ---------------------------------------------------------------------- *)

fun subspec th =
  case concl th of
    Imp(e as Atom(Rl("=",[Var x,t])),Imp(p,q)) =>
        let val th1 = imp_trans (genimp x (imp_swap th))
                            (exists_left_th x e q) in
        modusponens (imp_swap th1) (axiom_existseq x t)
        end
  | _ => raise Fail "subspec: wrong sort of theorem";

(* ---------------------------------------------------------------------- *)
(*    |- x = y ==> p[x] ==> q[y]  [x not in FV(q); y not in FV(p) or x == y] *)
(*  ------------------------------------------------------- subalpha       *)
(*                 |- (forall x. p) ==> (forall y. q)                     *)
(* ---------------------------------------------------------------------- *)

fun subalpha th =
   case concl th of
    Imp(Atom(Rl("=",[Var x,Var y])),Imp(p,q)) =>
        if x = y then genimp x (modusponens th (axiom_eqrefl(Var x)))
        else gen_right y (subspec th)
  | _ => raise Fail "subalpha: wrong sort of theorem";

(* ---------------------------------------------------------------------- *)
(*         ---------------------------------- isubst                      *)
(*            |- s = t ==> p[s] ==> p[t]                                   *)
(* ---------------------------------------------------------------------- *)

fun isubst s t sfm tfm =
  if sfm = tfm then add_assum (mk_eq s t) (imp_refl tfm) else
  case (sfm,tfm) of
    (Atom(Rl(p,sa)),Atom(Rl(p',ta))) =>
        if p = p' andalso List.length sa = List.length ta then
            let val ths = map2 (icongruence s t) sa ta
                val (ls,rs) = unzip (List.map (dest_eq o consequent o concl) ths) in
            imp_trans_chain ths (axiom_predcong p ls rs)
            end
        else
            raise Fail "isubst"
  | (Imp(sp,sq),Imp(tp,tq)) =>
        let val th1 = imp_trans (eq_sym s t) (isubst t s tp sp)
            val th2 = isubst s t sq tq in
        imp_trans_chain [th1, th2] (imp_mono_th sp tp sq tq)
        end
  | (Forall(x,p),Forall(y,q)) =>
        if x = y then
          imp_trans (gen_right x (isubst s t p q)) (axiom_allimp x p q)
        else
          let val z = Var(variant x (unions_str [fv p, fv q, fvt s, fvt t]))
              val th1 = isubst (Var x) z p (subst (x |==> z) p)
              val th2 = isubst z (Var y) (subst (y |==> z) q) q
              val th3 = subalpha th1
              val th4 = subalpha th2
              val th5 = isubst s t (consequent(concl th3))
                                   (antecedent(concl th4)) in
          imp_swap (imp_trans2 (imp_trans th3 (imp_swap th5)) th4)
          end
  | _ =>
        let val sth = iff_imp1(expand_connective sfm)
            val tth = iff_imp2(expand_connective tfm)
            val th1 = isubst s t (consequent(concl sth))
```

```
                            (antecedent(concl tth)) in
        imp_swap(imp_trans sth (imp_swap(imp_trans2 th1 tth)))
        end;

(* ------------------------------------------------------------------------- *)
(*                                                                           *)
(* ---------------------------------------- alpha "z" <!forall x. p[x]!> *)
(*   |- (forall x. p[x]) ==> (forall z. p'[z])                               *)
(*                                                                           *)
(* [Restriction that z is not free in the initial p[x].]                     *)
(* ------------------------------------------------------------------------- *)

fun alpha z fm =
  case fm of
    Forall(x,p) => let val p' = subst (x |==> Var z) p in
                   subalpha(isubst (Var x) (Var z) p p')
                   end
  | _ => raise Fail "alpha: not a universal formula";

(* ------------------------------------------------------------------------- *)
(*                                                                           *)
(* ------------------------------- ispec t <!forall x. p[x]!>                *)
(*   |- (forall x. p[x]) ==> p'[t]                                           *)
(* ------------------------------------------------------------------------- *)

fun ispec t fm =
  case fm of
    Forall(x,p) =>
      if mem x (fvt t) then
        let val th = alpha (variant x (union_str (fvt t) (var p))) fm in
        imp_trans th (ispec t (consequent(concl th)))
        end
      else subspec(isubst (Var x) t p (subst (x |==> t) p))
  | _ => raise Fail "ispec: non-universal formula";

(* ------------------------------------------------------------------------- *)
(* Specialization rule.                                                      *)
(* ------------------------------------------------------------------------- *)

fun spec t th = modusponens (ispec t (concl th)) th;

>
```

ML `<"lcffol.sml";`

```
(* ========================================================================= *)
(* First order tableau procedure using LCF setup.                            *)
(* ========================================================================= *)

(* ------------------------------------------------------------------------- *)
(* Unification of complementary literals.                                    *)
(* ------------------------------------------------------------------------- *)

fun unify_complementsf env =
  fn (Atom(Rl(p1,a1)),Imp(Atom(Rl(p2,a2)),Falsity)) => unify env [(Fn(p1,a1),Fn(p2,a2))]
   | (Imp(Atom(Rl(p1,a1)),Falsity),Atom(Rl(p2,a2))) => unify env [(Fn(p1,a1),Fn(p2,a2))]
   | _ => raise Fail "unify_complementsf";

(* ------------------------------------------------------------------------- *)
(*    |- (q ==> f) ==> ... ==> (q ==> p) ==> r                               *)
(* ------------------------------------------ use_laterimp <!q ==> p!>        *)
(*    |- (p ==> f) ==> ... ==> (q ==> p) ==> r                               *)
(* ------------------------------------------------------------------------- *)
```

```
fun use_laterimp i fm =
  case fm of
    Imp(_,Imp(i',_)) =>
      ( case (fm,i'=i) of
          (Imp(Imp(q',s),Imp(i' as Imp(q,p),r)),true) =>
             let val th1 = axiom_distribimp i (Imp(Imp(q,s),r)) (Imp(Imp(p,s),r))
                 val th2 = imp_swap(imp_trans_th q p s)
                 val th3 = imp_swap(imp_trans_th (Imp(p,s)) (Imp(q,s)) r) in
             imp_swap2(modusponens th1 (imp_trans th2 th3))
             end
        | (Imp(qs,Imp(a,b)),_) =>
             imp_swap2(imp_add_assum a (use_laterimp i (Imp(qs,b))))
      )
;

(* ------------------------------------------------------------------------- *)
(* The "closure" inference rules.                                            *)
(* ------------------------------------------------------------------------- *)

fun imp_false_rule' th es = imp_false_rule(th es);

fun imp_true_rule' th1 th2 es = imp_true_rule (th1 es) (th2 es);

fun imp_front' n thp es = imp_front n (thp es);

fun add_assum' fm thp (es as(e,s)) =
  add_assum (onformula e fm) (thp es);

fun eliminate_connective' fm thp (es as(e,s)) =
  imp_trans (eliminate_connective (onformula e fm)) (thp es);

fun spec' y fm n thp (e,s) =
  let val th = imp_swap(imp_front n (thp(e,s))) in
  imp_unduplicate(imp_trans (ispec (e y) (onformula e fm)) th)
  end;

fun ex_falso' fms (e,s) =
  ex_falso (itlist (mk_imp o onformula e) fms s);

fun complits' (p::fl,lits) i (e,s) =
  let val (l1,p'::l2) = chop_list i lits in
  itlist (imp_insert o onformula e) (fl @ l1)
         (imp_contr (onformula e p)
                    (itlist (mk_imp o onformula e) l2 s))
  end;

fun deskol' (skh:fol fm) thp (e,s) =
  let val th = thp (e,s) in
  modusponens (use_laterimp (onformula e skh) (concl th)) th
  end;

(* ------------------------------------------------------------------------- *)
(* Main refutation function.                                                 *)
(* ------------------------------------------------------------------------- *)

fun is_lit (Atom(_)) = true
  | is_lit (Imp(Atom(_),Falsity)) = true
  | is_lit _ = false;

fun is_uni (Forall(_,_)) = true
  | is_uni _ = false;

fun dest_uni (Forall(x,p)) = (x,p);
```

```
fun is_exi (Imp(Forall(_,_),Falsity)) = true
  | is_exi _ = false;

fun dest_exi (Imp(yp as Forall(y,p),Falsity)) = (y,p,yp);

fun lcftab skofun (fms,lits,n) cont (esk as (env,sks,k)) =
    if n < 0 then raise Fail "lcftab: no proof" else
    case fms of
      []     => raise Fail "lcftab: No contradiction"
    | fm::fl =>
        if is_false fm then (
            cont (ex_falso' (fl @ lits)) esk
        ) else if is_true fm then (
            lcftab skofun (fl,lits,n) (cont o add_assum' fm) esk
        ) else if is_conj fm then (
            let val (p,q)=dest_conj fm in
            lcftab skofun (p::Imp(q,Falsity)::fl,lits,n) (cont o imp_false_rule') esk
            end
        ) else if is_disj fm then (
            let val (p,q)=dest_disj fm in
            lcftab skofun (Imp(p,Falsity)::fl,lits,n) (fn th => lcftab skofun (q::fl,lits,n)
                                                                (cont o imp_true_rule' th)) esk
            end
        ) else if is_lit fm then (
            (tryfind (fn p' => (
                let val env' = unify_complementsf env (fm, p') in
                cont (complits' (fms, lits) (index p' lits)) (env', sks, k)
                end)) lits)
            handle Fail _ => (
                lcftab skofun (fl,fm::lits,n) (cont o imp_front' (List.length fl)) esk
            )
        ) else if is_uni fm then (
            let val (x,p) = dest_uni fm
                val y = Var("X_"^(Int.toString k)) in
            lcftab skofun ((subst (x |==> y) p)::fl@[fm],lits,n-1)
                    (cont o spec' y fm (List.length fms)) (env,sks,k+1)
            end
        ) else if is_exi fm then (
            let val (y,p,yp) = dest_exi fm
                val fx = skofun yp
                val p' = subst(y |==> fx) p
                val skh = Imp(p',Forall(y,p))
                val sks' = (Forall(y,p),fx)::sks in
            lcftab skofun (Imp(p',Falsity)::fl,lits,n) (cont o deskol' skh) (env,sks',k)
            end
        ) else ( (* is nonprimitive *)
           let val fm' = consequent(concl(eliminate_connective fm)) in
           lcftab skofun (fm'::fl,lits,n) (cont o eliminate_connective' fm) esk
           end
        )
;

(* ------------------------------------------------------------------------- *)
(* Identify quantified subformulas; true = exists, false = forall. This is   *)
(* taking into account the effective parity.                                 *)
(* ------------------------------------------------------------------------- *)

fun quantforms e fm =
  case fm of
    Not(p) => quantforms (not e) p
  | And(p,q) => union_folfm (quantforms e p) (quantforms e q)
  | Or(p,q)  => union_folfm (quantforms e p) (quantforms e q)
  | Imp(p,q) => quantforms e (Or(Not p,q))
  | Iff(p,q) => quantforms e (Or(And(p,q),And(Not p,Not q)))
```

```
  | Exists(x,p) => if e then fm::(quantforms e p) else quantforms e p
  | Forall(x,p) => if e then quantforms e p else fm::(quantforms e p)
  | _ => [];

(* ------------------------------------------------------------------------- *)
(* Now create some Skolem functions.                                         *)
(* ------------------------------------------------------------------------- *)

fun skolemfuns fm =
  let val fns = List.map (fn pr => fst pr) (functions fm)
      val skts = List.map (fn Exists(x,p) => Forall(x,Not p) | p => p)
                  (quantforms true fm)
      fun skofun i (ap as Forall(y,p)) =
            let val vars = List.map (fn v => Var v) (fv ap) in
            (ap,Fn(variant("f"^"_"^Int.toString i) fns,vars))
            end
  in
  map2 skofun (1--length skts) skts
  end;

(* ------------------------------------------------------------------------- *)
(* Matching.                                                                 *)
(* ------------------------------------------------------------------------- *)

fun form_match (fp as (f1,f2)) env =
  case fp of
    (Falsity,Falsity) => env
  | (Truth,Truth)     => env
  | (Atom(Rl(p,pa)),Atom(Rl(q,qa))) => term_match env [(Fn(p,pa),Fn(q,qa))]
  | (Not(p1),Not(p2)) => form_match (p1,p2) env
  | (And(p1,q1),And(p2,q2)) => form_match (p1,p2) (form_match (q1,q2) env)
  | (Or(p1,q1),Or(p2,q2))   => form_match (p1,p2) (form_match (q1,q2) env)
  | (Imp(p1,q1),Imp(p2,q2)) => form_match (p1,p2) (form_match (q1,q2) env)
  | (Iff(p1,q1),Iff(p2,q2)) => form_match (p1,p2) (form_match (q1,q2) env)
  | (Forall(x1,p1),Forall(x2,p2)) =>
        if (x1=x2) then
          let val z = variant x1 (union_str (fv p1) (fv p2))
              val inst_fn = subst (x1 |==> Var z) in
          undefine_str z (form_match (inst_fn p1,inst_fn p2) env)
          end
        else
          raise Fail "form_match"
  | (Exists(x1,p1),Exists(x2,p2)) =>
        if (x1=x2) then
          let val z = variant x1 (union_str (fv p1) (fv p2))
              val inst_fn = subst (x1 |==> Var z) in
          undefine_str z (form_match (inst_fn p1,inst_fn p2) env)
          end
        else
          raise Fail "form_match"
  | _ => raise Fail "form_match";

(* ------------------------------------------------------------------------- *)
(* With the current approach to picking Skolem functions.                    *)
(* ------------------------------------------------------------------------- *)

fun lcfrefute fm n cont =
  let val sl = skolemfuns fm
      fun find_skolem fm =
           tryfind(fn (f,t) => tsubst(form_match (f,fm) undefined) t) sl
  in
  lcftab find_skolem ([fm],[],n) cont (undefined,[],0)
  end;
```

```
fun mk_skol (Forall(y,p),fx) q =
  Imp(Imp(subst (y |==> fx) p,Forall(y,p)),q);

fun simpcont thp (env,sks,k) =
  let val ifn = tsubst(solve env) in
  thp(ifn,onformula ifn (itlist mk_skol sks Falsity))
  end;

(* ------------------------------------------------------------------------- *)
(*         |- (p(v) ==> forall x. p(x)) ==> q                                *)
(*       ------------------------------------ elim_skolemvar                 *)
(*                   |- q                                                    *)
(* ------------------------------------------------------------------------- *)

fun elim_skolemvar th =
  case concl th of
    Imp(Imp(pv,(apx as Forall(x,px))),q) =>
        let val [th1,th2] = List.map (imp_trans(imp_add_concl Falsity th))
                             (imp_false_conseqs pv apx)
            val v = hd(subtract_str (fv pv) (fv apx) @ [x])
            val th3 = gen_right v th1
            val th4 = imp_trans th3 (alpha x (consequent(concl th3))) in
        modusponens (axiom_doubleneg q) (right_mp th2 th4)
        end
  | _ => raise Fail "elim_skolemvar";

(* ------------------------------------------------------------------------- *)
(* Top continuation with careful sorting and variable replacement.           *)
(* Also need to delete post-instantiation duplicates! This shows up more     *)
(* often now that we have adequate sharing.                                  *)
(* ------------------------------------------------------------------------- *)

fun deskolcont thp (env,sks,k) =
  let val ifn = tsubst(solve env)
      val isk = setify_ftp(List.map (fn (p,t) => (onformula ifn p,ifn t)) sks)
      val ssk = sort (decreasing (termsize o snd)) isk
      val vs  = List.map (fn i => Var("Y_"^Int.toString i)) (1--List.length ssk)
      val vfn = replacet(itlist2 (fn (p,t) => fn v => t |---> v) ssk vs undefined)
      val th  = thp(vfn o ifn,onformula vfn (itlist mk_skol ssk Falsity)) in
  repeat (elim_skolemvar o imp_swap) th
  end;

(* ------------------------------------------------------------------------- *)
(* Overall first-order prover.                                               *)
(* ------------------------------------------------------------------------- *)

fun lcffol fm =
  let val fvs = fv fm
      val fm' = Imp(itlist mk_forall fvs fm,Falsity)
      val th1 = deepen (fn n => lcfrefute fm' n deskolcont) 0
      val th2 = modusponens (axiom_doubleneg (negatef fm')) th1 in
  itlist (fn v => spec(Var v)) (rev fvs) th2
  end;

>
```

ML `<"tactics.sml";`

```
(* ========================================================================= *)
(* Goals, LCF-like tactics and Mizar-like proofs.                            *)
(* ========================================================================= *)

datatype goals =
  Goals of ((string * fol fm) list * fol fm)list *
```

```
          (thm list -> thm);

(* ------------------------------------------------------------------------- *)
(* Printer for goals (just shows first goal plus total number).              *)
(* ------------------------------------------------------------------------- *)

val print_goal_aux =
    let fun print_hyp (l, fm) = (
        open_hbox();
        print_string (l^":");
        print_space ();
        print_formula_aux print_atom_aux fm;
        print_newline();
        close_box()
    ) in
    fn (Goals (gls, jfn)) =>
        case gls of
          [] =>
            print_string "No subgoals"
        | (asl, w) :: ogls =>(
            print_newline ();
            if ogls = [] then
                print_string "1 subgoal:"
            else (
                print_int (List.length gls);
                print_string " subgoals starting with"
            )
            ;
            print_newline();
            List.app print_hyp (List.rev asl);
            print_string "---> ";
            open_hvbox 0; print_formula_aux print_atom_aux w; close_box();
            print_newline ()
          )
    end;

fun print_goal g = (print_goal_aux g; print_flush ());

(* ------------------------------------------------------------------------- *)
(* Setting up goals and terminating them in a theorem.                       *)
(* ------------------------------------------------------------------------- *)

fun set_goal p =
  let fun chk th = if concl th = p then th else raise Fail "wrong theorem" in
  Goals([([],p)],fn [th] => chk(modusponens th truth))
  end;

fun extract_thm gls =
  case gls of
    Goals([],jfn) => jfn []
  | _ => raise Fail "extract_thm: unsolved goals";

fun tac_proof g prf = extract_thm(itlist (fn f => f) (List.rev prf) g);

fun prove p prf = tac_proof (set_goal p) prf;

(* ------------------------------------------------------------------------- *)
(* Conjunction introduction tactic.                                          *)
(* ------------------------------------------------------------------------- *)

fun conj_intro_tac (Goals((asl,And(p,q))::gls,jfn)) =
  let fun jfn' (thp::thq::ths) =
    jfn(imp_trans_chain [thp, thq] (and_pair p q)::ths) in
  Goals((asl,p)::(asl,q)::gls,jfn')
```

```
  end;

(* ------------------------------------------------------------------------- *)
(* Handy idiom for tactic that does not split subgoals.                      *)
(* ------------------------------------------------------------------------- *)

fun jmodify jfn tfn (th::oths) = jfn(tfn th :: oths);

(* ------------------------------------------------------------------------- *)
(* Version of gen_right with a bound variable change.                        *)
(* ------------------------------------------------------------------------- *)

fun gen_right_alpha y x th =
  let val th1 = gen_right y th in
  imp_trans th1 (alpha x (consequent(concl th1)))
  end;

(* ------------------------------------------------------------------------- *)
(* Universal introduction.                                                   *)
(* ------------------------------------------------------------------------- *)

fun forall_intro_tac y (Goals((asl,(fm as Forall(x,p)))::gls,jfn)) =
  if mem y (fv fm) orelse List.exists (mem y o fv o snd) asl
  then raise Fail "fix: variable already free in goal" else
  Goals((asl,subst(x |==> Var y) p)::gls,
        jmodify jfn (gen_right_alpha y x));

(* ------------------------------------------------------------------------- *)
(* Another inference rule: |- P[t] ==> exists x. P[x]                        *)
(* ------------------------------------------------------------------------- *)

fun right_exists x t p =
  let val th = contrapos(ispec t (Forall(x,Not p)))
      val Not(Not p') = antecedent(concl th) in
  end_itlist imp_trans
   [imp_contr p' Falsity, imp_add_concl Falsity (iff_imp1 (axiom_not p')),
    iff_imp2(axiom_not (Not p')), th, iff_imp2(axiom_exists x p)]
  end;

(* ------------------------------------------------------------------------- *)
(* Existential introduction.                                                 *)
(* ------------------------------------------------------------------------- *)

fun exists_intro_tac t (Goals((asl,Exists(x,p))::gls,jfn)) =
  Goals((asl,subst(x |==> t) p)::gls,
        jmodify jfn (fn th => imp_trans th (right_exists x t p))) ;

(* ------------------------------------------------------------------------- *)
(* Implication introduction tactic.                                          *)
(* ------------------------------------------------------------------------- *)

fun imp_intro_tac s (Goals((asl,Imp(p,q))::gls,jfn)) =
  let val jmod = if asl = [] then add_assum Truth else imp_swap o shunt in
  Goals(((s,p)::asl,q)::gls,jmodify jfn jmod)
  end;

(* ------------------------------------------------------------------------- *)
(* Append contextual hypothesis to unconditional theorem.                    *)
(* ------------------------------------------------------------------------- *)

fun assumptate (Goals((asl,w)::gls,jfn)) th =
  add_assum (list_conj (map snd asl)) th;

(* ------------------------------------------------------------------------- *)
```

```
(* Get the first assumption (quicker than head of assumps result).           *)
(* ------------------------------------------------------------------------- *)

fun firstassum asl =
  let val p = snd(hd asl)
      val q = list_conj(List.map snd (List.tl asl)) in
  if tl asl = [] then imp_refl p else and_left p q
  end;

(* ------------------------------------------------------------------------- *)
(* Import "external" theorem.                                                *)
(* ------------------------------------------------------------------------- *)

fun using ths p g =
  let val ths' = map (fn th => itlist gen (fv(concl th)) th) ths in
  List.map (assumptate g) ths'
  end;

(* ------------------------------------------------------------------------- *)
(* Turn assumptions p1,...,pn into theorems |- p1 /\ ... /\ pn ==> pi        *)
(* ------------------------------------------------------------------------- *)

fun assumps asl =
  case asl of
    [] => []
  | [(l,p)] => [(l,imp_refl p)]
  | (l,p)::lps =>
        let val ths = assumps lps
            val q = antecedent(concl(snd(List.hd ths)))
            val rth = and_right p q in
        (l,and_left p q)::List.map (fn (l,th) => (l,imp_trans rth th)) ths
        end;

(* ------------------------------------------------------------------------- *)
(* Produce canonical theorem from list of theorems or assumption labels.     *)
(* ------------------------------------------------------------------------- *)

fun by hyps p (Goals((asl,w)::gls,jfn)) =
  let val ths = assumps asl in
  List.map (fn s => assoc s ths) hyps
  end;

(* ------------------------------------------------------------------------- *)
(* Main automatic justification step.                                        *)
(* ------------------------------------------------------------------------- *)

local
  fun singleton [_] = true
    | singleton _ = false
in
  fun justify byfn hyps p g =
    let val ths = byfn hyps p g in
    if singleton ths andalso consequent(concl (List.hd ths)) = p then (
      List.hd ths
    ) else (
      let val th = lcffol(itlist (mk_imp o consequent o concl) ths p) in
      case ths of
        [] => assumptate g th
      | _  => imp_trans_chain ths th
      end
    )
    end
end;
```

```
(* ------------------------------------------------------------------------- *)
(* Nested subproof.                                                          *)
(* ------------------------------------------------------------------------- *)

fun proof tacs p (Goals((asl,w)::gls,jfn)) =
  [tac_proof (Goals([(asl,p)],fn [th] => th)) tacs];

(* ------------------------------------------------------------------------- *)
(* Trivial justification, producing no hypotheses.                           *)
(* ------------------------------------------------------------------------- *)

fun at once p gl = [];
val once = [];

(* ------------------------------------------------------------------------- *)
(* Hence an automated terminal tactic.                                       *)
(* ------------------------------------------------------------------------- *)

fun auto_tac byfn hyps (g as Goals((asl,w)::gls,jfn)) =
  let val th = justify byfn hyps w g in
  Goals(gls,fn ths => jfn(th::ths))
  end;

(* ------------------------------------------------------------------------- *)
(* A "lemma" tactic.                                                         *)
(* ------------------------------------------------------------------------- *)

fun lemma_tac s p byfn hyps (g as Goals((asl,w)::gls,jfn)) =
  let val tr = imp_trans(justify byfn hyps p g)
      val mfn = if asl = [] then tr else imp_unduplicate o tr o shunt in
  Goals(((s,p)::asl,w)::gls,jmodify jfn mfn)
  end;

(* ------------------------------------------------------------------------- *)
(* Elimination tactic for existential quantification.                        *)
(* ------------------------------------------------------------------------- *)

fun exists_elim_tac l fm byfn hyps (g as Goals((asl,w)::gls,jfn)) =
  let val Exists(x,p) = fm in
  if List.exists (mem x o fv) (w::List.map snd asl)
  then raise Fail "exists_elim_tac: variable free in assumptions" else
  let val th = justify byfn hyps (Exists(x,p)) g
      fun jfn' pth = imp_unduplicate(imp_trans th (exists_left x (shunt pth)))
  in
  Goals(((l,p)::asl,w)::gls,jmodify jfn jfn')
  end end;

(* ------------------------------------------------------------------------- *)
(* If |- p ==> r and |- q ==> r then |- p \/ q ==> r                         *)
(* ------------------------------------------------------------------------- *)

fun ante_disj th1 th2 =
  let val (p,r) = dest_imp(concl th1)
      val (q,s) = dest_imp(concl th2)
      val ths = map contrapos [th1, th2]
      val th3 = imp_trans_chain ths (and_pair (Not p) (Not q))
      val th4 = contrapos(imp_trans (iff_imp2(axiom_not r)) th3)
      val th5 = imp_trans (iff_imp1(axiom_or p q)) th4 in
  right_doubleneg(imp_trans th5 (iff_imp1(axiom_not(Imp(r,Falsity)))))
  end;

(* ------------------------------------------------------------------------- *)
(* Elimination tactic for disjunction.                                       *)
(* ------------------------------------------------------------------------- *)
```

```
fun disj_elim_tac l fm byfn hyps (g as Goals((asl,w)::gls,jfn) ) =
  let val th = justify byfn hyps fm g
      val Or(p,q) = fm
      fun jfn' (pth::qth::ths) =
         let val th1 = imp_trans th (ante_disj (shunt pth) (shunt qth)) in
         jfn(imp_unduplicate th1::ths)
         end
  in
  Goals(((l,p)::asl,w)::((l,q)::asl,w)::gls,jfn')
  end;

(* ------------------------------------------------------------------------- *)
(* Declarative proof.                                                        *)
(* ------------------------------------------------------------------------- *)

fun multishunt i th =
  let val th1 = imp_swap(funpow i (imp_swap o shunt) th) in
  imp_swap(funpow (i-1) (unshunt o imp_front 2) th1)
  end;

fun assume lps (Goals((asl,Imp(p,q))::gls,jfn)) =
  if end_itlist mk_and (map snd lps) <> p then raise Fail "assume" else
  let fun jfn' th =
    if asl = [] then add_assum Truth th else multishunt (length lps) th in
  Goals((lps@asl,q)::gls,jmodify jfn jfn')
  end;

fun note (l,p) = lemma_tac l p;

fun have p = note("",p);

fun so tac arg byfn =
  tac arg (fn hyps => fn p => fn (gl as Goals((asl,w)::_,_)) =>
                      firstassum asl :: byfn hyps p gl);

val fix = forall_intro_tac;

fun consider (x,p) = exists_elim_tac "" (Exists(x,p));

fun take tm gls = exists_intro_tac tm gls;

fun cases fm byfn hyps g = disj_elim_tac "" fm byfn hyps g;

(* ------------------------------------------------------------------------- *)
(* Thesis modification.                                                      *)
(* ------------------------------------------------------------------------- *)

fun conclude p byfn hyps (gl as Goals((asl,w)::gls,jfn)) =
  let val th = justify byfn hyps p gl in
  if p = w then Goals((asl,Truth)::gls,jmodify jfn (fn _ => th)) else
  let val (p',q) = dest_and w in
  if p' <> p then raise Fail "conclude: bad conclusion" else
  let fun mfn th' = imp_trans_chain [th, th'] (and_pair p q) in
  Goals((asl,q)::gls,jmodify jfn mfn)
  end end end;

(* ------------------------------------------------------------------------- *)
(* A useful shorthand for solving the whole goal.                            *)
(* ------------------------------------------------------------------------- *)

fun our thesis byfn hyps (gl as Goals((asl,w)::gls,jfn)) =
  conclude w byfn hyps gl;
val thesis = "";
```

```
(* ------------------------------------------------------------------------- *)
(* Termination.                                                              *)
(* ------------------------------------------------------------------------- *)

fun qed (gl as Goals((asl,w)::gls,jfn)) =
  if w = Truth then Goals(gls,fn ths => jfn(assumptate gl truth :: ths))
  else raise Fail "qed: non-trivial goal";

(* ------------------------------------------------------------------------- *)
(* A simple example.                                                         *)
(* ------------------------------------------------------------------------- *)

val ewd954 = prove
 (<!("(forall x y. x <= y <=> x * y = x) /\\ " ^
   "(forall x y. f(x * y) = f(x) * f(y)) " ^
   "==> forall x y. x <= y ==> f(x) <= f(y)")!>)
 [note("eq_sym",<!"forall x y. x = y ==> y = x"!>)
    using [eq_sym (<!|"x"|!>) (<!|"y"|!>)],
  note("eq_trans",<!"forall x y z. x = y /\\ y = z ==> x = z"!>)
    using [eq_trans (<!|"x"|!>) (<!|"y"|!>) (<!|"z"|!>)],
  note("eq_cong",<!"forall x y. x = y ==> f(x) = f(y)"!>)
    using [axiom_funcong "f" [(<!|"x"|!>)] [(<!|"y"|!>)]],
  assume [("le",<!"forall x y. x <= y <=> x * y = x"!>),
          ("hom",<!"forall x y. f(x * y) = f(x) * f(y)"!>)],
  fix "x", fix "y",
  assume [("xy",<!"x <= y"!>)],
  so have (<!"x * y = x"!>) by ["le"],
  so have (<!"f(x * y) = f(x)"!>) by ["eq_cong"],
  so have (<!"f(x) = f(x * y)"!>) by ["eq_sym"],
  so have (<!"f(x) = f(x) * f(y)"!>) by ["eq_trans", "hom"],
  so have (<!"f(x) * f(y) = f(x)"!>) by ["eq_sym"],
  so conclude (<!"f(x) <= f(y)"!>) by ["le"],
  qed];

(* ------------------------------------------------------------------------- *)
(* More examples not in the main text.                                       *)
(* ------------------------------------------------------------------------- *)

prove
 (<!("(exists x. p(x)) ==> (forall x. p(x) ==> p(f(x))) " ^
   "==> exists y. p(f(f(f(f(y)))))")!>)
  [assume [("A",<!"exists x. p(x)"!>)],
   assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
   note ("C",<!"forall x. p(x) ==> p(f(f(f(f(x)))))"!>)
   proof
    [have (<!"forall x. p(x) ==> p(f(f(x)))"!>) by ["B"],
     so conclude (<!"forall x. p(x) ==> p(f(f(f(f(x)))))"!>) at once,
     qed],
   consider ("a",<!"p(a)"!>) by ["A"],
   take (<!|"a"|!>),
   so conclude (<!"p(f(f(f(f(a)))))"!>) by ["C"],
   qed];

(* ------------------------------------------------------------------------- *)
(* Alternative formulation with lemma construct.                             *)
(* ------------------------------------------------------------------------- *)

let fun lemma (s,p) (gl as Goals((asl,w)::gls,jfn)) =
  Goals((asl,p)::((s,p)::asl,w)::gls,
        fn (thp::thw::oths) =>
            jfn(imp_unduplicate(imp_trans thp (shunt thw)) :: oths)) in
prove
 (<!("(exists x. p(x)) ==> (forall x. p(x) ==> p(f(x))) " ^
```

```
   "==> exists y. p(f(f(f(f(y)))))")!>)
  [assume [("A",<!"exists x. p(x)"!>)],
   assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
   lemma ("C",<!"forall x. p(x) ==> p(f(f(f(f(x)))))"!>),
     have (<!"forall x. p(x) ==> p(f(f(x)))"!>) by ["B"],
     so conclude (<!"forall x. p(x) ==> p(f(f(f(f(x)))))"!>) at once,
     qed,
   consider ("a",<!"p(a)"!>) by ["A"],
   take (<!|"a"|!>),
   so conclude (<!"p(f(f(f(f(a)))))"!>) by ["C"],
   qed]
end;

(* ------------------------------------------------------------------------ *)
(* Examples.                                                                *)
(* ------------------------------------------------------------------------ *)

prove (<!("p(a) ==> (forall x. p(x) ==> p(f(x))) " ^
        "==> exists y. p(y) /\\ p(f(y))")!>)
      [our thesis at once,
       qed];

prove
 (<!("(exists x. p(x)) ==> (forall x. p(x) ==> p(f(x))) " ^
   "==> exists y. p(f(f(f(y)))))")!>)
  [assume [("A",<!"exists x. p(x)"!>)],
   assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
   note ("C",<!"forall x. p(x) ==> p(f(f(f(f(x)))))"!>) proof
    [have (<!"forall x. p(x) ==> p(f(f(x)))"!>) by ["B"],
     so our thesis at once,
     qed],
   consider ("a",<!"p(a)"!>) by ["A"],
   take (<!|"a"|!>),
   so our thesis by ["C"],
   qed];

prove (<!("forall a. p(a) ==> (forall x. p(x) ==> p(f(x))) " ^
                  "==> exists y. p(y) /\\ p(f(y))")!>)
      [fix "c",
       assume [("A",<!"p(c)"!>)],
       assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
       take (<!|"c"|!>),
       conclude (<!"p(c)"!>) by ["A"],
       note ("C",<!"p(c) ==> p(f(c))"!>) by ["B"],
       so our thesis by ["C", "A"],
       qed];

prove (<!("p(c) ==> (forall x. p(x) ==> p(f(x))) " ^
                  "==> exists y. p(y) /\\ p(f(y))")!>)
      [assume [("A",<!"p(c)"!>)],
       assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
       take (<!|"c"|!>),
       conclude (<!"p(c)"!>) by ["A"],
       our thesis by ["A", "B"],
       qed];

prove (<!("forall a. p(a) ==> (forall x. p(x) ==> p(f(x))) " ^
                  "==> exists y. p(y) /\\ p(f(y))")!>)
      [fix "c",
       assume [("A",<!"p(c)"!>)],
       assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
       take (<!|"c"|!>),
       conclude (<!"p(c)"!>) by ["A"],
       note ("C",<!"p(c) ==> p(f(c))"!>) by ["B"],
```

```
       our thesis by ["C", "A"],
       qed];

prove (<!("forall a. p(a) ==> (forall x. p(x) ==> p(f(x))) " ^
                 "==> exists y. p(y) /\\ p(f(y))")!>)
      [fix "c",
       assume [("A",<!"p(c)"!>)],
       assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
       take (<!|"c"|!>),
       note ("D",<!"p(c)"!>) by ["A"],
       note ("C",<!"p(c) ==> p(f(c))"!>) by ["B"],
       our thesis by ["C", "A", "D"],
       qed];

prove (<!"(p(a) \\/ p(b)) ==> q ==> exists y. p(y)"!>)
  [assume [("A",<!"p(a) \\/ p(b)"!>)],
   assume [("",<!"q"!>)],
   cases (<!"p(a) \\/ p(b)"!>) by ["A"],
     take (<!|"a"|!>),
     so our thesis at once,
     qed,

     take (<!|"b"|!>),
     so our thesis at once,
     qed];

prove
  (<!"(p(a) \\/ p(b)) /\\ (forall x. p(x) ==> p(f(x))) ==> exists y. p(f(y))"!>)
  [assume [("base",<!"p(a) \\/ p(b)"!>),
           ("Step",<!"forall x. p(x) ==> p(f(x))"!>)],
   cases (<!"p(a) \\/ p(b)"!>) by ["base"],
     so note("A",<!"p(a)"!>) at once,
     note ("X",<!"p(a) ==> p(f(a))"!>) by ["Step"],
     take (<!|"a"|!>),
     our thesis by ["A", "X"],
     qed,

     take (<!|"b"|!>),
     so our thesis by ["Step"],
     qed];

prove
 (<!"(exists x. p(x)) ==> (forall x. p(x) ==> p(f(x))) ==> exists y. p(f(y))"!>)
  [assume [("A",<!"exists x. p(x)"!>)],
   assume [("B",<!"forall x. p(x) ==> p(f(x))"!>)],
   consider ("a",<!"p(a)"!>) by ["A"],
   so note ("concl",<!"p(f(a))"!>) by ["B"],
   take (<!|"a"|!>),
   our thesis by ["concl"],
   qed];

prove (<!("(forall x. p(x) ==> q(x)) ==> (forall x. q(x) ==> p(x)) " ^
       "==> (p(a) <=> q(a))")!>)
  [assume [("A",<!"forall x. p(x) ==> q(x)"!>)],
   assume [("B",<!"forall x. q(x) ==> p(x)"!>)],
   note ("von",<!"p(a) ==> q(a)"!>) by ["A"],
   note ("bis",<!"q(a) ==> p(a)"!>) by ["B"],
   our thesis by ["von", "bis"],
   qed];

>
```

Main Examples

ML_val ‹(* Hoare's Exercise ewd1062_1 & ewd1062_2 (Harrison has only a proof with tactics) *)

```
prove
  (<!"(forall x. x <= x) /\\ " ^
      "(forall x y z. x <= y /\\ y <= z ==> x <= z) /\\ " ^
      "(forall x y. f(x) <= y <=> x <= g(y)) " ^
      "==> (forall x y. x <= y ==> f(x) <= f(y)) /\\ " ^
          "(forall x y. x <= y ==> g(x) <= g(y))")!>)
  [
    assume [("A", <!"(forall x. x <= x) /\\ " ^
              "(forall x y z. x <= y /\\ y <= z ==> x <= z) /\\ " ^
              "(forall x y. f(x) <= y <=> x <= g(y))")!>)],
    conclude (<!"(forall x y. x <= y ==> f(x) <= f(y)) /\\ " ^
                "(forall x y. x <= y ==> g(x) <= g(y))")!>) proof
    [
      conclude (<!"(forall x y. x <= y ==> f(x) <= f(y))"!>) by ["A"],
      conclude (<!"(forall x y. x <= y ==> g(x) <= g(y))"!>) by ["A"],
      qed
    ],
    qed
  ]

›
```

ML_val ‹(* Pelletier p43 (Harrison has it in a comment but the proof seems not to finish) *)

```
prove
  (<!"(forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)) ==> forall x y. Q(x,y) <=> Q(y,x)"!>)
  [
    assume [("A", <!"forall x y. Q(x,y) <=> forall z. P(z,x) <=> P(z,y)"!>)],
    conclude (<!"forall x y. Q(x,y) <=> Q(y,x)"!>) proof
    [
      fix "x", fix "y",
      conclude (<!"Q(x,y) <=> Q(y,x)"!>) proof
      [
        have (<!"(Q(x,y) ==> Q(y,x)) /\\ (Q(y,x) ==> Q(x,y))"!>) proof
        [
          conclude (<!"Q(x,y) ==> Q(y,x)"!>) proof
          [
            assume [("", <!"Q(x,y)"!>)],
            so have (<!"forall z. P(z,x) <=> P(z,y)"!>) by ["A"],
            so have (<!"forall z. P(z,y) <=> P(z,x)"!>) at once,
            so conclude (<!"Q(y,x)"!>) by ["A"],
            qed
          ],
          conclude (<!"Q(y,x) ==> Q(x,y)"!>) proof
          [
            assume [("", <!"Q(y,x)"!>)],
            so have (<!"forall z. P(z,y) <=> P(z,x)"!>) by ["A"],
            so have (<!"forall z. P(z,x) <=> P(z,y)"!>) at once,
            so conclude (<!"Q(x,y)"!>) by ["A"],
            qed
          ],
          qed
        ],
        so our thesis at once,
        qed
      ],
      qed
    ],
    qed
  ]
```

```
>
```

ML_val `<(* Pelletier p46 (Harrison does not have it) *)`

```
  prove
    (<!("(forall x. P(x) /\\ (forall y. P(y) /\\ H(y,x) ==> G(y)) ==> G(x)) /\\ " ^
          "((exists x. P(x) /\\ ~G(x)) ==> " ^
            "(exists x. P(x) /\\ ~G(x) /\\ (forall y. P(y) /\\ ~G(y) ==> J(x,y)))) /\\ " ^
          "(forall x y. P(x) /\\ P(y) /\\ H(x,y) ==> ~J(y,x)) ==> " ^
          "(forall x. P(x) ==> G(x))")!>)
    [
      assume [("A", <!("(forall x. P(x) /\\ (forall y. P(y) /\\ H(y,x) ==> G(y)) ==> G(x)) /\\ " ^
          "((exists x. P(x) /\\ ~G(x)) ==> " ^
            "(exists x. P(x) /\\ ~G(x) /\\ (forall y. P(y) /\\ ~G(y) ==> J(x,y)))) /\\ " ^
          "(forall x y. P(x) /\\ P(y) /\\ H(x,y) ==> ~J(y,x))")!>)],
      conclude (<!"(forall x. P(x) ==> G(x))"!>) proof
      [
        fix "x",
        conclude (<!"P(x) ==> G(x)"!>) proof
        [
          assume [("B", <!"P(x)"!>)],
          conclude (<!"G(x)"!>) by ["B","A"], qed
        ], qed
      ], qed
    ]
```

```
>
```

Other Examples

ML `<`

```
fun by_mp (ab, a) p (Goals((asl,_)::_,_)) =
  let
    val ths = assumps asl
    val th = right_mp (assoc ab ths) (assoc a ths)
    handle Fail _ => raise Fail "by_mp: unapplicable assumptions"
  in
  if consequent (concl th) = p then [th] else raise Fail "by_mp: wrong conclusion"
  end
  | by_mp _ _ _ = raise Fail "Match by_mp"
;
```

```
>
```

ML `<fun auto s = prove (<!s!>) [our thesis at once, qed]>`

ML_val `<auto "A ==> A">`

ML_val `<auto "exists x. D(x) ==> forall x. D(x)">`

ML_val `<auto "(forall x. ~R(x) ==> R(f(x))) ==> exists x. R(x) /\\ R(f(f(x)))">`

ML_val `<(* Harrison p58 (as mentioned in the errata it is not Pelletier p58) *)`

```
auto "forall x. exists v w. forall y z. P(x) /\\ Q(y) ==> (P(v) \\/ R(w)) /\\ (R(z) ==> Q(v))"
```

```
>
```

ML_val `<(* Pelletier p1 *)`

```
auto "p ==> q <=> ~q ==> ~p"
```

```
>
```

ML__val `<(* Pelletier p2 *)`

```
auto "~ ~p <=> p"
```

```
>
```

ML__val `<(* Pelletier p3 *)`

```
auto "~(p ==> q) ==> q ==> p"
```

```
>
```

ML__val `<(* Pelletier p4 *)`

```
auto "~p ==> q <=> ~q ==> p"
```

```
>
```

ML__val `<(* Pelletier p5 *)`

```
auto "(p \\/ q ==> p \\/ r) ==> p \\/ (q ==> r)"
```

```
>
```

ML__val `<(* Pelletier p6 *)`

```
auto "p \\/ ~p"
```

```
>
```

ML__val `<(* Pelletier p7 *)`

```
auto "p \\/ ~ ~ ~p"
```

```
>
```

ML__val `<(* Pelletier p8 *)`

```
auto "((p ==> q) ==> p) ==> p"
```

```
>
```

ML__val `<(* Pelletier p9 *)`

```
auto "(p \\/ q) /\\ (~p \\/ q) /\\ (p \\/ ~q) ==> ~(~q \\/ ~q)"
```

```
>
```

ML__val `<(* Pelletier p10 *)`

```
auto "(q ==> r) /\\ (r ==> p /\\ q) /\\ (p ==> q /\\ r) ==> (p <=> q)"
```

```
>
```

ML__val `<(* Pelletier p11 *)`

```
auto "p <=> p"
```

```
>
```

```
ML_val <(* Pelletier p12 *)

auto "((p <=> q) <=> r) <=> (p <=> (q <=> r))"

>

ML_val <(* Pelletier p13 *)

auto "p \\/ q /\\ r <=> (p \\/ q) /\\ (p \\/ r)"

>

ML_val <(* Pelletier p14 *)

auto "(p <=> q) <=> (q \\/ ~p) /\\ (~q \\/ p)"

>

ML_val <(* Pelletier p15 *)

auto "p ==> q <=> ~p \\/ q"

>

ML_val <(* Pelletier p16 *)

auto "(p ==> q) \\/ (q ==> p)"

>

ML_val <(* Pelletier p17 *)

auto "p /\\ (q ==> r) ==> s <=> (~p \\/ q \\/ s) /\\ (~p \\/ ~r \\/ s)"

>

ML_val <(* Pelletier p18 *)

auto "exists y. forall x. P(y) ==> P(x)"

>

ML_val <(* Pelletier p19 *)

auto "exists x. forall y z. (P(y) ==> Q(z)) ==> P(x) ==> Q(x)"

>

ML_val <(* Pelletier p20 *)

auto ("(forall x y. exists z. forall w. P(x) /\\ Q(y) ==> R(z) /\\ U(w)) " ^
      "==> (exists x y. P(x) /\\ Q(y)) ==> (exists z. R(z))")

>

ML_val <(* Pelletier p21 *)

auto "(exists x. P ==> Q(x)) /\\ (exists x. Q(x) ==> P) ==> (exists x. P <=> Q(x))"

>

ML_val <(* Pelletier p22 *)
```

```
auto "(forall x. P <=> Q(x)) ==> (P <=> (forall x. Q(x)))"

>
```

ML_val `<(* Pelletier p23 *)`

```
auto "(forall x. P \\/ Q(x)) <=> P \\/ (forall x. Q(x))"

>
```

ML_val `<(* Pelletier p24 *)`

```
auto ("~(exists x. U(x) /\\ Q(x)) /\\ " ^
      "(forall x. P(x) ==> Q(x) \\/ R(x)) /\\ " ^
      "~(exists x. P(x) ==> (exists x. Q(x))) /\\ " ^
      "(forall x. Q(x) /\\ R(x) ==> U(x)) " ^
      "==> (exists x. P(x) /\\ R(x))")

>
```

ML_val `<(* Pelletier p25 *)`

```
auto ("(exists x. P(x)) /\\ " ^
      "(forall x. U(x) ==> ~G(x) /\\ R(x)) /\\ " ^
      "(forall x. P(x) ==> G(x) /\\ U(x)) /\\ " ^
      "((forall x. P(x) ==> Q(x)) \\/ (exists x. Q(x) /\\ P(x))) " ^
      "==> (exists x. Q(x) /\\ P(x))")

>
```

ML_val `<(* Pelletier p26 *)`

```
auto ("((exists x. P(x)) <=> (exists x. Q(x))) /\\ " ^
      "(forall x y. P(x) /\\ Q(y) ==> (R(x) <=> U(y))) " ^
      "==> ((forall x. P(x) ==> R(x)) <=> (forall x. Q(x) ==> U(x)))")

>
```

ML_val `<(* Pelletier p27 *)`

```
auto ("(exists x. P(x) /\\ ~Q(x)) /\\ " ^
      "(forall x. P(x) ==> R(x)) /\\ " ^
      "(forall x. U(x) /\\ V(x) ==> P(x)) /\\ " ^
      "(exists x. R(x) /\\ ~Q(x)) " ^
      "==> (forall x. V(x) ==> ~R(x)) ==> (forall x. U(x) ==> ~V(x))")

>
```

ML_val `<(* Pelletier p28 *)`

```
auto ("(forall x. P(x) ==> (forall x. Q(x))) /\\ " ^
      "((forall x. Q(x) \\/ R(x)) ==> (exists x. Q(x) /\\ R(x))) /\\ " ^
      "((exists x. R(x)) ==> (forall x. L(x) ==> M(x))) " ^
      "==> (forall x. P(x) /\\ L(x) ==> M(x))")

>
```

ML_val `<(* Pelletier p29 *)`

```
auto ("(exists x. P(x)) /\\ (exists x. G(x)) ==> " ^
      "((forall x. P(x) ==> H(x)) /\\ (forall x. G(x) ==> J(x)) " ^
      "<=> (forall x y. P(x) /\\ G(y) ==> H(x) /\\ J(y)))")

>
```

ML_val `<(* Pelletier p30 *)`

```
auto ("(forall x. P(x) \\/ G(x) ==> ~H(x)) /\\ " ^
      "(forall x. (G(x) ==> ~U(x)) ==> P(x) /\\ H(x)) " ^
      "==> (forall x. U(x))")

>
```

ML_val `<(* Pelletier p31 *)`

```
auto ("~(exists x. P(x) /\\ (G(x) \\/ H(x))) /\\ " ^
      "(exists x. Q(x) /\\ P(x)) /\\ " ^
      "(forall x. ~H(x) ==> J(x)) " ^
      "==> (exists x. Q(x) /\\ J(x))")

>
```

ML_val `<(* Pelletier p32 *)`

```
auto ("(forall x. P(x) /\\ (G(x) \\/ H(x)) ==> Q(x)) /\\ " ^
      "(forall x. Q(x) /\\ H(x) ==> J(x)) /\\ " ^
      "(forall x. R(x) ==> H(x)) " ^
      "==> (forall x. P(x) /\\ R(x) ==> J(x))")

>
```

ML_val `<(* Pelletier p33 *)`

```
auto ("(forall x. P(a) /\\ (P(x) ==> P(b)) ==> P(c)) " ^
      "<=> (forall x. P(a) ==> P(x) \\/ P(c)) /\\ (P(a) ==> P(b) ==> P(c))")

>
```

ML_val `<(* Pelletier p34 *)`

```
prove
(<!("((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) <=>"
^   "((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y))))")!>)
[

  note ("directions",
  <!("(((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) ==>"
^    "((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))) /\\"
^    "(((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))) ==>"
^    "((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))))")!>)
  proof
  [
    conclude
    (<!("((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) ==>"
^       "((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y))))")!>)
    proof
    [
      assume [("A",
      <!"(exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))"!>)],

      note ("ant",
      <!("((exists x. forall y. Q(x) <=> Q(y)) ==> ((exists x. P(x)) <=> (forall y. P(y)))) /\\"
^        "(((exists x. P(x)) <=> (forall y. P(y))) ==> ((exists x. forall y. Q(x) <=> Q(y))))")!>)
      proof
      [

        conclude
        (<!"(exists x. forall y. Q(x) <=> Q(y)) ==> ((exists x. P(x)) <=> (forall y. P(y)))"!>)
```

```
      proof
      [
        assume [("", <!"exists x. forall y. Q(x) <=> Q(y)"!>)],
        so have (<!"(exists x. Q(x)) <=> (forall y. Q(y))"!>) at once,
        so have (<!"exists x. forall y. P(x) <=> P(y)"!>) by ["A"],
        so our thesis at once,
        qed
      ],

      conclude
      (<!"((exists x. P(x)) <=> (forall y. P(y))) ==> (exists x. forall y. Q(x) <=> Q(y))"!>)
      proof
      [
        note ("imp",
        <!("((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) ==>"
^          "((exists x. forall y. P(x) <=> P(y)) ==> ((exists x. Q(x)) <=> (forall y. Q(y))))")!>)
        using [axiom_iffimp1
          (<!"exists x. forall y. P(x) <=> P(y)"!>)
          (<!"(exists x. Q(x)) <=> (forall y. Q(y))"!>)],
        note ("A1",
         <!"(exists x. forall y. P(x) <=> P(y)) ==> ((exists x. Q(x)) <=> (forall y. Q(y)))"!>)
        by_mp ("imp", "A"),

        assume [("", <!"((exists x. P(x)) <=> (forall y. P(y)))"!>)],
        so have (<!"(exists x. forall y. P(x) <=> P(y))"!>) at once,
        so have (<!"((exists x. Q(x)) <=> (forall y. Q(y)))"!>) by ["A1"],
        so our thesis at once,
        qed
      ],
      qed
    ],

    note ("imp",
    <!("((exists x. forall y. Q(x) <=> Q(y)) ==> ((exists x. P(x)) <=> (forall y. P(y)))) /\\"
^      "(((exists x. P(x)) <=> (forall y. P(y))) ==> ((exists x. forall y. Q(x) <=> Q(y)))) ==>"
^      "((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y))))")!>)
    using [unshunt (axiom_impiff
      (<!"exists x. forall y. Q(x) <=> Q(y)"!>)
      (<!"(exists x. P(x)) <=> (forall y. P(y))"!>))],

    our thesis by_mp ("imp", "ant"),
    qed
  ],

  conclude
  (<!("((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))) ==>"
^     "((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y))))")!>)
  proof
  [
    assume [("A",
     <!"(exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))"!>)],

    note ("ant",
    <!("((exists x. forall y. P(x) <=> P(y)) ==> ((exists x. Q(x)) <=> (forall y. Q(y)))) /\\"
^      "(((exists x. Q(x)) <=> (forall y. Q(y))) ==> (exists x. forall y. P(x) <=> P(y)))")!>)
    proof
    [

      conclude
      (<!"(exists x. forall y. P(x) <=> P(y)) ==> ((exists x. Q(x)) <=> (forall y. Q(y)))"!>)
      proof
      [
        assume [("", <!"exists x. forall y. P(x) <=> P(y)"!>)],
        so have (<!"(exists x. P(x)) <=> (forall y. P(y))"!>) at once,
```

```
          so have (<!"exists x. forall y. Q(x) <=> Q(y)"!>) by ["A"],
          so our thesis at once,
          qed
        ],

        conclude
        (<!"((exists x. Q(x)) <=> (forall y. Q(y))) ==> (exists x. forall y. P(x) <=> P(y))"!>)
        proof
        [
          note ("imp",
          <!("((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))) ==>"
^           "((exists x. forall y. Q(x) <=> Q(y)) ==> ((exists x. P(x)) <=> (forall y. P(y))))")!>)
          using [axiom_iffimp1
            (<!"exists x. forall y. Q(x) <=> Q(y)"!>)
            (<!"(exists x. P(x)) <=> (forall y. P(y))"!>)],
          note ("A1",
          <!"(exists x. forall y. Q(x) <=> Q(y)) ==> ((exists x. P(x)) <=> (forall y. P(y)))"!>)
          by_mp ("imp", "A"),

          assume [("", <!"(exists x. Q(x)) <=> (forall y. Q(y))"!>)],
          so have (<!"(exists x. forall y. Q(x) <=> Q(y))"!>) at once,
          so have (<!"((exists x. P(x)) <=> (forall y. P(y)))"!>) by ["A1"],
          so our thesis at once,
          qed
        ],
        qed
      ],

      note ("imp",
      <!("((exists x. forall y. P(x) <=> P(y)) ==> ((exists x. Q(x)) <=> (forall y. Q(y)))) /\\"
^        "(((exists x. Q(x)) <=> (forall y. Q(y))) ==> (exists x. forall y. P(x) <=> P(y))) ==>"
^        "((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y))))")!>)
      using [unshunt (axiom_impiff
        (<!"exists x. forall y. P(x) <=> P(y)"!>)
        (<!"(exists x. Q(x)) <=> (forall y. Q(y))"!>))],

      our thesis by_mp ("imp", "ant"),
      qed
    ],
    qed
  ],

  note ("impiff",
  <!(("(((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) ==>"
^     "((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))) /\\"
^     "(((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))) ==>"
^     "((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) ==>"
^     "(((exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))) <=>"
^     "((exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))))")!>)
  using [unshunt (axiom_impiff
    (<!("(exists x. forall y. P(x) <=> P(y)) <=> ((exists x. Q(x)) <=> (forall y. Q(y)))")!>)
    (<!("(exists x. forall y. Q(x) <=> Q(y)) <=> ((exists x. P(x)) <=> (forall y. P(y)))")!>))],

  our thesis by_mp ("impiff", "directions"),
  qed
]

>

ML_val <(* Pelletier p35 *)

auto "exists x y. P(x,y) ==> (forall x y. P(x,y))"

>
```

```
ML_val <(* Pelletier p36 *)

auto ("(forall x. exists y. P(x,y)) /\\ " ^
      "(forall x. exists y. G(x,y)) /\\ " ^
      "(forall x y. P(x,y) \\/ G(x,y) ==> (forall z. P(y,z) \\/ G(y,z) ==> H(x,z))) " ^
      "==> (forall x. exists y. H(x,y))")

>

ML_val <(* Pelletier p37 *)

auto ("(forall z. " ^
        "exists w. forall x. exists y. (P(x,z) ==> P(y,w)) /\\ P(y,z) /\\ " ^
        "(P(y,w) ==> (exists u. Q(u,w)))) /\\ " ^
      "(forall x z. ~P(x,z) ==> (exists y. Q(y,z))) /\\ " ^
      "((exists x y. Q(x,y)) ==> (forall x. R(x,x))) " ^
      "==> (forall x. exists y. R(x,y))")

>

ML_val <(* Pelletier p38 *)

auto ("(forall x. " ^
        "P(a) /\\ (P(x) ==> (exists y. P(y) /\\ R(x,y))) ==> " ^
        "(exists z w. P(z) /\\ R(x,w) /\\ R(w,z))) <=> " ^
      "(forall x. " ^
        "(~P(a) \\/ P(x) \\/ (exists z w. P(z) /\\ R(x,w) /\\ R(w,z))) /\\ " ^
        "(~P(a) \\/ ~(exists y. P(y) /\\ R(x,y)) \\/ " ^
        "(exists z w. P(z) /\\ R(x,w) /\\ R(w,z))))")

>

ML_val <(* Pelletier p39 *)

auto "~(exists x. forall y. P(y,x) <=> ~P(y,y))"

>

ML_val <(* Pelletier p40 *)

auto ("(exists y. forall x. P(x,y) <=> P(x,x)) " ^
      "==> ~(forall x. exists y. forall z. P(z,y) <=> ~P(z,x))")

>

ML_val <(* Pelletier p41 *)

auto ("(forall z. exists y. forall x. P(x,y) <=> P(x,z) /\\ ~P(x,x)) " ^
      "==> ~(exists z. forall x. P(x,z))")

>

ML_val <(* Pelletier p42 *)

auto "~(exists y. forall x. P(x,y) <=> ~(exists z. P(x,z) /\\ P(z,x)))"

>

ML_val <(* Pelletier p44 *)

auto ("(forall x. P(x) ==> (exists y. G(y) /\\ H(x,y)) /\\ " ^
      "(exists y. G(y) /\\ ~H(x,y))) /\\ " ^
      "(exists x. J(x) /\\ (forall y. G(y) ==> H(x,y))) ==> " ^
```

```
      "(exists x. J(x) /\\ ~P(x))")

>

ML_val <(* Pelletier p45 *)

auto ("(forall x. " ^
        "P(x) /\\ (forall y. G(y) /\\ H(x,y) ==> J(x,y)) ==> " ^
             "(forall y. G(y) /\\ H(x,y) ==> R(y))) /\\ " ^
      "~(exists y. L(y) /\\ R(y)) /\\ " ^
      "(exists x. P(x) /\\ (forall y. H(x,y) ==> " ^
        "L(y)) /\\ (forall y. G(y) /\\ H(x,y) ==> J(x,y))) ==> " ^
      "(exists x. P(x) /\\ ~(exists y. G(y) /\\ H(x,y)))")

>

ML_val <(* Pelletier p55 *)

auto ("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(agatha,agatha) /\\ " ^
             "~killed(butler,agatha) /\\ " ^
             "~killed(charles,agatha)")

>

ML_val <(* Pelletier p57 *)

auto ("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)) " ^
      "==> P(f(a,b),f(a,c))")

>

ML_val <(* Pelletier p59 *)

auto "(forall x. P(x) <=> ~P(f(x))) ==> (exists x. P(x) /\\ ~P(f(x)))"

>

ML_val <(* Pelletier p60 *)

auto "forall x. P(x,f(x)) <=> exists y. (forall z. P(z,y) ==> P(z,f(x))) /\\ P(x,y)"

>

ML_val <(* gilmore_3 *)

auto ("exists x. forall y z. " ^
             "((F(y,z) ==> (G(y) ==> H(x))) ==> F(x,x)) /\\ " ^
             "((F(z,x) ==> G(x)) ==> H(z)) /\\ " ^
             "F(x,y) " ^
             "==> F(z,z)")

>
```

```
ML_val <(* gilmore_4 *)

auto ("exists x y. forall z. " ^
             "(F(x,y) ==> F(y,z) /\\ F(z,z)) /\\ " ^
             "(F(x,y) /\\ G(x,y) ==> G(x,z) /\\ G(z,z))")

>

ML_val <(* gilmore_5 *)

auto ("(forall x. exists y. F(x,y) \\/ F(y,x)) /\\ " ^
      "(forall x y. F(y,x) ==> F(y,y)) " ^
      "==> exists z. F(z,z)")

>

ML_val <(* gilmore_6 *)

auto ("forall x. exists y. " ^
             "(exists u. forall v. F(u,x) ==> G(v,u) /\\ G(u,x)) " ^
             "==> (exists u. forall v. F(u,y) ==> G(v,u) /\\ G(u,y)) \\/ " ^
                       "(forall u v. exists w. G(v,u) \\/ H(w,y,u) ==> G(u,w))")

>

ML_val <(* gilmore_7 *)

auto ("(forall x. K(x) ==> exists y. L(y) /\\ (F(x,y) ==> G(x,y))) /\\ " ^
      "(exists z. K(z) /\\ forall u. L(u) ==> F(z,u)) " ^
      "==> exists v w. K(v) /\\ L(w) /\\ G(v,w)")

>

ML_val <(* gilmore_8 *)

auto ("exists x. forall y z. " ^
             "((F(y,z) ==> (G(y) ==> (forall u. exists v. H(u,v,x)))) ==> F(x,x)) /\\ " ^
             "((F(z,x) ==> G(x)) ==> (forall u. exists v. H(u,v,z))) /\\ " ^
             "F(x,y) " ^
             "==> F(z,z)")

>

ML_val <(* gilmore_9 *)

auto ("forall x. exists y. forall z. " ^
             "((forall u. exists v. F(y,u,v) /\\ G(y,u) /\\ ~H(y,x)) " ^
                  "==> (forall u. exists v. F(x,u,v) /\\ G(z,u) /\\ ~H(x,z)) " ^
                  "==> (forall u. exists v. F(x,u,v) /\\ G(y,u) /\\ ~H(x,y))) /\\ " ^
             "((forall u. exists v. F(x,u,v) /\\ G(y,u) /\\ ~H(x,y)) " ^
                 "==> ~(forall u. exists v. F(x,u,v) /\\ G(z,u) /\\ ~H(x,z)) " ^
                 "==> (forall u. exists v. F(y,u,v) /\\ G(y,u) /\\ ~H(y,x)) /\\ " ^
                       "(forall u. exists v. F(z,u,v) /\\ G(y,u) /\\ ~H(z,y)))")

>

ML_val <(* davis_putnam_example *)

auto ("exists x. exists y. forall z. " ^
      "(F(x,y) ==> (F(y,z) /\\ F(z,z))) /\\ " ^
      "((F(x,y) /\\ G(x,y)) ==> (G(x,z) /\\ G(z,z)))")

>
```

ML__val ‹(* ewd1062_1 *)

```
auto ("(forall x. x <= x) /\\ " ^
      "(forall x y z. x <= y /\\ y <= z ==> x <= z) /\\ " ^
      "(forall x y. f(x) <= y <=> x <= g(y)) " ^
      "==> (forall x y. x <= y ==> f(x) <= f(y))")
```

›

ML__val ‹(* ewd1062_2 *)

```
auto ("(forall x. x <= x) /\\ " ^
      "(forall x y z. x <= y /\\ y <= z ==> x <= z) /\\ " ^
      "(forall x y. f(x) <= y <=> x <= g(y)) " ^
      "==> (forall x y. x <= y ==> g(x) <= g(y))")
```

›

end

Acknowledgements

The SML code is based on the OCaml code accompanying John Harrison's Handbook of Practical Logic and Automated Reasoning, Cambridge University Press, 2009

Thanks to Jasmin Blanchette, Asta Halkjær From, John Bruntse Larsen, Andrei Popescu and Tom Ridge for discussions.